

---

# **OpenSeesPy Documentation**

***Release 0.4.2019.7***

**Minjie Zhu**

**Dec 09, 2019**



---

## Contents

---

<b>1 Developer</b>	<b>3</b>
--------------------	----------

<b>Index</b>	<b>349</b>
--------------	------------



---

**Important:** Version 0.5.8 is released!

Mac version is supported!

OpenSeesPy has been official in *DesignSafe*.

The Linux Version of OpenSeesPy can now run on Windows 10 (see *Windows Sybsystem for Linux*).

The latest version of this document can be found at <https://openseespydoc.readthedocs.io/en/latest/>.

---

**Note:** Please send any questions to [github issues](#).

You are very welcome to contribute to OpenSeesPy with new command documents and examples by sending pull requests through [github pulls](#).

---

OpenSeesPy is a Python 3 interpreter of OpenSees. A minimum script is shown below:

```
# import OpenSeesPy
import openseespy.opensees as ops

# wipe model
ops.wipe()

# create model
ops.model('-ndm', ndm, '-ndf', ndf)
```

Most of OpenSeesPy commands have the same syntax and arguments as the OpenSees [Tcl commands](#). The conversion from Tcl to Python is easy and straightforward as demonstrated with commands below.



# CHAPTER 1

---

## Developer

---

*Minjie Zhu <zhum@oregonstate.edu>*

Research Associate Post Doc  
Civil and Construction Engineering  
Oregon State University

## 1.1 Installation

1. *PyPi*
2. *DesignSafe*
3. *Windows Subsystem for Linux*

### 1.1.1 PyPi

#### Install Python

- Install Anaconda

#### Install OpenSeesPy

- To install

```
python -m pip install openseespy
python -m pip install --user openseespy
```

- To upgrade

```
python -m pip install --upgrade openseespy  
python -m pip install --user --upgrade openseespy
```

### Import OpenSeesPy

```
import openseespy.opensees as ops
```

### 1.1.2 DesignSafe

OpenSeesPy has been official in DesignSafe.

1. *OpenSeesPy in DesignSafe*
2. *Paraview in DesignSafe*

### OpenSeesPy in DesignSafe

Follow steps below to run OpenSeesPy in DesignSafe.

---

#### Tip:

- Go To DesignSafe website and click register



---

#### Tip:

- Register an account for free and log in

**REGISTER AN ACCOUNT**

Users may register for only one account using the form below. DesignSafe uses the TACC Identity Service, thus you will be registering for a TACC account. Upon submitting this form check your email, possibly your Spam folder, for an email message containing a link to confirm and activate your new account.

**First name\***

**Last name\***

**Email\***

**Confirm Email\***

**Phone\***

**Institution\***

[? My Institution is not listed](#)

**Position/TITLE\***

**Country of residence\***

**TACC**

**Already have an Account? [Log in.](#)**

If you already have a TACC account, simply [log in](#) with your TACC username and password to access DesignSafe.

You may have a TACC account if you have used XSEDE or TeraGrid in the past and had an allocation on a TACC resource.

---

**Tip:**

- Land on your own portal and go to workspace

The screenshot shows the main interface of the DesignSafe-CI Research Workbench. At the top, there's a navigation bar with links for "Research Workbench", "Learning Center", "NHERI Facilities", "NHERI Community", "About", and "Help". A search bar and a user profile dropdown ("Welcome, Minjel!") are also at the top right. Below the header, there's a banner for "NHERI: A NATURAL HAZARDS ENGINEERING RESEARCH INFRASTRUCTURE" featuring a map of the United States. The main content area includes a sidebar with links to "Data Depot", "Workspace", "Recon Portal", "SimCenter Research Tools", and "User Guides". It also features three large cards: one showing "0 Jobs", another showing "38 Available Apps", and a third showing "0.0 bytes Data Stored". To the right, there's a "Quick Links" sidebar with links to "Manage Account", "Data Depot", "Workspace", "Recon Portal", and "Training". Below these are sections for "My Jobs" (with a message about recent jobs), "Notifications" (showing 0 notifications), "My Tickets" (with a "Create New" button), and "Recent Apps" (with a message about recent apps). At the bottom, there's a note from the National Science Foundation (NSF) supporting the project.

**Tip:**

- In your workspace, select Jupyter, launch it, and start my server

NHERI: A NATURAL HAZARDS ENGINEERING RESEARCH INFRASTRUCTURE

Research Workbench ▾ Learning Center ▾ NHERI Facilities ▾ NHERI Community ▾ About Help ▾

Search DesignSafe 

## WORKSPACE

Learn About the Workspace.

Simulation [9] Visualization [7] Data Processing [2] Partner Data Apps [4] Utilities [2] My Apps [5]

PADCIRC SWAN (Stampede2) ADCIRC	Jupyter	MATLAB R2017b	Potree Converter	Potree Viewer	
------------------------------------	---------	---------------	------------------	---------------	--

Browsing: zhum

File name	Size
.ipynb_checkpoints	4 kB
.Trash-458981	4 kB
examples	4 kB
FireConvert	4 kB
Truss.ipynb	3 kB

Select an application from the tray above.

The Workspace allows users to perform simulations and analyze data using popular simulation codes including OpenSees, ADCIRC, and OpenFOAM, as well as data analysis and visualization tools including Jupyter, MATLAB, Paraview and VisIt.



DesignSafe-CI is supported by multiple grants from the National Science Foundation:  
[Cyberinfrastructure](#), [NCO](#), [SimCenter](#).

**Tip:**

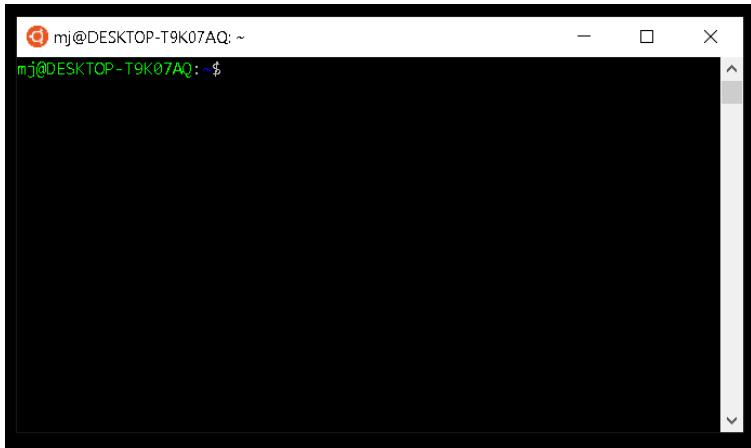
- Now you should be in the Jupyter Notebook
- Go to mydata and always save your data under this folder



---

**Tip:**

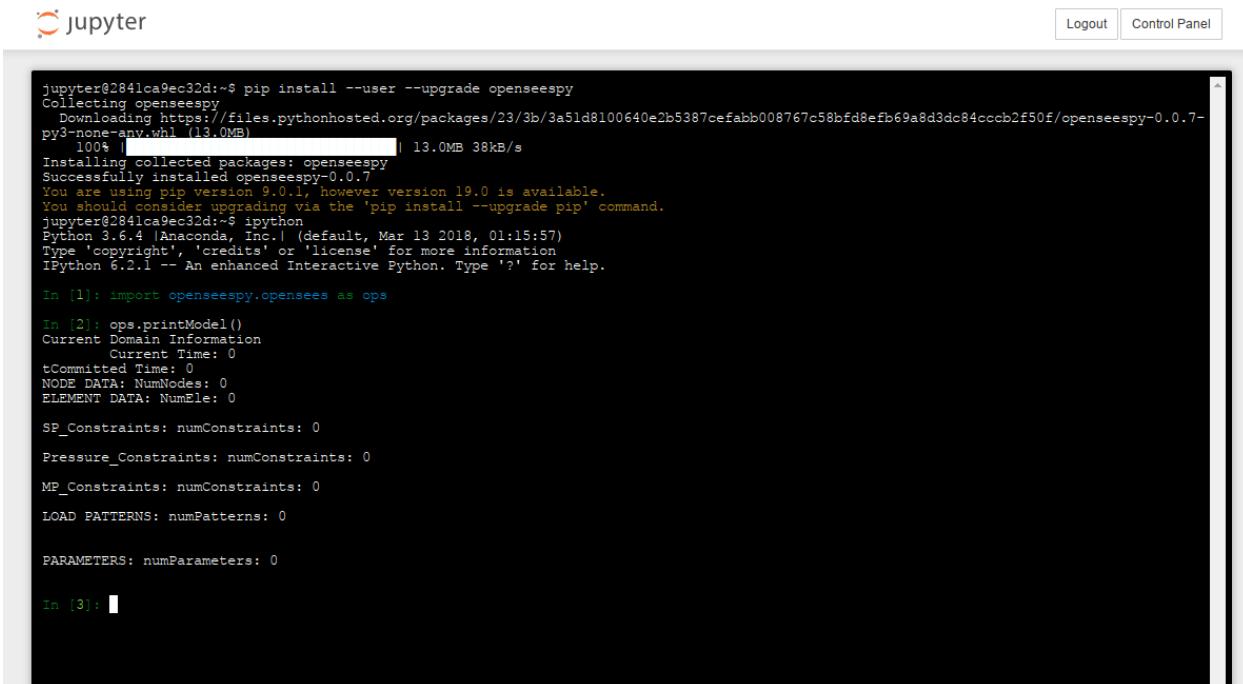
- In mydata folder, select New and then Terminal



---

**Tip:**

- The version on DesignSafe is not the latest.
- To update to the latest:
  - Type `pip install --user --upgrade openseespy` in the terminal
  - Check the installation by running `ipython` and `import openseespy.opensees as ops`
  - Everytime you start a new session, you have to update `openseespy` (this is not ideal, but it works for now).
- Now you may close the terminal by closing the brower tab.



The screenshot shows a terminal window within a Jupyter Notebook interface. The terminal output is as follows:

```
jupyter@2841ca9ec32d:~$ pip install --user --upgrade openseespy
Collecting openseespy
  Downloading https://files.pythonhosted.org/packages/23/3b/3a51d8100640e2b5387cefabb008767c58bfd8efb69a8d3dc84cccb2f50f/openseespy-0.0.7-py3-none-any.whl (13.0MB)
    100% |██████████| 13.0MB 38kB/s
Installing collected packages: openseespy
Successfully installed openseespy-0.0.7
You are using pip version 9.0.1, however version 19.0 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
jupyter@2841ca9ec32d:~$ ipython
Python 3.6.4 |Anaconda, Inc.| (default, Mar 13 2018, 01:15:57)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import openseespy.opensees as ops
In [2]: ops.printModel()
Current Domain Information
  Current Time: 0
  tCommitted Time: 0
  NODE DATA: NumNodes: 0
  ELEMENT DATA: NumEle: 0
  SP_Constraints: numConstraints: 0
  Pressure_Constraints: numConstraints: 0
  MP_Constraints: numConstraints: 0
  LOAD PATTERNS: numPatterns: 0
  PARAMETERS: numParameters: 0

In [3]:
```

---

**Tip:**

- Back to mydata folder, select New and then Python 3



---

**Tip:**

- Now you can write OpenSeesPy script in Jupyter Notebook, run the script, and show results

- To show figure in the Notebook, you should include `%matplotlib inline` at the beginning

The screenshot shows a Jupyter Notebook interface. At the top, there's a toolbar with File, Edit, View, Insert, Cell, Kernel, Widgets, Help, and a Trusted button. Below the toolbar, there are buttons for Run, Cell, Kernel, Help, and a Code dropdown. The main area has two code cells:

```
In [2]: import openseespy.opensees as ops  
%matplotlib inline  
import numpy as np  
import matplotlib.pyplot as plt  
  
ops.wipe()  
  
# set modelbuilder  
ops.model('basic', '-ndm', 2, '-ndf', 2)  
  
# create nodes  
ops.node(1, 0.0, 0.0)  
ops.node(2, 144.0, 0.0)  
ops.node(3, 168.0, 0.0)  
ops.node(4, 72.0, 96.0)  
  
# set boundary condition  
ops.fix(1, 1, 1)  
ops.fix(2, 1, 1)  
ops.fix(3, 1, 1)  
  
# plot x coordinates  
plt.plot([1,2,3,4],[ops.nodeCoord(1,1),ops.nodeCoord(2,1),ops.nodeCoord(3,1),ops.nodeCoord(4,1)])
```

Out[2]: [`<matplotlib.lines.Line2D at 0x7f1564b70da0>`]

Below the code, a plot is displayed showing a piecewise linear function. The x-axis ranges from 10 to 40, and the y-axis ranges from 0 to 175. The plot consists of four segments connecting points (10, 0), (20, 150), (30, 165), and (40, 75).

---

## Paraview in DesignSafe

DesignSafe provides paraview for viewing OpenSeesPy results:

---

### Tip:

- Make sure the steps in *OpenSeesPy in DesignSafe* are completed.
- 

---

### Tip:

- Go to Workspace and select Visualizaion and Paraview.

The screenshot shows the DesignSafe-CI workspace interface. At the top, there's a navigation bar with links for Research Workbench, Learning Center, NHERI Facilities, NHERI Community, About, Help, and a search bar. A "Welcome, Minjie!" message is displayed with a notification icon. Below the navigation is a section titled "WORKSPACE" with a "Learn About the Workspace" link. A grid of tool icons is shown, categorized by tabs: Simulation [9], Visualization [7], Data Processing [2], Partner Data Apps [4], Utilities [2], and My Apps [5]. The "Visualization" tab is selected, showing icons for FigureGen (F), Hazmapper (a globe icon), Paraview (three parallel bars), Potree Converter (P), Potree Viewer (P), and QGIS Desktop (Q). Below this is a "Visit" section featuring the VisIt logo. To the left of the workspace is a sidebar with a file browser showing a directory structure with files like ipynb\_checkpoints, Trash-458981, archive, examples, FireConvert, demo.ipynb, and Truss.ipynb. To the right of the workspace is a vertical sidebar labeled "Jobs Status". A text box in the center states: "as well as data analysis and visualization tools including Jupyter, MATLAB, Paraview and VisIt."

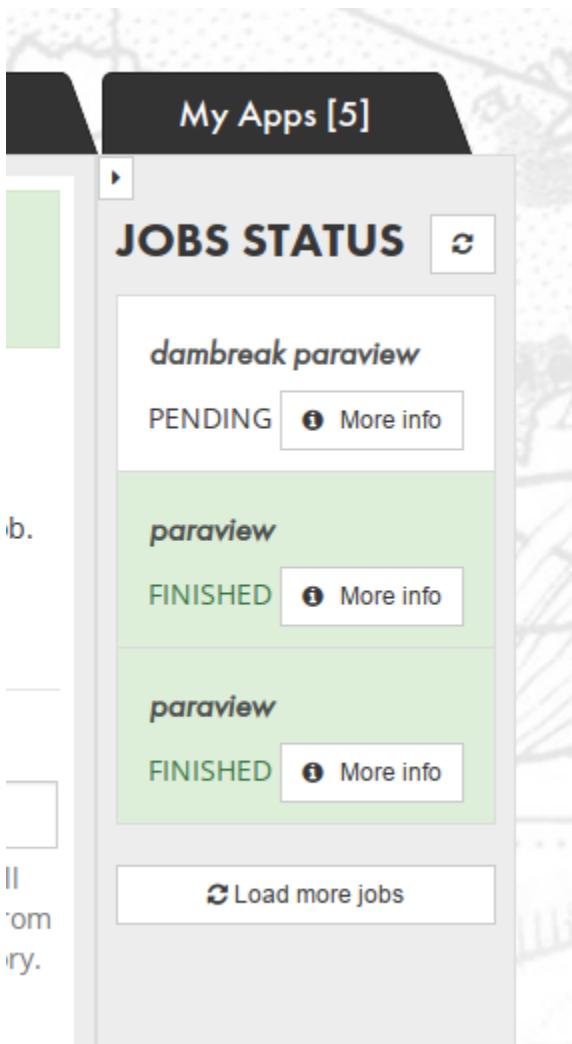
**Tip:**

- In the Job Submission windows, you should select Working Directory, Maximum job runtime, Job name, and Node Count, and click Run.

The screenshot shows the Data Depot Browser interface for running Paraview. At the top, there are tabs for Simulation [9], Visualization [7], Data Processing [2], Partner Data Apps [4], Utilities [2], and My Apps [5]. Below the tabs, the main area is titled "DATA DEPOT BROWSER" and "RUN PARAVIEW ver. 4.3.1". A sub-section titled "Select data source" shows a dropdown menu set to "My Data". On the left, a sidebar lists files under "Browsing: zhum", including ".ipynb\_checkpoints", ".Trash-458981", "archive", "examples", "FireConvert", "demo.ipynb", and "Truss.ipynb". The main panel contains sections for "Working Directory" (set to "agave://designsafe.storage.default/zhum/examples"), "Job details" (maximum runtime set to "01:00:00" in HH:MM:SS format), "Job name" (set to "paraview"), "Job output archive location (optional)" (set to "<username>/archive/jobs/\${YYYY-MM-DD}/\${JOB\_NAME}-\${JOB\_ID}"), and "Node Count" (set to "1"). At the bottom are "Run" and "Close" buttons.

## Tip:

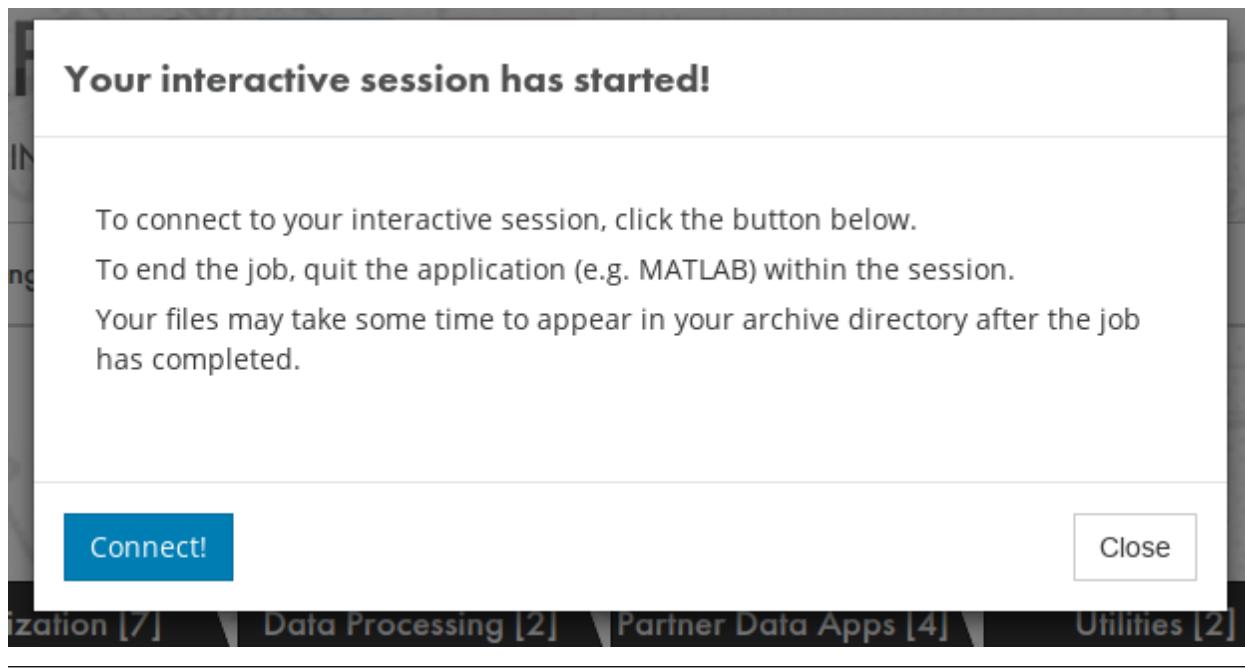
- You can see the job status on the right



---

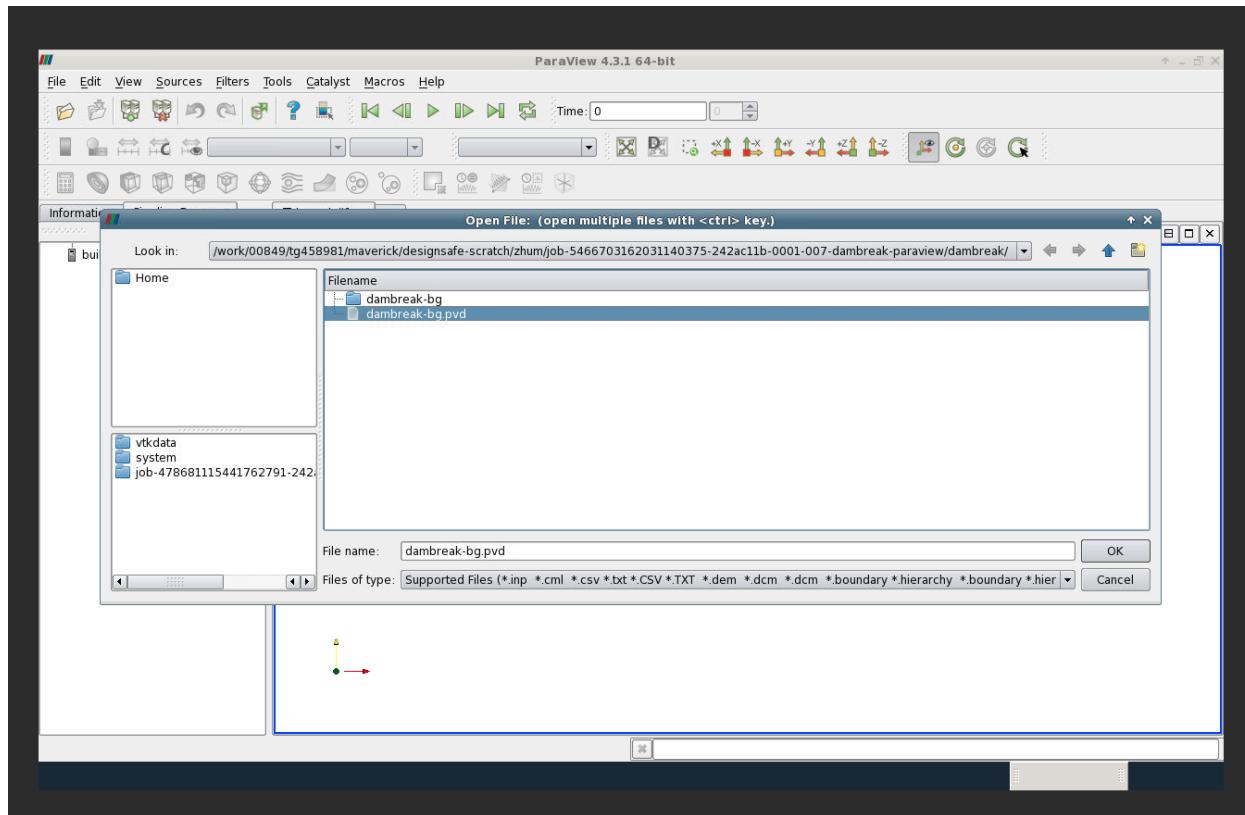
**Tip:**

- Wait until see this windows and connect to Paraview



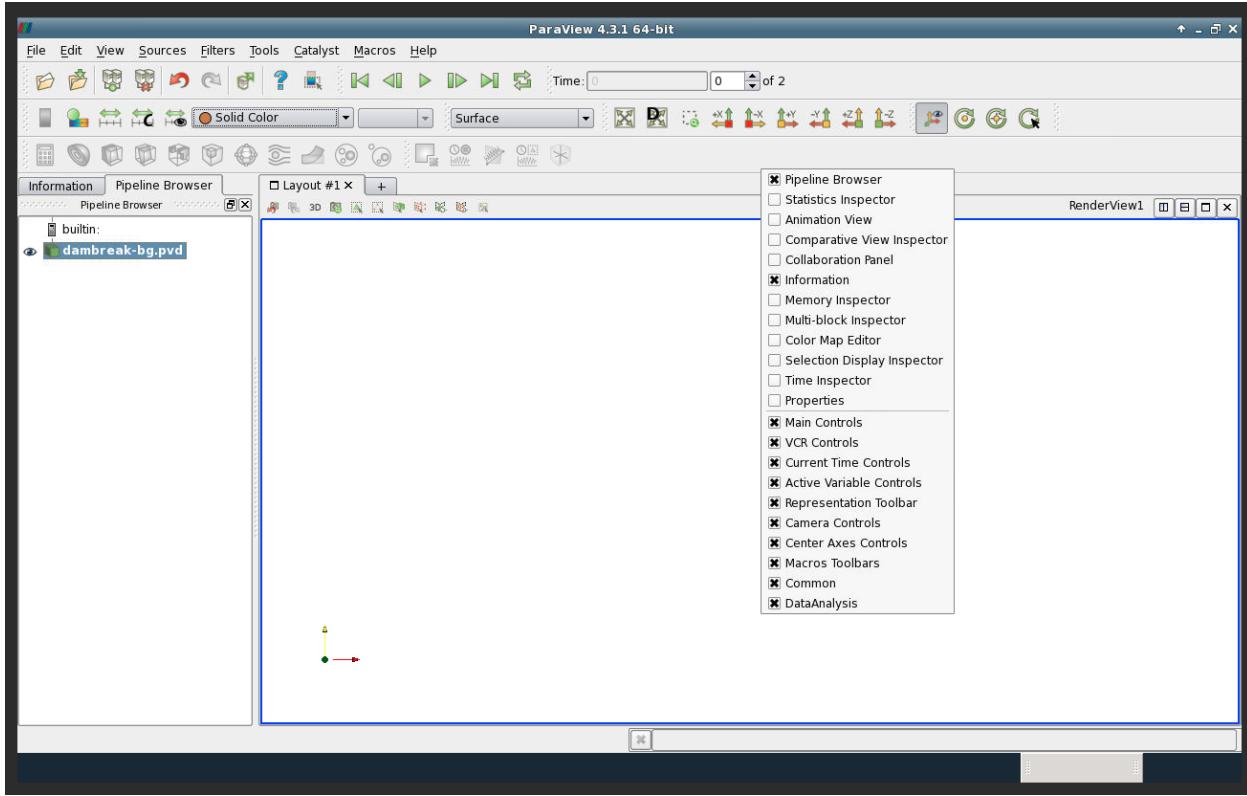
**Tip:**

- Now open the pvd file in Paraview

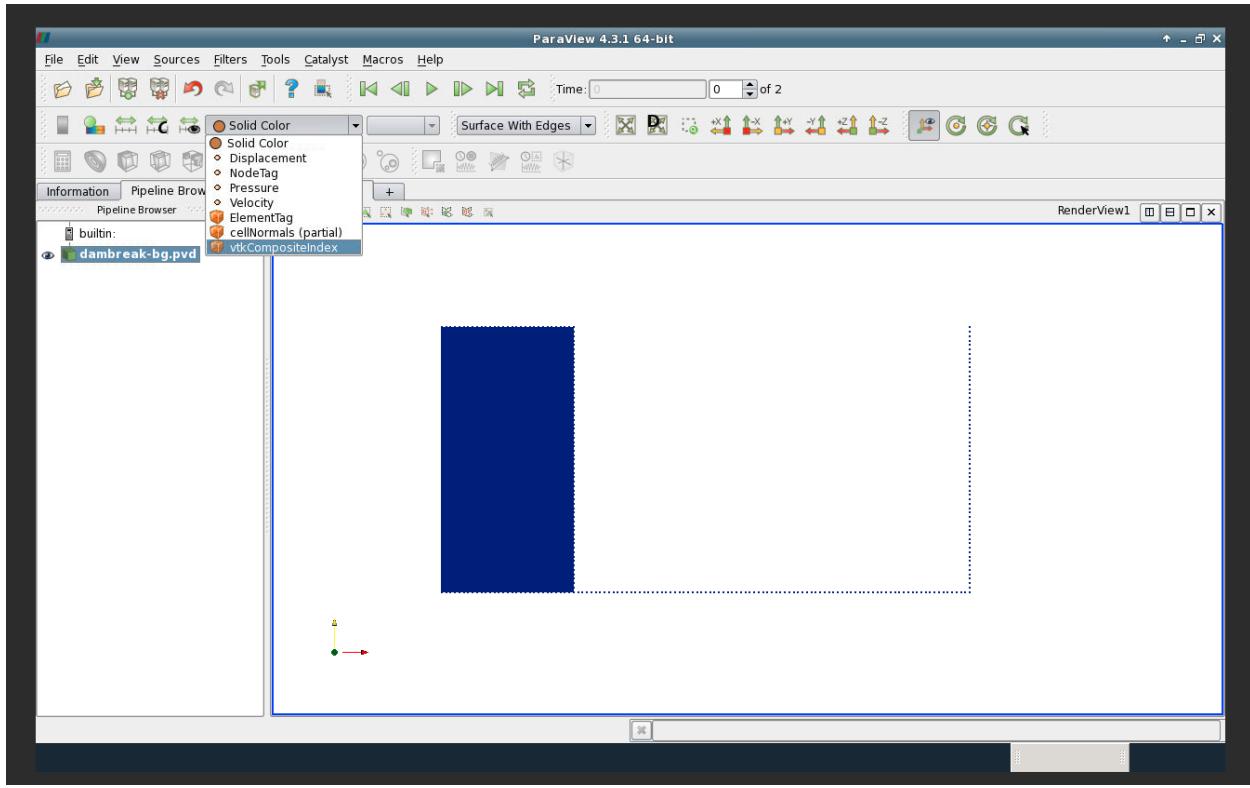


**Tip:**

- Initially, you see nothing
- Check the Pipeline Browser
- Click on the eye left to the file in Pipeline Browser

**Tip:**

- Change the Solid Color to other variables
- Change Surface to Surface With Edges

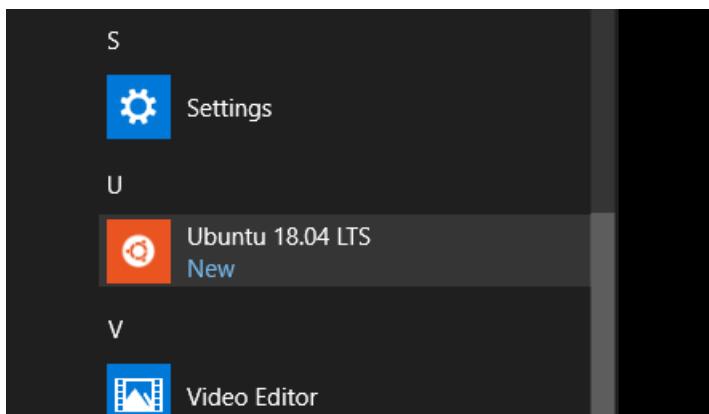


### 1.1.3 Windows Sybsystem for Linux

This is a real Linux sybsystem for you to run OpenSeesPy Linux version on Windows.

#### Install the Windows Sybsystem for Linux

Follow the instruction on [here](#) to install Windows Subsystem for Linux on Windows 10. There are a couple of Linux distributions available and Ubuntu is recommended. Once the Linux is installed, it will show as an application in the start menu.



## Run OpenSeesPy with Windows Sybsystem for Linux

- Run the subsystem from start menu and a terminal window will show.



- Download Anaconda Linux version with command

```
~$ wget https://repo.anaconda.com/archive/Anaconda3-2019.10-Linux-x86_64.sh
```

- Install Anconda Linux version with commands

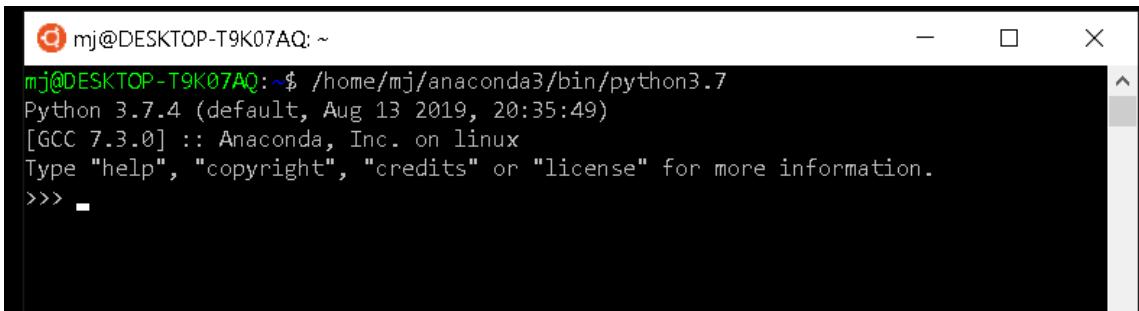
```
~$ bash Anaconda3-2019.10-Linux-x86_64.sh

>>> Please answer 'yes' or 'no':
>>> yes

>>> Anaconda3 will not be installed into this location:
[~/home/username/anaconda3] >>> (enter)
```

- Run Anaconda with follwing command, where *username* is your username of your computer. Please use the *username* shown in last step

```
~$ /home/username/anaconda3/bin/python3.7
```

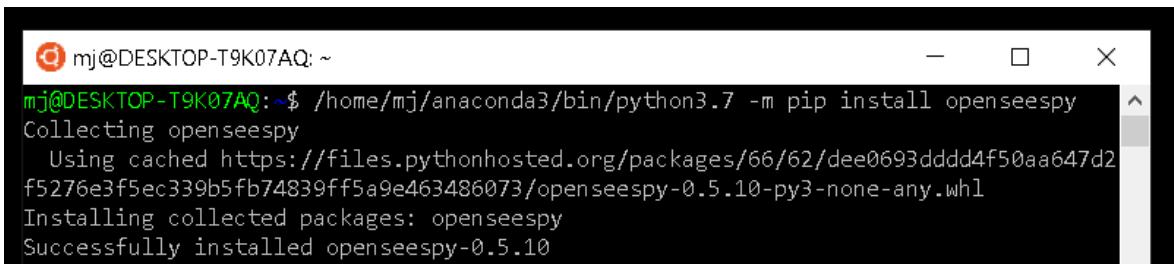


A terminal window titled 'mj@DESKTOP-T9K07AQ: ~'. It displays the Python 3.7.4 startup message:

```
mj@DESKTOP-T9K07AQ:~$ /home/mj/anaconda3/bin/python3.7
Python 3.7.4 (default, Aug 13 2019, 20:35:49)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

- Install or Upgrade OpenSeesPy with commands

```
~$ /home/username/anaconda3/bin/python3.7 -m pip install openseespy
~$ /home/username/anaconda3/bin/python3.7 -m pip install --upgrade openseespy
```



A terminal window titled 'mj@DESKTOP-T9K07AQ: ~'. It shows the execution of the pip command to install openseespy:

```
mj@DESKTOP-T9K07AQ:~$ /home/mj/anaconda3/bin/python3.7 -m pip install openseespy
Collecting openseespy
  Using cached https://files.pythonhosted.org/packages/66/62/dee0693dddd4f50aa647d2
f5276e3f5ec339b5fb74839ff5a9e463486073/openseespy-0.5.10-py3-none-any.whl
Installing collected packages: openseespy
Successfully installed openseespy-0.5.10
```

- Run OpenSeesPy

First run Anaconda with

```
~$ /home/username/anaconda3/bin/python3.7
```

Then import OpenSeesPy with

```
import openseespy.opensees as ops
ops.printModel()
```

```
mj@DESKTOP-T9K07AQ:~$ /home/mj/anaconda3/bin/python3.7
Python 3.7.4 (default, Aug 13 2019, 20:35:49)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import openseespy.opensees as ops
>>> ops.printModel()
Current Domain Information
    Current Time: 0
    tCommitted Time: 0
    NODE DATA: NumNodes: 0
    ELEMENT DATA: NumEle: 0

    SP_Constraints: numConstraints: 0

    Pressure_Constraints: numConstraints: 0

    MP_Constraints: numConstraints: 0

    LOAD PATTERNS: numPatterns: 0

    PARAMETERS: numParameters: 0

>>>
```

## Run OpenSeesPy scripts

In the terminal,

```
/home/username/anaconda3/bin/python3.7 script.py
```

## 1.2 Compilation

Following is the OpenSees Compilation using QT by

*Stevan Gavrilovic* [github](#)

PhD Candidate  
University of British Columbia

Original instructions can be found at [here](#).

### 1.2.1 A Qt build environment for OpenSees

OpenSeesQt

A Qt build environment for OpenSees – Open System For Earthquake Engineering Simulation Pacific Earthquake Engineering Research Center(<http://opensees.berkeley.edu>).

Qt is an open source, cross-platform development environment that is free for many uses. Please see the [license](#).

## 1.2.2 Dependency

The purpose of this project is to create a package that will allow aspiring developers to get started on writing code without having to worry about the compilation environment. A program as large as OpenSees relies on many third-party libraries, often referred to as dependencies. It can be a daunting task assembling, compiling, and linking these libraries. Many times, these libraries depend on other libraries, and so on. The current list of dependencies includes:

- MUMPS (5.1.2)
- Scalapack (2.0.2.13)
- UMFPACK (5.7.7) contained in Suite-Sparse (5.3.0)
- SUPERLU (5.2.1)
- SUPERLUMT (3.0)
- SUPERLUDIST (5.1.0)
- Openblas (0.3.5)
- Parmetis (4.0.3)
- ARPACK (3.6.3)
- Libevent (2.1.8)
- GCC(8.2.0)
- TCL (8.6.9)\*\* Tcl only
- Python(3.7.2)\*\* OpenSeesPy only

Please ensure that your project complies with each library's licensing requirements.

Another feature of this build platform is modularity. Developers can select from a list of build options that can be turned on and off as required. The basic configuration builds the structural analysis core. Other build options include parallel processing, reliability, and the particle finite element method (PFEM) modules. Python and Tcl interpreters are additional build options. These options are located in single configuration file called qmake.conf. By default, the environment is configured to build the core along with the parallel processing module. Other options can be turned on by deleting the '#' symbol that precedes the option; this includes the option into the build environment.

---

**Note:** Note: This build environment comes with pre-compiled dependencies. Although this makes getting started easier, the caveat is that your build environment (compiler version) must match that of the environment used to compile the dependencies. The supported environments are listed below. There are no guarantees that it will work with other build environments. In other words, make sure your compiler type and version (i.e., clang-1000.11.45.5) matches the version below listed under the heading 'Supported Build Environments'. Otherwise, bad things might happen. Also, this project is still a work in progress. Currently, only building in OS X is supported. Windows support will be added shortly. Moreover, not all build options are supported. For example, compiling with fortran is not supported.

---

**Note:** This project uses qmake and Qt Creator. qmake is a build tool for compiling and linking applications. Qt Creator is a free IDE (interactive development environment) that bundles code writing/editing and application building

within one program. Qt Creator uses project (.pro) files. The project files contain all information required by qmake to build an application.

---

### 1.2.3 Getting started:

1. Download and install Qt open source from <https://www.qt.io/download> The version of the Qt library is not important in this case since the library is not used in the OpenSees project (although I use Qt in other projects and recommend it)
2. Download the OpenSeesQt source code into a folder of your choice (<https://github.com/steva44/OpenSees/archive/master.zip>)
3. In the directory containing the code, double click on the ‘OpenSees.pro’ file. If compiling OpenSeesPy, open the ‘OpenSeesPy.pro’ file. The .pro files are project files that will automatically open the project in Qt Creator.
4. Select a build configuration, for example ‘Desktop Qt 5.12.1 clang 64bit’. The project will automatically configure itself. You only have to do this once.
5. The left-hand pane should now display the project directory structure. In the left-hand pane, under the heading qmake, open the ‘qmake.conf’ file. Review and select additional build options, if any. Note that this is still a work in progress and not all build options are supported.
6. Click on the ‘start’ button in the bottom lefthand corner of Qt Creator to compile. Clicking on the small computer symbol above the start button allows for switching between the debug and release deploy configurations. The release deployment results in faster program execution but it does not allow for debugging or stepping through the code. The start button with the bug symbol opens the debugger.
7. Go and have a coffee, it will take a few minutes to finish compiling!

### 1.2.4 Building OpenSeesPy:

OpenSeesPy builds OpenSees as a library object that can be used within Python.

Steps: Follow steps 1-4 under the heading getting started above.

1. The left-hand pane should now display the project directory structure. In the left-hand pane, under the heading qmake, open the ‘qmake.conf’ file. Under the heading #INTERPRETERS, uncomment the \_PYTHON option by removing the '#' symbol. Everything else should be configured automatically going forward. Python automatically compiles with the reliability, parallel, and PFEM modules.
2. The last few lines at the end of the ‘OpenSeesPy.pro’ file contain the location of the Python framework. Update this so that it matches the location of Python on your build system.
3. Click on the ‘start’ button in the bottom lefthand corner of Qt Creator to start compiling. Clicking on the small computer symbol allows for switching between the debug and release deploy configurations. The release deployment results in faster program execution but it does not allow for debugging or stepping through the code. Build in release mode if using OpenSees as a library in a Python project.
4. Go and have a coffee, it will take a few minutes to finish compiling!
5. After successful compilation, the library will be in the ‘bin’ folder. The bin folder is located in the ‘build’ folder which is created, by default, one directory higher than the OpenSeesQt source code. The name of the build folder should look something like this: build-OpenSeesPy-Desktop\_Qt\_5\_12\_1\_clang\_64bit-Debug
6. OS X only

OS X automatically prepends a ‘lib’ to the library file. Remove this ‘lib’ and rename the file to be ‘opensees.dylib’ Next, a symbolic link is required for a Python project to import the library. To create a symbolic

link, cd the directory containing the OpenSees library in terminal and run the following command to create a symbolic link:

```
ln -s opensees.dylib opensees.so
```

There should now be a .so (shared object) file in addition to the .dylib file. Finally, copy both the .dylib and the .so ‘link’ into your python environment folder to import it into your project. Directions for using OpenSeesPy can be found at the project website: <https://openseespydoc.readthedocs.io/en/latest/index.html>

## 1.2.5 Supported Build Environments:

### OSX

Build Environment:

- OSX 10.14.3 (Mojave)
- Qt 5.12.1
- Qt Creator 4.8.1

Compiler:

- Apple LLVM version 10.0.0 (clang-1000.11.45.5)
- Target: x86\_64-apple-darwin18.2.0
- Thread model: posix 64-BIT architecture

To find the version of clang on your computer, type the following in terminal:

```
clang --version
```

---

**Note:** This project comes with pre-built libraries for everything except Python. Therefore, you do not have to go through the trouble of building any libraries unless you are using a special build system or you want to experiment. The precompiled library files are located in the ‘OpenSeesLibs’ folder. In the event that you are feeling adventurous and you want to compile the libraries on your own, instructions are given below for each library, for each operating system. After successful compilation, note the installation directory. This directory contains the locations of the ‘include’ and ‘lib’ folders for that library. If replacing or adding new libraries, the file paths should be updated in the ‘OpenSeesLibs.pri’ file. This is required so that the compiler knows where to find the header files and to link the libraries to your project.

---

### OSX

On OSX, the dependencies are built/installed with Homebrew. Homebrew is a free and open-source software package management system that simplifies the installation of software on Apple’s macOS operating system and Linux. Homebrew maintains its own folder within /usr/local/ directory aptly named the ‘Cellar’:

```
/usr/local/Cellar/
```

Each dependency installed through Homebrew will have its own subfolder within the Cellar directory. Each subfolder contains that dependencies ‘include’ and ‘lib’ folders.

## 1.2.6 MUMPS

MULTifrontal Massively Parallel sparse direct Solver, or MUMPS, is a sparse direct solver used for parallel solving of a system of equations

Installing MUMPS via brew: Dominique Orban has written a Homebrew formula (<http://brew.sh>) for Mac OSX users. Homebrew MUMPS is now available via the OpenBLAS tap. Build instructions are as follows:

In terminal, copy and paste each command individually and execute:

```
brew tap dpo/openblas  
brew tap-pin dpo/openblas  
brew options mumps # to discover build options  
brew install mumps [options...]
```

The options can be left blank, i.e., with default options so the last line will look like:

```
brew install mumps
```

Mumps requires the following dependencies that will automatically be installed:

```
-Scalapack
```

## 1.2.7 OpenMPI

OpenMPI is a high performance message passing library (<https://www.open-mpi.org/>)

Installing OpenMpi via brew: In terminal, copy and paste the following command and execute:

```
brew install open-mpi
```

OpenMPI requires the following dependencies that will automatically be installed:

- GCC (GNU compiler collection)
- libevent (Asynchronous event library: <https://libevent.org/>)

## 1.2.8 UMFPACK

UMFPACK is a set of routines for solving unsymmetric sparse linear systems of the form Ax=b, using the Unsymmetric MultiFrontal method (Matrix A is not required to be symmetric). UMFPACK is part of suite-sparse library in homebrew/science

In terminal, copy and paste each command individually and execute:

```
brew tap homebrew/science  
brew install suite-sparse
```

UMFPACK requires the following dependencies that will automatically be installed:

- Metis ('METIS' is a type of GraphPartitioner and numberer - An Unstructured Graph Partitioning And Sparse Matrix Ordering System', developed by G. Karypis and V. Kumar at the University of Minnesota.

## 1.2.9 SUPERLU

SUPERLU is a general purpose library for the direct solution of large, sparse, nonsymmetric systems of linear equations. The library is written in C and is callable from either C or Fortran program. It uses MPI, OpenMP and CUDA to support various forms of parallelism.

Installing SUPERLU via brew In terminal, copy and paste the following command and execute:

```
brew install superlu
```

Should install by default with option --with-openmp enabled. Open MP is needed for parallel analysis.

SUPERLU requires the following dependencies that will automatically be installed:

- GCC (GNU compiler collection)
- openblas (In scientific computing, OpenBLAS is an open source implementation of the BLAS API with many hand-crafted optimizations for specific processor types)

## 1.2.10 SUPERLUMT

SUPERLU but for for shared memory parallel machines. Provides Pthreads and OpenMP interfaces.

Installing SUPERLUMT via brew: In terminal, copy and paste the following command and execute:

```
brew install superlu_mt
```

SUPERLUMT requires the following dependencies that will automatically be installed:

- openblas

## 1.2.11 SUPERLUDIST

SUPERLU but for for distributed memory parallel machines. Supports manycore heterogeneous node architecture: MPI is used for interprocess communication, OpenMP is used for on-node threading, CUDA is used for computing on GPUs.

Installing SUPERLUDIST via brew: In terminal, copy and paste the following command and execute:

```
brew install superlu_dist
```

SUPERLUDIST requires the following dependencies that will automatically be installed:

- GCC (GNU compiler collection)
- openblas (In scientific computing, OpenBLAS is an open source implementation of the BLAS API with many hand-crafted optimizations for specific processor types)
- OpenMPI (a high performance message passing library (<https://www.open-mpi.org/>))
- Parmetis (MPI library for graph/mesh partitioning and fill-reducing orderings)

## 1.2.12 LAPACK (SCALAPACK)

The Linear Algebra PACKage, or LAPACK, is written in Fortran 90 and provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) are also provided, as are related computations such as reordering of the Schur factorizations and estimating condition numbers. Dense and banded matrices are handled, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices, in both single and double precision.

LAPACK is given as a system library in OSX, you may have to update the locations of your system library in ‘OpenSeesLibs.pri’

## 1.2.13 BLAS

The BLAS (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations.

BLAS is given as a system library in OSX, you may have to update the locations of your system library in 'OpenSeesLibs.pri'

## 1.2.14 ARPACK

ARPACK contains routines to solve large scale eigenvalue problems

Installing ARPACK via brew: In terminal, copy and paste the following command and execute:

```
brew install arpack
```

ARPACK requires the following dependencies that will automatically be installed:

- GCC (GNU compiler collection)
- openblas (In scientific computing, OpenBLAS is an open source implementation of the BLAS API with many hand-crafted optimizations for specific processor types)

## 1.2.15 GCC

Many of the dependencies require fortran (there is still a lot of legacy fortran code floating around in the engineering world). On OSX, I found the best solution is to use the pre-bundled fortran capabilities in the GNU compiler collection or GCC. In addition to its fortran capabilities, GCC is a dependency for many other libraries.

Installing GCC via brew: In terminal, copy and paste the following command and execute:

```
brew install GCC
```

## 1.2.16 PYTHON

Python is an interpreted, high-level, general-purpose programming language. It is used in OpenSees as an interpreter in the OpenSeesPy version. In OpenSeesPy, Python version 3 is used.

Installing PYTHON via brew:

```
brew install python
```

## 1.2.17 MISC. NOTES

For the SUPERLU library. The file supermatrix.h throws an undefined error for the type `int_t`. It is actually defined in the file `slu_ddefs.h`, but for some reason the compiler is not linking the two. Add the following line, copied from `slu_ddefs.h` to `supermatrix.h` around line 17:

```
typedef int int_t; /* default */
```

## 1.3 Change Log

- **Version 0.5.4**

- Support Mac
- Support Python3.8 on Windows and Linux

- **Version 0.5.3**

- Fix bug in LimitState UniadialMaterial
- Automatic trimming spaces for string inputs
- Some output commands return lists instead of ints, such as nodeDisp etc.

- **Version 0.5.2**

- Add package openseespy.postprocessing
- Add setStartNodeTag command
- modalDamping: bug fixes
- Add Steel02Fatiuge material
- Add Concrete02IS material
- Add HardeningMaterial2 material
- Add hystereticBackone command
- Add stiffnessDegradation command
- Add strengthDegradation command
- Add unloadingRule command

- **Version 0.4.2019.7**

- Parallel: the Linux version is enabled with parallel capability
- Python stream: add no echo
- Mesh: add CorotTruss
- TriMesh: can create line elements
- QuadMesh: can create line and triangular elements
- Python inputs: more flexible input types
- Commands: add ExplicitDifference integrator

- **Version 0.3.0**

- Add logFile command
- Add partial uniform load fo ForceBeamColumn
- Add ShellDKGT element
- Add ‘-V’ option in Newmark and HHT
- Fix bugs in wipe and Mesh
- Various PFEM updates
- Update to OpenSees 3.0.3

- **Version 0.2.0** ([8a3d622](#))

- OpenSeesPy now can print messages and errors in Jupyter Notebook and other Windows based Python applications
- Add setParameter command
- Add nodeDOFs command
- Add setNumThread and getNumThread commands in a multi-threaded environment
- Add logFile command
- printA and prinbB can return matrix and vector as lists
- Fix bugs in updateMaterialStage
- PM4Sand improvements
- Add CatenaryCable element to OpenSeesPy

- **Version 0.1.1** ([f9f45fe](#))

- Update to OpenSees 3.0.2

- **Version 0.0.7** ([b75db21](#))

- Add “2D wheel-rail” element
- PVD recorder allows to set a path
- Add “sdfResponse” function for single dof dynamic analysis
- Fix a bug in Joint2D
- Fix typo in UCSD UP elements
- Fix bugs in PressureIndependMultiYield
- Add JSON print options to some materials and elements

- **Version 0.0.6** ([cead6e8](#))

- Add “nonlinearBeamColumn” element for backward compatibility
- Add “updateMaterialStage” function
- Add “RCCircular” sectiton
- Add “quadr” patch for backward compatibility
- Fix bugs in “Steel01Thermal” material
- Fix bugs in Truss
- Fix bugs in eleNodes function
- Fix bugs in ZeroLength element
- Fix bugs in FiberSection2d
- Fix bugs in PFEMLinSOE

- **Version 0.0.5** ([215c63d](#))

- Update to OpenSees 3.0.0

## 1.4 Model Commands

The model or domain in OpenSees is a collection (an aggregation in object-oriented terms) of elements, nodes, single- and multi-point constraints and load patterns. It is the aggregation of these components which define the type of model that is being analyzed.

1. *model command*
2. *element commands*
3. *node command*
4. *sp constraint commands*
5. *mp constraint commands*
6. *timeSeries commands*
7. *pattern commands*
8. *mass command*
9. *region command*
10. *rayleigh command*
11. *block commands*
12. *beamIntegration commands*
13. *uniaxialMaterial commands*
14. *nDMaterial commands*
15. *section commands*
16. *frictionModel commands*
17. *geomTransf commands*

### 1.4.1 model command

**model** ('basic', '-ndm', ndm, '-ndf', ndf=ndm\*(ndm+1)/2)

Set the default model dimensions and number of dofs.

ndm ( <a href="#">int</a> )	number of dimensions (1,2,3)
ndf ( <a href="#">int</a> )	number of dofs (optional)

### 1.4.2 element commands

**element** (eleType, eleTag, \*eleNodes, \*eleArgs)

Create a OpenSees element.

eleType ( <a href="#">str</a> )	element type
eleTag ( <a href="#">int</a> )	element tag.
eleNodes ( <a href="#">list</a> ( <a href="#">int</a> ))	a list of element nodes, must be preceded with *.
eleArgs ( <a href="#">list</a> )	a list of element arguments, must be preceded with *.

For example,

```

eleType = 'truss'
eleTag = 1
eleNodes = [iNode, jNode]
eleArgs = [A, matTag]
element(eleType, eleTag, *eleNodes, *eleArgs)

```

The following contain information about available `eleType`:

## Zero-Length Element

1. `zeroLength Element`
2. `zeroLengthND Element`
3. `zeroLengthSection Element`
4. `CoupledZeroLength Element`
5. `zeroLengthContact Element`
6. `zeroLengthContactNTS2D`
7. `zeroLengthInterface2D`
8. `zeroLengthImpact3D`

## `zeroLength Element`

```

element ('zeroLength', eleTag, *eleNodes, '-mat', *matTags, '-dir', *dirs[, '-doRayleigh', rFlag=0 ][, '-orient', *vecx, *vecyp ])

```

This command is used to construct a zeroLength element object, which is defined by two nodes at the same location. The nodes are connected by multiple UniaxialMaterial objects to represent the force-deformation relationship for the element.

<code>eleTag (int)</code>	unique element object tag
<code>eleNodes (list (int))</code>	a list of two element nodes
<code>matTags (list (int))</code>	a list of tags associated with previously-defined Uni-axialMaterials
<code>dirs (list (int))</code>	a list of material directions: <ul style="list-style-type: none"> <li>• 1,2,3 - translation along local x,y,z axes, respectively;</li> <li>• 4,5,6 - rotation about local x,y,z axes, respectively</li> </ul>
<code>vecx (list (float))</code>	a list of vector components in global coordinates defining local x-axis (optional)
<code>vecyp (list (float))</code>	a list of vector components in global coordinates defining vector yp which lies in the local x-y plane for the element. (optional)
<code>rFlag (float)</code>	optional, default = 0 <ul style="list-style-type: none"> <li>• <code>rFlag</code> = 0 NO RAYLEIGH DAMPING (default)</li> <li>• <code>rFlag</code> = 1 include rayleigh damping</li> </ul>

---

**Note:** If the optional orientation vectors are not specified, the local element axes coincide with the global axes. Otherwise the local z-axis is defined by the cross product between the vectors x and yp vectors specified on the command line.

---

**See also:**[Notes](#)**zeroLengthND Element****element** ('zeroLengthND', eleTag, \*eleNodes, matTag[, uniTag ][, '-orient', \*vecx, vecyp ])

This command is used to construct a zeroLengthND element object, which is defined by two nodes at the same location. The nodes are connected by a single NDMaterial object to represent the force-deformation relationship for the element.

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of two element nodes
matTag (int)	tag associated with previously-defined ndMaterial object
uniTag (int)	tag associated with previously-defined UniaxialMaterial object which may be used to represent uncoupled behavior orthogonal to the plane of the NDmaterial response. SEE NOTES 2 and 3.
vecx (list (float))	a list of vector components in global coordinates defining local x-axis (optional)
vecyp (list (float))	a list of vector components in global coordinates defining vector yp which lies in the local x-y plane for the element. (optional)

---

**Note:**

1. The zeroLengthND element only represents translational response between its nodes
  2. If the NDMaterial object is of order two, the response lies in the element local x-y plane and the UniaxialMaterial object may be used to represent the uncoupled behavior orthogonal to this plane, i.e. along the local z-axis.
  3. If the NDMaterial object is of order three, the response is along each of the element local axes.
  4. If the optional orientation vectors are not specified, the local element axes coincide with the global axes. Otherwise the local z-axis is defined by the cross product between the vectors x and yp vectors specified on the command line.
  5. The valid queries to a zero-length element when creating an ElementRecorder object are ‘force’, ‘deformation’, and ‘material matArg1 matArg2 ...’
- 

**See also:**[Notes](#)

## zeroLengthSection Element

```
element ('zeroLengthSection', eleTag, *eleNodes, sectTag[, '-orient', *vecx, *vecyp ][, '-doRayleigh', rFlag ])
```

This command is used to construct a zero length element object, which is defined by two nodes at the same location. The nodes are connected by a single section object to represent the force-deformation relationship for the element.

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of two element nodes
sectTag (int)	tag associated with previously-defined Section object
vecx (list (float))	a list of vector components in global coordinates defining local x-axis (optional)
vecyp (list (float))	a list of vector components in global coordinates defining vector yp which lies in the local x-y plane for the element. (optional)
rFlag (float)	optional, default = 0 <ul style="list-style-type: none"> <li>• rFlag = 0 NO RAYLEIGH DAMPING (default)</li> <li>• rFlag = 1 include rayleigh damping</li> </ul>

### See also:

Notes

## CoupledZeroLength Element

```
element ('CoupledZeroLength', eleTag, *eleNodes, dirn1, dirn2, matTag[, rFlag=1 ])
```

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of two element nodes
matTag (float)	tags associated with previously-defined UniaxialMaterial
dir1 dir2 (int)	the two directions, 1 through ndof.
rFlag (float)	optional, default = 0 <ul style="list-style-type: none"> <li>• rFlag = 0 NO RAYLEIGH DAMPING (default)</li> <li>• rFlag = 1 include rayleigh damping</li> </ul>

### See also:

Notes

## zeroLengthContact Element

```
element ('zeroLengthContact2D', eleTag, *eleNodes, Kn, Kt, mu, '-normal', Nx, Ny)
```

This command is used to construct a zeroLengthContact2D element, which is Node-to-node frictional contact

element used in two dimensional analysis and three dimensional analysis:

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of a slave and a master nodes
Kn (float)	Penalty in normal direction
Kt (float)	Penalty in tangential direction
mu (float)	friction coefficient

**element ('zeroLengthContact3D', eleTag, \*eleNodes, Kn, Kt, mu, c, dir)**

This command is used to construct a zeroLengthContact3D element, which is Node-to-node frictional contact element used in two dimensional analysis and three dimensional analysis:

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of a slave and a master nodes
Kn (float)	Penalty in normal direction
Kt (float)	Penalty in tangential direction
mu (float)	friction coefficient
c (float)	cohesion (not available in 2D)
dir (int)	Direction flag of the contact plane (3D), it can be: <ul style="list-style-type: none"><li>• 1 Out normal of the master plane pointing to +X direction</li><li>• 2 Out normal of the master plane pointing to +Y direction</li><li>• 3 Out normal of the master plane pointing to +Z direction</li></ul>

**See also:**

Notes

**zeroLengthContactNTS2D****element ('zeroLengthContactNTS2D', eleTag, '-sNdNum', sNdNum, '-mNdNum', mNdNum, '-Nodes', \*Nodes, Kn, kt, phi)**

eleTag (int)	unique element object tag
sNdNum (int)	Number of Slave Nodes
mNdNum (int)	Number of Master nodes
Nodes (list (int))	Slave and master node tags respectively
Kn (float)	Penalty in normal direction
Kt (float)	Penalty in tangential direction
phi (float)	Friction angle in degrees

---

**Note:**

1. The contact element is node-to-segment (NTS) contact. The relation follows Mohr-Coulomb frictional law:  $T = N \times \tan(\phi)$ , where  $T$  is the tangential force,  $N$  is normal force across the interface and  $\phi$  is friction angle.
2. For 2D contact, slave nodes and master nodes must be 2 DOF and notice that the slave and master nodes must be entered in counterclockwise order.

- 
3. The resulting tangent from the contact element is non-symmetric. Switch to the non-symmetric matrix solver if convergence problem is experienced.
  4. As opposed to node-to-node contact, predefined normal vector for node-to-segment (NTS) element is not required because contact normal will be calculated automatically at each step.
  5. contact element is implemented to handle large deformations.
- 

**See also:**[Notes](#)**zeroLengthInterface2D**

```
element ('zeroLengthInterface2D', eleTag, '-sNdNum', sNdNum, '-mNdNum', mNdNum, '-dof', sdof, mdof,
'-Nodes', *Nodes, Kn, kt, phi)
```

eleTag ( <a href="#">int</a> )	unique element object tag
sNdNum ( <a href="#">int</a> )	Number of Slave Nodes
mNdNum ( <a href="#">int</a> )	Number of Master nodes
sdof, mdof ( <a href="#">int</a> )	Slave and Master degree of freedom
Nodes ( <a href="#">list (int)</a> )	Slave and master node tags respectively
Kn ( <a href="#">float</a> )	Penalty in normal direction
Kt ( <a href="#">float</a> )	Penalty in tangential direction
phi ( <a href="#">float</a> )	Friction angle in degrees

**Note:**

1. The contact element is node-to-segment (NTS) contact. The relation follows Mohr-Coulomb frictional law:  $T = N \times \tan(\phi)$ , where  $T$  is the tangential force,  $N$  is normal force across the interface and  $\phi$  is friction angle.
  2. For 2D contact, slave nodes and master nodes must be 2 DOF and notice that the slave and master nodes must be entered in counterclockwise order.
  3. The resulting tangent from the contact element is non-symmetric. Switch to the non-symmetric matrix solver if convergence problem is experienced.
  4. As opposed to node-to-node contact, predefined normal vector for node-to-segment (NTS) element is not required because contact normal will be calculated automatically at each step.
  5. contact element is implemented to handle large deformations.
- 

**See also:**[Notes](#)**zeroLengthImpact3D**

```
element ('zeroLengthImpact3D', eleTag, *eleNodes, direction, initGap, frictionRatio, Kt, Kn, Kn2, Delta_y,
cohesion)
```

This command constructs a node-to-node zero-length contact element in 3D space to simulate the impact/pounding and friction phenomena.

eleTag ( <a href="#">int</a> )	unique element object tag
eleNodes ( <a href="#">list (int)</a> )	a list of a slave and a master nodes
direction ( <a href="#">int</a> )	<ul style="list-style-type: none"><li>• 1 if out-normal vector of master plane points to +X direction</li><li>• 2 if out-normal vector of master plane points to +Y direction</li><li>• 3 if out-normal vector of master plane points to +Z direction</li></ul>
initGap ( <a href="#">float</a> )	Initial gap between master plane and slave plane
frictionRatio ( <a href="#">float</a> )	Friction ratio in two tangential directions (parallel to master and slave planes)
Kt ( <a href="#">float</a> )	Penalty in two tangential directions
Kn ( <a href="#">float</a> )	Penalty in normal direction (normal to master and slave planes)
Kn2 ( <a href="#">float</a> )	Penalty in normal direction after yielding based on Hertz impact model
Delta_y ( <a href="#">float</a> )	Yield deformation based on Hertz impact model
cohesion ( <a href="#">float</a> )	Cohesion, if no cohesion, it is zero

---

**Note:**

1. This element has been developed on top of the “zeroLengthContact3D”. All the notes available in “zeroLengthContact3D” wiki page would apply to this element as well. It includes the definition of master and slave nodes, the number of degrees of freedom in the domain, etc.
  2. Regarding the number of degrees of freedom (DOF), the end nodes of this element should be defined in 3DOF domain. For getting information on how to use 3DOF and 6DOF domain together, please refer to OpenSees documentation and forums or see the zip file provided in the EXAMPLES section below.
  3. This element adds the capabilities of “ImpactMaterial” to “zeroLengthContact3D.”
  4. For simulating a surface-to-surface contact, the element can be defined for connecting the nodes on slave surface to the nodes on master surface.
  5. The element was found to be fast-converging and eliminating the need for extra elements and nodes in the modeling process.
- 

**See also:**[Notes](#)**Truss Elements**

1. [Truss Element](#)
2. [Corotational Truss Element](#)

**Truss Element**

This command is used to construct a truss element object. There are two ways to construct a truss element object:

---

**element** ('Truss', eleTag, \*eleNodes, A, matTag[, '-rho', rho ][, '-cMass', cFlag ][, '-doRayleigh', rFlag ])

One way is to specify an area and a UniaxialMaterial identifier:

**element** ('TrussSection', eleTag, \*eleNodes, A, secTag[, '-rho', rho ][, '-cMass', cFlag ][, '-doRayleigh', rFlag ])

the other is to specify a Section identifier:

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of two element nodes
A (float)	cross-sectional area of element
matTag (int)	tag associated with previously-defined UniaxialMaterial
secTag (int)	tag associated with previously-defined Section
rho (float)	mass per unit length, optional, default = 0.0
cFlag (float)	consistent mass flag, optional, default = 0 <ul style="list-style-type: none"> <li>• cFlag = 0 lumped mass matrix (default)</li> <li>• cFlag = 1 consistent mass matrix</li> </ul>
rFlag (float)	Rayleigh damping flag, optional, default = 0 <ul style="list-style-type: none"> <li>• rFlag = 0 NO RAYLEIGH DAMPING (default)</li> <li>• rFlag = 1 include Rayleigh damping</li> </ul>

---

#### Note:

1. The truss element DOES NOT include geometric nonlinearities, even when used with beam-columns utilizing P-Delta or Corotational transformations.
2. When constructed with a UniaxialMaterial object, the truss element considers strain-rate effects, and is thus suitable for use as a damping element.
3. The valid queries to a truss element when creating an ElementRecorder object are 'axialForce,' 'forces,' 'localForce', 'deformations,' 'material matArg1 matArg2...', 'section sectArg1 sectArg2...' There will be more queries after the interface for the methods involved have been developed further.

---

#### See also:

Notes

### Corotational Truss Element

This command is used to construct a corotational truss element object. There are two ways to construct a corotational truss element object:

**element** ('corotTruss', eleTag, \*eleNodes, A, matTag[, '-rho', rho ][, '-cMass', cFlag ][, '-doRayleigh', rFlag ])

One way is to specify an area and a UniaxialMaterial identifier:

**element** ('corotTrussSection', eleTag, \*eleNodes, secTag[, '-rho', rho ][, '-cMass', cFlag ][, '-doRayleigh', rFlag ])

the other is to specify a Section identifier:

eleTag ( <a href="#">int</a> )	unique element object tag
eleNodes ( <a href="#">list (int)</a> )	a list of two element nodes
A ( <a href="#">float</a> )	cross-sectional area of element
matTag ( <a href="#">int</a> )	tag associated with previously-defined UniaxialMaterial
sectTag ( <a href="#">int</a> )	tag associated with previously-defined Section
rho ( <a href="#">float</a> )	mass per unit length, optional, default = 0.0
cFlag ( <a href="#">float</a> )	consistent mass flag, optional, default = 0 <ul style="list-style-type: none"><li>• cFlag = 0 lumped mass matrix (default)</li><li>• cFlag = 1 consistent mass matrix</li></ul>
rFlag ( <a href="#">float</a> )	Rayleigh damping flag, optional, default = 0 <ul style="list-style-type: none"><li>• rFlag = 0 NO RAYLEIGH DAMPING (default)</li><li>• rFlag = 1 include Rayleigh damping</li></ul>

---

**Note:**

1. When constructed with a UniaxialMaterial object, the corotational truss element considers strain-rate effects, and is thus suitable for use as a damping element.
  2. The valid queries to a truss element when creating an ElementRecorder object are ‘axialForce,’ ‘stiff,’ deformations,’ ‘material matArg1 matArg2...,’ ‘section sectArg1 sectArg2...’ There will be more queries after the interface for the methods involved have been developed further.
  3. CorotTruss DOES NOT include Rayleigh damping by default.
- 

**See also:**

[Notes](#)

## Beam-Column Elements

1. [\*Elastic Beam Column Element\*](#)
2. [\*Elastic Beam Column Element with Stiffness Modifiers\*](#)
3. [\*Elastic Timoshenko Beam Column Element\*](#)
4. [\*Beam With Hinges Element\*](#)
5. [\*dispBeamColumn\*](#)
6. [\*forceBeamColumn\*](#)
7. [\*nonlinearBeamColumn\*](#)
8. [\*Flexure-Shear Interaction Displacement-Based Beam-Column Element\*](#)
9. [\*MVLEM - Multiple-Vertical-Line-Element-Model for RC Walls\*](#)
10. [\*SFI MVLEM - Cyclic Shear-Flexure Interaction Model for RC Walls\*](#)

## Elastic Beam Column Element

This command is used to construct an elasticBeamColumn element object. The arguments for the construction of an elastic beam-column element depend on the dimension of the problem, ndm:

```
element ('elasticBeamColumn', eleTag, *eleNodes, A, E, Iz, transfTag[, '-mass', massDens ][, '-cMass'])
```

For a two-dimensional problem

```
element ('elasticBeamColumn', eleTag, *eleNodes, A, E, G, J, Iy, Iz, transfTag[, '-mass', massDens ][, '-cMass'])
```

For a three-dimensional problem

<code>eleTag (int)</code>	unique element object tag
<code>eleNodes (list (int))</code>	a list of two element nodes
<code>A (float)</code>	cross-sectional area of element
<code>E (float)</code>	Young's Modulus
<code>G (float)</code>	Shear Modulus
<code>J (float)</code>	torsional moment of inertia of cross section
<code>Iz (float)</code>	second moment of area about the local z-axis
<code>Iy (float)</code>	second moment of area about the local y-axis
<code>transfTag (int)</code>	identifier for previously-defined coordinate-transformation (CrdTransf) object
<code>massDens (float)</code>	element mass per unit length (optional, default = 0.0)
<code>'-cMass' (str)</code>	to form consistent mass matrix (optional, default = lumped mass matrix)

### See also:

[Notes](#)

## Elastic Beam Column Element with Stiffness Modifiers

This command is used to construct a ModElasticBeam2d element object. The arguments for the construction of an elastic beam-column element with stiffness modifiers is applicable for 2-D problems. This element should be used for modelling of a structural element with an equivalent combination of one elastic element with stiffness-proportional damping, and two springs at its two ends with no stiffness proportional damping to represent a prismatic section. The modelling technique is based on a number of analytical studies discussed in Zareian and Medina (2010) and Zareian and Krawinkler (2009) and is utilized in order to solve problems related to numerical damping in dynamic analysis of frame structures with concentrated plasticity springs.

```
element ('ModElasticBeam2d', eleTag, *eleNodes, A, E, Iz, K11, K33, K44, transfTag[, '-mass', massDens ][, '-cMass'])
```

<code>eleTag (int)</code>	unique element object tag
<code>eleNodes (list (int))</code>	a list of two element nodes
<code>A (float)</code>	cross-sectional area of element
<code>E (float)</code>	Young's Modulus
<code>Iz (float)</code>	second moment of area about the local z-axis
<code>K11 (float)</code>	stiffness modifier for translation
<code>K33 (float)</code>	stiffness modifier for translation
<code>K44 (float)</code>	stiffness modifier for rotation
<code>transfTag (int)</code>	identifier for previously-defined coordinate-transformation (CrdTransf) object
<code>massDens (float)</code>	element mass per unit length (optional, default = 0.0)
<code>'-cMass' (str)</code>	to form consistent mass matrix (optional, default = lumped mass matrix)

**See also:**[Notes](#)**Elastic Timoshenko Beam Column Element**

This command is used to construct an `ElasticTimoshenkoBeam` element object. A Timoshenko beam is a frame member that accounts for shear deformations. The arguments for the construction of an elastic Timoshenko beam element depend on the dimension of the problem, ndm:

```
element ('ElasticTimoshenkoBeam', eleTag, *eleNodes, E, G, A, Iz, Avy, transfTag[, '-mass', massDens ][, '-cMass'])
```

For a two-dimensional problem:

```
element ('ElasticTimoshenkoBeam', eleTag, *eleNodes, E, G, A, Iz, Jx, Iy, Iz, Avy, Avz, transfTag[, '-mass', massDens ][, '-cMass'])
```

For a three-dimensional problem:

<code>eleTag</code> ( <code>int</code> )	unique element object tag
<code>eleNodes</code> ( <code>list</code> ( <code>int</code> ))	a list of two element nodes
<code>E</code> ( <code>float</code> )	Young's Modulus
<code>G</code> ( <code>float</code> )	Shear Modulus
<code>A</code> ( <code>float</code> )	cross-sectional area of element
<code>Jx</code> ( <code>float</code> )	torsional moment of inertia of cross section
<code>Iy</code> ( <code>float</code> )	second moment of area about the local y-axis
<code>Iz</code> ( <code>float</code> )	second moment of area about the local z-axis
<code>Avy</code> ( <code>float</code> )	Shear area for the local y-axis
<code>Avz</code> ( <code>float</code> )	Shear area for the local z-axis
<code>transfTag</code> ( <code>int</code> )	identifier for previously-defined coordinate-transformation (CrdTransf) object
<code>massDens</code> ( <code>float</code> )	element mass per unit length (optional, default = 0.0)
'-cMass' ( <code>str</code> )	to form consistent mass matrix (optional, default = lumped mass matrix)

**See also:**[Notes](#)**Beam With Hinges Element**

This command is used to construct a `forceBeamColumn` element object, which is based on the non-iterative (or iterative) flexibility formulation. The locations and weights of the element integration points are based on so-called plastic hinge integration, which allows the user to specify plastic hinge lengths at the element ends. Two-point Gauss integration is used on the element interior while two-point Gauss-Radau integration is applied over lengths of  $4L_{pI}$  and  $4L_{pJ}$  at the element ends, viz. "modified Gauss-Radau plastic hinge integration". A total of six integration points are used in the element state determination (two for each hinge and two for the interior).

Users may be familiar with the `beamWithHinges` command format (see below); however, the format shown here allows for the simple but important case of using a material nonlinear section model on the element interior. The previous `beamWithHinges` command constrained the user to an elastic interior, which often led to unconservative estimates of the element resisting force when plasticity spread beyond the plastic hinge regions into the element interior.

The advantages of this new format over the previous `beamWithHinges` command are

- Plasticity can spread beyond the plastic hinge regions
- Hinges can form on the element interior, e.g., due to distributed member loads

To create a beam element with hinges, one has to use a `forceBeamColumn` element with following `beamIntegration()`.

**Note:**

- 'HingeRadau' – two-point Gauss-Radau applied to the hinge regions over 4LpI and 4LpJ (six element integration points)
- 'HingeRadauTwo' – two-point Gauss-Radau in the hinge regions applied over LpI and LpJ (six element integration points)
- 'HingeMidpoint' – midpoint integration over the hinge regions (four element integration points)
- 'HingeEndpoint' – endpoint integration over the hinge regions (four element integration points)

**See also:**

For more information on the behavior, advantages, and disadvantages of these approaches to plastic hinge integration, see

Scott, M.H. and G.L. Fenves. "Plastic Hinge Integration Methods for Force-Based Beam-Column Elements", Journal of Structural Engineering, 132(2):244-252, February 2006.

Scott, M.H. and K.L. Ryan. "Moment-Rotation Behavior of Force-Based Plastic Hinge Elements", Earthquake Spectra, 29(2):597-607, May 2013.

The primary advantages of HingeRadau are

- The user can specify a physically meaningful plastic hinge length
- The largest bending moment is captured at the element ends
- The exact numerical solution is recovered for a linear-elastic prismatic beam
- The characteristic length is equal to the user-specified plastic hinge length when deformations localize at the element ends

while the primary disadvantages are

- The element post-yield response is too flexible for strain-hardening section response (consider using HingeRadauTwo)
- The user needs to know the plastic hinge length a priori (empirical equations are available)

## dispBeamColumn

**element ('dispBeamColumn', eleTag, iNode, jNode, transfTag, integrationTag, '-cMass', '-mass', mass=0.0)**

Create a ForceBeamColumn element.

<code>eleTag (int)</code>	tag of the element
<code>iNode (int)</code>	tag of node i
<code>jNode (int)</code>	tag of node j
<code>transfTag (int)</code>	tag of transformation
<code>integrationTag (int)</code>	tag of <code>beamIntegration()</code>
<code>'-cMass'</code>	to form consistent mass matrix (optional, default = lumped mass matrix)
<code>mass (float)</code>	element mass density (per unit length), from which a lumped-mass matrix is formed (optional)

## forceBeamColumn

```
element ('forceBeamColumn', eleTag, iNode, jNode, transfTag, integrationTag, '-iter', maxIter=10, tol=1e-12, '-mass', mass=0.0)
```

Create a ForceBeamColumn element.

eleTag (int)	tag of the element
iNode (int)	tag of node i
jNode (int)	tag of node j
transfTag (int)	tag of transformation
integrationTag (int)	tag of <a href="#">beamIntegration()</a>
maxIter (float)	maximum number of iterations to undertake to satisfy element compatibility (optional)
tol (float)	tolerance for satisfaction of element compatibility (optional)
mass (float)	element mass density (per unit length), from which a lumped-mass matrix is formed (optional)

## nonlinearBeamColumn

```
element ('nonlinearBeamColumn', eleTag, iNode, jNode, numIntgrPts, sectTag, transfTag, '-iter', maxIter=10, tol=1e-12, '-mass', mass=0.0, '-integration', intType)
```

Create a nonlinearBeamColumn element. This element is for backward compatibility.

eleTag (int)	tag of the element
iNode (int)	tag of node i
jNode (int)	tag of node j
numIntgrPts (int)	number of integration points.
sectTag (int)	tag of section
transfTag (int)	tag of transformation
maxIter (float)	maximum number of iterations to undertake to satisfy element compatibility (optional)
tol (float)	tolerance for satisfaction of element compatibility (optional)
mass (float)	element mass density (per unit length), from which a lumped-mass matrix is formed (optional)
intType (str)	integration type (optional, default is 'Lobatto') <ul style="list-style-type: none"><li>• 'Lobatto'</li><li>• 'Legendre'</li><li>• 'Radau'</li><li>• 'NewtonCotes'</li><li>• 'Trapezoidal'</li></ul>

## Flexure-Shear Interaction Displacement-Based Beam-Column Element

This command is used to construct a dispBeamColumnInt element object, which is a distributed-plasticity, displacement-based beam-column element which includes interaction between flexural and shear components.

```
element ('dispBeamColumnInt', eleTag, *eleNodes, numIntgrPts, sectTag, transfTag, cRot['-mass', massDens])
```

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of two element nodes
numIntgrPt (int)	number of integration points along the element.
secTag (int)	identifier for previously-defined section object
transfTag (int)	identifier for previously-defined coordinate-transformation (CrdTransf) object
cRot (float)	identifier for element center of rotation (or center of curvature distribution). Fraction of the height distance from bottom to the center of rotation (0 to 1)
massDens (float)	element mass density (per unit length), from which a lumped-mass matrix is formed (optional, default=0.0)

**See also:**

Notes

**MVLEM - Multiple-Vertical-Line-Element-Model for RC Walls**

The MVLEM element command is used to generate a two-dimensional Multiple-Vertical-Line-Element-Model (MVLEM; Vulcano et al., 1988; Orakcal et al., 2004, Kolozvari et al., 2015) for simulation of flexure-dominated RC wall behavior. A single model element incorporates six global degrees of freedom, three of each located at the center of rigid top and bottom beams, as illustrated in Figure 1a. The axial/flexural response of the MVLEM is simulated by a series of uniaxial elements (or macro-fibers) connected to the rigid beams at the top and bottom (e.g., floor) levels, whereas the shear response is described by a shear spring located at height ch from the bottom of the wall element (Figure 1a). Shear and flexural responses of the model element are uncoupled. The relative rotation between top and bottom faces of the wall element occurs about the point located on the central axis of the element at height ch (Figure 1b). Rotations and resulting transverse displacements are calculated based on the wall curvature, derived from section and material properties, corresponding to the bending moment at height ch of each element (Figure 1b). A value of c=0.4 was recommended by Vulcano et al. (1988) based on comparison of the model response with experimental results.

```
element ('MVLEM', eleTag, Dens, *eleNodes, m, c, '-thick', *Thicknesses, '-width', *Widths, '-rho', *Reinforcing_ratios, '-matConcrete', *Concrete_tags, '-matSteel', *Steel_tags, '-matShear', Shear_tag)
```

eleTag (int)	unique element object tag
Dens (float)	Wall density
eleNodes (list (int))	a list of two element nodes
m (int)	Number of element macro-fibers
c (float)	Location of center of rotation from the iNode, c = 0.4 (recommended)
Thicknesses (list (float))	a list of m macro-fiber thicknesses
Widths (list (float))	a list of m macro-fiber widths
Reinforcing_ratios (list (float))	a list of m reinforcing ratios corresponding to macro-fibers; for each fiber: $\rho_{oi} = A_{s,i}/A_{gross,i}$ ( $1 < i < m$ )
Concrete_tags (list (int))	a list of m uniaxialMaterial tags for concrete
Steel_tags (list (int))	a list of m uniaxialMaterial tags for steel
Shear_tag (int)	Tag of uniaxialMaterial for shear material

**See also:**[Notes](#)**SFI MVLEM - Cyclic Shear-Flexure Interaction Model for RC Walls**

The SFI\_MVLEM command is used to construct a Shear-Flexure Interaction Multiple-Vertical-Line-Element Model (SFI-MVLEM, Kolozvari et al., 2015a, b, c), which captures interaction between axial/flexural and shear behavior of RC structural walls and columns under cyclic loading. The SFI\_MVLEM element (Figure 1) incorporates 2-D RC panel behavior described by the Fixed-Strut-Angle-Model (nDMaterial FSAM; Ulugtekin, 2010; Orakcal et al., 2012), into a 2-D macroscopic fiber-based model (MVLEM). The interaction between axial and shear behavior is captured at each RC panel (macro-fiber) level, which further incorporates interaction between shear and flexural behavior at the SFI\_MVLEM element level.

**element** ( ", eleTag, \*eleNodes, m, c, '-thick', \*Thicknesses, '-width', \*Widths, '-mat', \*Material\_tags)

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of two element nodes
m (int)	Number of element macro-fibers
c (float)	Location of center of rotation with from the iNode, $c = 0.4$ (recommended)
Thicknesses (list (float))	a list of m macro-fiber thicknesses
Widths (list (float))	a list of m macro-fiber widths
Material_tags (list (int))	a list of m macro-fiber nDMaterial1 tags

**See also:**[Notes](#)**Joint Elements**

1. [BeamColumnJoint Element](#)
2. [ElasticTubularJoint Element](#)
3. [Joint2D Element](#)

**BeamColumnJoint Element**

This command is used to construct a two-dimensional beam-column-joint element object. The element may be used with both two-dimensional and three-dimensional structures; however, load is transferred only in the plane of the element.

**element** ('beamColumnJoint', eleTag, \*eleNodes, Mat1, Mat2, Mat3, Mat4, Mat5, Mat6, Mat7, Mat8, Mat9, Mat10, Mat11, Mat12, Mat13[, eleHeightFac=1.0, eleWidthFac=1.0])

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of four element nodes
Mat1 (int)	uniaxial material tag for left bar-slip spring at node 1
Mat2 (int)	uniaxial material tag for right bar-slip spring at node 1
Mat3 (int)	uniaxial material tag for interface-shear spring at node 1
Mat4 (int)	uniaxial material tag for lower bar-slip spring at node 2
Mat5 (int)	uniaxial material tag for upper bar-slip spring at node 2
Mat6 (int)	uniaxial material tag for interface-shear spring at node 2
Mat7 (int)	uniaxial material tag for left bar-slip spring at node 3
Mat8 (int)	uniaxial material tag for right bar-slip spring at node 3
Mat9 (int)	uniaxial material tag for interface-shear spring at node 3
Mat10 (int)	uniaxial material tag for lower bar-slip spring at node 4
Mat11 (int)	uniaxial material tag for upper bar-slip spring at node 4
Mat12 (int)	uniaxial material tag for interface-shear spring at node 4
Mat13 (int)	uniaxial material tag for shear-panel
eleHeightF (float)	floating point value (as a ratio to the total height of the element) to be considered for determination of the distance in between the tension-compression couples (optional, default: 1.0)
eleWidthF (float)	floating point value (as a ratio to the total width of the element) to be considered for determination of the distance in between the tension-compression couples (optional, default: 1.0)

**See also:**[Notes](#)**ElasticTubularJoint Element**

This command is used to construct an ElasticTubularJoint element object, which models joint flexibility of tubular joints in two dimensional analysis of any structure having tubular joints.

```
element ('ElasticTubularJoint', eleTag, *eleNodes, Brace_Diameter, Brace_Angle, E, Chord_Diameter,
Chord_Thickness, Chord_Angle)
```

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of two element nodes
Brace_Diameter (float)	outer diameter of brace
Brace_Angle (float)	angle between brace and chord axis $0 < \text{Brace\_Angle} < 90$
E (float)	Young's Modulus
Chord_Diameter (float)	outer diameter of chord
Chord_Thickness (float)	thickness of chord
Chord_Angle (float)	angle between chord axis and global x-axis $0 < \text{Chord\_Angle} < 180$

**See also:**[Notes](#)

## Joint2D Element

This command is used to construct a two-dimensional beam-column-joint element object. The two dimensional beam-column joint is idealized as a parallelogram shaped shear panel with adjacent elements connected to its mid-points. The midpoints of the parallelogram are referred to as external nodes. These nodes are the only analysis components that connect the joint element to the surrounding structure.

```
element ('Joint2D', eleTag, *eleNodes[, Mat1, Mat2, Mat3, Mat4], MatC, LrgDspTag[, '-damage', DmgTag
][], '-damage', Dmg1 Dmg2 Dmg3 Dmg4 DmgC ])
```

eleTag (int)	unique element object tag
eleNode (list (int))	a list of five element nodes = [nd1, nd2, nd3, nd4, ndC]. ndC is the central node of beam-column joint. (the tag ndC is used to generate the internal node, thus, the node should not exist in the domain or be used by any other node)
Mat 1 (int)	uniaxial material tag for interface rotational spring at node 1. Use a zero tag to indicate the case that a beam-column element is rigidly framed to the joint. (optional)
Mat 2 (int)	uniaxial material tag for interface rotational spring at node 2. Use a zero tag to indicate the case that a beam-column element is rigidly framed to the joint. (optional)
Mat 3 (int)	uniaxial material tag for interface rotational spring at node 3. Use a zero tag to indicate the case that a beam-column element is rigidly framed to the joint. (optional)
Mat 4 (int)	uniaxial material tag for interface rotational spring at node 4. Use a zero tag to indicate the case that a beam-column element is rigidly framed to the joint. (optional)
Mat C (int)	uniaxial material tag for rotational spring of the central node that describes shear panel behavior
LrgDspT (int)	an integer indicating the flag for considering large deformations: * 0 - for small deformations and constant geometry * 1 - for large deformations and time varying geometry * 2 - for large deformations ,time varying geometry and length correction
DmgTag (int)	damage model tag
Dmg1 (int)	damage model tag for Mat1
Dmg2 (int)	damage model tag for Mat2
Dmg3 (int)	damage model tag for Mat3
Dmg4 (int)	damage model tag for Mat4
DmgC (int)	panel damage model tag

### See also:

Notes

## Link Elements

1. *Two Node Link Element*

## Two Node Link Element

This command is used to construct a twoNodeLink element object, which is defined by two nodes. The element can have zero or non-zero length. This element can have 1 to 6 degrees of freedom, where only the transverse and rotational degrees of freedom are coupled as long as the element has non-zero length. In addition, if the element length is larger than zero, the user can optionally specify how the P-Delta moments around the local x- and y-axis are distributed among a moment at node i, a moment at node j, and a shear couple. The sum of these three ratios is always equal to 1. In addition the shear center can be specified as a fraction of the element length from the iNode. The element does not contribute to the Rayleigh damping by default. If the element has non-zero length, the local x-axis is determined from the nodal geometry unless the optional x-axis vector is specified in which case the nodal geometry is ignored and the user-defined orientation is utilized. It is important to recognize that if this element has zero length, it does not consider the geometry as given by the nodal coordinates, but utilizes the user-defined orientation vectors to determine the directions of the springs.

```
element ('twoNodeLink', eleTag, *eleNodes, '-mat', *matTags, '-dir', *dirs[, '-orient', *vecx, *vecy][, '-pDelta', *Mratio][, '-shearDist', *sDratios][, '-doRayleigh'][, '-mass', m])
```

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of two element nodes
matTags (list (int))	a list of tags associated with previously-defined Uni-axialMaterial objects
dirs (list (int))	a list material directions: <ul style="list-style-type: none"> <li>• 2D-case: 1 , 2 - translations along local x,y axes; 3 - rotation about local z axis</li> <li>• 3D-case: 1, 2, 3 - translations along local x,y,z axes; 4, 5, 6 - rotations about local x,y,z axes</li> </ul>
vecx (list (float))	vector components in global coordinates defining local x-axis (optional)
vecy (list (float))	vector components in global coordinates defining local y-axis (optional)
Mratios (list (float))	P-Delta moment contribution ratios, size of ratio vector is 2 for 2D-case and 4 for 3D-case (entries: [My_iNode, My_jNode, Mz_iNode, Mz_jNode]) My_iNode + My_jNode <= 1.0, Mz_iNode + Mz_jNode <= 1.0. Remaining P-Delta moments are resisted by shear couples. (optional)
sDratios (list (float))	shear distances from iNode as a fraction of the element length, size of ratio vector is 1 for 2D-case and 2 for 3D-case. (entries: [dy_iNode, dz_iNode]) (optional, default = [0.5, 0.5])
'-doRayleigh' (str)	to include Rayleigh damping from the element (optional, default = no Rayleigh damping contribution)
m (float)	element mass (optional, default = 0.0)

### See also:

Notes

## Bearing Elements

1. *Elastomeric Bearing (Plasticity) Element*

2. *Elastomeric Bearing (Bouc-Wen) Element*
3. *Flat Slider Bearing Element*
4. *Single Friction Pendulum Bearing Element*
5. *Triple Friction Pendulum Bearing Element*
6. *Triple Friction Pendulum Element*
7. *MultipleShearSpring Element*
8. *KikuchiBearing Element*
9. *YamamotoBiaxialHDR Element*
10. *ElastomericX*
11. *LeadRubberX*
12. *HDR*
13. *RJ-Watson EQS Bearing Element*
14. *FPBearingPTV*

## Elastomeric Bearing (Plasticity) Element

This command is used to construct an elastomericBearing element object, which is defined by two nodes. The element can have zero length or the appropriate bearing height. The bearing has unidirectional (2D) or coupled (3D) plasticity properties for the shear deformations, and force-deformation behaviors defined by UniaxialMaterials in the remaining two (2D) or four (3D) directions. By default ( $sDratio = 0.5$ ) P-Delta moments are equally distributed to the two end-nodes. To avoid the introduction of artificial viscous damping in the isolation system (sometimes referred to as “damping leakage in the isolation system”), the bearing element does not contribute to the Rayleigh damping by default. If the element has non-zero length, the local x-axis is determined from the nodal geometry unless the optional x-axis vector is specified in which case the nodal geometry is ignored and the user-defined orientation is utilized.

```
element ('elastomericBearingPlasticity', eleTag, *eleNodes, kInit, qd, alpha1, alpha2, mu, '-P', matTag, '-Mz', matTag[, '-orient', x1, x2, x3, y1, y2, y3][, '-shearDist', sDratio][, '-doRayleigh'][, '-mass', m])
```

For a two-dimensional problem

```
element ('elastomericBearingPlasticity', eleTag, *eleNodes, kInit, qd, alpha1, alpha2, mu, '-P', matTag, '-T', matTag, '-My', matTag, '-Mz', matTag[, '-orient'[, x1, x2, x3], y1, y2, y3][, '-shearDist', sDratio][, '-doRayleigh'][, '-mass', m])
```

For a three-dimensional problem

<code>eleTag (int)</code>	unique element object tag
<code>eleNodes (list (int))</code>	a list of two element nodes
<code>kInit (float)</code>	initial elastic stiffness in local shear direction
<code>qd (float)</code>	characteristic strength
<code>alpha1 (float)</code>	post yield stiffness ratio of linear hardening component
<code>alpha2 (float)</code>	post yield stiffness ratio of non-linear hardening component
<code>mu (float)</code>	exponent of non-linear hardening component
<code>'-P' matTag (int)</code>	tag associated with previously-defined UniaxialMaterial in axial direction
<code>'-T' matTag (int)</code>	tag associated with previously-defined UniaxialMaterial in torsional direction
<code>'-My' matTag (int)</code>	tag associated with previously-defined UniaxialMaterial in moment direction around local y-axis
<code>'-Mz' matTag (int)</code>	tag associated with previously-defined UniaxialMaterial in moment direction around local z-axis
<code>x1 x2 x3 (float)</code>	vector components in global coordinates defining local x-axis (optional)
<code>y1 y2 y3 (float)</code>	vector components in global coordinates defining local y-axis (optional)
<code>sDratio (float)</code>	shear distance from iNode as a fraction of the element length (optional, default = 0.5)
<code>'-doRayleigh' (str)</code>	to include Rayleigh damping from the bearing (optional, default = no Rayleigh damping contribution)
<code>m (float)</code>	element mass (optional, default = 0.0)

**See also:**[Notes](#)**Elastomeric Bearing (Bouc-Wen) Element**

This command is used to construct an elastomericBearing element object, which is defined by two nodes. The element can have zero length or the appropriate bearing height. The bearing has unidirectional (2D) or coupled (3D) plasticity properties for the shear deformations, and force-deformation behaviors defined by UniaxialMaterials in the remaining two (2D) or four (3D) directions. By default (`sDratio = 0.5`) P-Delta moments are equally distributed to the two end-nodes. To avoid the introduction of artificial viscous damping in the isolation system (sometimes referred to as “damping leakage in the isolation system”), the bearing element does not contribute to the Rayleigh damping by default. If the element has non-zero length, the local x-axis is determined from the nodal geometry unless the optional x-axis vector is specified in which case the nodal geometry is ignored and the user-defined orientation is utilized.

```
element ('elastomericBearingBoucWen', eleTag, *eleNodes, kInit, qd, alpha1, alpha2, mu, eta, beta, gamma
        '-P', matTag '-Mz', matTag[-orient', x1, x2, x3, y1, y2, y3][-shearDist', sDratio][-doRayleigh][-mass', m])
```

For a two-dimensional problem

```
element ('elastomericBearingBoucWen', eleTag, *eleNodes, kInit, qd, alpha1, alpha2, mu, eta, beat, gamma,
        '-P', matTag, '-T', matTag, '-My', matTag, '-Mz', matTag[-orient[x1, x2, x3], y1, y2, y3][-shearDist', sDratio][-doRayleigh][-mass', m])
```

For a three-dimensional problem

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of two element nodes
kInit (float)	initial elastic stiffness in local shear direction
qd (float)	characteristic strength
alpha1 (float)	post yield stiffness ratio of linear hardening component
alpha2 (float)	post yield stiffness ratio of non-linear hardening component
mu (float)	exponent of non-linear hardening component
eta (float)	yielding exponent (sharpness of hysteresis loop corners) (default = 1.0)
beta (float)	first hysteretic shape parameter (default = 0.5)
gamma (float)	second hysteretic shape parameter (default = 0.5)
'-P' matTag (int)	tag associated with previously-defined UniaxialMaterial in axial direction
'-T' matTag (int)	tag associated with previously-defined UniaxialMaterial in torsional direction
'-My' matTag (int)	tag associated with previously-defined UniaxialMaterial in moment direction around local y-axis
'-Mz' matTag (int)	tag associated with previously-defined UniaxialMaterial in moment direction around local z-axis
x1 x2 x3 (float)	vector components in global coordinates defining local x-axis (optional)
y1 y2 y3 (float)	vector components in global coordinates defining local y-axis (optional)
sDratio (float)	shear distance from iNode as a fraction of the element length (optional, default = 0.5)
'-doRayleigh' (str)	to include Rayleigh damping from the bearing (optional, default = no Rayleigh damping contribution)
m (float)	element mass (optional, default = 0.0)

**See also:**[Notes](#)**Flat Slider Bearing Element**

This command is used to construct a flatSliderBearing element object, which is defined by two nodes. The iNode represents the flat sliding surface and the jNode represents the slider. The element can have zero length or the appropriate bearing height. The bearing has unidirectional (2D) or coupled (3D) friction properties for the shear deformations, and force-deformation behaviors defined by UniaxialMaterials in the remaining two (2D) or four (3D) directions. To capture the uplift behavior of the bearing, the user-specified UniaxialMaterial in the axial direction is modified for no-tension behavior. By default (sDratio = 0.0) P-Delta moments are entirely transferred to the flat sliding surface (iNode). It is important to note that rotations of the flat sliding surface (rotations at the iNode) affect the shear behavior of the bearing. To avoid the introduction of artificial viscous damping in the isolation system (sometimes referred to as “damping leakage in the isolation system”), the bearing element does not contribute to the Rayleigh damping by default. If the element has non-zero length, the local x-axis is determined from the nodal geometry unless the optional x-axis vector is specified in which case the nodal geometry is ignored and the user-defined orientation is utilized.

```
element ('flatSliderBearing', eleTag, *eleNodes, frnMdlTag, kInit, '-P', matTag, '-Mz', matTag[], '-orient',
         x1, x2, x3, y1, y2, y3][, '-shearDist', sDratio][, '-doRayleigh'][, '-mass', m][, '-iter', maxIter,
         tol])
```

For a two-dimensional problem

```
element ('flatSliderBearing', eleTag, *eleNodes, frnMdlTag, kInit, '-P', matTag, '-T', matTag, '-My', matTag,
         '-Mz', matTag[], '-orient'[, x1, x2, x3], y1, y2, y3][, '-shearDist', sDratio][, '-doRayleigh'][, '-mass',
         m][, '-iter', maxIter, tol])
```

For a three-dimensional problem

<code>eleTag (int)</code>	unique element object tag
<code>eleNodes (list (int))</code>	a list of two element nodes
<code>frnMdlTag (float)</code>	tag associated with previously-defined FrictionModel
<code>kInit (float)</code>	initial elastic stiffness in local shear direction
<code>'-P' matTag (int)</code>	tag associated with previously-defined UniaxialMaterial in axial direction
<code>'-T' matTag (int)</code>	tag associated with previously-defined UniaxialMaterial in torsional direction
<code>'-My' matTag (int)</code>	tag associated with previously-defined UniaxialMaterial in moment direction around local y-axis
<code>'-Mz' matTag (int)</code>	tag associated with previously-defined UniaxialMaterial in moment direction around local z-axis
<code>x1 x2 x3 (float)</code>	vector components in global coordinates defining local x-axis (optional)
<code>y1 y2 y3 (float)</code>	vector components in global coordinates defining local y-axis (optional)
<code>sDratio (float)</code>	shear distance from iNode as a fraction of the element length (optional, default = 0.0)
<code>'-doRayleigh' (str)</code>	to include Rayleigh damping from the bearing (optional, default = no Rayleigh damping contribution)
<code>m (float)</code>	element mass (optional, default = 0.0)
<code>maxIter (int)</code>	maximum number of iterations to undertake to satisfy element equilibrium (optional, default = 20)
<code>tol (float)</code>	convergence tolerance to satisfy element equilibrium (optional, default = 1E-8)

**See also:**[Notes](#)**Single Friction Pendulum Bearing Element**

This command is used to construct a singleFPBearing element object, which is defined by two nodes. The iNode represents the concave sliding surface and the jNode represents the articulated slider. The element can have zero length or the appropriate bearing height. The bearing has unidirectional (2D) or coupled (3D) friction properties (with post-yield stiffening due to the concave sliding surface) for the shear deformations, and force-deformation behaviors defined by UniaxialMaterials in the remaining two (2D) or four (3D) directions. To capture the uplift behavior of the bearing, the user-specified UniaxialMaterial in the axial direction is modified for no-tension behavior. By default (`sDratio = 0.0`) P-Delta moments are entirely transferred to the concave sliding surface (iNode). It is important to note that rotations of the concave sliding surface (rotations at the iNode) affect the shear behavior of the bearing. To avoid the introduction of artificial viscous damping in the isolation system (sometimes referred to as “damping leakage in the isolation system”), the bearing element does not contribute to the Rayleigh damping by default. If the element has non-zero length, the local x-axis is determined from the nodal geometry unless the optional x-axis vector is specified in which case the nodal geometry is ignored and the user-defined orientation is utilized.

```
element ('singleFPBearing', eleTag, *eleNodes, frnMdlTag, Reff, kInit, '-P', matTag, '-Mz', matTag[], '-orient', x1, x2, x3, y1, y2, y3][, '-shearDist', sDratio][, '-doRayleigh'][, '-mass', m][, '-iter', maxIter, tol])
```

For a two-dimensional problem

```
element ('singleFPBearing', eleTag, *eleNodes, frnMdlTag, Reff, kInit, '-P', matTag, '-T', matTag, '-My', matTag, '-Mz', matTag[], '-orient'][, x1, x2, x3], y1, y2, y3][, '-shearDist', sDratio][, '-doRayleigh'][, '-mass', m][, '-iter', maxIter, tol])
```

For a three-dimensional problem

eleTag ( <a href="#">int</a> )	unique element object tag
eleNodes ( <a href="#">list</a> ( <a href="#">int</a> ))	a list of two element nodes
frnMdlTag ( <a href="#">float</a> )	tag associated with previously-defined FrictionModel
Reff ( <a href="#">float</a> )	effective radius of concave sliding surface
kInit ( <a href="#">float</a> )	initial elastic stiffness in local shear direction
'-P' matTag ( <a href="#">int</a> )	tag associated with previously-defined UniaxialMaterial in axial direction
'-T' matTag ( <a href="#">int</a> )	tag associated with previously-defined UniaxialMaterial in torsional direction
'-My' matTag ( <a href="#">int</a> )	tag associated with previously-defined UniaxialMaterial in moment direction around local y axis
'-Mz' matTag ( <a href="#">int</a> )	tag associated with previously-defined UniaxialMaterial in moment direction around local z-axis
x1 x2 x3 ( <a href="#">float</a> )	vector components in global coordinates defining local x-axis (optional)
y1 y2 y3 ( <a href="#">float</a> )	vector components in global coordinates defining local y-axis (optional)
sDratio ( <a href="#">float</a> )	shear distance from iNode as a fraction of the element length (optional, default = 0.0)
'-doRayleigh' ( <a href="#">str</a> )	to include Rayleigh damping from the bearing (optional, default = no Rayleigh damping contribution)
m ( <a href="#">float</a> )	element mass (optional, default = 0.0)
maxIter ( <a href="#">int</a> )	maximum number of iterations to undertake to satisfy element equilibrium (optional, default = 20)
tol ( <a href="#">float</a> )	convergence tolerance to satisfy element equilibrium (optional, default = 1E-8)

**See also:**[Notes](#)**Triple Friction Pendulum Bearing Element**

This command is used to construct a Triple Friction Pendulum Bearing element object, which is defined by two nodes. The element can have zero length or the appropriate bearing height. The bearing has unidirectional (2D) or coupled (3D) friction properties (with post-yield stiffening due to the concave sliding surface) for the shear deformations, and force-deformation behaviors defined by UniaxialMaterials in the remaining two (2D) or four (3D) directions. To capture the uplift behavior of the bearing, the user-specified UniaxialMaterial in the axial direction is modified for no-tension behavior. P-Delta moments are entirely transferred to the concave sliding surface (iNode). It is important to note that rotations of the concave sliding surface (rotations at the iNode) affect the shear behavior of the bearing. If the element has non-zero length, the local x-axis is determined from the nodal geometry unless the optional x-axis vector is specified in which case the nodal geometry is ignored and the user-defined orientation is utilized.

```
element ('TFP', eleTag, *eleNodes, R1, R2, R3, R4, D1, D2, D3, D4, d1, d2, d3, d4, mu1, mu2, mu3, mu4, h1, h2, h3, h4, H0, colLoad[, K ])
```

<code>eleTag (int)</code>	unique element object tag
<code>eleNodes (list (int))</code>	a list of two element nodes
<code>R1 (float)</code>	Radius of inner bottom sliding surface
<code>R2 (float)</code>	Radius of inner top sliding surface
<code>R3 (float)</code>	Radius of outer bottom sliding surface
<code>R4 (float)</code>	Radius of outer top sliding surface
<code>D1 (float)</code>	Diameter of inner bottom sliding surface
<code>D2 (float)</code>	Diameter of inner top sliding surface
<code>D3 (float)</code>	Diameter of outer bottom sliding surface
<code>D4 (float)</code>	Diameter of outer top sliding surface
<code>d1 (float)</code>	diameter of inner slider
<code>d2 (float)</code>	diameter of inner slider
<code>d3 (float)</code>	diameter of outer bottom slider
<code>d4 (float)</code>	diameter of outer top slider
<code>mu1 (float)</code>	friction coefficient of inner bottom sliding surface
<code>mu2 (float)</code>	friction coefficient of inner top sliding surface
<code>mu3 (float)</code>	friction coefficient of outer bottom sliding surface
<code>mu4 (float)</code>	friction coefficient of outer top sliding surface
<code>h1 (float)</code>	height from inner bottom sliding surface to center of bearing
<code>h2 (float)</code>	height from inner top sliding surface to center of bearing
<code>h3 (float)</code>	height from outer bottom sliding surface to center of bearing
<code>h4 (float)</code>	height from inner top sliding surface to center of bearing
<code>H0 (float)</code>	total height of bearing
<code>colLoad (float)</code>	initial axial load on bearing (only used for first time step then load come from model)
<code>K (float)</code>	optional, stiffness of spring in vertical dirn (dof 2 if ndm= 2, dof 3 if ndm = 3) (default=1.0e15)

See also:

Notes

### Triple Friction Pendulum Element

```
element ('TripleFrictionPendulum', eleTag, *eleNodes, frnTag1, frnTag2, frnTag3, vertMatTag, rotZMatTag,
          rotXMatTag, rotYMatTag, L1, L2, L3, d1, d2, d3, W, uy, kvt, minFv, tol)
```

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of two element nodes
frnTag1, frnTag2 frnTag3 (int)	= tags associated with previously-defined FrictionModels at the three sliding interfaces
vertMatTag (int)	Pre-defined material tag for COMPRESSION behavior of the bearing
rotZMatTag rotXMatTag rotYMatTag (int)	Pre-defined material tags for rotational behavior about 3-axis, 1-axis and 2-axis, respectively.
L1 L2 L3 (float)	= effective radii. $L_i = R_i - h_i$ (see Figure 1)
d1 d2 d3 (float)	= displacement limits of pendulums (Figure 1). Displacement limit of the bearing is $2 \cdot d_1 + d_2 + d_3 + L_1 \cdot d_3 / L_3 - L_1 \cdot d_2 / L_2$
W (float)	= axial force used for the first trial of the first analysis step.
uy (float)	= lateral displacement where sliding of the bearing starts. Recommended value = 0.25 to 1 mm. A smaller value may cause convergence problem.
kvt (float)	= Tension stiffness $k_{vt}$ of the bearing.
minFv (>=0) (float)	= minimum vertical compression force in the bearing used for computing the horizontal tangent stiffness matrix from the normalized tangent stiffness matrix of the element. $minFv$ is substituted for the actual compressive force when it is less than $minFv$ , and prevents the element from using a negative stiffness matrix in the horizontal direction when uplift occurs. The vertical nodal force returned to nodes is always computed from $kvc$ (or $kvt$ ) and vertical deformation, and thus is not affected by $minFv$ .
tol (float)	= relative tolerance for checking the convergence of the element. Recommended value = 1.e-10 to 1.e-3.

**See also:**

Notes

### MultipleShearSpring Element

This command is used to construct a multipleShearSpring (MSS) element object, which is defined by two nodes. This element consists of a series of identical shear springs arranged radially to represent the isotropic behavior in the local y-z plane.

```
element ('multipleShearSpring', eleTag, *eleNodes, nSpring, '-mat', matTag[], '-lim', dsp][, '-orient'][, x1, x2, x3], yp1, yp2, yp3][, '-mass', m])
```

<code>eleTag (int)</code>	unique element object tag
<code>eleNodes (list (int))</code>	a list of two element nodes
<code>nSpring (int)</code>	number of springs
<code>matTag (int)</code>	tag associated with previously-defined UniaxialMaterial object
<code>dsp (float)</code>	minimum deformation to calculate equivalent coefficient (see note 1)
<code>x1 x2 x3 (float)</code>	vector components in global coordinates defining local x-axis
<code>yp1 yp2 yp3 (float)</code>	vector components in global coordinates defining vector yp which lies in the local x-y plane for the element
<code>m (float)</code>	element mass

**Note:** If `dsp` is positive and the shear deformation of MSS exceeds `dsp`, this element calculates equivalent coefficient to adjust force and stiffness of MSS. The adjusted MSS force and stiffness reproduce the behavior of the previously defined uniaxial material under monotonic loading in every direction. If `dsp` is zero, the element does not calculate the equivalent coefficient.

#### See also:

[Notes](#)

### KikuchiBearing Element

This command is used to construct a KikuchiBearing element object, which is defined by two nodes. This element consists of multiple shear spring model (MSS) and multiple normal spring model (MNS).

```
element ('KikuchiBearing', eleTag, *eleNodes, '-shape', shape, '-size', size, totalRubber[], '-totalHeight',
         totalHeight], '-nMSS', nMSS, '-matMSS', matMSSTag[], '-limDisp', limDisp], '-nMNS', nMNS, '-matMNS',
         matMNSTag[], '-lambda', lambda][, '-orient'][, x1, x2, x3], yp1, yp2, yp3][, '-mass', m][,
         '-noPDIInput'][, '-noTilt'][, '-adjustPDOOutput', ci, cj][, '-doBalance', limFo, limFi, nIter])
```

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of two element nodes
shape (float)	following shapes are available: round, square
size (float)	diameter (round shape), length of edge (square shape)
totalRubber (float)	total rubber thickness
totalHeight (float)	total height of the bearing (default: distance between iNode and jNode)
nMSS (int)	number of springs in MSS = nMSS
matMSSTag (int)	matTag for MSS
limDisp (float)	minimum deformation to calculate equivalent coefficient of MSS (see note 1)
nMNS (int)	number of springs in MNS = nMNS*nMNS (for round and square shape)
matMNSTag (int)	matTag for MNS
lambda (float)	parameter to calculate compression modulus distribution on MNS (see note 2)
x1 x2 x3 (float)	vector components in global coordinates defining local x-axis
yp1 yp2 yp3 (float)	vector components in global coordinates defining vector yp which lies in the local x-y plane for the element
m (float)	element mass
'-noPDIInput' (str)	not consider P-Delta moment
'-noTilt' (str)	not consider tilt of rigid link
ci cj (float)	P-Delta moment adjustment for reaction force (default: ci =0.5, cj =0.5)
limFo limFi nIter (float)	tolerance of external unbalanced force (limFo), tolerance of internal unbalanced force (limFi), number of iterations to get rid of internal unbalanced force (nIter)

**See also:**[Notes](#)**YamamotoBiaxialHDR Element**

This command is used to construct a YamamotoBiaxialHDR element object, which is defined by two nodes. This element can be used to represent the isotropic behavior of high-damping rubber bearing in the local y-z plane.

```
element ('YamamotoBiaxialHDR', eleTag, *eleNodes, Tp, DDo, DDi, Hr[, '-coRS', cr, cs ][, '-orient'[, x1, x2, x3 ], y1, y2, y3 ][, '-mass', m ])
```

<code>eleTag (int)</code>	unique element object tag
<code>eleNodes (list (int))</code>	a list of two element nodes
<code>Tp (int)</code>	compound type = 1 : X0.6R manufactured by Bridgestone corporation.
<code>DDo (float)</code>	outer diameter [m]
<code>DDi (float)</code>	bore diameter [m]
<code>Hr (float)</code>	total thickness of rubber layer [m] Optional Data
<code>cr cs (float)</code>	coefficients for shear stress components of $\tau_r$ and $\tau_s$
<code>x1 x2 x3 (float)</code>	vector components in global coordinates defining local x-axis
<code>yp1 yp2 yp3 (float)</code>	vector components in global coordinates defining vector $yp$ which lies in the local x-y plane for the element
<code>m (float)</code>	element mass [kg]

**See also:**

Notes

**ElastomericX**

This command is used to construct an ElastomericX bearing element object in three-dimension. The 3D continuum geometry of an elastomeric bearing is modeled as a 2-node, 12 DOF discrete element. This elements extends the formulation of Elastomeric\_Bearing\_(Bouc-Wen)\_Element element. However, instead of the user providing material models as input arguments, it only requires geometric and material properties of an elastomeric bearing as arguments. The material models in six direction are formulated within the element from input arguments. The time-dependent values of mechanical properties (e.g., shear stiffness, buckling load capacity) can also be recorded using the “parameters” recorder.

```
element ('ElastomericX', eleTag, *eleNodes, Fy, alpha, Gr, Kbulk, D1, D2, ts, tr, n[[x1, x2, x3]], y1, y2, y3
        ][, kc ][, PhiM ][, ac ][, sDratio ][, m ][, cd ][, tc ][, tag1 ][, tag2 ][, tag3 ][, tag4 ])
For 3D problem
```

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of two element nodes
Fy (float)	yield strength
alpha (float)	post-yield stiffness ratio
Gr (float)	shear modulus of elastomeric bearing
Kbulk (float)	bulk modulus of rubber
D1 (float)	internal diameter
D2 (float)	outer diameter (excluding cover thickness)
ts (float)	single steel shim layer thickness
tr (float)	single rubber layer thickness
n (int)	number of rubber layers
x1 x2 x3 (float)	vector components in global coordinates defining local x-axis (optional)
y1 y2 y3 (float)	vector components in global coordinates defining local y-axis (optional)
kc (float)	cavitation parameter (optional, default = 10.0)
PhiM (float)	damage parameter (optional, default = 0.5)
ac (float)	strength reduction parameter (optional, default = 1.0)
sDratio (float)	shear distance from iNode as a fraction of the element length (optional, default = 0.5)
m (float)	element mass (optional, default = 0.0)
cd (float)	viscous damping parameter (optional, default = 0.0)
tc (float)	cover thickness (optional, default = 0.0)
tag1 (float)	Tag to include cavitation and post-cavitation (optional, default = 0)
tag2 (float)	Tag to include buckling load variation (optional, default = 0)
tag3 (float)	Tag to include horizontal stiffness variation (optional, default = 0)
tag4 (float)	Tag to include vertical stiffness variation (optional, default = 0)

---

**Note:** Because default values of heating parameters are in SI units, user must override the default heating parameters values if using Imperial units

User should distinguish between yield strength of elastomeric bearing ( $F_y$ ) and characteristic strength ( $Q_d$ ):  $Q_d = F_y * (1 - \alpha)$

---

#### See also:

Notes

### LeadRubberX

This command is used to construct a LeadRubberX bearing element object in three-dimension. The 3D continuum geometry of a lead rubber bearing is modeled as a 2-node, 12 DOF discrete element. It extends the formulation of ElastomericX by including strength degradation in lead rubber bearing due to heating of the lead-core. The Lead-RubberX element requires only the geometric and material properties of an elastomeric bearing as arguments. The material models in six direction are formulated within the element from input arguments. The time-dependent values of mechanical properties (e.g., shear stiffness, buckling load capacity, temperature in the lead-core, yield strength) can also be recorded using the “parameters” recorder.

```
element ('LeadRubberX', eleTag, *eleNodes, Fy, alpha, Gr, Kbulk, D1, D2, ts, tr, n[[x1, x2, x3]], y1, y2, y3
        ][, kc][, PhiM][, ac][, sDratio][, m][, cd][, tc][, qL][, cL][, kS][, aS][, tag1][, tag2][,
        tag3][, tag4][, tag5])
```

<code>eleTag (int)</code>	unique element object tag
<code>eleNodes (list (int))</code>	a list of two element nodes
<code>Fy (float)</code>	yield strength
<code>alpha (float)</code>	post-yield stiffness ratio
<code>Gr (float)</code>	shear modulus of elastomeric bearing
<code>Kbulk (float)</code>	bulk modulus of rubber
<code>D1 (float)</code>	internal diameter
<code>D2 (float)</code>	outer diameter (excluding cover thickness)
<code>ts (float)</code>	single steel shim layer thickness
<code>tr (float)</code>	single rubber layer thickness
<code>n (int)</code>	number of rubber layers
<code>x1 x2 x3 (float)</code>	vector components in global coordinates defining local x-axis (optional)
<code>y1 y2 y3 (float)</code>	vector components in global coordinates defining local y-axis (optional)
<code>kc (float)</code>	cavitation parameter (optional, default = 10.0)
<code>PhiM (float)</code>	damage parameter (optional, default = 0.5)
<code>ac (float)</code>	strength reduction parameter (optional, default = 1.0)
<code>sDratio (float)</code>	shear distance from iNode as a fraction of the element length (optional, default = 0.5)
<code>m (float)</code>	element mass (optional, default = 0.0)
<code>cd (float)</code>	viscous damping parameter (optional, default = 0.0)
<code>tc (float)</code>	cover thickness (optional, default = 0.0)
<code>qL (float)</code>	density of lead (optional, default = 11200 kg/m <sup>3</sup> )
<code>cL (float)</code>	specific heat of lead (optional, default = 130 N·m/kg °C)
<code>kS (float)</code>	thermal conductivity of steel (optional, default = 50 W/m °C)
<code>aS (float)</code>	thermal diffusivity of steel (optional, default = 1.41e-05 m <sup>2</sup> /s)
<code>tag1 (int)</code>	Tag to include cavitation and post-cavitation (optional, default = 0)
<code>tag2 (int)</code>	Tag to include buckling load variation (optional, default = 0)
<code>tag3 (int)</code>	Tag to include horizontal stiffness variation (optional, default = 0)
<code>tag4 (int)</code>	Tag to include vertical stiffness variation (optional, default = 0)
<code>tag5 (int)</code>	Tag to include strength degradation in shear due to heating of lead core (optional, default = 0)

**Note:** Because default values of heating parameters are in SI units, user must override the default heating parameters values if using Imperial units

User should distinguish between yield strength of elastomeric bearing ( $F_y$ ) and characteristic strength ( $Q_d$ ):  $Q_d = F_y * (1 - \text{alpha})$

## See also:

[Notes](#)

## HDR

This command is used to construct an HDR bearing element object in three-dimension. The 3D continuum geometry of an high damping rubber bearing is modeled as a 2-node, 12 DOF discrete element. This is the third element in the series of elements developed for analysis of base-isolated structures under extreme loading (others being ElastomericX and LeadRubberX). The major difference between HDR element with ElastomericX is the hysteresis model in shear. The HDR element uses a model proposed by Grant et al. (2004) to capture the shear behavior of a high damping rubber bearing. The time-dependent values of mechanical properties (e.g., vertical stiffness, buckling load capacity) can also be recorded using the “parameters” recorder.

```
element ('HDR', eleTag, *eleNodes, Gr, Kbulk, D1, D2, ts, tr, n, a1, a2, a3, b1, b2, b3, c1, c2, c3, c4[[, x1,
    x2, x3], y1, y2, y3][[, kc ][, PhiM ][, ac ][, sDratio ][, m ][, tc ]])
```

For 3D problem

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of two element nodes
Gr (float)	shear modulus of elastomeric bearing
Kbulk (float)	bulk modulus of rubber
D1 (float)	internal diameter
D2 (float)	outer diameter (excluding cover thickness)
ts (float)	single steel shim layer thickness
tr (float)	single rubber layer thickness
n (int)	number of rubber layers
a1 a2 a3 b1 b2 b3 c1 c2 c3 c4 (float)	parameters of the Grant model
x1 x2 x3 (float)	vector components in global coordinates defining local x-axis (optional)
y1 y2 y3 (float)	vector components in global coordinates defining local y-axis (optional)
kc (float)	cavitation parameter (optional, default = 10.0)
PhiM (float)	damage parameter (optional, default = 0.5)
ac (float)	strength reduction parameter (optional, default = 1.0)
sDratio (float)	shear distance from iNode as a fraction of the element length (optional, default = 0.5)
m (float)	element mass (optional, default = 0.0)
tc (float)	cover thickness (optional, default = 0.0)

#### See also:

Notes

### RJ-Watson EQS Bearing Element

This command is used to construct a RJWatsonEqsBearing element object, which is defined by two nodes. The iNode represents the masonry plate and the jNode represents the sliding surface plate. The element can have zero length or the appropriate bearing height. The bearing has unidirectional (2D) or coupled (3D) friction properties (with post-yield stiffening due to the mass-energy-regulator (MER) springs) for the shear deformations, and force-deformation behaviors defined by UniaxialMaterials in the remaining two (2D) or four (3D) directions. To capture the uplift behavior of the bearing, the user-specified UniaxialMaterial in the axial direction is modified for no-tension behavior. By default (sDratio = 1.0) P-Delta moments are entirely transferred to the sliding surface (jNode). It is important to note that rotations of the sliding surface (rotations at the jNode) affect the shear behavior of the bearing. To avoid the introduction of artificial viscous damping in the isolation system (sometimes referred to as “damping leakage in the isolation system”), the bearing element does not contribute to the Rayleigh damping by default. If the element has non-zero length, the local x-axis is determined from the nodal geometry unless the optional x-axis vector is specified in which case the nodal geometry is ignored and the user-defined orientation is utilized.

```
element ('RJWatsonEqsBearing', eleTag, *eleNodes, frnMdlTag, kInit, '-P', matTag, '-Vy', matTag, '-Mz',
    matTag[, '-orient', x1, x2, x3, y1, y2, y3][[, '-shearDist', sDratio ][, '-doRayleigh'][, '-mass', m
    ][, '-iter', maxIter, tol]])
```

For a two-dimensional problem

```
element ('RJWatsonEqsBearing', eleTag, *eleNodes, frnMdlTag, kInit, '-P', matTag, '-Vy', matTag, '-Vz',
    matTag, '-T', matTag, '-My', matTag, '-Mz', matTag[, '-orient'][, x1, x2, x3], y1, y2, y3][[, '-shearDist', sDratio ][, '-doRayleigh'][, '-mass', m ][, '-iter', maxIter, tol]])
```

For a three-dimensional problem

<code>eleTag (int)</code>	unique element object tag
<code>eleNodes (list (int))</code>	a list of two element nodes
<code>frnMdlTag (float)</code>	tag associated with previously-defined FrictionModel
<code>kInit (float)</code>	initial stiffness of sliding friction component in local shear direction
<code>'-P' matTag (int)</code>	tag associated with previously-defined UniaxialMaterial in axial direction
<code>'-Vy' matTag (int)</code>	tag associated with previously-defined UniaxialMaterial in shear direction along local y-axis (MER spring behavior not including friction)
<code>'-Vz' matTag (int)</code>	tag associated with previously-defined UniaxialMaterial in shear direction along local z-axis (MER spring behavior not including friction)
<code>'-T' matTag (int)</code>	tag associated with previously-defined UniaxialMaterial in torsional direction
<code>'-My' matTag (int)</code>	tag associated with previously-defined UniaxialMaterial in moment direction around local y-axis
<code>'-Mz' matTag (int)</code>	tag associated with previously-defined UniaxialMaterial in moment direction around local z-axis
<code>x1 x2 x3 (float)</code>	vector components in global coordinates defining local x-axis (optional)
<code>y1 y2 y3 (float)</code>	vector components in global coordinates defining local y-axis (optional)
<code>sDratio (float)</code>	shear distance from iNode as a fraction of the element length (optional, default = 0.0)
<code>'-doRayleigh (str)</code>	to include Rayleigh damping from the bearing (optional, default = no Rayleigh damping contribution)
<code>m (float)</code>	element mass (optional, default = 0.0)
<code>maxIter (int)</code>	maximum number of iterations to undertake to satisfy element equilibrium (optional, default = 20)
<code>tol (float)</code>	convergence tolerance to satisfy element equilibrium (optional, default = 1E-8)

### See also:

Notes

### FPBearingPTV

The FPBearingPTV command creates a single Friction Pendulum bearing element, which is capable of accounting for the changes in the coefficient of friction at the sliding surface with instantaneous values of the sliding velocity, axial pressure and temperature at the sliding surface. The constitutive modelling is similar to the existing singleFPBearing element, otherwise. The FPBearingPTV element has been verified and validated in accordance with the ASME guidelines, details of which are presented in Chapter 4 of Kumar et al. (2015a).

```
element ('FPBearingPTV', eleTag, *eleNodes, MuRef, IsPressureDependent, pRef, isTemperatureDependent, Diffusivity, Conductivity, IsVelocityDependent, rateParameter, ReflectiveFP, Radius_Contact, kInitial, theMaterialA, theMaterialB, theMaterialC, theMaterialD, x1, x2, x3, y1, y2, y3, shearDist, doRayleigh, mass, iter, tol, unit)
```

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of two element nodes
MuRef (float)	Reference coefficient of friction
IsPressureDependent (int)	1 if the coefficient of friction is a function of instantaneous axial pressure
pRef (float)	Reference axial pressure (the bearing pressure under static loads)
IsTemperatureDependent (int)	1 if the coefficient of friction is a function of instantaneous temperature at the sliding surface
Diffusivity (float)	Thermal diffusivity of steel
Conductivity (float)	Thermal conductivity of steel
IsVelocityDependent (int)	1 if the coefficient of friction is a function of instantaneous velocity at the sliding surface
rateParameter (float)	The exponent that determines the shape of the coefficient of friction vs. sliding velocity curve
ReffectiveFP (float)	Effective radius of curvature of the sliding surface of the FPbearing
Radius_Contact (float)	Radius of contact area at the sliding surface
kInitial (float)	Lateral stiffness of the sliding bearing before sliding begins
theMaterialA (int)	Tag for the uniaxial material in the axial direction
theMaterialB (int)	Tag for the uniaxial material in the torsional direction
theMaterialC (int)	Tag for the uniaxial material for rocking about local Y axis
theMaterialD (int)	Tag for the uniaxial material for rocking about local Z axis
x1 x2 x3 (float)	Vector components to define local X axis
y1 y2 y3 (float)	Vector components to define local Y axis
shearDist (float)	Shear distance from iNode as a fraction of the length of the element
doRayleigh (int)	To include Rayleigh damping from the bearing
mass (float)	Element mass
iter (int)	Maximum number of iterations to satisfy the equilibrium of element
tol (float)	Convergence tolerance to satisfy the equilibrium of the element
unit (int)	Tag to identify the unit from the list below. <ul style="list-style-type: none"><li>• 1: N, m, s, C</li><li>• 2: kN, m, s, C</li><li>• 3: N, mm, s, C</li><li>• 4: kN, mm, s, C</li><li>• 5: lb, in, s, C</li><li>• 6: kip, in, s, C</li><li>• 7: lb, ft, s, C</li><li>• 8: kip, ft, s, C</li></ul>

**See also:**

Notes

## Quadrilateral Elements

1. [Quad Element](#)
2. [Shell Element](#)
3. [ShellDKGQ](#)
4. [ShellDKGT](#)
5. [ShellNLDKGQ](#)
6. [ShellNLDKGT](#)
7. [ShellNL](#)
8. [Bbar Plane Strain Quadrilateral Element](#)
9. [Enhanced Strain Quadrilateral Element](#)
10. [SSPquad Element](#)

### Quad Element

This command is used to construct a FourNodeQuad element object which uses a bilinear isoparametric formulation.

**element** ('quad', eleTag, \*eleNodes, thick, type, matTag[, pressure, rho, b1, b2])

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of four element nodes in counter-clockwise order
thick (float)	element thickness
type (str)	string representing material behavior. The type parameter can be either 'PlaneStrain' or 'PlaneStress'
matTag (int)	tag of nDMaterial
pressure (float)	surface pressure (optional, default = 0.0)
rho (float)	element mass density (per unit volume) from which a lumped element mass matrix is computed (optional, default=0.0)
b1 b2 (float)	constant body forces defined in the isoparametric domain (optional, default=0.0)

---

#### Note:

1. Consistent nodal loads are computed from the pressure and body forces.
  2. The valid queries to a Quad element when creating an ElementRecorder object are ‘forces’, ‘stresses,’ and ‘material \$matNum matArg1 matArg2 ...’ Where \$matNum refers to the material object at the integration point corresponding to the node numbers in the isoparametric domain.
- 

#### See also:

Notes

## Shell Element

This command is used to construct a ShellMITC4 element object, which uses a bilinear isoparametric formulation in combination with a modified shear interpolation to improve thin-plate bending performance.

**element** ('ShellMITC4', eleTag, \*eleNodes, secTag)

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of four element nodes in counter-clockwise order
secTag (int)	tag associated with previously-defined SectionForceDeformation object. Currently must be either a 'PlateFiberSection', or 'ElasticMembranePlateSection'

---

### Note:

1. The valid queries to a Quad element when creating an ElementRecorder object are ‘forces’, ‘stresses,’ and ‘material \$matNum matArg1 matArg2 ...’ Where \$matNum refers to the material object at the integration point corresponding to the node numbers in the isoparametric domain.
  2. It is a 3D element with 6 dofs and CAN NOT be used in 2D domain.
- 

### See also:

Notes

## ShellDKGQ

This command is used to construct a ShellDKGQ element object, which is a quadrilateral shell element based on the theory of generalized conforming element.

**element** ('ShellDKGQ', eleTag, \*eleNodes, secTag)

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of four element nodes in counter-clockwise order
secTag (int)	tag associated with previously-defined SectionForceDeformation object. Currently can be a 'PlateFiberSection', a 'ElasticMembranePlateSection' and a 'LayeredShell' section

### See also:

Notes

## ShellDKGT

This command is used to construct a ShellDKGT element object, which is a triangular shell element based on the theory of generalized conforming element.

**element ('ShellDKGT', eleTag, \*eleNodes, secTag)**

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of three element nodes in clockwise or counter-clockwise order
secTag (int)	tag associated with previously-defined SectionForceDeformation object. currently can be a 'PlateFiberSection', a 'ElasticMembranePlateSection' and a 'LayeredShell' section

**See also:**

[Notes](#)

### ShellINLDKGQ

This command is used to construct a ShellINLDKGQ element object accounting for the geometric nonlinearity of large deformation using the updated Lagrangian formula, which is developed based on the ShellDKGT element.

**element ('ShellINLDKGQ', eleTag, \*eleNodes, secTag)**

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of four element nodes in counter-clockwise order
secTag (int)	tag associated with previously-defined SectionForceDeformation object. currently can be a 'PlateFiberSection', a 'ElasticMembranePlateSection' and a 'LayeredShell' section

**See also:**

[Notes](#)

### ShellINLDKGT

This command is used to construct a ShellINLDKGT element object accounting for the geometric nonlinearity of large deformation using the updated Lagrangian formula, which is developed based on the ShellDKGT element.

**element ('ShellINLDKGT', eleTag, \*eleNodes, secTag)**

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of three element nodes in clockwise or counter-clockwise order around the element
secTag (int)	tag associated with previously-defined SectionForceDeformation object. currently can be a 'PlateFiberSection', a 'ElasticMembranePlateSection' and a 'LayeredShell' section

**See also:**

Notes

## ShellNL

**element** ('ShellNL', eleTag, \*eleNodes, sectTag)

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of nine element nodes, input is the typical, firstly four corner nodes counter-clockwise, then mid-side nodes counter-clockwise and finally the central node
sectTag (int)	tag associated with previously-defined SectionForceDeformation object. currently can be a 'PlateFiberSection', a 'ElasticMembranePlateSection' and a 'LayeredShell' section

See also:

Notes

## Bbar Plane Strain Quadrilateral Element

This command is used to construct a four-node quadrilateral element object, which uses a bilinear isoparametric formulation along with a mixed volume/pressure B-bar assumption. This element is for plane strain problems only.

**element** ('bbarQuad', eleTag, \*eleNodes, thick, matTag)

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of four element nodes in counter-clockwise order
thick (float)	element thickness
matTag (int)	tag of nDMaterial

---

Note:

1. PlainStrain only.
  2. The valid queries to a Quad element when creating an ElementRecorder object are ‘forces’, ‘stresses,’ and ‘material \$matNum matArg1 matArg2 ...’ Where \$matNum refers to the material object at the integration point corresponding to the node numbers in the isoparametric domain.
- 

See also:

Notes

## Enhanced Strain Quadrilateral Element

This command is used to construct a four-node quadrilateral element, which uses a bilinear isoparametric formulation with enhanced strain modes.

**element ('enhancedQuad', eleTag, \*eleNodes, thick, type, matTag)**

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of four element nodes in counter-clockwise order
thick (float)	element thickness
type (str)	string representing material behavior. Valid options depend on the NDMaterial object and its available material formulations. The type parameter can be either 'PlaneStrain' or 'PlaneStress'
matTag (int)	tag of nDMaterial

**See also:**

Notes

### SSPquad Element

This command is used to construct a SSPquad element object.

**element ('SSPquad', eleTag, \*eleNodes, matTag, type, thick[, b1, b2])**

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of four element nodes in counter-clockwise order
thick (float)	thickness of the element in out-of-plane direction
type (str)	string to relay material behavior to the element, can be either 'PlaneStrain' or 'PlaneStress'
matTag (int)	unique integer tag associated with previously-defined nDMaterial object
b1 b2 (float)	constant body forces in global x- and y-directions, respectively (optional, default = 0.0)

The SSPquad element is a four-node quadrilateral element using physically stabilized single-point integration (SSP → Stabilized Single Point). The stabilization incorporates an assumed strain field in which the volumetric dilation and the shear strain associated with the hourglass modes are zero, resulting in an element which is free from volumetric and shear locking. The elimination of shear locking results in greater coarse mesh accuracy in bending dominated problems, and the elimination of volumetric locking improves accuracy in nearly-incompressible problems. Analysis times are generally faster than corresponding full integration elements. The formulation for this element is identical to the solid phase portion of the SSPquadUP element as described by McGann et al. (2012).

---

**Note:**

1. Valid queries to the SSPquad element when creating an ElementalRecorder object correspond to those for the nDMaterial object assigned to the element (e.g., 'stress', 'strain'). Material response is recorded at the single integration point located in the center of the element.
2. The SSPquad element was designed with intentions of duplicating the functionality of the Quad Element. If an example is found where the SSPquad element cannot do something that works for the Quad Element, e.g., material updating, please contact the developers listed below so the bug can be fixed.

---

**See also:**

Notes

## Triangular Elements

1. *Tri31 Element*

### Tri31 Element

This command is used to construct a constant strain triangular element (Tri31) which uses three nodes and one integration points.

**element** ('Tri31', eleTag, \*eleNodes, thick, type, matTag[, pressure, rho, b1, b2])

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of three element nodes in counter-clockwise order
thick (float)	element thickness
type (str)	string representing material behavior. The type parameter can be either 'PlaneStrain' or 'PlaneStress'
matTag (int)	tag of nDMaterial
pressure (float)	surface pressure (optional, default = 0.0)
rho (float)	element mass density (per unit volume) from which a lumped element mass matrix is computed (optional, default=0.0)
b1 b2 (float)	constant body forces defined in the domain (optional, default=0.0)

---

**Note:**

1. Consistent nodal loads are computed from the pressure and body forces.
  2. The valid queries to a Tri31 element when creating an ElementRecorder object are ‘forces’, ‘stresses,’ and ‘material \$matNum matArg1 matArg2 ...’ Where \$matNum refers to the material object at the integration point corresponding to the node numbers in the domain.
- 

**See also:**

Notes

## Brick Elements

1. *Standard Brick Element*
2. *Bbar Brick Element*
3. *Twenty Node Brick Element*
4. *SSPbrick Element*

## Standard Brick Element

This element is used to construct an eight-node brick element object, which uses a trilinear isoparametric formulation.

**element** ('stdBrick', eleTag, \*eleNodes, matTag[, b1, b2, b3])

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of eight element nodes in bottom and top faces and in counter-clockwise order
matTag (int)	tag of nDMaterial
b1 b2 b3 (float)	body forces in global x,y,z directions

---

### Note:

1. The valid queries to a Brick element when creating an ElementRecorder object are ‘forces’, ‘stresses,’ (‘strains’ version > 2.2.0) and ‘material \$matNum matArg1 matArg2 ...’ Where \$matNum refers to the material object at the integration point corresponding to the node numbers in the isoparametric domain.
  2. This element can only be defined in -ndm 3 -ndf 3
- 

### See also:

[Notes](#)

## Bbar Brick Element

This command is used to construct an eight-node mixed volume/pressure brick element object, which uses a trilinear isoparametric formulation.

**element** ('bbarBrick', eleTag, \*eleNodes, matTag[, b1, b2, b3])

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of eight element nodes in bottom and top faces and in counter-clockwise order
matTag (int)	tag of nDMaterial
b1 b2 b3 (float)	body forces in global x,y,z directions

---

### Note:

1. Node numbering for this element is different from that for the eight-node brick (Brick8N) element.
  2. The valid queries to a Quad element when creating an ElementRecorder object are ‘forces’, ‘stresses’, ‘strains’, and ‘material \$matNum matArg1 matArg2 ...’ Where \$matNum refers to the material object at the integration point corresponding to the node numbers in the isoparametric domain.
- 

### See also:

[Notes](#)

## Twenty Node Brick Element

The element is used to construct a twenty-node three dimensional element object

**element** ('Brick20N', *eleTag*, \**eleNodes*, *matTag*, *bf1*, *bf2*, *bf3*, *massDen*)

<i>eleTag</i> (int)	unique element object tag
<i>eleNodes</i> (list (int))	a list of twenty element nodes, input order is shown in notes below
<i>matTag</i> (int)	material tag associated with previously-defined NDMaterial object
<i>bf1</i> <i>bf2</i> <i>bf3</i> (float)	body force in the direction of global coordinates x, y and z
<i>massDen</i> (float)	mass density (mass/volume)

---

**Note:** The valid queries to a Brick20N element when creating an ElementRecorder object are ‘force,’ ‘stiffness,’ ‘stress’, ‘gausspoint’ or ‘plastic’. The output is given as follows:

1. ‘stress’

the six stress components from each Gauss points are output by the order: sigma\_xx, sigma\_yy, sigma\_zz, sigma\_xy, sigma\_xz, sigma\_yz

2. ‘gausspoint’

the coordinates of all Gauss points are printed out

3. ‘plastic’

the equivalent deviatoric plastic strain from each Gauss point is output in the same order as the coordinates are printed

---

### See also:

Notes

## SSPbrick Element

This command is used to construct a SSPbrick element object.

**element** ('SSPbrick', *eleTag*, \**eleNodes*, *matTag*[, *b1*, *b2*, *b3*])

<i>eleTag</i> (int)	unique element object tag
<i>eleNodes</i> (list (int))	a list of eight element nodes in bottom and top faces and in counter-clockwise order
<i>matTag</i> (int)	unique integer tag associated with previously-defined nDMaterial object
<i>b1</i> <i>b2</i> <i>b3</i> (float)	constant body forces in global x-, y-, and z-directions, respectively (optional, default = 0.0)

The SSPbrick element is an eight-node hexahedral element using physically stabilized single-point integration (SSP → Stabilized Single Point). The stabilization incorporates an enhanced assumed strain field, resulting in an element which is free from volumetric and shear locking. The elimination of shear locking results in greater coarse mesh accuracy in bending dominated problems, and the elimination of volumetric locking improves accuracy in nearly-incompressible problems. Analysis times are generally faster than corresponding full integration elements.

---

**Note:**

1. Valid queries to the SSPbrick element when creating an ElementalRecorder object correspond to those for the nDMaterial object assigned to the element (e.g., ‘stress’, ‘strain’). Material response is recorded at the single integration point located in the center of the element.
  2. The SSPbrick element was designed with intentions of duplicating the functionality of the stdBrick Element. If an example is found where the SSPbrick element cannot do something that works for the stdBrick Element, e.g., material updating, please contact the developers listed below so the bug can be fixed.
- 

**See also:**

Notes

## Tetrahedron Elements

1. *FourNodeTetrahedron*

### FourNodeTetrahedron

This command is used to construct a standard four-node tetrahedron element objec with one-point Gauss integration.

**element** ('FourNodeTetrahedron', eleTag, \*eleNodes, matTag[, b1, b2, b3])

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of four element nodes
mat Tag (int)	tag of nDMaterial
b1 b2 b3 (float)	body forces in global x,y,z directions

**See also:**

Notes

## UC San Diego u-p element (saturated soil)

1. *Four Node Quad u-p Element*
2. *Brick u-p Element*
3. *BbarQuad u-p Element*
4. *BbarBrick u-p Element*
5. *Nine Four Node Quad u-p Element*
6. *Twenty Eight Node Brick u-p Element*

### Four Node Quad u-p Element

FourNodeQuadUP is a four-node plane-strain element using bilinear isoparametric formulation. This element is implemented for simulating dynamic response of solid-fluid fully coupled material, based on Biot’s theory of porous medium. Each element node has 3 degrees-of-freedom (DOF): DOF 1 and 2 for solid displacement (u) and DOF 3 for fluid pressure (p).

**element** ('', eleTag, \*eleNodes, thick, matTag, bulk, fmass, hPerm, vPerm[, b1=0, b2=0, t=0 ])

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of four element nodes in counter-clockwise order
thick (float)	Element thickness
matTag (int)	Tag of an NDMaterial object (previously defined) of which the element is composed
bulk (float)	Combined undrained bulk modulus $B_c$ relating changes in pore pressure and volumetric strain, may be approximated by: $B_c \approx B_f/n$ where $B_f$ is the bulk modulus of fluid phase ( $2.2 \times 10^6$ kPa (or $3.191 \times 10^5$ psi) for water), and $n$ the initial porosity.
fmass (float)	Fluid mass density
hPerm, vPerm (float)	Permeability coefficient in horizontal and vertical directions respectively.
b1, b2 (float)	Optional gravity acceleration components in horizontal and vertical directions respectively (defaults are 0.0)
t (float)	Optional uniform element normal traction, positive in tension (default is 0.0)

**See also:**

[Notes](#)

### Brick u-p Element

BrickUP is an 8-node hexahedral linear isoparametric element. Each node has 4 degrees-of-freedom (DOF): DOFs 1 to 3 for solid displacement (u) and DOF 4 for fluid pressure (p). This element is implemented for simulating dynamic response of solid-fluid fully coupled material, based on Biot's theory of porous medium.

**element** ('brickUP', eleTag, \*eleNodes, matTag, bulk, fmass, PermX, PermY, PermZ[, bX=0, bY=0, bZ=0 ])

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of eight element nodes
matTag (int)	Tag of an NDMaterial object (previously defined) of which the element is composed
bulk (float)	Combined undrained bulk modulus $B_c$ relating changes in pore pressure and volumetric strain, may be approximated by: $B_c \approx B_f/n$ where $B_f$ is the bulk modulus of fluid phase ( $2.2 \times 10^6$ kPa (or $3.191 \times 10^5$ psi) for water), and n the initial porosity.
fmass (float)	Fluid mass density
permX, permY, permZ (float)	Permeability coefficients in x, y, and z directions respectively.
bX, bY, bZ (float)	Optional gravity acceleration components in x, y, and z directions directions respectively (defaults are 0.0)

**See also:**

Notes

**BbarQuad u-p Element**

bbarQuadUP is a four-node plane-strain mixed volume/pressure element, which uses a tri-linear isoparametric formulation. This element is implemented for simulating dynamic response of solid-fluid fully coupled material, based on Biot's theory of porous medium. Each element node has 3 degrees-of-freedom (DOF): DOF 1 and 2 for solid displacement (u) and DOF 3 for fluid pressure (p).

```
element ('bbarQuadUP', eleTag, *eleNodes, thick, matTag, bulk, fmass, hPerm, vPerm[, b1=0, b2=0, t=0])
```

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of four element nodes in counter-clockwise order
thick (float)	Element thickness
matTag (int)	Tag of an NDMaterial object (previously defined) of which the element is composed
bulk (float)	Combined undrained bulk modulus $B_c$ relating changes in pore pressure and volumetric strain, may be approximated by: $B_c \approx B_f/n$ where $B_f$ is the bulk modulus of fluid phase ( $2.2 \times 10^6$ kPa (or $3.191 \times 10^5$ psi) for water), and $n$ the initial porosity.
fmass (float)	Fluid mass density
hPerm, vPerm (float)	Permeability coefficient in horizontal and vertical directions respectively.
b1, b2 (float)	Optional gravity acceleration components in horizontal and vertical directions respectively (defaults are 0.0)
t (float)	Optional uniform element normal traction, positive in tension (default is 0.0)

See also:

Notes

### BbarBrick u-p Element

bbarBrickUP is a 8-node mixed volume/pressure element, which uses a tri-linear isoparametric formulation.

Each node has 4 degrees-of-freedom (DOF): DOFs 1 to 3 for solid displacement (u) and DOF 4 for fluid pressure (p). This element is implemented for simulating dynamic response of solid-fluid fully coupled material, based on Biot's theory of porous medium.

```
element ('bbarBrickUP', eleTag, *eleNodes, matTag, bulk, fmass, PermX, PermY, PermZ[, bX=0, bY=0, bZ=0])
```

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of eight element nodes
matTag (int)	Tag of an NDMaterial object (previously defined) of which the element is composed
bulk (float)	Combined undrained bulk modulus $B_c$ relating changes in pore pressure and volumetric strain, may be approximated by: $B_c \approx B_f/n$ where $B_f$ is the bulk modulus of fluid phase ( $2.2 \times 10^6$ kPa (or $3.191 \times 10^5$ psi) for water), and n the initial porosity.
fmass (float)	Fluid mass density
permX, permY, permZ (float)	Permeability coefficients in x, y, and z directions respectively.
bX, bY, bZ (float)	Optional gravity acceleration components in x, y, and z directions directions respectively (defaults are 0.0)

**See also:**

Notes

**Nine Four Node Quad u-p Element**

Nine\_Four\_Node\_QuadUP is a 9-node quadrilateral plane-strain element. The four corner nodes have 3 degrees-of-freedom (DOF) each: DOF 1 and 2 for solid displacement (u) and DOF 3 for fluid pressure (p). The other five nodes have 2 DOFs each for solid displacement. This element is implemented for simulating dynamic response of solid-fluid fully coupled material, based on Biot's theory of porous medium.

```
element ('9_4_QuadUP', eleTag, *eleNodes, thick, matTag, bulk, fmass, hPerm, vPerm[, b1=0, b2=0])
```

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of nine element nodes
thick (float)	Element thickness
matTag (int)	Tag of an NDMaterial object (previously defined) of which the element is composed
bulk (float)	Combined undrained bulk modulus $B_c$ relating changes in pore pressure and volumetric strain, may be approximated by: $B_c \approx B_f/n$ where $B_f$ is the bulk modulus of fluid phase ( $2.2 \times 10^6$ kPa (or $3.191 \times 10^5$ psi) for water), and n the initial porosity.
fmass (float)	Fluid mass density
hPerm, vPerm (float)	Permeability coefficient in horizontal and vertical directions respectively.
b1, b2 (float)	Optional gravity acceleration components in horizontal and vertical directions respectively (defaults are 0.0)

**See also:**[Notes](#)

## Twenty Eight Node Brick u-p Element

wenty\_Eight\_Node\_BrickUP is a 20-node hexahedral isoparametric element.

The eight corner nodes have 4 degrees-of-freedom (DOF) each: DOFs 1 to 3 for solid displacement (u) and DOF 4 for fluid pressure (p). The other nodes have 3 DOFs each for solid displacement. This element is implemented for simulating dynamic response of solid-fluid fully coupled material, based on Biot's theory of porous medium.

```
element ('bbarBrickUP', eleTag, *eleNodes, matTag, bulk, fmass, PermX, PermY, PermZ[, bX=0, bY=0, bZ=0])
```

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of twenty element nodes
matTag (int)	Tag of an NDMaterial object (previously defined) of which the element is composed
bulk (float)	Combined undrained bulk modulus $B_c$ relating changes in pore pressure and volumetric strain, may be approximated by: $B_c \approx B_f/n$ where $B_f$ is the bulk modulus of fluid phase ( $2.2 \times 10^6$ kPa (or $3.191 \times 10^5$ psi) for water), and n the initial porosity.
fmass (float)	Fluid mass density
permX, permY, permZ (float)	Permeability coefficients in x, y, and z directions respectively.
bX, bY, bZ (float)	Optional gravity acceleration components in x, y, and z directions directions respectively (defaults are 0.0)

**See also:**[Notes](#)

## Other u-p elements

1. [SSPquadUP Element](#)
2. [SSPbrickUP Element](#)

## SSPquadUP Element

This command is used to construct a SSPquadUP element object.

```
element ('SSPquadUP', eleTag, *eleNodes, matTag, thick, fBulk, fDen, k1, k2, void, alpha[, b1, b2])
```

<code>eleTag (int)</code>	unique element object tag
<code>eleNodes (list (int))</code>	a list of four element nodes in counter-clockwise order
<code>matTag (int)</code>	unique integer tag associated with previously-defined nDMaterial object
<code>thick (float)</code>	thickness of the element in out-of-plane direction
<code>fBulk (float)</code>	bulk modulus of the pore fluid
<code>fDen (float)</code>	mass density of the pore fluid
<code>k1 k2 (float)</code>	permeability coefficients in global x- and y-directions, respectively
<code>void (float)</code>	voids ratio
<code>alpha (float)</code>	spatial pressure field stabilization parameter (see discussion below for more information)
<code>b1 b2 (float)</code>	constant body forces in global x- and y-directions, respectively (optional, default = 0.0) - See Note 3

The SSPquadUP element is an extension of the SSPquad Element for use in dynamic plane strain analysis of fluid saturated porous media. A mixed displacement-pressure ( $u-p$ ) formulation is used, based upon the work of Biot as extended by Zienkiewicz and Shiomi (1984).

The physical stabilization necessary to allow for reduced integration incorporates an assumed strain field in which the volumetric dilation and the shear strain associated with the hourglass modes are zero, resulting in an element which is free from volumetric and shear locking. The elimination of shear locking results in greater coarse mesh accuracy in bending dominated problems, and the elimination of volumetric locking improves accuracy in nearly-incompressible problems. Analysis times are generally faster than corresponding full integration elements.

Equal-order interpolation is used for the displacement and pressure fields, thus, the SSPquadUP element does not inherently pass the inf-sup condition, and is not fully acceptable in the incompressible-impermeable limit (the QuadUP Element has the same issue). A stabilizing parameter is employed to permit the use of equal-order interpolation for the SSPquadUP element. This parameter \$alpha can be computed as

$$\alpha = 0.25 * (h^2) / (den * c^2)$$

where h is the element size, c is the speed of elastic wave propagation in the solid phase, and den is the mass density of the solid phase. The \$alpha parameter should be a small number. With a properly defined \$alpha parameter, the SSPquadUP element can produce comparable results to a higher-order element such as the 9\_4\_QualUP Element at a significantly lower computational cost and with a greater ease in mesh generation.

The full formulation for the SSPquadUP element can be found in McGann et al. (2012) along with several example applications.

#### Note:

1. The SSPquadUP element will only work in dynamic analysis.
2. For saturated soils, the mass density input into the associated nDMaterial object should be the saturated mass density.
3. When modeling soil, the body forces input into the SSPquadUP element should be the components of the gravitational vector, not the unit weight.
4. Fixing the pore pressure degree-of-freedom (dof 3) at a node is a drainage boundary condition at which zero pore pressure will be maintained throughout the analysis. Leaving the third dof free allows pore pressures to build at that node.
5. Valid queries to the SSPquadUP element when creating an ElementalRecorder object correspond to those for the nDMaterial object assigned to the element (e.g., ‘stress’, ‘strain’). Material response is recorded at the single integration point located in the center of the element.

6. The SSPquadUP element was designed with intentions of duplicating the functionality of the QuadUP Element. If an example is found where the SSPquadUP element cannot do something that works for the QuadUP Element, e.g., material updating, please contact the developers listed below so the bug can be fixed.
- 

**See also:**

Notes

## SSPbrickUP Element

This command is used to construct a SSPbrickUP element object.

**element** ('SSPbrickUP', eleTag, \*eleNodes, matTag, fBulk, fDen, k1, k2, k3, void, alpha[], b1, b2, b3])

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of eight element nodes in counter-clockwise order
matTag (float)	unique integer tag associated with previously-defined nDMaterial object
fBulk (float)	bulk modulus of the pore fluid
fDen (float)	mass density of the pore fluid
k1 k2 k3 (float)	permeability coefficients in global x-, y-, and z-directions, respectively
void (float)	voids ratio
alpha (float)	spatial pressure field stabilization parameter (see discussion below for more information)
b1 b2 b3 (float)	constant body forces in global x-, y-, and z-directions, respectively (optional, default = 0.0) - See Note 3

The SSPbrickUP element is an extension of the SSPbrick Element for use in dynamic 3D analysis of fluid saturated porous media. A mixed displacement-pressure (u-p) formulation is used, based upon the work of Biot as extended by Zienkiewicz and Shiomi (1984).

The physical stabilization necessary to allow for reduced integration incorporates an enhanced assumed strain field, resulting in an element which is free from volumetric and shear locking. The elimination of shear locking results in greater coarse mesh accuracy in bending dominated problems, and the elimination of volumetric locking improves accuracy in nearly-incompressible problems. Analysis times are generally faster than corresponding full integration elements.

Equal-order interpolation is used for the displacement and pressure fields, thus, the SSPbrickUP element does not inherently pass the inf-sup condition, and is not fully acceptable in the incompressible-impermeable limit (the brickUP Element has the same issue). A stabilizing parameter is employed to permit the use of equal-order interpolation for the SSPbrickUP element. This parameter \$alpha can be computed as

$$\alpha = h^2 / (4 * (K_s + (4/3) * G_s))$$

where  $h$  is the element size, and  $K_s$  and  $G_s$  are the bulk and shear moduli for the solid phase. The  $\alpha$  parameter should be a small number. With a properly defined  $\alpha$  parameter, the SSPbrickUP element can produce comparable results to a higher-order element such as the 20\_8\_BrickUP Element at a significantly lower computational cost and with a greater ease in mesh generation.

---

**Note:**

1. The SSPbrickUP element will only work in dynamic analysis.

2. For saturated soils, the mass density input into the associated nDMaterial object should be the saturated mass density.
  3. When modeling soil, the body forces input into the SSPbrickUP element should be the components of the gravitational vector, not the unit weight.
  4. Fixing the pore pressure degree-of-freedom (dof 4) at a node is a drainage boundary condition at which zero pore pressure will be maintained throughout the analysis. Leaving the fourth dof free allows pore pressures to build at that node.
  5. Valid queries to the SSPbrickUP element when creating an ElementalRecorder object correspond to those for the nDMaterial object assigned to the element (e.g., ‘stress’, ‘strain’). Material response is recorded at the single integration point located in the center of the element.
  6. The SSPbrickUP element was designed with intentions of duplicating the functionality of the brickUP Element. If an example is found where the SSPbrickUP element cannot do something that works for the brickUP Element, e.g., material updating, please contact the developers listed below so the bug can be fixed.
- 

**See also:**[Notes](#)**Contact Elements**

1. [\*SimpleContact2D\*](#)
2. [\*SimpleContact3D\*](#)
3. [\*BeamContact2D\*](#)
4. [\*BeamContact3D\*](#)
5. [\*BeamEndContact3D\*](#)

**SimpleContact2D**

This command is used to construct a SimpleContact2D element object.

```
element ('SimpleContact2D', eleTag, iNode, jNode, sNode, lNode, matTag, gTol, fTol)
```

<code>eleTag (int)</code>	unique element object tag
<code>iNode jNode (int)</code>	master nodes (-ndm 2 -ndf 2)
<code>sNode (int)</code>	slave node (-ndm 2 -ndf 2)
<code>lNode (int)</code>	Lagrange multiplier node (-ndm 2 -ndf 2)
<code>matTag (int)</code>	unique integer tag associated with previously-defined nDMaterial object
<code>gTol (float)</code>	gap tolerance
<code>fTol (float)</code>	force tolerance

The SimpleContact2D element is a two-dimensional node-to-segment contact element which defines a frictional contact interface between two separate bodies. The master nodes are the nodes which define the endpoints of a line segment on the first body, and the slave node is a node from the second body. The Lagrange multiplier node is required to enforce the contact condition. This node should not be shared with any other element in the domain. Information on the theory behind this element can be found in, e.g. Wriggers (2002).

**Note:**

1. The SimpleContact2D element has been written to work exclusively with the ContactMaterial2D nDMaterial object.
  2. The valid recorder queries for this element are:
    1. force - returns the contact force acting on the slave node in vector form.
    2. frictionforce - returns the frictional force acting on the slave node in vector form.
    3. forcescalar - returns the scalar magnitudes of the normal and tangential contact forces.
  4. The SimpleContact2D elements are set to consider frictional behavior as a default, but the frictional state of the SimpleContact2D element can be changed from the input file using the setParameter command. When updating, value of 0 corresponds to the frictionless condition, and a value of 1 signifies the inclusion of friction. An example command for this update procedure is provided below
  3. The SimpleContact2D element works well in static and pseudo-static analysis situations.
  4. In transient analysis, the presence of the contact constraints can effect the stability of commonly-used time integration methods in the HHT or Newmark family (e.g., Laursen, 2002). For this reason, use of alternative time-integration methods which numerically damp spurious high frequency behavior may be required. The TRBDF2 integrator is an effective method for this purpose. The Newmark integrator can also be effective with proper selection of the gamma and beta coefficients. The trapezoidal rule, i.e., Newmark with gamma = 0.5 and beta = 0.25, is particularly prone to instability related to the contact constraints and is not recommended.
- 

**See also:**

Notes

### SimpleContact3D

This command is used to construct a SimpleContact3D element object.

**element** ('SimpleContact3D', eleTag, iNode, jNode, kNode, lNode, sNode, LNode, matTag, gTol, fTol)

eleTag (int)	unique element object tag
iNode jNode kNode lNode (int)	master nodes (-ndm 3 -ndf 3)
sNode (int)	slave node (-ndm 3 -ndf 3)
LNode (int)	Lagrange multiplier node (-ndm 3 -ndf 3)
matTag (int)	unique integer tag associated with previously-defined nDMaterial object
gTol (float)	gap tolerance
fTol (float)	force tolerance

The SimpleContact3D element is a three-dimensional node-to-surface contact element which defines a frictional contact interface between two separate bodies. The master nodes are the nodes which define a surface of a hexahedral element on the first body, and the slave node is a node from the second body. The Lagrange multiplier node is required to enforce the contact condition. This node should not be shared with any other element in the domain. Information on the theory behind this element can be found in, e.g. Wriggers (2002).

---

**Note:**

1. The SimpleContact3D element has been written to work exclusively with the ContactMaterial3D nDMaterial object.

2. The valid recorder queries for this element are:
    1. force - returns the contact force acting on the slave node in vector form.
    2. frictionforce - returns the frictional force acting on the slave node in vector form.
    3. forcescalar - returns the scalar magnitudes of the single normal and two tangential contact forces.
    4. The SimpleContact3D elements are set to consider frictional behavior as a default, but the frictional state of the SimpleContact3D element can be changed from the input file using the setParameter command. When updating, value of 0 corresponds to the frictionless condition, and a value of 1 signifies the inclusion of friction. An example command for this update procedure is provided below
  3. The SimpleContact3D element works well in static and pseudo-static analysis situations.
  4. In transient analysis, the presence of the contact constraints can effect the stability of commonly-used time integration methods in the HHT or Newmark family (e.g., Laursen, 2002). For this reason, use of alternative time-integration methods which numerically damp spurious high frequency behavior may be required. The TRBDF2 integrator is an effective method for this purpose. The Newmark integrator can also be effective with proper selection of the gamma and beta coefficients. The trapezoidal rule, i.e., Newmark with gamma = 0.5 and beta = 0.25, is particularly prone to instability related to the contact constraints and is not recommended.
- 

**See also:**[Notes](#)**BeamContact2D**

This command is used to construct a BeamContact2D element object.

**element** ('BeamContact2D', eleTag, matTag, width, gTol, fTol[, cFlag])

eleTag (int)	unique element object tag
iNode jNode (int)	master nodes (-ndm 2 -ndf 3)
sNode (int)	slave node (-ndm 2 -ndf 2)
lNode (int)	Lagrange multiplier node (-ndm 2 -ndf 2)
matTag (int)	unique integer tag associated with previously-defined nDMaterial object
width (float)	the width of the wall represented by the beam element in plane strain
gTol (float)	gap tolerance
fTol (float)	force tolerance
cFlag (int)	optional initial contact flag cFlag = 0 >> contact between bodies is initially assumed (DEFAULT) cFlag = 1 >> no contact between bodies is initially assumed

The BeamContact2D element is a two-dimensional beam-to-node contact element which defines a frictional contact interface between a beam element and a separate body. The master nodes (3 DOF) are the endpoints of the beam

element, and the slave node (2 DOF) is a node from a second body. The Lagrange multiplier node (2 DOF) is required to enforce the contact condition. Each contact element should have a unique Lagrange multiplier node. The Lagrange multiplier node should not be fixed, otherwise the contact condition will not work.

Under plane strain conditions in 2D, a beam element represents a unit thickness of a wall. The width is the dimension of this wall in the 2D plane. This width should be built-in to the model to ensure proper enforcement of the contact condition. The Excavation Supported by Cantilevered Sheet Pile Wall practical example provides some further examples and discussion on the usage of this element.

---

**Note:**

1. The BeamContact2D element has been written to work exclusively with the ContactMaterial2D nDMaterial object.
  2. The valid recorder queries for this element are:
    1. force - returns the contact force acting on the slave node in vector form.
    2. frictionforce - returns the frictional force acting on the slave node in vector form.
    3. forcescalar - returns the scalar magnitudes of the normal and tangential contact forces.
    4. masterforce - returns the reactions (forces and moments) acting on the master nodes.
  5. The BeamContact2D elements are set to consider frictional behavior as a default, but the frictional state of the BeamContact2D element can be changed from the input file using the setParameter command. When updating, value of 0 corresponds to the frictionless condition, and a value of 1 signifies the inclusion of friction. An example command for this update procedure is provided below
  3. The BeamContact2D element works well in static and pseudo-static analysis situations.
  4. In transient analysis, the presence of the contact constraints can effect the stability of commonly-used time integration methods in the HHT or Newmark family (e.g., Laursen, 2002). For this reason, use of alternative time-integration methods which numerically damp spurious high frequency behavior may be required. The TRBDF2 integrator is an effective method for this purpose. The Newmark integrator can also be effective with proper selection of the gamma and beta coefficients. The trapezoidal rule, i.e., Newmark with gamma = 0.5 and beta = 0.25, is particularly prone to instability related to the contact constraints and is not recommended.
- 

**See also:**

Notes

**BeamContact3D**

This command is used to construct a BeamContact3D element object.

```
element ('BeamContact3D', eleTag, iNode, jNode, sNode, lNode, radius, crdTransf, matTag, gTol, fTol[,  
cFlag])
```

eleTag (int)	unique element object tag
iNode jNode (int)	master nodes (-ndm 3 -ndf 6)
sNode (int)	slave node (-ndm 3 -ndf 3)
lNode (int)	Lagrange multiplier node (-ndm 3 -ndf 3)
radius (float)	constant radius of circular beam associated with beam element
crdTransf (int)	unique integer tag associated with previously-defined geometricTransf object
matTag (int)	unique integer tag associated with previously-defined nDMaterial object
gTol (float)	gap tolerance
fTol (float)	force tolerance
cFlag (int)	optional initial contact flag <code>cFlag = 0 &gt;&gt; contact between bodies is initially assumed (DEFAULT)</code> <code>cFlag = 1 &gt;&gt; no contact between bodies is initially assumed</code>

The BeamContact3D element is a three-dimensional beam-to-node contact element which defines a frictional contact interface between a beam element and a separate body. The master nodes (6 DOF) are the endpoints of the beam element, and the slave node (3 DOF) is a node from a second body. The Lagrange multiplier node (3 DOF) is required to enforce the contact condition. Each contact element should have a unique Lagrange multiplier node. The Lagrange multiplier node should not be fixed, otherwise the contact condition will not work.

---

**Note:**

1. The BeamContact3D element has been written to work exclusively with the ContactMaterial3D nDMaterial object.
2. The valid recorder queries for this element are:
  1. force - returns the contact force acting on the slave node in vector form.
  2. frictionforce - returns the frictional force acting on the slave node in vector form.
  3. forcescalar - returns the scalar magnitudes of the single normal and two tangential contact forces.
  4. masterforce - returns the reactions (forces only) acting on the master nodes.
  5. mastermoment - returns the reactions (moments only) acting on the master nodes.
  6. masterreaction - returns the full reactions (forces and moments) acting on the master nodes.
7. The BeamContact3D elements are set to consider frictional behavior as a default, but the frictional state of the BeamContact3D element can be changed from the input file using the setParameter command. When updating, value of 0 corresponds to the frictionless condition, and a value of 1 signifies the inclusion of friction. An example command for this update procedure is provided below
3. The BeamContact3D element works well in static and pseudo-static analysis situations.
4. In transient analysis, the presence of the contact constraints can effect the stability of commonly-used time integration methods in the HHT or Newmark family (e.g., Laursen, 2002). For this reason, use of alternative time-integration methods which numerically damp spurious high frequency behavior may be required. The

TRBDF2 integrator is an effective method for this purpose. The Newmark integrator can also be effective with proper selection of the gamma and beta coefficients. The trapezoidal rule, i.e., Newmark with gamma = 0.5 and beta = 0.25, is particularly prone to instability related to the contact constraints and is not recommended.

---

**See also:**

Notes

**BeamEndContact3D**

This command is used to construct a BeamEndContact3D element object.

**element** ('BeamEndContact3D', eleTag, iNode, jNode, sNode, lNode, radius, gTol, fTol[, cFlag])

eleTag (int)	unique element object tag
iNode (int)	master node from the beam (-ndm 3 -ndf 6)
jNode (int)	the remaining node on the beam element with iNode (-ndm 3 -ndf 6)
sNode (int)	slave node (-ndm 3 -ndf 3)
lNode (int)	Lagrange multiplier node (-ndm 3 -ndf 3)
radius (float)	radius of circular beam associated with beam element
gTol (float)	gap tolerance
fTol (float)	force tolerance
cFlag (float)	optional initial contact flag cFlag = 0 >> contact between bodies is initially assumed (DEFAULT) cFlag1 = 1 >> no contact between bodies is initially assumed

The BeamEndContact3D element is a node-to-surface contact element which defines a normal contact interface between the end of a beam element and a separate body. The first master node (\$iNode) is the beam node which is at the end of the beam (i.e. only connected to a single beam element), the second node (\$jNode) is the remaining node on the beam element in question. The slave node is a node from a second body. The Lagrange multiplier node is required to enforce the contact condition. This node should not be shared with any other element in the domain, and should be created with the same number of DOF as the slave node.

The BeamEndContact3D element enforces a contact condition between a fictitious circular plane associated with a beam element and a node from a second body. The normal direction of the contact plane coincides with the endpoint tangent of the beam element at the master beam node (\$iNode). The extents of this circular plane are defined by the radius input parameter. The master beam node can only come into contact with a slave node which is within the extents of the contact plane. There is a lag step associated with changing between the ‘in contact’ and ‘not in contact’ conditions.

This element was developed for use in establishing a contact condition for the tip of a pile modeled as using beam elements and the underlying soil elements in three-dimensional analysis.

---

**Note:**

1. The BeamEndContact3D element does not use a material object.
2. The valid recorder queries for this element are:
  1. force - returns the contact force acting on the slave node in vector form.
  2. masterforce - returns the reactions (forces and moments) acting on the master node.
  3. The BeamEndContact3D element works well in static and pseudo-static analysis situations.
3. In transient analysis, the presence of the contact constraints can effect the stability of commonly-used time integration methods in the HHT or Newmark family (e.g., Laursen, 2002). For this reason, use of alternative time-integration methods which numerically damp spurious high frequency behavior may be required. The TRBDF2 integrator is an effective method for this purpose. The Newmark integrator can also be effective with proper selection of the gamma and beta coefficients. The trapezoidal rule, i.e., Newmark with gamma = 0.5 and beta = 0.25, is particularly prone to instability related to the contact constraints and is not recommended.

**See also:**[Notes](#)**Cable Elements**

1. [\*CatenaryCableElement\*](#)

**CatenaryCableElement**

This command is used to construct a catenary cable element object.

```
element ('CatenaryCable', eleTag, iNode, jNode, weight, E, A, L0, alpha, temperature_change, rho, errorTol, Nsubsteps, massType)
```

eleTag ( <a href="#">int</a> )	unique element object tag
iNode jNode ( <a href="#">int</a> )	end nodes (3 dof per node)
E ( <a href="#">float</a> )	elastic modulus of the cable material
A ( <a href="#">float</a> )	cross-sectional area of element
L0 ( <a href="#">float</a> )	unstretched length of the cable
alpha ( <a href="#">float</a> )	coefficient of thermal expansion
temperature_change ( <a href="#">float</a> )	temperature change for the element
rho ( <a href="#">float</a> )	mass per unit length
errortol ( <a href="#">float</a> )	allowed tolerance for within-element equilibrium (Newton-Raphson iterations)
Nsubsteps ( <a href="#">int</a> )	number of within-element substeps into which equilibrium iterations are subdivided (not number of steps to convergence)
massType ( <a href="#">int</a> )	Mass matrix model to use (massType = 0 lumped mass matrix, massType = 1 rigid-body mass matrix (in development))

This cable is a flexibility-based formulation of the catenary cable. An iterative scheme is used internally to compute equilibrium. At each iteration, node i is considered fixed while node j is free. End-forces are applied at node-j and its displacements computed. Corrections to these forces are applied iteratively using a Newton-Raphson scheme (with optional sub-stepping via \$Nsubsteps) until nodal displacements are within the provided tolerance (\$errortol). When convergence is reached, a stiffness matrix is computed by inversion of the flexibility matrix and rigid-body mode injection.

---

**Note:**

1. The stiffness of the cable comes from the large-deformation interaction between loading and cable shape. Therefore, all cables must have distributed forces applied to them. See example. Should not work for only nodal forces.
  2. Valid queries to the CatenaryCable element when creating an ElementalRecorder object correspond to ‘forces’, which output the end-forces of the element in global coordinates (3 for each node).
  3. Only the lumped-mass formulation is currently available.
  4. The element does up 100 internal iterations. If convergence is not achieved, will result in error and some diagnostic information is printed out.
- 

**See also:**

Notes

## PFEM Elements

1. [PFEMElementBubble](#)
2. [PFEMElementCompressible](#)

### PFEMElementBubble

**element ('PFEMElementBubble',eleTag,nd1,nd2,nd3[,nd4]rho,mu,b1,b2[,b3],[thickness,kappa])**

Create a PFEM Bubble element, which is a fluid element for FSI analysis.

eleTag (int)	tag of the element
nd1 (int)	tag of node 1
nd2 (int)	tag of node 2
nd3 (int)	tag of node 3
nd4 (int)	tag of node 4 (required for 3D)
rho (float)	fluid density
mu (float)	fluid viscosity
b1 (float)	body body acceleration in x direction
b2 (float)	body body acceleration in y direction
b3 (float)	body body acceleration in z direction (required for 3D)
thickness (float)	element thickness (required for 2D)
kappa (float)	fluid bulk modulus (optional)

### PFEMElementCompressible

**element ('PFEMElementCompressible', eleTag, nd1, nd2, nd3, nd4, rho, mu, b1, b2[, thickness, kappa])**

Create a PFEM compressible element, which is a fluid element for FSI analysis.

<code>eleTag (int)</code>	tag of the element
<code>nd1 (int)</code>	tag of node 1
<code>nd2 (int)</code>	tag of node 2
<code>nd3 (int)</code>	tag of node 3
<code>nd4 (int)</code>	tag of node 4 (middle node)
<code>rho (float)</code>	fluid density
<code>mu (float)</code>	fluid viscosity
<code>b1 (float)</code>	body body acceleration in x direction
<code>b2 (float)</code>	body body acceleration in y direction
<code>thickness (float)</code>	element thickness (optional)
<code>kappa (float)</code>	fluid bulk modulus (optional)

**Misc.**

1. [SurfaceLoad Element](#)
2. [VS3D4](#)
3. [AC3D8](#)
4. [ASI3D8](#)
5. [AV3D4](#)

**SurfaceLoad Element**

This command is used to construct a SurfaceLoad element object.

**element** ('SurfaceLoad', *eleTag*, \**eleNodes*, *p*)

<code>eleTag (int)</code>	unique element object tag
<code>eleNodes (list (int))</code>	the four nodes defining the element, input in counterclockwise order (-ndm 3 -ndf 3)
<code>p (float)</code>	applied pressure loading normal to the surface, outward is positive, inward is negative

The SurfaceLoad element is a four-node element which can be used to apply surface pressure loading to 3D brick elements. The SurfaceLoad element applies energetically-conjugate forces corresponding to the input scalar pressure to the nodes associated with the element. As these nodes are shared with a 3D brick element, the appropriate nodal loads are therefore applied to the brick.

**Note:**

1. There are no valid ElementalRecorder queries for the SurfaceLoad element. Its sole purpose is to apply nodal forces to the adjacent brick element.
2. The pressure loading from the SurfaceLoad element can be applied in a load pattern. See the analysis example below.

**See also:**

[Notes](#)

## VS3D4

This command is used to construct a four-node 3D viscous-spring boundary quad element object based on a bilinear isoparametric formulation.

**element** ('VS3D4', eleTag, \*eleNodes, E, G, rho, R, alphaN, alphaT)

eleTag (int)	unique element object tag
eleNodes (list (int))	4 end nodes
E (float)	Young's Modulus of element material
G (float)	Shear Modulus of element material
rho (float)	Mass Density of element material
R (float)	distance from the scattered wave source to the boundary
alphaN (float)	correction parameter in the normal direction
alphaT (float)	correction parameter in the tangential direction

---

**Note:** Reference: Liu J, Du Y, Du X, et al. 3D viscous-spring artificial boundary in time domain. Earthquake Engineering and Engineering Vibration, 2006, 5(1):93-102

---

### See also:

[Notes](#)

## AC3D8

This command is used to construct an eight-node 3D brick acoustic element object based on a trilinear isoparametric formulation.

**element** ('AC3D8', eleTag, \*eleNodes, matTag)

eleTag (int)	unique element object tag
eleNodes (list (int))	8 end nodes
matTag (int)	Material Tag of previously defined nD material

---

**Note:** Reference: ABAQUS theory manual. (2.9.1 Coupled acoustic-structural medium analysis)

---

### See also:

[Notes](#)

## ASI3D8

This command is used to construct an eight-node zero-thickness 3D brick acoustic-structure interface element object based on a bilinear isoparametric formulation. The nodes in the acoustic domain share the same coordinates with the nodes in the solid domain.

---

**element** ('ASI3D8', *eleTag*, \**eleNodes1*, \**eleNodes2*)

<i>eleTag</i> (int)	unique element object tag
* <i>eleNodes`</i> (list (int))	four nodes defining structure domain of element boundaries
* <i>eleNodes2</i> (list (int))	four nodes defining acoustic domain of element boundaries

---

**Note:** Reference: ABAQUS theory manual. (2.9.1 Coupled acoustic-structural medium analysis)

---

**See also:**

Notes

## AV3D4

This command is used to construct a four-node 3D acoustic viscous boundary quad element object based on a bilinear isoparametric formulation.

**element** ('AV3D4', *eleTag*, \**eleNodes*, *matTag*)

<i>eleTag</i> (int)	unique element object tag
<i>eleNodes</i> (list (int))	4 end nodes
<i>matTag</i> (int)	Material Tag of previously defined nD material

**See also:**

Notes

## 1.4.3 node command

**node** (*nodeTag*, \**crds*, '-*ndf*', *ndf*, '-*mass*', \**mass*, '-*disp*', \**disp*, '-*vel*', \**vel*, '-*accel*', \**accel*)

Create a OpenSees node.

<i>nodeTag</i> (int)	node tag.
<i>crds</i> (list (float))	nodal coordinates.
<i>ndf</i> (float)	nodal ndf. (optional)
<i>mass</i> (list (float))	nodal mass. (optional)
<i>vel</i> (list (float))	nodal velocities. (optional)
<i>accel</i> (list (float))	nodal accelerations. (optional)

## 1.4.4 sp constraint commands

Create constraints for a single dof of a node.

1. *fix command*
2. *fixX command*
3. *fixY command*
4. *fixZ command*

## fix command

**fix** (*nodeTag*, \**constrValues*)

Create a homogeneous SP constraint.

nodeTag (int)	tag of node to be constrained
constrValues (list (int))	a list of constraint values (0 or 1), must be preceded with *. <ul style="list-style-type: none"><li>• 0 free</li><li>• 1 fixed</li></ul>

For example,

```
# fully fixed
vals = [1, 1, 1]
fix(nodeTag, *vals)
```

## fixX command

**fixX** (*x*, \**constrValues*, '-*tol*', *tol*=*1e-10*)

Create homogeneous SP constraints.

x (float)	x-coordinate of nodes to be constrained
constrValues (list (int))	a list of constraint values (0 or 1), must be preceded with *. <ul style="list-style-type: none"><li>• 0 free</li><li>• 1 fixed</li></ul>
tol (float)	user-defined tolerance (optional)

## fixY command

**fixY** (*y*, \**constrValues*, '-*tol*', *tol*=*1e-10*)

Create homogeneous SP constraints.

y (float)	y-coordinate of nodes to be constrained
constrValues (list (int))	a list of constraint values (0 or 1), must be preceded with *. <ul style="list-style-type: none"><li>• 0 free</li><li>• 1 fixed</li></ul>
tol (float)	user-defined tolerance (optional)

## fixZ command

**fixZ** (*z*, \**constrValues*, '-*tol*', *tol*=*1e-10*)

Create homogeneous SP constraints.

<code>z (float)</code>	z-coordinate of nodes to be constrained
<code>constrValues (list (int))</code>	a list of constraint values (0 or 1), must be preceded with *. <ul style="list-style-type: none"><li>• 0 free</li><li>• 1 fixed</li></ul>
<code>tol (float)</code>	user-defined tolerance (optional)

## 1.4.5 mp constraint commands

Create constraints for multiple dofs of multiple nodes.

1. *equalDOF command*
2. *equalDOF\_Mixed command*
3. *rigidDiaphragm command*
4. *rigidLink command*

### equalDOF command

**equalDOF** (*rNodeTag*, *cNodeTag*, *\*dofs*)

Create a multi-point constraint between nodes.

<code>rNodeTag (int)</code>	integer tag identifying the retained, or master node.
<code>cNodeTag (int)</code>	integer tag identifying the constrained, or slave node.
<code>dofs (list (int))</code>	nodal degrees-of-freedom that are constrained at the cNode to be the same as those at the rNode Valid range is from 1 through ndf, the number of nodal degrees-of-freedom.

### equalDOF\_Mixed command

**equalDOF\_Mixed** (*rNodeTag*, *cNodeTag*, *numDOF*, *\*rcdofs*)

Create a multi-point constraint between nodes.

<code>rNodeTag (int)</code>	integer tag identifying the retained, or master node.
<code>cNodeTag (int)</code>	integer tag identifying the constrained, or slave node.
<code>numDOF (int)</code>	number of dofs to be constrained
<code>rcdofs (list (int))</code>	nodal degrees-of-freedom that are constrained at the cNode to be the same as those at the rNode Valid range is from 1 through ndf, the number of nodal degrees-of-freedom. <code>rcdofs = [rdof1, cdof1, rdof2, cdof2, ...]</code>

## rigidDiaphragm command

**rigidDiaphragm** (*perpDirn*, *rNodeTag*, \**cNodeTags*)

Create a multi-point constraint between nodes. These objects will constraint certain degrees-of-freedom at the listed slave nodes to move as if in a rigid plane with the master node. To enforce this constraint, Transformation constraint is recommended.

<i>perpDirn</i> (int)	direction perpendicular to the rigid plane (i.e. direction 3 corresponds to the 1-2 plane)
<i>rNodeTag</i> (int)	integer tag identifying the master node
<i>cNodeTags</i> (list int)	integer tags identifying the slave nodes

## rigidLink command

**rigidLink** (*type*, *rNodeTag*, *cNodeTag*)

Create a multi-point constraint between nodes.

<i>type</i> (str)	string-based argument for rigid-link type: • 'bar': only the translational degree-of-freedom will be constrained to be exactly the same as those at the master node • 'beam': both the translational and rotational degrees of freedom are constrained.
<i>rNodeTag</i> (int)	integer tag identifying the master node
<i>cNodeTag</i> (int)	integer tag identifying the slave node

## 1.4.6 timeSeries commands

**timeSeries** (*tsType*, *tsTag*, \**tsArgs*)

This command is used to construct a TimeSeries object which represents the relationship between the time in the domain,  $t$ , and the load factor applied to the loads,  $\lambda$ , in the load pattern with which the TimeSeries object is associated, i.e.  $\lambda = F(t)$ .

<i>tsType</i> (str)	time series type.
<i>tsTag</i> (int)	time series tag.
<i>tsArgs</i> (list)	a list of time series arguments

The following contain information about available *tsType*:

1. *Constant TimeSeries*
2. *Linear TimeSeries*
3. *Trigonometric TimeSeries*
4. *Triangular TimeSeries*
5. *Rectangular TimeSeries*
6. *Pulse TimeSeries*
7. *Path TimeSeries*

## Constant TimeSeries

**timeSeries** ('Constant', tag, '-factor', factor=1.0)

This command is used to construct a TimeSeries object in which the load factor applied remains constant and is independent of the time in the domain, i.e.  $\lambda = f(t) = C$ .

tag (int)	unique tag among TimeSeries objects.
factor (float)	the load factor applied (optional)

## Linear TimeSeries

**timeSeries** ('Linear', tag, '-factor', factor=1.0)

This command is used to construct a TimeSeries object in which the load factor applied is linearly proportional to the time in the domain, i.e.

$$\lambda = f(t) = cFactor * t.$$

tag (int)	unique tag among TimeSeries objects.
factor (float)	Linear factor. (optional)

## Trigonometric TimeSeries

**timeSeries** ('Trig', tag, tStart, tEnd, period, '-factor', factor=1.0, '-shift', shift=0.0, '-zeroShift', zeroShift=0.0)

This command is used to construct a TimeSeries object in which the load factor is some trigonometric function of the time in the domain

$$\lambda = f(t) = \begin{cases} cFactor * \sin(\frac{2.0\pi(t-tStart)}{period} + \phi), & tStart \leq t \leq tEnd \\ 0.0, & \text{otherwise} \end{cases}$$

$$\phi = shift - \frac{period}{2.0\pi} * \arcsin(\frac{zeroShift}{cFactor})$$

tag (int)	unique tag among TimeSeries objects.
tStart (float)	Starting time of non-zero load factor.
tEnd (float)	Ending time of non-zero load factor.
period (float)	Characteristic period of sine wave.
shift (float)	Phase shift in radians. (optional)
factor (float)	Load factor. (optional)
zeroShift (float)	Zero shift. (optional)

## Triangular TimeSeries

**timeSeries** ('Triangle', tag, tStart, tEnd, period, '-factor', factor=1.0, '-shift', shift=0.0, '-zeroShift', zeroShift=0.0)

This command is used to construct a TimeSeries object in which the load factor is some triangular function of the time in the domain.

$$\lambda = f(t) = \begin{cases} slope * k * period + zeroShift, & k < 0.25 \\ cFactor - slope * (k - 0.25) * period + zeroShift, & k < 0.75 \\ -cFactor + slope * (k - 0.75) * period + zeroShift, & k < 1.0 \\ 0.0, & \text{otherwise} \end{cases}$$

$$\begin{aligned}slope &= \frac{cFactor}{period/4} \\ k &= \frac{t + \phi - tStart}{period} - \text{floor}\left(\frac{t + \phi - tStart}{period}\right) \\ \phi &= shift - \frac{zeroShift}{slope}\end{aligned}$$

tag (int)	unique tag among TimeSeries objects.
tStart (float)	Starting time of non-zero load factor.
tEnd (float)	Ending time of non-zero load factor.
period (float)	Characteristic period of sine wave.
shift (float)	Phase shift in radians. (optional)
factor (float)	Load factor. (optional)
zeroShift (float)	Zero shift. (optional)

## Rectangular TimeSeries

**timeSeries ('Rectangular', tag, tStart, tEnd, '-factor', factor=1.0)**

This command is used to construct a TimeSeries object in which the load factor is constant for a specified period and 0 otherwise, i.e.

$$\lambda = f(t) = \begin{cases} cFactor, & tStart \leq t \leq tEnd \\ 0.0, & \text{otherwise} \end{cases}$$

tag (int)	unique tag among TimeSeries objects.
tStart (float)	Starting time of non-zero load factor.
tEnd (float)	Ending time of non-zero load factor.
factor (float)	Load factor. (optional)

## Pulse TimeSeries

**timeSeries ('Pulse', tag, tStart, tEnd, period, '-width', width=0.5, '-shift', shift=0.0, '-factor', factor=1.0, '-zeroShift', zeroShift=0.0)**

This command is used to construct a TimeSeries object in which the load factor is some pulse function of the time in the domain.

$$\lambda = f(t) = \begin{cases} cFactor + zeroShift, & k < width \\ zeroShift, & k < 1 \\ 0.0, & \text{otherwise} \end{cases}$$

$$k = \frac{t + shift - tStart}{period} - \text{floor}\left(\frac{t + shift - tStart}{period}\right)$$

tag (int)	unique tag among TimeSeries objects.
tStart (float)	Starting time of non-zero load factor.
tEnd (float)	Ending time of non-zero load factor.
period (float)	Characteristic period of pulse.
width (float)	Pulse width as a fraction of the period. (optional)
shift (float)	Phase shift in seconds. (optional)
factor (float)	Load factor. (optional)
zeroShift (float)	Zero shift. (optional)

## Path TimeSeries

```
timeSeries ('Path', tag, '-dt', dt=0.0, '-values', *values, '-time', *time, '-filepath', filePath=", '-fileTime', fileTime=", '-factor', factor=1.0, '-startTime', startTime=0.0, '-useLast', '-prependZero')
```

The relationship between load factor and time is input by the user as a series of discrete points in the 2d space (load factor, time). The input points can come from a file or from a list in the script. When the time specified does not match any of the input points, linear interpolation is used between points. There are many ways to specify the load path, for example, the load factors set with values or filePath, and the time set with dt, time, or fileTime.

tag (int)	unique tag among TimeSeries objects.
dt (float)	Time interval between specified points. (optional)
values (list (float))	Load factor values in a (list). (optional)
time (list (float))	Time values in a (list). (optional)
filePath (str)	File containing the load factors values. (optional)
fileTime (str)	File containing the time values for corresponding load factors. (optional)
factor (float)	A factor to multiply load factors by. (optional)
startTime (float)	Provide a start time for provided load factors. (optional)
'-useLast' (str)	Use last value after the end of the series. (optional)
'-prependZero' (str)	Prepend a zero value to the series of load factors. (optional)

- Linear interpolation between points.
- If the specified time is beyond last point (AND WATCH FOR NUMERICAL ROUNDOFF), 0.0 is returned. Specify '-useLast' to use the last data point instead of 0.0.
- The transient integration methods in OpenSees assume zero initial conditions. So it is important that any timeSeries that is being used in a transient analysis starts from zero (first data point in the timeSeries = 0.0). To guarantee that this is the case the optional parameter '-prependZero' can be specified to prepend a zero value to the provided TimeSeries.

## 1.4.7 pattern commands

```
pattern (patternType, patternTag, *patternArgs)
```

The pattern command is used to construct a LoadPattern and add it to the Domain. Each LoadPattern in OpenSees has a TimeSeries associated with it. In addition it may contain ElementLoads, NodalLoads and SinglePointConstraints. Some of these SinglePoint constraints may be associated with GroundMotions.

patternType (str)	pattern type.
patternTag (int)	pattern tag.
patternArgs (list)	a list of pattern arguments

The following contain information about available patternType:

1. Plain Pattern
2. UniformExcitation Pattern
3. Multi-Support Excitation Pattern

## Plain Pattern

**pattern** ('Plain', patternTag, tsTag, '-fact', factor)

This command allows the user to construct a LoadPattern object. Each plain load pattern is associated with a TimeSeries object and can contain multiple NodalLoads, ElementalLoads and SP\_Constraint objects. The command to generate LoadPattern object contains in { } the commands to generate all the loads and the single-point constraints in the pattern. To construct a load pattern and populate it, the following command is used:

patternTag (int)	unique tag among load patterns.
tsTag (int)	the tag of the time series to be used in the load pattern
factor (float)	constant factor. (optional)

---

**Note:** the commands below to generate all the loads and sp constraints will be included in last called pattern command.

---

## load command

**load** (nodeTag, \*loadValues)

This command is used to construct a NodalLoad object and add it to the enclosing LoadPattern.

nodeTag (int)	tag of node to which load is applied.
loadValues (list (float))	ndf reference load values.

---

**Note:** The load values are reference loads values. It is the time series that provides the load factor. The load factor times the reference values is the load that is actually applied to the node.

---

## eleLoad command

**eleLoad** ('-ele', \*eleTags, '-range', eleTag1, eleTag2, '-type', '-beamUniform', Wy, Wz=0.0, Wx=0.0, '-beamPoint', Py, Pz=0.0, xL, Px=0.0, '-beamThermal', \*tempPts)

The eleLoad command is used to construct an ElementalLoad object and add it to the enclosing LoadPattern.

eleTags (list (int))	tag of PREVIOUSLY DEFINED element
eleTag1 (int)	element tag
eleTag2 (int)	element tag
Wx (float)	mag of uniformly distributed ref load acting in direction along member length. (optional)
Wy (float)	mag of uniformly distributed ref load acting in local y direction of element
Wz (float)	mag of uniformly distributed ref load acting in local z direction of element. (optional and only for 3D)
Px (float)	mag of ref point load acting in direction along member length. (optional)
Py (float)	mag of ref point load acting in local y direction of element
Pz (float)	mag of ref point load acting in local z direction of element. (optional and only for 3D)
xL (float)	location of point load relative to node I, prescribed as fraction of element length
tempPts (list (float))	temperature points: tempPts = [T1, y1, T2, y2, ..., T9, y9] Each point (T1, y1) define a temperature and location. This command may accept 2,5 or 9 temperature points.

**Note:**

1. The load values are reference loads values, it is the time series that provides the load factor. The load factor times the reference values is the load that is actually applied to the node.
2. At the moment, eleLoads do not work with 3D beam-column elements if Corotational geometric transformation is used.

**sp command****sp** (*nodeTag, dof, \*dofValues*)

This command is used to construct a single-point constraint object and add it to the enclosing LoadPattern.

nodeTag (int)	tag of node to which load is applied.
dof (int)	the degree-of-freedom at the node to which constraint is applied (1 through ndf)
dofValues (list (float))	ndf reference constraint values.

---

**Note:** The dofValue is a reference value, it is the time series that provides the load factor. The load factor times the reference value is the constraint that is actually applied to the node.

---

**UniformExcitation Pattern**

```
pattern ('UniformExcitation', patternTag, dir, '-disp', dispSeriesTag, '-vel', velSeriesTag, '-accel', accelSeriesTag, '-vel0', vel0, '-fact', factor)
```

The UniformExcitation pattern allows the user to apply a uniform excitation to a model acting in a certain direction. The command is as follows:

<code>patternTag (int)</code>	unique tag among load patterns
<code>dir (int)</code>	direction in which ground motion acts 1. corresponds to translation along the global X axis 2. corresponds to translation along the global Y axis 3. corresponds to translation along the global Z axis 4. corresponds to rotation about the global X axis 5. corresponds to rotation about the global Y axis 6. corresponds to rotation about the global Z axis
<code>dispSeriesTag (int)</code>	tag of the TimeSeries series defining the displacement history. (optional)
<code>velSeriesTag (int)</code>	tag of the TimeSeries series defining the velocity history. (optional)
<code>accelSeriesTag (int)</code>	tag of the TimeSeries series defining the acceleration history. (optional)
<code>vel0 (float)</code>	the initial velocity (optional, default=0.0)
<code>factor (float)</code>	constant factor (optional, default=1.0)

---

**Note:**

1. The responses obtained from the nodes for this type of excitation are RELATIVE values, and not the absolute values obtained from a multi-support case.
  2. must set one of the disp, vel or accel time series
- 

## Multi-Support Excitation Pattern

`pattern ('MultipleSupport', patternTag)`

The Multi-Support pattern allows similar or different prescribed ground motions to be input at various supports in the structure. In OpenSees, the prescribed motion is applied using single-point constraints, the single-point constraints taking their constraint value from user created ground motions.

---

**Note:**

1. The results for the responses at the nodes are the ABSOLUTE values, and not relative values as in the case of a UniformExcitation.
  2. The non-homogeneous single point constraints require an appropriate choice of constraint handler.
- 

## Plain Ground Motion

`groundMotion (gmTag, 'Plain', '-disp', dispSeriesTag, '-vel', velSeriesTag, '-accel', accelSeriesTag, '-int', tsInt='Trapezoidal', 'fact', factor=1.0)`

This command is used to construct a plain GroundMotion object. Each GroundMotion object is associated with a number of TimeSeries objects, which define the acceleration, velocity and displacement records for that ground motion. T

---

<code>gmTag (int)</code>	unique tag among ground motions in load pattern
<code>dispSeriesTag (int)</code>	tag of the TimeSeries series defining the displacement history. (optional)
<code>velSeriesTag (int)</code>	tag of the TimeSeries series defining the velocity history. (optional)
<code>accelSeriesTag (int)</code>	tag of the TimeSeries series defining the acceleration history. (optional)
<code>tsInt (str)</code>	'Trapezoidal' or 'Simpson' numerical integration method
<code>factor (float)</code>	constant factor. (optional)

---

**Note:**

1. The displacements are the ones used in the ImposedMotions to set nodal response.
  2. If only the acceleration TimeSeries is provided, numerical integration will be used to determine the velocities and displacements.
  3. For earthquake excitations it is important that the user provide the displacement time history, as the one generated using the trapezoidal method will not provide good results.
  4. Any combination of the acceleration, velocity and displacement time-series can be specified.
- 

**Interpolated Ground Motion****groundMotion** (`gmTag`, '*Interpolated*', \*`gmTags`, '-`fact`', `facts`)

This command is used to construct an interpolated GroundMotion object, where the motion is determined by combining several previously defined ground motions in the load pattern.

<code>gmTag (int)</code>	unique tag among ground motions in load pattern
<code>gmTags (list (int))</code>	the tags of existing ground motions in pattern to be used for interpolation
<code>facts (list (float))</code>	the interpolation factors. (optional)

**Imposed Motion****imposedMotion** (`nodeTag`, `dof`, `gmTag`)

This command is used to construct an ImposedMotionSP constraint which is used to enforce the response of a dof at a node in the model. The response enforced at the node at any give time is obtained from the GroundMotion object associated with the constraint.

<code>nodeTag (int)</code>	tag of node on which constraint is to be placed
<code>dof (int)</code>	dof of enforced response. Valid range is from 1 through ndf at node.
<code>gmTag (int)</code>	pre-defined GroundMotion object tag

**1.4.8 mass command****mass** (`nodeTag`, \*`massValues`)

This command is used to set the mass at a node

<code>nodeTag (int)</code>	integer tag identifying node whose mass is set
<code>massValues (list (float))</code>	ndf nodal mass values corresponding to each DOF

### 1.4.9 region command

```
region (regTag, '-ele', *eles, '-eleOnly', *eles, '-eleRange', startEle, endEle, '-eleOnlyRange', startEle, endEle, '-node', *nodes, '-nodeOnly', *nodes, '-nodeRange', startNode, endNode, '-nodeOnlyRange', startNode, endNode, '-rayleigh', alphaM, betaKinit, betaKcomm)
```

The region command is used to label a group of nodes and elements. This command is also used to assign rayleigh damping parameters to the nodes and elements in this region. The region is specified by either elements or nodes, not both. If elements are defined, the region includes these elements and the all connected nodes, unless the -eleOnly option is used in which case only elements are included. If nodes are specified, the region includes these nodes and all elements of which all nodes are prescribed to be in the region, unless the -nodeOnly option is used in which case only the nodes are included.

regTag (int)	unique integer tag
eles (list (int))	tags of selected elements in domain to be included in region (optional)
nodes (list (int))	tags of selected nodes in domain to be included in region (optional)
startEle (int)	tag for start element (optional)
endEle (int)	tag for end element (optional)
startNode (int)	tag for start node (optional)
endNode (int)	tag for end node (optional)
alphaM (float)	factor applied to elements or nodes mass matrix (optional)
betaK (float)	factor applied to elements current stiffness matrix (optional)
betaKinit (float)	factor applied to elements initial stiffness matrix (optional)
betaKcomm (float)	factor applied to elements committed stiffness matrix (optional)

---

**Note:** The user cannot prescribe the region by BOTH elements and nodes.

---

### 1.4.10 rayleigh command

```
rayleigh (alphaM, betaK, betaKinit, betaKcomm)
```

This command is used to assign damping to all previously-defined elements and nodes. When using rayleigh damping in OpenSees, the damping matrix for an element or node, D is specified as a combination of stiffness and mass-proportional damping matrices:

$$D = \alpha_M * M + \beta_K * K_{curr} + \beta_{Kinit} * K_{init} + \beta_{Kcomm} * K_{commit}$$

alphaM (float)	factor applied to elements or nodes mass matrix
betaK (float)	factor applied to elements current stiffness matrix.
betaKinit (float)	factor applied to elements initial stiffness matrix.
betaKcomm (float)	factor applied to elements committed stiffness matrix.

### 1.4.11 block commands

Create a block of mesh

#### block2D command

```
block2D (numX, numY, startNode, startEle, eleType, *eleArgs, *crds)
```

Create mesh of quadrilateral elements

numX (int)	number of elements in local x directions of the block.
numY (int)	number of elements in local y directions of the block.
startNode (int)	node from which the mesh generation will start.
startEle (int)	element from which the mesh generation will start.
eleType (str)	element type ('quad', 'shell', 'bbarQuad', 'enhancedQuad', or 'SSPquad')
eleArgs (list)	a list of element parameters.
crds (list)	coordinates of the block elements with the format: [1, x1, y1, <z1>, 2, x2, y2, <z2>, 3, x3, y3, <z3>, 4, x4, y4, <z4>, <5>, <x5>, <y5>, <z5>, <6>, <x6>, <y6>, <z6>, <7>, <x7>, <y7>, <z7>, <8>, <x8>, <y8>, <z8>, <9>, <x9>, <y9>, <z9>] <> means optional

### block3D command

**block3D** (*numX*, *numY*, *numZ*, *startNode*, *startEle*, *eleType*, \**eleArgs*, \**crds*)  
 Create mesh of quadrilateral elements

numX	number of elements in local x directions of the block.
(int)	
numY	number of elements in local y directions of the block.
(int)	
numZ	number of elements in local z directions of the block.
(int)	
startNode	node from which the mesh generation will start.
(int)	
startElement	element from which the mesh generation will start.
(int)	
elementType	element type ('stdBrick', 'bbarBrick', 'Brick20N')
(str)	
elementList	a list of element parameters.
(list)	
crds	coordinates of the block elements with the format:
(list)	[1, x1, y1, z1, 2, x2, y2, z2, 3, x3, y3, z3, 4, x4, y4, z4, 5, x5, y5, z5, 6, x6, y6, z6, 7, x7, y7, z7, 8, x8, y8, z8, 9, x9, y9, z9, <10>, <x10>, <y10>, <z10>, <11>, <x11>, <y11>, <z11>, <12>, <x12>, <y12>, <z12>, <13>, <x13>, <y13>, <z13>, <14>, <x14>, <y14>, <z14>, <15>, <x15>, <y15>, <z15>, <16>, <x16>, <y16>, <z16>, <17>, <x17>, <y17>, <z17>, <18>, <x18>, <y18>, <z18>, <19>, <x19>, <y19>, <z19>, <20>, <x20>, <y20>, <z20>, <21>, <x21>, <y21>, <z21>, <22>, <x22>, <y22>, <z22>, <23>, <x23>, <y23>, <z23>, <24>, <x24>, <y24>, <z24>, <25>, <x25>, <y25>, <z25>, <26>, <x26>, <y26>, <z26>, <27>, <x27>, <y27>, <z27>] <> means optional

### 1.4.12 beamIntegration commands

**beamIntegration** (*type*, *tag*, \**args*)

A wide range of numerical integration options are available in OpenSees to represent distributed plasticity or non-prismatic section details in Beam-Column Elements, i.e., across the entire element domain [0, L].

Following are beamIntegration types available in the OpenSees:

Integration Methods for Distributed Plasticity. Distributed plasticity methods permit yielding at any integration point

along the element length.

1. [Lobatto](#)
2. [Legendre](#)
3. [NewtonCotes](#)
4. [Radau](#)
5. [Trapezoidal](#)
6. [CompositeSimpson](#)
7. [UserDefined](#)
8. [FixedLocation](#)
9. [LowOrder](#)
10. [MidDistance](#)

## Lobatto

**beamIntegration ('Lobatto', tag, secTag, N)**

Create a Gauss-Lobatto beamIntegration object. Gauss-Lobatto integration is the most common approach for evaluating the response of [forceBeamColumn](#) (Neuenhofer and Filippou 1997) because it places an integration point at each end of the element, where bending moments are largest in the absence of interior element loads.

tag (int)	tag of the beam integration.
secTag (int)	A previous-defined section object.
N (int)	Number of integration points along the element.

## Legendre

**beamIntegration ('Legendre', tag, secTag, N)**

Create a Gauss-Legendre beamIntegration object. Gauss-Legendre integration is more accurate than Gauss-Lobatto; however, it is not common in force-based elements because there are no integration points at the element ends.

Places N Gauss-Legendre integration points along the element. The location and weight of each integration point are tabulated in references on numerical analysis. The force deformation response at each integration point is defined by the section. The order of accuracy for Gauss-Legendre integration is 2N-1.

Arguments and examples see [Lobatto](#).

## NewtonCotes

**beamIntegration ('NewtonCotes', tag, secTag, N)**

Create a Newton-Cotes beamIntegration object. Newton-Cotes places integration points uniformly along the element, including a point at each end of the element.

Places N Newton-Cotes integration points along the element. The weights for the uniformly spaced integration points are tabulated in references on numerical analysis. The force deformation response at each integration point is defined by the section. The order of accuracy for Gauss-Radau integration is N-1.

Arguments and examples see [Lobatto](#).

## Radau

**beamIntegration** ('Radau', tag, secTag, N)

Create a Gauss-Radau beamIntegration object. Gauss-Radau integration is not common in force-based elements because it places an integration point at only one end of the element; however, it forms the basis for optimal plastic hinge integration methods.

Places N Gauss-Radau integration points along the element with a point constrained to be at ndI. The location and weight of each integration point are tabulated in references on numerical analysis. The force-deformation response at each integration point is defined by the section. The order of accuracy for Gauss-Radau integration is  $2N-2$ .

Arguments and examples see [Lobatto](#).

## Trapezoidal

**beamIntegration** ('Trapezoidal', tag, secTag, N)

Create a Trapezoidal beamIntegration object.

Arguments and examples see [Lobatto](#).

## CompositeSimpson

**beamIntegration** ('CompositeSimpson', tag, secTag, N)

Create a CompositeSimpson beamIntegration object.

Arguments and examples see [Lobatto](#).

## UserDefined

**beamIntegration** ('UserDefined', tag, N, \*secTags, \*locs, \*wts)

Create a UserDefined beamIntegration object. This option allows user-specified locations and weights of the integration points.

tag (int)	tag of the beam integration
N (int)	number of integration points along the element.
secTags (list (int))	A list previous-defined section objects.
locs (list (float))	Locations of integration points along the element.
wts (list (float))	weights of integration points.

```
locs = [0.1, 0.3, 0.5, 0.7, 0.9]
wts = [0.2, 0.15, 0.3, 0.15, 0.2]
secs = [1, 2, 2, 2, 1]
beamIntegration('UserDefined', 1, len(secs), *secs, *locs, *wts)
```

Places N integration points along the element, which are defined in `locs` on the natural domain [0, 1]. The weight of each integration point is defined in the `wts` also on the [0, 1] domain. The force-deformation response at each integration point is defined by the `secs`. The `locs`, `wts`, and `secs` should be of length N. In general, there is no accuracy for this approach to numerical integration.

## FixedLocation

**beamIntegration ('FixedLocation', tag, N, \*secTags, \*locs)**

Create a FixedLocation beamIntegration object. This option allows user-specified locations of the integration points. The associated integration weights are computed by the method of undetermined coefficients (Vandermonde system)

$$\sum_{i=1}^N x_i^{j-1} w_i = \int_0^1 x^{j-1} dx = \frac{1}{j}, \quad (j = 1, \dots, N)$$

Note that *NewtonCotes* integration is recovered when the integration point locations are equally spaced.

tag (int)	tag of the beam integration
N (int)	number of integration points along the element.
secTags (list (int))	A list previous-defined section objects.
locs (list (float))	Locations of integration points along the element.

Places N integration points along the element, whose locations are defined in `locs`. on the natural domain [0, 1]. The force-deformation response at each integration point is defined by the `secs`. Both the `locs` and `secs` should be of length N. The order of accuracy for Fixed Location integration is N-1.

## LowOrder

**beamIntegration ('LowOrder', tag, N, \*secTags, \*locs, \*wts)**

Create a LowOrder beamIntegration object. This option is a generalization of the *FixedLocation* and *UserDefined* integration approaches and is useful for moving load analysis (Kidarsa, Scott and Higgins 2008). The locations of the integration points are user defined, while a selected number of weights are specified and the remaining weights are computed by the method of undetermined coefficients.

$$\sum_{i=1}^{N_f} x_{fi}^{j-1} w_{fi} = \frac{1}{j} - \sum_{i=1}^{N_c} x_{ci}^{j-1} w_{ci}$$

Note that *FixedLocation* integration is recovered when `Nc` is zero.

tag (int)	tag of the beam integration
N (int)	number of integration points along the element.
secTags (list (int))	A list previous-defined section objects.
locs (list (float))	Locations of integration points along the element.
wts (list (float))	weights of integration points.

```
locs = [0.0, 0.2, 0.5, 0.8, 1.0]
wts = [0.2, 0.2]
secs = [1, 2, 2, 2, 1]
beamIntegration('LowOrder', 1, len(secs), *secs, *locs, *wts)
```

Places N integration points along the element, which are defined in `locs`. on the natural domain [0, 1]. The force-deformation response at each integration point is defined by the `secs`. Both the `locs` and `secs` should be of length N. The `wts` at user-selected integration points are specified on [0, 1], which can be of length `Nc` equals 0 up to N. These specified weights are assigned to the first `Nc` entries in the `locs` and `secs`, respectively. The order of accuracy for Low Order integration is N-Nc-1.

---

**Note:**  $N_c$  is determined from the length of the `wts` list. Accordingly, `FixedLocation` integration is recovered when `wts` is an empty list and `UserDefined` integration is recovered when the `wts` and `locs` lists are of equal length.

---

## MidDistance

**beamIntegration ('MidDistance', tag, N, \*secTags, \*locs)**

Create a MidDistance beamIntegration object. This option allows user-specified locations of the integration points. The associated integration weights are determined from the midpoints between adjacent integration point locations.  $w_i = (x_{i+1} - x_{i-1})/2$  for  $i = 2 \dots N-1$ ,  $w_1 = (x_1 + x_2)/2$ , and  $w_N = 1 - (x_{N-1} + x_N)/2$ .

<code>tag (int)</code>	tag of the beam integration
<code>N (int)</code>	number of integration points along the element.
<code>secTags (list (int))</code>	A list previous-defined section objects.
<code>locs (list (float))</code>	Locations of integration points along the element.

```
locs = [0.0, 0.2, 0.5, 0.8, 1.0]
secs = [1,2,2,2,1]
beamIntegration('MidDistance', 1, len(secs), *secs, *locs)
```

Places `N` integration points along the element, whose locations are defined in `locs` on the natural domain [0, 1]. The force-deformation response at each integration point is defined by the `secs`. Both the `locs` and `secs` should be of length `N`. This integration rule can only integrate constant functions exactly since the sum of the integration weights is one.

For the `locs` shown above, the associated integration weights will be [0.15, 0.2, 0.3, 0.2, 0.15].

Plastic Hinge Integration Methods. Plastic hinge integration methods confine material yielding to regions of the element of specified length while the remainder of the element is linear elastic. A summary of plastic hinge integration methods is found in (Scott and Fenves 2006).

1. [UserHinge](#)
2. [HingeMidpoint](#)
3. [HingeRadau](#)
4. [HingeRadauTwo](#)
5. [HingeEndpoint](#)

## UserHinge

**beamIntegration ('UserHinge', tag, secE, npL, \*secsL, \*locsL, \*wtsL, npR, \*secsR, \*locsR, \*wtsR)**

Create a UserHinge beamIntegration object.

<code>tag (int)</code>	tag of the beam integration
<code>secE (int)</code>	A previous-defined section objects for non-hinge area.
<code>npL (int)</code>	number of integration points along the left hinge.
<code>secsL (list (int))</code>	A list of previous-defined section objects for left hinge area.
<code>locsL (list (float))</code>	A list of locations of integration points for left hinge area.
<code>wtsL (list (float))</code>	A list of weights of integration points for left hinge area.
<code>npR (int)</code>	number of integration points along the right hinge.
<code>secsR (list (int))</code>	A list of previous-defined section objects for right hinge area.
<code>locsR (list (float))</code>	A list of locations of integration points for right hinge area.
<code>wtsR (list (float))</code>	A list of weights of integration points for right hinge area.

```

tag = 1
secE = 5

npL = 2
secsL = [1,2]
locsL = [0.1,0.2]
wtsL = [0.5,0.5]

npR = 2
secsR = [3,4]
locsR = [0.8,0.9]
wtsR = [0.5,0.5]

beamIntegration ('UserHinge',tag,secE,npL,*secsL,*locsL,*wtsL,npR,*secsR,*locsR,
                ↵*wtsR)

```

## HingeMidpoint

`beamIntegration ('HingeMidpoint', tag, secI, lpI, secJ, lpJ, secE)`

Create a HingeMidpoint beamIntegration object. Midpoint integration over each hinge region is the most accurate one-point integration rule; however, it does not place integration points at the element ends and there is a small integration error for linear curvature distributions along the element.

<code>tag (int)</code>	tag of the beam integration.
<code>secI (int)</code>	A previous-defined section object for hinge at I.
<code>lpI (float)</code>	The plastic hinge length at I.
<code>secJ (int)</code>	A previous-defined section object for hinge at J.
<code>lpJ (float)</code>	The plastic hinge length at J.
<code>secE (int)</code>	A previous-defined section object for the element interior.

The plastic hinge length at end I (J) is equal to `lpI` (`lpJ`) and the associated force deformation response is defined by the `secI` (`secJ`). The force deformation response of the element interior is defined by the `secE`. Typically, the interior section is linear-elastic, but this is not necessary.

```

lpI = 0.1
lpJ = 0.2
beamIntegration ('HingeMidpoint',secI,lpI,secJ,lpJ,secE)

```

## HingeRadau

**beamIntegration** ('HingeRadau', tag, secI, lpI, secJ, lpJ, secE)

Create a HingeRadau beamIntegration object. Two-point Gauss-Radau integration over each hinge region places an integration point at the element ends and at 2/3 the hinge length inside the element. This approach represents linear curvature distributions exactly; however, the characteristic length for softening plastic hinges is not equal to the assumed plastic hinge length.

Arguments and examples see [HingeMidpoint](#).

## HingeRadauTwo

**beamIntegration** ('HingeRadauTwo', tag, secI, lpI, secJ, lpJ, secE)

Create a HingeRadauTwo beamIntegration object. Modified two-point Gauss-Radau integration over each hinge region places an integration point at the element ends and at 8/3 the hinge length inside the element. This approach represents linear curvature distributions exactly and the characteristic length for softening plastic hinges is equal to the assumed plastic hinge length.

Arguments and examples see [HingeMidpoint](#).

## HingeEndpoint

**beamhingeEndpoint** (tag, secI, lpI, secJ, lpJ, secE)

Create a HingeEndpoint beamIntegration object. Endpoint integration over each hinge region moves the integration points to the element ends; however, there is a large integration error for linear curvature distributions along the element.

Arguments and examples see [HingeMidpoint](#).

## 1.4.13 uniaxialMaterial commands

**uniaxialMaterial** (matType, matTag, \*matArgs)

This command is used to construct a UniaxialMaterial object which represents uniaxial stress-strain (or force-deformation) relationships.

matType (str)	material type
matTag (int)	material tag.
matArgs (list)	a list of material arguments, must be preceded with *.

For example,

```
matType = 'Steel01'  
matTag = 1  
matArgs = [Fy, E0, b]  
uniaxialMaterial(matType, matTag, *matArgs)
```

The following contain information about available matType:

### Steel & Reinforcing-Steel Materials

1. [Steel01](#)
2. [Steel02](#)

3. *Steel4*
4. *Hysteretic*
5. *ReinforcingSteel*
6. *Dodd\_Restrepo*
7. *RambergOsgoodSteel*
8. *SteelMPF*
9. *Steel01Thermal*

## Steel01

**uniaxialMaterial ('Steel01', matTag, Fy, E0, b, a1, a2, a3, a4)**

This command is used to construct a uniaxial bilinear steel material object with kinematic hardening and optional isotropic hardening described by a non-linear evolution equation (REF: Fedeas).

matTag (int)	integer tag identifying material
Fy (float)	yield strength
E0 (float)	initial elastic tangent
b (float)	strain-hardening ratio (ratio between post-yield tangent and initial elastic tangent)
a1 (float)	isotropic hardening parameter, increase of compression yield envelope as proportion of yield strength after a plastic strain of $a_2 * (F_y/E_0)$ (optional)
a2 (float)	isotropic hardening parameter (see explanation under a1). (optional).
a3 (float)	isotropic hardening parameter, increase of tension yield envelope as proportion of yield strength after a plastic strain of $a_4 * (F_y/E_0)$ . (optional)
a4 (float)	isotropic hardening parameter (see explanation under a3). (optional)

---

**Note:** If strain-hardening ratio is zero and you do not expect softening of your system use BandSPD solver.

---

## Steel02

**uniaxialMaterial ('Steel02', matTag, Fy, E0, b, \*params, a1=a2\*Fy/E0, a2=1.0, a3=a4\*Fy/E0, a4=1.0, sigInit=0.0)**

This command is used to construct a uniaxial Giuffre-Menegotto-Pinto steel material object with isotropic strain hardening.

matTag ( <a href="#">int</a> )	integer tag identifying material
Fy ( <a href="#">float</a> )	yield strength
E0 ( <a href="#">float</a> )	initial elastic tangent
b ( <a href="#">float</a> )	strain-hardening ratio (ratio between post-yield tangent and initial elastic tangent)
params ( <a href="#">list (float)</a> )	parameters to control the transition from elastic to plastic branches. params=[R0, cR1, cR2]. Recommended values: R0=between 10 and 20, cR1=0.925, cR2=0.15
a1 ( <a href="#">float</a> )	isotropic hardening parameter, increase of compression yield envelope as proportion of yield strength after a plastic strain of $a_2 * (F_y/E_0)$ (optional)
a2 ( <a href="#">float</a> )	isotropic hardening parameter (see explanation under a1). (optional).
a3 ( <a href="#">float</a> )	isotropic hardening parameter, increase of tension yield envelope as proportion of yield strength after a plastic strain of $a_4 * (F_y/E_0)$ . (optional)
a4 ( <a href="#">float</a> )	isotropic hardening parameter (see explanation under a3). (optional)
sigInit ( <a href="#">float</a> )	Initial Stress Value (optional, default: 0.0) the strain is calculated from $\text{epsP}=\text{sigInit}/E$  <b>if</b> (sigInit!= 0.0) { double epsInit = sigInit/E; eps = trialStrain+epsInit; } <b>else</b> { eps = trialStrain; }

**See also:**[Steel02](#)[Steel4](#)

```
uniaxialMaterial ('Steel4', matTag, Fy, E0, '-asym', '-kin', b_k, R_0, r_1, r_2, b_kc, R_0c, r_1c, r_2c,  
'-iso', b_i, rho_i, b_l, R_i, l_yp, b_ic, rho_ic, b_lc, R_ic, '-ult', f_u, R_u, f_uc, R_uc,  
'-init', sig_init, '-mem', cycNum)
```

This command is used to construct a general uniaxial material with combined kinematic and isotropic hardening and optional non-symmetric behavior.

<code>matTag (int)</code>	integer tag identifying material
<code>Fy (float)</code>	yield strength
<code>E0 (float)</code>	initial elastic tangent
<code>'-kin' (str)</code>	apply kinematic hardening
<code>b_k (float)</code>	hardening ratio ( $E_k/E_0$ )
<code>R_0, r_1, r_2 (float)</code>	control the exponential transition from linear elastic to hardening asymptote recommended values: $R_0 = 20$ , $r_1 = 0.90$ , $r_2 = 0.15$
<code>'-iso' (str)</code>	apply isotropic hardening
<code>b_i (float)</code>	initial hardening ratio ( $E_i/E_0$ )
<code>b_l (float)</code>	saturated hardening ratio ( $E_{is}/E_0$ )
<code>rho_i (float)</code>	specifies the position of the intersection point between initial and saturated hardening asymptotes
<code>R_i (float)</code>	control the exponential transition from initial to saturated asymptote
<code>l_yp (float)</code>	length of the yield plateau in $\epsilon_{y0} = f_y / E_0$ units
<code>'-ult' (str)</code>	apply an ultimate strength limit
<code>f_u (float)</code>	ultimate strength
<code>R_u (float)</code>	control the exponential transition from kinematic hardening to perfectly plastic asymptote
<code>'-asym' (str)</code>	assume non-symmetric behavior
<code>'-init' (str)</code>	apply initial stress
<code>sig_init (float)</code>	initial stress value
<code>'-mem' (str)</code>	configure the load history memory
<code>cycNum (float)</code>	expected number of half-cycles during the loading process Efficiency of the material can be slightly increased by correctly setting this value. The default value is <code>cycNum = 50</code> Load history memory can be turned off by setting <code>cycNum = 0</code> .

**See also:**`Steel4`

## Hysteretic

```
uniaxialMaterial ('Hysteretic', matTag, *p1, *p2, *p3=p2, *n1, *n2, *n3=n2, pinchX, pinchY, damage1, damage2, beta)
```

This command is used to construct a uniaxial bilinear hysteretic material object with pinching of force and deformation, damage due to ductility and energy, and degraded unloading stiffness based on ductility.

matTag (int)	integer tag identifying material
p1 (list (float))	p1=[s1p, e1p], stress and strain (or force & deformation) at first point of the envelope in the positive direction
p2 (list (float))	p2=[s2p, e2p], stress and strain (or force & deformation) at second point of the envelope in the positive direction
p3 (list (float))	p3=[s3p, e3p], stress and strain (or force & deformation) at third point of the envelope in the positive direction
n1 (list (float))	n1=[s1n, e1n], stress and strain (or force & deformation) at first point of the envelope in the negative direction
n2 (list (float))	n2=[s2n, e2n], stress and strain (or force & deformation) at second point of the envelope in the negative direction
n3 (list (float))	n3=[s3n, e3n], stress and strain (or force & deformation) at third point of the envelope in the negative direction
pinchx (float)	pinching factor for strain (or deformation) during reloading
pinchy (float)	pinching factor for stress (or force) during reloading
damage1 (float)	damage due to ductility: D1(mu-1)
damage2 (float)	damage due to energy: D2(Eii/Eult)
beta (float)	power used to determine the degraded unloading stiffness based on ductility, mu-beta (optional, default=0.0)

### See also:

[Steel4](#)

## ReinforcingSteel

```
uniaxialMaterial ('ReinforcingSteel', matTag, fy, fu, Es, Esh, esh, eult, '-GABuck', lsr, beta, r, gama, '-DMBuck', lsr, alpha=1.0, '-CMFatigue', Cf, alpha, Cd, '-IsoHard', al=4.3, limit=1.0, '-MPCurveParams', R1=0.333, R2=18.0, R3=4.0)
```

This command is used to construct a ReinforcingSteel uniaxial material object. This object is intended to be used in a reinforced concrete fiber section as the steel reinforcing material.

<code>matTag (int)</code>	integer tag identifying material
<code>f<sub>y</sub> (float)</code>	Yield stress in tension
<code>f<sub>u</sub> (float)</code>	Ultimate stress in tension
<code>E<sub>s</sub> (float)</code>	Initial elastic tangent
<code>E<sub>sh</sub> (float)</code>	Tangent at initial strain hardening
<code>e<sub>sh</sub> (float)</code>	Strain corresponding to initial strain hardening
<code>e<sub>ult</sub> (float)</code>	Strain at peak stress
<code>'-GABuck' (str)</code>	Buckling Model Based on Gomes and Appleton (1997)
<code>lsr (float)</code>	Slenderness Ratio
<code>beta (float)</code>	Amplification factor for the buckled stress strain curve.
<code>r (float)</code>	Buckling reduction factor r can be a real number between [0.0 and 1.0] r=1.0 full reduction (no buckling) r=0.0 no reduction 0.0<r<1.0 linear interpolation between buckled and unbuckled curves
<code>gamma (float)</code>	Buckling constant
<code>'-DMBuck' (str)</code>	Buckling model based on Dhakal and Maekawa (2002)
<code>lsr (float)</code>	Slenderness Ratio
<code>alpha (float)</code>	Adjustment Constant usually between 0.75 and 1.0 Default: alpha=1.0, this parameter is optional.
<code>'-CMFatigue' (str)</code>	Coffin-Manson Fatigue and Strength Reduction
<code>C<sub>f</sub> (float)</code>	Coffin-Manson constant C
<code>alpha (float)</code>	Coffin-Manson constant a
<code>C<sub>d</sub> (float)</code>	Cyclic strength reduction constant
<code>'-IsoHard; (str)</code>	Isotropic Hardening / Diminishing Yield Plateau
<code>a<sub>1</sub> (float)</code>	Hardening constant (default = 4.3)
<code>limit (float)</code>	Limit for the reduction of the yield plateau. % of original plateau length to remain (0.01 < limit < 1.0 ) Limit =1.0, then no reduction takes place (default =0.01)
<code>'-MPCurvePa (str)</code>	Menegotto and Pinto Curve Parameters
<code>R<sub>1</sub> (float)</code>	(default = 0.333)
<code>R<sub>2</sub> (float)</code>	(default = 18)
<code>R<sub>3</sub> (float)</code>	(default = 4)

See also:

Notes

### Dodd\_Restrepo

`uniaxialMaterial ('Dodd_Restrepo', matTag, Fy, Fsu, ESH, ESU, Youngs, ESHI, FSHI, OmegaFac=1.0)`

This command is used to construct a Dodd-Restrepo steel material

matTag (int)	integer tag identifying material
Fy (float)	Yield strength
Fsu (float)	Ultimate tensile strength (UTS)
ESH (float)	Tensile strain at initiation of strain hardening
ESU (float)	Tensile strain at the UTS
Youngs (float)	Modulus of elasticity
ESHI (float)	Tensile strain for a point on strain hardening curve, recommended range of values for ESHI: [(ESU + 5*ESH)/6, (ESU + 3*ESH)/4]
FSHI (float)	Tensile stress at point on strain hardening curve corresponding to ESHI
OmegaFac (float)	Roundedness factor for Bauschinger curve in cycle reversals from the strain hardening curve. Range: [0.75, 1.15]. Largest value tends to near a bilinear Bauschinger curve. Default = 1.0.

See also:

Notes

### RambergOsgoodSteel

**uniaxialMaterial ('RambergOsgoodSteel', matTag, fy, E0, a, n)**

This command is used to construct a Ramberg–Osgood steel material object.

matTag (int)	integer tag identifying material
f <sub>y</sub> (float)	Yield strength
E <sub>0</sub> (float)	initial elastic tangent
a (float)	“yield offset” and the Commonly used value for a is 0.002
n (float)	Parameters to control the transition from elastic to plastic branches. And controls the hardening of the material by increasing the “n” hardening ratio will be decreased. Commonly used values for n are ~5 or greater.

See also:

Notes

### SteelMPF

**uniaxialMaterial ('SteelMPF', matTag, f<sub>yp</sub>, f<sub>yn</sub>, E<sub>0</sub>, bp, bn, R<sub>0</sub>, cR<sub>1</sub>, cR<sub>2</sub>, a1=0.0, a2=1.0, a3=0.0, a4=1.0)**

This command is used to construct a uniaxialMaterial SteelMPF (Kolozvari et al., 2015), which represents the well-known uniaxial constitutive nonlinear hysteretic material model for steel proposed by Menegotto and Pinto (1973), and extended by Filippou et al. (1983) to include isotropic strain hardening effects.

matTag (int)	integer tag identifying material
fyp (float)	Yield strength in tension (positive loading direction)
fyn (float)	Yield strength in compression (negative loading direction)
E0 (float)	Initial tangent modulus
bp (float)	Strain hardening ratio in tension (positive loading direction)
bn (float)	Strain hardening ratio in compression (negative loading direction)
R0 (float)	Initial value of the curvature parameter R (R0 = 20 recommended)
cR1 (float)	Curvature degradation parameter (a1 = 0.925 recommended)
cR2 (float)	Curvature degradation parameter (a2 = 0.15 or 0.0015 recommended)
a1 (float)	Isotropic hardening in compression parameter (optional, default = 0.0). Shifts compression yield envelope by a proportion of compressive yield strength after a maximum plastic tensile strain of a2(fyp/E0)
a2 (float)	Isotropic hardening in compression parameter (optional, default = 1.0).
a3 (float)	Isotropic hardening in tension parameter (optional, default = 0.0). Shifts tension yield envelope by a proportion of tensile yield strength after a maximum plastic compressive strain of a3(fyn/E0).
a4 (float)	Isotropic hardening in tension parameter (optional, default = 1.0). See explanation of a3.

**See also:**[Notes](#)**Steel01Thermal****uniaxialMaterial ('Steel01Thermal', matTag, Fy, E0, b, a1, a2, a3, a4)**

This command is the thermal version for 'Steel01'.

matTag (int)	integer tag identifying material
Fy (float)	yield strength
E0 (float)	initial elastic tangent
b (float)	strain-hardening ratio (ratio between post-yield tangent and initial elastic tangent)
a1 (float)	isotropic hardening parameter, increase of compression yield envelope as proportion of yield strength after a plastic strain of $a_2 * (F_y/E_0)$ (optional)
a2 (float)	isotropic hardening parameter (see explanation under a1). (optional).
a3 (float)	isotropic hardening parameter, increase of tension yield envelope as proportion of yield strength after a plastic strain of $a_4 * (F_y/E_0)$ . (optional)
a4 (float)	isotropic hardening parameter (see explanation under a3). (optional)

## Concrete Materials

1. [Concrete01](#)
2. [Concrete02](#)
3. [Concrete04](#)
4. [Concrete06](#)
5. [Concrete07](#)
6. [Concrete01WithSITC](#)
7. [ConfinedConcrete01](#)
8. [ConcreteD](#)
9. [FRPConfinedConcrete](#)
10. [ConcreteCM](#)

### Concrete01

**uniaxialMaterial ('Concrete01', matTag, fpc, epsc0, fpcu, epsU)**

This command is used to construct a uniaxial Kent-Scott-Park concrete material object with degraded linear unloading/reloading stiffness according to the work of Karsan-Jirsa and no tensile strength. (REF: Fedeas).

matTag ( <a href="#">int</a> )	integer tag identifying material
fpc ( <a href="#">float</a> )	concrete compressive strength at 28 days (compression is negative)
epsc0 ( <a href="#">float</a> )	concrete strain at maximum strength
fpcu ( <a href="#">float</a> )	concrete crushing strength
epsU ( <a href="#">float</a> )	concrete strain at crushing strength

---

#### Note:

1. Compressive concrete parameters should be input as negative values (if input as positive, they will be converted to negative internally).
  2. The initial slope for this model is  $(2*fpc/epsc0)$
- 

#### See also:

[Notes](#)

### Concrete02

**uniaxialMaterial ('Concrete02', matTag, fpc, epsc0, fpcu, epsU, lambda, ft, Ets)**

This command is used to construct a uniaxial Kent-Scott-Park concrete material object with degraded linear unloading/reloading stiffness according to the work of Karsan-Jirsa and no tensile strength. (REF: Fedeas).

<code>matTag (int)</code>	integer tag identifying material
<code>fpc (float)</code>	concrete compressive strength at 28 days (compression is negative)
<code>epsc0 (float)</code>	concrete strain at maximum strength
<code>fpcu (float)</code>	concrete crushing strength
<code>epsU (float)</code>	concrete strain at crushing strength
<code>lambda (float)</code>	ratio between unloading slope at \$epscu and initial slope
<code>ft (float)</code>	tensile strength
<code>Ets (float)</code>	tension softening stiffness (absolute value) (slope of the linear tension softening branch)

---

**Note:**

1. Compressive concrete parameters should be input as negative values (if input as positive, they will be converted to negative internally).
  2. The initial slope for this model is  $(2*fpc/epsc0)$
- 

**See also:**

[Notes](#)

**Concrete04**

**uniaxialMaterial ('Concrete04', matTag, fc, ec, ecu, Ec, fct, et, beta)**

This command is used to construct a uniaxial Popovics concrete material object with degraded linear unloading/reloading stiffness according to the work of Karsan-Jirsa and tensile strength with exponential decay.

<code>matTag (int)</code>	integer tag identifying material
<code>fc (float)</code>	floating point values defining concrete compressive strength at 28 days (compression is negative)
<code>ec (float)</code>	floating point values defining concrete strain at maximum strength
<code>ecu (float)</code>	floating point values defining concrete strain at crushing strength
<code>Ec (float)</code>	floating point values defining initial stiffness
<code>fct (float)</code>	floating point value defining the maximum tensile strength of concrete (optional)
<code>et (float)</code>	floating point value defining ultimate tensile strain of concrete (optional)
<code>beta (float)</code>	floating point value defining the exponential curve parameter to define the residual stress (as a factor of ft) at etu

---

**Note:**

1. Compressive concrete parameters should be input as negative values.
2. The envelope of the compressive stress-strain response is defined using the model proposed by Popovics (1973). If the user defines  $Ec = 57000 * \sqrt{|fc|}$  (in psi) then the envelope curve is identical to proposed by Mander et al. (1988).
3. Model Characteristic: For loading in compression, the envelope to the stress-strain curve follows the model proposed by Popovics (1973) until the concrete crushing strength is achieved and also for strains beyond that corresponding to the crushing strength. For unloading and reloading in compression, the Karsan-Jirsa model (1969) is used to determine the slope of the curve. For tensile loading, an exponential curve is used to define the envelope to the stress-strain curve. For unloading and reloading in tensile, the secant stiffness is used to define the path.

---

**See also:**[Notes](#)**Concrete06****uniaxialMaterial ('Concrete06', matTag, fc, e0, n, k, alpha1, fcr, ecr, b, alpha2)**

This command is used to construct a uniaxial concrete material object with tensile strength, nonlinear tension stiffening and compressive behavior based on Thorenfeldt curve.

matTag (int)	integer tag identifying material
fc (float)	concrete compressive strength (compression is negative)
e0 (float)	strain at compressive strength
n (float)	compressive shape factor
k (float)	post-peak compressive shape factor
alpha1 (float)	$\alpha_1$ parameter for compressive plastic strain definition
fcr (float)	tensile strength
ecr (float)	tensile strain at peak stress (fcr)
b (float)	exponent of the tension stiffening curve
alpha2 (float)	$\alpha_2$ parameter for tensile plastic strain definition

---

**Note:**

1. Compressive concrete parameters should be input as negative values.
- 

**See also:**[Notes](#)**Concrete07****uniaxialMaterial ('Concrete07', matTag, fc, ec, Ec, ft, et, xp, xn, r)**

Concrete07 is an implementation of Chang & Mander's 1994 concrete model with simplified unloading and reloading curves. Additionally the tension envelope shift with respect to the origin proposed by Chang and Mander has been removed. The model requires eight input parameters to define the monotonic envelope of confined and unconfined concrete in the following form:

matTag (int)	integer tag identifying material
fc (float)	concrete compressive strength (compression is negative)
ec (float)	concrete strain at maximum compressive strength
Ec (float)	Initial Elastic modulus of the concrete
ft (float)	tensile strength of concrete (tension is positive)
et (float)	tensile strain at max tensile strength of concrete
xp (float)	Non-dimensional term that defines the strain at which the straight line descent begins in tension
xn (float)	Non-dimensional term that defines the strain at which the straight line descent begins in compression
r (float)	Parameter that controls the nonlinear descending branch

**See also:**[Notes](#)**Concrete01WithSITC****uniaxialMaterial ('Concrete01WithSITC', matTag, fpc, epsec0, fpcu, epsU, endStrainSITC=0.01)**

This command is used to construct a modified uniaxial Kent-Scott-Park concrete material object with degraded linear unloading/reloading stiffness according to the work of Karsan-Jirsa and no tensile strength. The modification is to model the effect of Stuff In The Cracks (SITC).

matTag (int)	integer tag identifying material
fpc (float)	concrete compressive strength at 28 days (compression is negative)
epsec0 (float)	concrete strain at maximum strength
fpcu (float)	concrete crushing strength
epsU (float)	concrete strain at crushing strength
endStrainSITC (float)	optional, default = 0.03

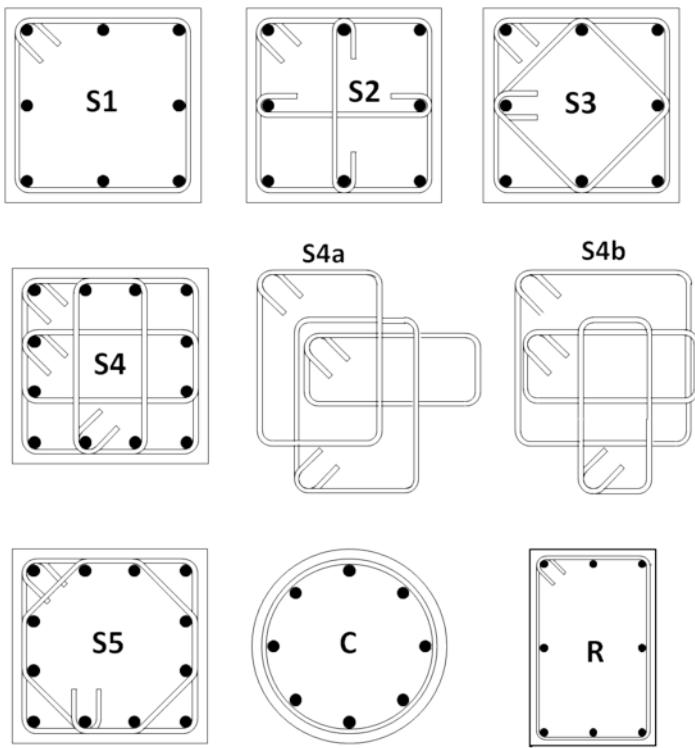
---

**Note:**

1. Compressive concrete parameters should be input as negative values (if input as positive, they will be converted to negative internally).
  2. The initial slope for this model is ( $2*fpc/epsec0$ )
- 

**See also:**[Notes](#)**ConfinedConcrete01****uniaxialMaterial ('ConfinedConcrete01', matTag, secType, fpc, Ec, '-epscu', epscu, '-gamma', gamma, '-nu', nu, '-varub', '-varnoub', L1, L2, L3, phis, S, fyh, Es0, haRatio, mu, phiLon, '-internal', \*intArgs, '-wrap', \*wrapArgs, '-gravel', '-silica', '-tol', tol, '-maxNumIter', maxNumIter, '-epscuLimit', epscuLimit, '-stRatio', stRatio)**

<code>matTag (int)</code>	integer tag identifying material
<code>sectType (str)</code>	<p>tag for the transverse reinforcement configuration. see image below.</p> <ul style="list-style-type: none"> <li>• 'S1' square section with S1 type of transverse reinforcement with or without external FRP wrapping</li> <li>• 'S2' square section with S2 type of transverse reinforcement with or without external FRP wrapping</li> <li>• 'S3' square section with S3 type of transverse reinforcement with or without external FRP wrapping</li> <li>• 'S4a' square section with S4a type of transverse reinforcement with or without external FRP wrapping</li> <li>• 'S4b' square section with S4b type of transverse reinforcement with or without external FRP wrapping</li> <li>• 'S5' square section with S5 type of transverse reinforcement with or without external FRP wrapping</li> <li>• 'C' circular section with or without external FRP wrapping</li> <li>• 'R' rectangular section with or without external FRP wrapping.</li> </ul>
<code>fpc (float)</code>	unconfined cylindrical strength of concrete specimen.
<code>Ec (float)</code>	initial elastic modulus of unconfined concrete.
<code>epscu (float)</code>	confined concrete ultimate strain. (optional)
<code>gamma (float)</code>	the ratio of the strength corresponding to ultimate strain to the peak strength of the confined concrete stress-strain curve. If <code>gamma</code> cannot be achieved in the range [0, <code>epscuLimit</code> ] then <code>epscuLimit</code> (optional, default: 0.05) will be assumed as ultimate strain.
<code>nu (float)</code>	Poisson's Ratio.
<code>'-varub' (float)</code>	Poisson's ratio is defined as a function of axial strain by means of the expression proposed by Braga et al. (2006) with the upper bound equal to 0.5
<code>'-varnoub' (float)</code>	Poisson's ratio is defined as a function of axial strain by means of the expression proposed by Braga et al. (2006) without any upper bound.
<code>L1 (float)</code>	length/diameter of square/circular core section measured respect to the hoop center line.
<code>L2 (float)</code>	additional dimensions when multiple hoops are being used.
<code>L3 (float)</code>	additional dimensions when multiple hoops are being used.
<code>phis (float)</code>	hoop diameter. If section arrangement has multiple hoops it refers to the external hoop.
<code>S (float)</code>	hoop spacing.
<code>fyh (float)</code>	yielding strength of the hoop steel.
<code>Es0 (float)</code>	elastic modulus of the hoop steel.
<code>haRatio (float)</code>	hardening ratio of the hoop steel.
<code>mu (float)</code>	ductility factor of the hoop steel.
<code>phiLon (float)</code>	diameter of longitudinal bars.
<code>intArgs (list (float))</code>	<code>intArgs= [phisi, Si, fyhi, Es0i, haRatioi, mui]</code> optional parameters for defining the internal transverse reinforcement. If they



See also:

Notes

## ConcreteD

**uniaxialMaterial ('ConcreteD', matTag, fc, epsc, ft, epst, Ec, alphac, alphat, cesp=0.25, etap=1.15)**

This command is used to construct a concrete material based on the Chinese design code.

<code>matTag (int)</code>	integer tag identifying material
<code>fc (float)</code>	concrete compressive strength
<code>epsc (float)</code>	concrete strain at corresponding to compressive strength
<code>ft (float)</code>	concrete tensile strength
<code>epst (float)</code>	concrete strain at corresponding to tensile strength
<code>Ec (float)</code>	concrete initial Elastic modulus
<code>alphac (float)</code>	compressive descending parameter
<code>alphat (float)</code>	tensile descending parameter
<code>cesp (float)</code>	plastic parameter, recommended values: 0.2~0.3
<code>etap (float)</code>	plastic parameter, recommended values: 1.0~1.3

---

**Note:**

1. Concrete compressive strength and the corresponding strain should be input as negative values.
  2. The value `fc/epsc` and `ft/epst` should be smaller than `Ec`.
- 

See also:

[Notes](#)**FRPConfinedConcrete**

```
uniaxialMaterial ('FRPConfinedConcrete', matTag, fpc1, fpc2, epsc0, D, c, Ej, Sj, tj, eju, S, fyl, fyh,  
dlong, dtrans, Es, vo, k, useBuck)
```

This command is used to construct a uniaxial Megalooikonomou-Monti-Santini concrete material object with degraded linear unloading/reloading stiffness according to the work of Karsan-Jirsa and no tensile strength.

matTag (int)	integer tag identifying material
fpc1 (float)	concrete core compressive strength.
fpc2 (float)	concrete cover compressive strength.
epsc0 (float)	strain corresponding to unconfined concrete strength.
D (float)	diameter of the circular section.
c (float)	dimension of concrete cover (until the outer edge of steel stirrups)
Ej (float)	elastic modulus of the fiber reinforced polymer (FRP) jacket.
Sj (float)	clear spacing of the FRP strips - zero if FRP jacket is continuous.
tj (float)	total thickness of the FRP jacket.
eju (float)	rupture strain of the FRP jacket from tensile coupons.
S (float)	spacing of the steel spiral/stirrups.
fyl (float)	yielding strength of longitudinal steel bars.
fyh (float)	yielding strength of the steel spiral/stirrups.
dlong (float)	diameter of the longitudinal bars of the circular section.
dtrans (float)	diameter of the steel spiral/stirrups.
Es (float)	elastic modulus of steel.
vo (float)	initial Poisson's coefficient for concrete.
k (float)	reduction factor for the rupture strain of the FRP jacket, recommended values 0.5-0.8.
useBuck (float)	FRP jacket failure criterion due to buckling of longitudinal compressive steel bars (0 = not include it, 1= to include it).

---

**Note:** #.IMPORTANT: The units of the input parameters should be in MPa, N, mm. #.Concrete compressive strengths and the corresponding strain should be input as positive values. #.When rupture of FRP jacket occurs due to dilation of concrete (lateral concrete strain exceeding reduced rupture strain of FRP jacket), the analysis is not terminated. Only a message "FRP Rupture" is plotted on the screen. #.When \$useBuck input parameter is on (equal to 1) and the model's longitudinal steel buckling conditions are fulfilled, a message "Initiation of Buckling of Long.Bar under Compression" is plotted on the screen. #.When rupture of FRP jacket occurs due to its interaction with buckled longitudinal compressive steel bars, the analysis is not terminated. Only a message "FRP Rupture due to Buckling of Long.Bar under compression" is plotted on the screen.

---

**See also:**[Notes](#)

## ConcreteCM

**uniaxialMaterial ('ConcreteCM', matTag, fpcc, epcc, Ec, rc, xcrn, ft, et, rt, xcrp, '-GapClose', gap=0)**

This command is used to construct a uniaxialMaterial ConcreteCM (Kolozvari et al., 2015), which is a uniaxial hysteretic constitutive model for concrete developed by Chang and Mander (1994).

matTag (int)	integer tag identifying material
fpcc (float)	Compressive strength ( $f'_c$ )
epcc (float)	Strain at compressive strength ( $\epsilon'_c$ )
Ec (float)	Initial tangent modulus ( $E_c$ )
rc (float)	Shape parameter in Tsai's equation defined for compression ( $r_c$ )
xcrn (float)	Non-dimensional critical strain on compression envelope ( $\epsilon_{cr}^-$ , where the envelope curve starts following a straight line)
ft (float)	Tensile strength ( $f_t$ )
et (float)	Strain at tensile strength ( $\epsilon_t$ )
rt (float)	Shape parameter in Tsai's equation defined for tension ( $r_t$ )
xcrp (float)	Non-dimensional critical strain on tension envelope ( $\epsilon_{cr}^+$ , where the envelope curve starts following a straight line – large value [e.g., 10000] recommended when tension stiffening is considered)
gap (float)	gap = 0, less gradual gap closure (default); gap = 1, more gradual gap closure

### See also:

[Notes](#)

## Standard Uniaxial Materials

1. [Elastic Uniaxial Material](#)
2. [Elastic-Perfectly Plastic Material](#)
3. [Elastic-Perfectly Plastic Gap Material](#)
4. [Elastic-No Tension Material](#)
5. [Parallel Material](#)
6. [Series Material](#)

## Elastic Uniaxial Material

**uniaxialMaterial ('Elastic', matTag, E, eta=0.0, Eneg=E)**

This command is used to construct an elastic uniaxial material object.

matTag (int)	integer tag identifying material
E (float)	tangent
eta (float)	damping tangent (optional, default=0.0)
Eneg (float)	tangent in compression (optional, default=E)

**See also:**

Notes

### Elastic-Perfectly Plastic Material

**uniaxialMaterial ('ElasticPP', matTag, E, epsyP, epsyN=epsyP, eps0=0.0)**

This command is used to construct an elastic perfectly-plastic uniaxial material object.

matTag (int)	integer tag identifying material
E (float)	tangent
epsyP (float)	strain or deformation at which material reaches plastic state in tension
epsyN (float)	strain or deformation at which material reaches plastic state in compression. (optional, default is tension value)
eps0 (float)	initial strain (optional, default: zero)

**See also:**

Notes

### Elastic-Perfectly Plastic Gap Material

**uniaxialMaterial ('ElasticPPGap', matTag, E, Fy, gap, eta=0.0, damage='noDamage')**

This command is used to construct an elastic perfectly-plastic gap uniaxial material object.

matTag (int)	integer tag identifying material
E (float)	tangent
Fy (float)	stress or force at which material reaches plastic state
gap (float)	initial gap (strain or deformation)
eta (float)	hardening ratio (=Eh/E), which can be negative
damage (str)	an optional string to specify whether to accumulate damage or not in the material. With the default string, 'noDamage' the gap material will re-center on load reversal. If the string 'damage' is provided this recentering will not occur and gap will grow.

**See also:**

Notes

## Elastic-No Tension Material

**uniaxialMaterial ('ENT', matTag, E)**

This command is used to construct a uniaxial elastic-no tension material object.

matTag (int)	integer tag identifying material
E (float)	tangent

See also:

Notes

## Parallel Material

**uniaxialMaterial ('Parallel', matTag, \*tags, '-factor', \*facts)**

This command is used to construct a parallel material object made up of an arbitrary number of previously-constructed UniaxialMaterial objects.

matTag (int)	integer tag identifying material
tags (list (int))	identification tags of materials making up the material model
facts (list (float))	factors to create a linear combination of the specified materials. Factors can be negative to subtract one material from another. (optional, default = 1.0)

See also:

Notes

## Series Material

**uniaxialMaterial ('Series', matTag, \*tags)**

This command is used to construct a series material object made up of an arbitrary number of previously-constructed UniaxialMaterial objects.

matTag (int)	integer tag identifying material
tags (list (int))	identification tags of materials making up the material model

See also:

Notes

## PyTzQz uniaxial materials for p-y, t-z and q-z elements for modeling soil-structure interaction through the piles in a structural foundation

1. *PySimple1 Material*
2. *TzSimple1 Material*
3. *QzSimple1 Material*
4. *PyLiq1 Material*

## 5. *TzLiq1 Material*

### PySimple1 Material

**uniaxialMaterial ('PySimple1', matTag, soilType, pult, Y50, Cd, c=0.0)**

This command is used to construct a PySimple1 uniaxial material object.

matTag (int)	integer tag identifying material
soilType (int)	soilType = 1 Backbone of p-y curve approximates Matlock (1970) soft clay relation. soilType = 2 Backbone of p-y curve approximates API (1993) sand relation.
pult (float)	Ultimate capacity of the p-y material. Note that “p” or “pult” are distributed loads [force per length of pile] in common design equations, but are both loads for this uniaxialMaterial [i.e., distributed load times the tributary length of the pile].
Y50 (float)	Displacement at which 50% of pult is mobilized in monotonic loading.
Cd (float)	Variable that sets the drag resistance within a fully-mobilized gap as Cd*pult.
c (float)	The viscous damping term (dashpot) on the far-field (elastic) component of the displacement rate (velocity). (optional Default = 0.0). Nonzero c values are used to represent radiation damping effects

#### See also:

Notes

### TzSimple1 Material

**uniaxialMaterial ('TzSimple1', matTag, tzType, tult, z50, c=0.0)**

This command is used to construct a TzSimple1 uniaxial material object.

matTag (int)	integer tag identifying material
soilType (int)	soilType = 1 Backbone of t-z curve approximates Reese and O'Neill (1987). soilType = 2 Backbone of t-z curve approximates Mosher (1984) relation.
tult (float)	Ultimate capacity of the t-z material. SEE NOTE 1.
z50 (float)	Displacement at which 50% of tult is mobilized in monotonic loading.
c (float)	The viscous damping term (dashpot) on the far-field (elastic) component of the displacement rate (velocity). (optional Default = 0.0). See NOTE 2.

---

#### Note:

1. The argument tult is the ultimate capacity of the t-z material. Note that “t” or “tult” are shear stresses [force per unit area of pile surface] in common design equations, but are both loads for this uniaxialMaterial [i.e., shear stress times the tributary area of the pile].
  2. Nonzero c values are used to represent radiation damping effects
- 

#### See also:

---

Notes

### **QzSimple1 Material**

**uniaxialMaterial ('QzSimple1', matTag, qzType, qult, Z50, suction=0.0, c=0.0)**

This command is used to construct a QzSimple1 uniaxial material object.

matTag (int)	integer tag identifying material
qzType (int)	qzType = 1 Backbone of q-z curve approximates Reese and O'Neill's (1987) relation for drilled shafts in clay. qzType = 2 Backbone of q-z curve approximates Vijayvergyia's (1977) relation for piles in sand.
qult (float)	Ultimate capacity of the q-z material. SEE NOTE 1.
Z50 (float)	Displacement at which 50% of qult is mobilized in monotonic loading. SEE NOTE 2.
suction (float)	Uplift resistance is equal to suction*qult. Default = 0.0. The value of suction must be 0.0 to 0.1.*
c (float)	The viscous damping term (dashpot) on the far-field (elastic) component of the displacement rate (velocity). Default = 0.0. Nonzero c values are used to represent radiation damping effects.*

---

**Note:**

1. qult: Ultimate capacity of the q-z material. Note that q1 or qult are stresses [force per unit area of pile tip] in common design equations, but are both loads for this uniaxialMaterial [i.e., stress times tip area].
2. Y50: Displacement at which 50% of pult is mobilized in monotonic loading. Note that Vijayvergyia's relation (qzType=2) refers to a "critical" displacement (zcrit) at which qult is fully mobilized, and that the corresponding z50 would be 0.125zcrit.
3. optional args suction and c must either both be omitted or both provided.

---

**See also:**

Notes

### **PyLiq1 Material**

**uniaxialMaterial ('PyLiq1', matTag, soilType, pult, Y50, Cd, c, pRes, ele1, ele2)**

**uniaxialMaterial ('PyLiq1', matTag, soilType, pult, Y50, Cd, c, pRes, '-timeSeries', tag)**

This command constructs a uniaxial p-y material that incorporates liquefaction effects. This p-y material is used with a zeroLength element to connect a pile (beam-column element) to a 2 D plane-strain FE mesh or displacement boundary condition. The p-y material obtains the average mean effective stress (which decreases with increasing excess pore pressure) either from two specified soil elements, or from a time series. Currently, the implementation requires that the specified soil elements consist of FluidSolidPorousMaterials in FourNode-Quad elements, or PressureDependMultiYield or PressureDependMultiYield02 materials in FourNodeQuadUP or NineFourQuadUP elements. There are two possible forms:

<code>matTag (int)</code>	integer tag identifying material
<code>soilType (int)</code>	<code>soilType = 1</code> Backbone of p-y curve approximates Matlock (1970) soft clay relation. <code>soilType = 2</code> Backbone of p-y curve approximates API (1993) sand relation.
<code>pult (float)</code>	Ultimate capacity of the p-y material. Note that “p” or “pult” are distributed loads [force per length of pile] in common design equations, but are both loads for this uniaxialMaterial [i.e., distributed load times the tributary length of the pile].
<code>z50 (float)</code>	Displacement at which 50% of pult is mobilized in monotonic loading.
<code>Cd (float)</code>	Variable that sets the drag resistance within a fully-mobilized gap as Cd*pult.
<code>c (float)</code>	The viscous damping term (dashpot) on the far-field (elastic) component of the displacement rate (velocity). (optional Default = 0.0). Nonzero c values are used to represent radiation damping effects
<code>pRes (float)</code>	sets the minimum (or residual) peak resistance that the material retains as the adjacent solid soil elements liquefy
<code>ele1 ele2 (float)</code>	are the eleTag (element numbers) for the two solid elements from which PyLiq1 will obtain mean effective stresses and excess pore pressures
<code>seriesTag (float)</code>	Alternatively, mean effective stress can be supplied by a time series by specifying the text string '-timeSeries' and the tag of the series seriesTag.

**See also:**[Notes](#)**TzLiq1 Material**`uniaxialMaterial ('TzLiq1', matTag, tzType, tult, z50, c, ele1, ele2)``uniaxialMaterial ('TzLiq1', matTag, tzType, tult, z50, c, '-timeSeries', seriesTag)`

The command constructs a uniaxial t-z material that incorporates liquefaction effects. This t-z material is used with a zeroLength element to connect a pile (beam-column element) to a 2 D plane-strain FE mesh. The t-z material obtains the average mean effective stress (which decreases with increasing excess pore pressure) from two specified soil elements. Currently, the implementation requires that the specified soil elements consist of FluidSolidPorousMaterials in FourNodeQuad elements.

<code>matTag (int)</code>	integer tag identifying material
<code>soilType (int)</code>	<code>soilType = 1</code> Backbone of t-z curve approximates Reese and O'Neill (1987). <code>soilType = 2</code> Backbone of t-z curve approximates Mosher (1984) relation.
<code>tult (float)</code>	Ultimate capacity of the t-z material. SEE NOTE 1.
<code>z50 (float)</code>	Displacement at which 50% of tult is mobilized in monotonic loading.
<code>c (float)</code>	The viscous damping term (dashpot) on the far-field (elastic) component of the displacement rate (velocity).
<code>ele1 ele2 (float)</code>	are the eleTag (element numbers) for the two solid elements from which PyLiq1 will obtain mean effective stresses and excess pore pressures
<code>seriesTag (float)</code>	Alternatively, mean effective stress can be supplied by a time series by specifying the text string '-timeSeries' and the tag of the seriesm seriesTag.

---

**Note:**

1. The argument `tult` is the ultimate capacity of the t-z material. Note that “t” or “`tult`” are shear stresses [force per unit area of pile surface] in common design equations, but are both loads for this `uniaxialMaterial` [i.e., shear stress times the tributary area of the pile].
  2. Nonzero `c` values are used to represent radiation damping effects
  3. To model the effects of liquefaction with `TzLiq1`, it is necessary to use the material stage updating command:
- 

**See also:**

Notes

**Other Uniaxial Materials**

1. [\*Hardening Material\*](#)
2. [\*CastFuse Material\*](#)
3. [\*ViscousDamper Material\*](#)
4. [\*BilinearOilDamper Material\*](#)
5. [\*Modified Ibarra-Medina-Krawinkler Deterioration Model with Bilinear Hysteretic Response \(Bilin Material\)\*](#)
6. [\*Modified Ibarra-Medina-Krawinkler Deterioration Model with Peak-Oriented Hysteretic Response \(ModIMK-PeakOriented Material\)\*](#)
7. [\*Modified Ibarra-Medina-Krawinkler Deterioration Model with Pinched Hysteretic Response \(ModIMKPinching Material\)\*](#)
8. [\*SAWS Material\*](#)
9. [\*BarSlip Material\*](#)
10. [\*Bond SP01 - - Strain Penetration Model for Fully Anchored Steel Reinforcing Bars\*](#)
11. [\*Fatigue Material\*](#)
12. [\*Impact Material\*](#)
13. [\*Hyperbolic Gap Material\*](#)
14. [\*Limit State Material\*](#)
15. [\*MinMax Material\*](#)
16. [\*ElasticBilin Material\*](#)
17. [\*ElasticMultiLinear Material\*](#)
18. [\*MultiLinear\*](#)
19. [\*Initial Strain Material\*](#)
20. [\*Initial Stress Material\*](#)
21. [\*PathIndependent Material\*](#)
22. [\*Pinching4 Material\*](#)
23. [\*Engineered Cementitious Composites Material\*](#)
24. [\*SelfCentering Material\*](#)

25. *Viscous Material*
26. *BoucWen Material*
27. *BWBN Material*
28. *KikuchiAikenHDR Material*
29. *KikuchiAikenLRB Material*
30. *AxialSp Material*
31. *AxialSpHD Material*
32. *Pinching Limit State Material*
33. *CFSWSWP Wood-Sheathed Cold-Formed Steel Shear Wall Panel*
34. *CFSSSWP Steel-Sheathed Cold-formed Steel Shear Wall Panel*

## Hardening Material

**uniaxialMaterial** ('Hardening', matTag, E, sigmaY, H\_iso, H\_kin, eta=0.0)

This command is used to construct a uniaxial material object with combined linear kinematic and isotropic hardening. The model includes optional visco-plasticity using a Perzyna formulation.

matTag ( <a href="#">int</a> )	integer tag identifying material
E ( <a href="#">float</a> )	tangent stiffness
sigmaY ( <a href="#">float</a> )	yield stress or force
H_iso ( <a href="#">float</a> )	isotropic hardening Modulus
H_kin ( <a href="#">float</a> )	kinematic hardening Modulus
eta ( <a href="#">float</a> )	visco-plastic coefficient (optional, default=0.0)

See also:

[Notes](#)

## CastFuse Material

**uniaxialMaterial** ('Cast', matTag, n, bo, h, fy, E, L, b, Ro, cR1, cR2, a1=s2\*Pp/Kp, a2=1.0, a3=a4\*Pp/Kp, a4=1.0)

This command is used to construct a parallel material object made up of an arbitrary number of previously-constructed UniaxialMaterial objects.

<code>matTag (int)</code>	integer tag identifying material
<code>n (int)</code>	Number of yield fingers of the CSF-brace
<code>bo (float)</code>	Width of an individual yielding finger at its base of the CSF-brace
<code>h (float)</code>	Thickness of an individual yielding finger
<code>f<sub>y</sub> (float)</code>	Yield strength of the steel material of the yielding finger
<code>E (float)</code>	Modulus of elasticity of the steel material of the yielding finger
<code>L (float)</code>	Height of an individual yielding finger
<code>b (float)</code>	Strain hardening ratio
<code>Ro (float)</code>	Parameter that controls the Bauschinger effect. Recommended Values for \$Ro=between 10 to 30
<code>cR1 (float)</code>	Parameter that controls the Bauschinger effect. Recommended Value cR1=0.925
<code>cR2 (float)</code>	Parameter that controls the Bauschinger effect. Recommended Value cR2=0.150
<code>a1 (float)</code>	isotropic hardening parameter, increase of compression yield envelope as proportion of yield strength after a plastic deformation of a2*(P <sub>p</sub> /K <sub>p</sub> )
<code>a2 (float)</code>	isotropic hardening parameter (see explanation under a1). (optional default = 1.0)
<code>a3 (float)</code>	isotropic hardening parameter, increase of tension yield envelope as proportion of yield strength after a plastic deformation of a4*(P <sub>p</sub> /K <sub>p</sub> )
<code>a4 (float)</code>	isotropic hardening parameter (see explanation under a3). (optional default = 1.0)

Gray et al. [1] showed that the monotonic backbone curve of a CSF-brace with known properties ( $n$ ,  $bo$ ,  $h$ ,  $L$ ,  $f_y$ ,  $E$ ) after yielding can be expressed as a close-form solution that is given by,  $P = P_p / \cos(2d/L)$ , in which  $d$  is the axial deformation of the brace at increment  $i$  and  $P_p$  is the yield strength of the CSF-brace and is given by the following expression

$$P_p = nb_oh^2 f_y / 4L$$

The elastic stiffness of the CSF-brace is given by,

$$K_p = nb_oEh^3 f_y / 6L^3$$

**See also:**

[Notes](#)

## ViscousDamper Material

```
uniaxialMaterial ('ViscousDamper', matTag, K, Cd, alpha, LGap=0.0, NM=1, RelTol=1e-6, AbsTol=1e-10, MaxHalf=15)
```

This command is used to construct a ViscousDamper material, which represents the Maxwell Model (linear spring and nonlinear dashpot in series). The ViscousDamper material simulates the hysteretic response of nonlinear viscous dampers. An adaptive iterative algorithm has been implemented and validated to solve numerically the constitutive equations within a nonlinear viscous damper with a high-precision accuracy.

matTag ( <a href="#">int</a> )	integer tag identifying material
K ( <a href="#">float</a> )	Elastic stiffness of linear spring to model the axial flexibility of a viscous damper (e.g. combined stiffness of the supporting brace and internal damper portion)
Cd ( <a href="#">float</a> )	Damping coefficient
alpha ( <a href="#">float</a> )	Velocity exponent
LGap ( <a href="#">float</a> )	Gap length to simulate the gap length due to the pin tolerance
NM ( <a href="#">int</a> )	Employed adaptive numerical algorithm (default value NM = 1; <ul style="list-style-type: none"><li>• 1 = Dormand-Prince54,</li><li>• 2 = 6th order Adams-Bashforth-Moulton,</li><li>• 3 = modified Rosenbrock Triple)</li></ul>
RelTol ( <a href="#">float</a> )	Tolerance for absolute relative error control of the adaptive iterative algorithm (default value 10^-6)
AbsTol ( <a href="#">float</a> )	Tolerance for absolute error control of adaptive iterative algorithm (default value 10^-10)
MaxHalf ( <a href="#">int</a> )	Maximum number of sub-step iterations within an integration step (default value 15)

**See also:**[Notes](#)**BilinearOilDamper Material**

```
uniaxialMaterial ('BilinearOilDamper', matTag, K, Cd, Fr=1.0, p=1.0, LGap=0.0, NM=1, RelTol=1e-6, AbsTol=1e-10, MaxHalf=15)
```

This command is used to construct a BilinearOilDamper material, which simulates the hysteretic response of bilinear oil dampers with relief valve. Two adaptive iterative algorithms have been implemented and validated to solve numerically the constitutive equations within a bilinear oil damper with a high-precision accuracy.

matTag ( <a href="#">int</a> )	integer tag identifying material
K ( <a href="#">float</a> )	Elastic stiffness of linear spring to model the axial flexibility of a viscous damper (e.g. combined stiffness of the supporting brace and internal damper portion)
Cd ( <a href="#">float</a> )	Damping coefficient
Fr ( <a href="#">float</a> )	Damper relief load (default=1.0, Damper property)
p ( <a href="#">float</a> )	Post-relief viscous damping coefficient ratio (default=1.0, linear oil damper)
LGap ( <a href="#">float</a> )	Gap length to simulate the gap length due to the pin tolerance
NM ( <a href="#">int</a> )	Employed adaptive numerical algorithm (default value NM = 1; • 1 = Dormand-Prince54, • 2 = 6th order Adams-Bashforth-Moulton, • 3 = modified Rosenbrock Triple)
RelTol ( <a href="#">float</a> )	Tolerance for absolute relative error control of the adaptive iterative algorithm (default value 10^-6)
AbsTol ( <a href="#">float</a> )	Tolerance for absolute error control of adaptive iterative algorithm (default value 10^-10)
MaxHalf ( <a href="#">int</a> )	Maximum number of sub-step iterations within an integration step (default value 15)

**See also:**[Notes](#)**Modified Ibarra-Medina-Krawinkler Deterioration Model with Bilinear Hysteretic Response (Bilin Material)**

```
uniaxialMaterial ('Bilin', matTag, K0, as_Plus, as_Neg, My_Plus, My_Neg, Lamda_S, Lamda_C,
Lamda_A, Lamda_K, c_S, c_C, c_A, c_K, theta_p_Plus, theta_p_Neg, theta_pc_Plus,
theta_pc_Neg, Res_Pos, Res_Neg, theta_u_Plus, theta_u_Neg, D_Plus, D_Neg, nFactor=0.0)
```

This command is used to construct a bilin material. The bilin material simulates the modified Ibarra-Krawinkler deterioration model with bilinear hysteretic response. Note that the hysteretic response of this material has been calibrated with respect to more than 350 experimental data of steel beam-to-column connections and multivariate regression formulas are provided to estimate the deterioration parameters of the model for different connection types. These relationships were developed by Lignos and Krawinkler (2009, 2011) and have been adopted by PEER/ATC (2010). The input parameters for this component model can be computed interactively from this [link](#). Use the module Component Model.

matTag (int)	integer tag identifying material
K0 (float)	elastic stiffness
as_Plus (float)	strain hardening ratio for positive loading direction
as_Neg (float)	strain hardening ratio for negative loading direction
My_Plus (float)	effective yield strength for positive loading direction
My_Neg (float)	effective yield strength for negative loading direction (negative value)
Lamda_S (float)	Cyclic deterioration parameter for strength deterioration [ $E_t = \text{Lamda\_S} * M_y$ ; set Lamda_S = 0 to disable this mode of deterioration]
Lamda_C (float)	Cyclic deterioration parameter for post-capping strength deterioration [ $E_t = \text{Lamda\_C} * M_y$ ; set Lamda_C = 0 to disable this mode of deterioration]
Lamda_A (float)	Cyclic deterioration parameter for acceleration reloading stiffness deterioration (is not a deterioration mode for a component with Bilinear hysteretic response) [Input value is required, but not used; set Lamda_A = 0].
Lamda_K (float)	Cyclic deterioration parameter for unloading stiffness deterioration [ $E_t = \text{Lamda\_K} * M_y$ ; set Lamda_k = 0 to disable this mode of deterioration]
c_S (float)	rate of strength deterioration. The default value is 1.0.
c_C (float)	rate of post-capping strength deterioration. The default value is 1.0.
c_A (float)	rate of accelerated reloading deterioration. The default value is 1.0.
c_K (float)	rate of unloading stiffness deterioration. The default value is 1.0.
theta_a_p_P (float)	pre-capping rotation for positive loading direction (often noted as plastic rotation capacity)
theta_a_p_N (float)	pre-capping rotation for negative loading direction (often noted as plastic rotation capacity) (positive value)
theta_a_pc_P (float)	post-capping rotation for positive loading direction
theta_a_pc_N (float)	post-capping rotation for negative loading direction (positive value)
Res_Pos (float)	residual strength ratio for positive loading direction
Res_Neg (float)	residual strength ratio for negative loading direction (positive value)
theta_a_u_P (float)	ultimate rotation capacity for positive loading direction
theta_a_u_N (float)	ultimate rotation capacity for negative loading direction (positive value)
D_Plus (float)	rate of cyclic deterioration in the positive loading direction (this parameter is used to create assymetric hysteretic behavior for the case of a composite beam). For symmetric hysteretic response use 1.0.
D_Neg (float)	rate of cyclic deterioration in the negative loading direction (this parameter is used to create assymetric hysteretic behavior for the case of a composite beam). For symmetric hysteretic response use 1.0.
nFactor (float)	elastic stiffness amplification factor, mainly for use with concentrated plastic hinge elements (optional, default = 0).

See also:

[Notes](#)

## Modified Ibarra-Medina-Krawinkler Deterioration Model with Peak-Oriented Hysteretic Response (ModIMKPeakOriented Material)

```
uniaxialMaterial ('ModIMKPeakOriented', matTag, K0, as_Plus, as_Neg, My_Plus, My_Neg, Lamda_S,  
Lamda_C, Lamda_A, Lamda_K, c_S, c_C, c_A, c_K, theta_p_Plus, theta_p_Neg,  
theta_pc_Plus, theta_pc_Neg, Res_Pos, Res_Neg, theta_u_Plus, theta_u_Neg,  
D_Plus, D_Neg)
```

This command is used to construct a ModIMKPeakOriented material. This material simulates the modified Ibarra-Medina-Krawinkler deterioration model with peak-oriented hysteretic response. Note that the hysteretic response of this material has been calibrated with respect to 200 experimental data of RC beams in order to estimate the deterioration parameters of the model. This information was developed by Lignos and Krawinkler (2012). NOTE: before you use this material make sure that you have downloaded the latest OpenSees version. A youtube video presents a summary of this model including the way to be used within openSees [youtube link](#).

matTag (int)	integer tag identifying material
K0 (float)	elastic stiffness
as_Plus (float)	strain hardening ratio for positive loading direction
as_Neg (float)	strain hardening ratio for negative loading direction
My_Plus (float)	effective yield strength for positive loading direction
My_Neg (float)	effective yield strength for negative loading direction (negative value)
Lamda_S (float)	Cyclic deterioration parameter for strength deterioration [ $E_t = \text{Lamda}_S * M_y$ , see Lignos and Krawinkler (2011); set Lamda_S = 0 to disable this mode of deterioration]
Lamda_C (float)	Cyclic deterioration parameter for post-capping strength deterioration [ $E_t = \text{Lamda}_C * M_y$ , see Lignos and Krawinkler (2011); set Lamda_C = 0 to disable this mode of deterioration]
Lamda_A (float)	Cyclic deterioration parameter for accelerated reloading stiffness deterioration [ $E_t = \text{Lamda}_A * M_y$ , see Lignos and Krawinkler (2011); set Lamda_A = 0 to disable this mode of deterioration]
Lamda_K (float)	Cyclic deterioration parameter for unloading stiffness deterioration [ $E_t = \text{Lamda}_K * M_y$ , see Lignos and Krawinkler (2011); set Lamda_K = 0 to disable this mode of deterioration]
c_S (float)	rate of strength deterioration. The default value is 1.0.
c_C (float)	rate of post-capping strength deterioration. The default value is 1.0.
c_A (float)	rate of accelerated reloading deterioration. The default value is 1.0.
c_K (float)	rate of unloading stiffness deterioration. The default value is 1.0.
theta_p_P (float)	pre-capping rotation for positive loading direction (often noted as plastic rotation capacity)
theta_p_N (float)	pre-capping rotation for negative loading direction (often noted as plastic rotation capacity) (must be defined as a positive value)
theta_pc_P (float)	post-capping rotation for positive loading direction
theta_pc_N (float)	post-capping rotation for negative loading direction (must be defined as a positive value)
Res_Pos (float)	residual strength ratio for positive loading direction
Res_Neg (float)	residual strength ratio for negative loading direction (must be defined as a positive value)
theta_u_P (float)	ultimate rotation capacity for positive loading direction
theta_u_N (float)	ultimate rotation capacity for negative loading direction (must be defined as a positive value)
D_Plus (float)	rate of cyclic deterioration in the positive loading direction (this parameter is used to create assymetric hysteretic behavior for the case of a composite beam). For symmetric hysteretic response use 1.0.
D_Neg (float)	rate of cyclic deterioration in the negative loading direction (this parameter is used to create assymetric hysteretic behavior for the case of a composite beam). For symmetric hysteretic response use 1.0.

**See also:**

Notes

## Modified Ibarra-Medina-Krawinkler Deterioration Model with Pinched Hysteretic Response (ModIMKPinching Material)

```
uniaxialMaterial ('ModIMKPinching', matTag, K0, as_Plus, as_Neg, My_Plus, My_Neg, FprPos,  
FprNeg, A_pinch, Lamda_S, Lamda_C, Lamda_A, Lamda_K, c_S, c_C, c_A,  
c_K, theta_p_Plus, theta_p_Neg, theta_pc_Plus, theta_pc_Neg, Res_Pos, Res_Neg,  
theta_u_Plus, theta_u_Neg, D_Plus, D_Neg)
```

This command is used to construct a ModIMKPinching material. This material simulates the modified Ibarra-Medina-Krawinkler deterioration model with pinching hysteretic response. NOTE: **before you use this material make sure that you have downloaded the latest OpenSees version**. A youtube video presents a summary of this model including the way to be used within openSees [youtube link](#).

matTag (int)	integer tag identifying material
K0 (float)	elastic stiffness
as_Plus (float)	strain hardening ratio for positive loading direction
as_Neg (float)	strain hardening ratio for negative loading direction
My_Plus (float)	effective yield strength for positive loading direction
My_Neg (float)	effective yield strength for negative loading direction (Must be defined as a negative value)
FprPos (float)	Ratio of the force at which reloading begins to force corresponding to the maximum historic deformation demand (positive loading direction)
FprNeg (float)	Ratio of the force at which reloading begins to force corresponding to the absolute maximum historic deformation demand (negative loading direction)
A_Pinch (float)	Ratio of reloading stiffness
Lamda_S (float)	Cyclic deterioration parameter for strength deterioration [ $E_t = \text{Lamda\_S} * M_y$ , see Lignos and Krawinkler (2011); set Lamda_S = 0 to disable this mode of deterioration]
Lamda_C (float)	Cyclic deterioration parameter for post-capping strength deterioration [ $E_t = \text{Lamda\_C} * M_y$ , see Lignos and Krawinkler (2011); set Lamda_C = 0 to disable this mode of deterioration]
Lamda_A (float)	Cyclic deterioration parameter for accelerated reloading stiffness deterioration [ $E_t = \text{Lamda\_A} * M_y$ , see Lignos and Krawinkler (2011); set Lamda_A = 0 to disable this mode of deterioration]
Lamda_K (float)	Cyclic deterioration parameter for unloading stiffness deterioration [ $E_t = \text{Lamda\_K} * M_y$ , see Lignos and Krawinkler (2011); set Lamda_K = 0 to disable this mode of deterioration]
c_S (float)	rate of strength deterioration. The default value is 1.0.
c_C (float)	rate of post-capping strength deterioration. The default value is 1.0.
c_A (float)	rate of accelerated reloading deterioration. The default value is 1.0.
c_K (float)	rate of unloading stiffness deterioration. The default value is 1.0.
theta_a_p_Pl (float)	pre-capping rotation for positive loading direction (often noted as plastic rotation capacity)
theta_a_p_Ne (float)	pre-capping rotation for negative loading direction (often noted as plastic rotation capacity) (must be defined as a positive value)
theta_a_pc_P (float)	post-capping rotation for positive loading direction
theta_a_pc_N (float)	post-capping rotation for negative loading direction (must be defined as a positive value)
Res_Pos (float)	residual strength ratio for positive loading direction
Res_Neg (float)	residual strength ratio for negative loading direction (must be defined as a positive value)
theta_u_Pl (float)	ultimate rotation capacity for positive loading direction
theta_u_Ne (float)	ultimate rotation capacity for negative loading direction (must be defined as a positive value)
D_Plus (float)	rate of cyclic deterioration in the positive loading direction (this parameter is used to create assymetric hysteretic behavior for the case of a composite beam). For symmetric hysteretic response use 1.0.
D_Neg (float)	rate of cyclic deterioration in the negative loading direction (this parameter is used to create assymetric hysteretic behavior for the case of a composite beam). For symmetric hysteretic response use 1.0.

**See also:**

Notes

**SAWS Material****uniaxialMaterial ('SAWS', matTag, F0, FI, DU, S0, R1, R2, R3, R4, alph, beta)**

This file contains the class definition for SAWSMaterial. SAWSMaterial provides the implementation of a one-dimensional hysteretic model developed as part of the CUREe Caltech wood frame project.

<code>matTag (int)</code>	integer tag identifying material
<code>F0 (float)</code>	Intercept strength of the shear wall spring element for the asymptotic line to the envelope curve $F0 > FI > 0$
<code>FI (float)</code>	Intercept strength of the spring element for the pinching branch of the hysteretic curve. ( $FI > 0$ ).
<code>DU (float)</code>	Spring element displacement at ultimate load. ( $DU > 0$ ).
<code>S0 (float)</code>	Initial stiffness of the shear wall spring element ( $S0 > 0$ ).
<code>R1 (float)</code>	Stiffness ratio of the asymptotic line to the spring element envelope curve. The slope of this line is $R1 S0$ . ( $0 < R1 < 1.0$ ).
<code>R2 (float)</code>	Stiffness ratio of the descending branch of the spring element envelope curve. The slope of this line is $R2 S0$ . ( $R2 < 0$ ).
<code>R3 (float)</code>	Stiffness ratio of the unloading branch off the spring element envelope curve. The slope of this line is $R3 S0$ . ( $R3 > 1$ ).
<code>R4 (float)</code>	Stiffness ratio of the pinching branch for the spring element. The slope of this line is $R4 S0$ . ( $R4 > 0$ ).
<code>alpha (float)</code>	Stiffness degradation parameter for the shear wall spring element. ( $ALPHA > 0$ ).
<code>beta (float)</code>	Stiffness degradation parameter for the spring element. ( $BETA > 0$ ).

**See also:**

Notes

**BarSlip Material****uniaxialMaterial ('BarSlip', matTag, fc, fy, Es, fu, Eh, db, ld, nb, depth, height, ancLratio=1.0, bsFlag, type, damage='Damage', unit='psi')**

This command is used to construct a uniaxial material that simulates the bar force versus slip response of a reinforcing bar anchored in a beam-column joint. The model exhibits degradation under cyclic loading. Cyclic degradation of strength and stiffness occurs in three ways: unloading stiffness degradation, reloading stiffness degradation, strength degradation.

matTag (int)	integer tag identifying material
fc (float)	positive floating point value defining the compressive strength of the concrete in which the reinforcing bar is anchored
fy (float)	positive floating point value defining the yield strength of the reinforcing steel
Es (float)	floating point value defining the modulus of elasticity of the reinforcing steel
fu (float)	positive floating point value defining the ultimate strength of the reinforcing steel
Eh (float)	floating point value defining the hardening modulus of the reinforcing steel
ld (float)	floating point value defining the development length of the reinforcing steel
db (float)	point value defining the diameter of reinforcing steel
nb (float)	an integer defining the number of anchored bars
depth (float)	floating point value defining the dimension of the member (beam or column) perpendicular to the dimension of the plane of the paper
height (float)	floating point value defining the height of the flexural member, perpendicular to direction in which the reinforcing steel is placed, but in the plane of the paper
ancLratio (float)	floating point value defining the ratio of anchorage length used for the reinforcing bar to the dimension of the joint in the direction of the reinforcing bar (optional, default: 1.0)
bsFlag (str)	string indicating relative bond strength for the anchored reinforcing bar (options: 'Strong' or 'Weak')
type (str)	string indicating where the reinforcing bar is placed. (options: 'beamtop', 'beambot' or 'column')
damage (str)	string indicating type of damage: whether there is full damage in the material or no damage (optional, options: 'Damage', 'NoDamage' ; default: 'Damage')
unit (str)	string indicating the type of unit system used (optional, options: 'psi', 'MPa', 'Pa', 'psf', 'ksi', 'ksf') (default: 'psi' / 'MPa')

**See also:**

Notes

**Bond SP01 -- Strain Penetration Model for Fully Anchored Steel Reinforcing Bars**

**uniaxialMaterial ('Bond\_SP01', matTag, Fy, Sy, Fu, Su, b, R)**

This command is used to construct a uniaxial material object for capturing strain penetration effects at the column-to-footing, column-to-bridge bent caps, and wall-to-footing intersections. In these cases, the bond slip associated with strain penetration typically occurs along a portion of the anchorage length. This model can also be applied to the beam end regions, where the strain penetration may include slippage of the bar along the entire anchorage length, but the model parameters should be chosen appropriately.

This model is for fully anchored steel reinforcement bars that experience bond slip along a portion of the anchorage length due to strain penetration effects, which are usually the case for column and wall longitudinal bars anchored into footings or bridge joints

matTag (int)	integer tag identifying material
Fy (float)	Yield strength of the reinforcement steel
Sy (float)	Rebar slip at member interface under yield stress. (see NOTES below)
Fu (float)	Ultimate strength of the reinforcement steel
Su (float)	Rebar slip at the loaded end at the bar fracture strength
b (float)	Initial hardening ratio in the monotonic slip vs. bar stress response (0.3~0.5)
R (float)	Pinching factor for the cyclic slip vs. bar response (0.5~1.0)

**See also:**

Notes

## Fatigue Material

```
uniaxialMaterial ('Fatigue', matTag, tag, '-E0', E0=0.191, '-m', m=-0.458, '-min', min=-1e16, '-max',
                   max=1e16)
```

The fatigue material uses a modified rainflow cycle counting algorithm to accumulate damage in a material using Miner's Rule. Element stress/strain relationships become zero when fatigue life is exhausted.

matTag (int)	integer tag identifying material
tag (float)	Unique material object integer tag for the material that is being wrapped
E0 (float)	Value of strain at which one cycle will cause failure (default 0.191)
m (float)	Slope of Coffin-Manson curve in log-log space (default -0.458)
min (float)	Global minimum value for strain or deformation (default -1e16)
max (float)	Global maximum value for strain or deformation (default 1e16)

See also:

Notes

## Impact Material

```
uniaxialMaterial ('ImpactMaterial', matTag, K1, K2, sigy, gap)
```

This command is used to construct an impact material object

matTag (int)	integer tag identifying material
K1 (float)	initial stiffness
K2 (float)	secondary stiffness
sigy (float)	yield displacement
gap (float)	initial gap

See also:

Notes

## Hyperbolic Gap Material

```
uniaxialMaterial ('HyperbolicGapMaterial', matTag, Kmax, Kur, Rf, Fult, gap)
```

This command is used to construct a hyperbolic gap material object.

matTag (int)	integer tag identifying material
Kmax (float)	initial stiffness
Kur (float)	unloading/reloading stiffness
Rf (float)	failure ratio
Fult (float)	ultimate (maximum) passive resistance
gap (float)	initial gap

---

Note:

1. This material is implemented as a compression-only gap material. `Fult` and `gap` should be input as negative values.
  2. Recomended Values:
    - $K_{max} = 20300 \text{ kN/m}$  of abutment width
    - $K_{cur} = K_{max}$
    - $R_f = 0.7$
    - $F_{ult} = -326 \text{ kN}$  per meter of abutment width
    - $gap = -2.54 \text{ cm}$
- 

**See also:**[Notes](#)**Limit State Material**

**uniaxialMaterial** ('*LimitState*', *matTag*, *s1p*, *e1p*, *s2p*, *e2p*, *s3p*, *e3p*, *s1n*, *e1n*, *s2n*, *e2n*, *s3n*, *e3n*, *pinchX*,  
*pinchY*, *damage1*, *damage2*, *beta*, *curveTag*, *curveType*)

This command is used to construct a uniaxial hysteretic material object with pinching of force and deformation, damage due to ductility and energy, and degraded unloading stiffness based on ductility. Failure of the material is defined by the associated Limit Curve.

<code>matTag</code> ( <code>int</code> )	integer tag identifying material
<code>s1p</code> <code>e1p</code> ( <code>float</code> )	stress and strain (or force & deformation) at first point of the envelope in the positive direction
<code>s2p</code> <code>e2p</code> ( <code>float</code> )	stress and strain (or force & deformation) at second point of the envelope in the positive direction
<code>s3p</code> <code>e3p</code> ( <code>float</code> )	stress and strain (or force & deformation) at third point of the envelope in the positive direction
<code>s1n</code> <code>e1n</code> ( <code>float</code> )	stress and strain (or force & deformation) at first point of the envelope in the negative direction
<code>s2n</code> <code>e2n</code> ( <code>float</code> )	stress and strain (or force & deformation) at second point of the envelope in the negative direction
<code>s3n</code> <code>e3n</code> ( <code>float</code> )	stress and strain (or force & deformation) at third point of the envelope in the negative direction
<code>pinchX</code> ( <code>float</code> )	pinching factor for strain (or deformation) during reloading
<code>pinchY</code> ( <code>float</code> )	pinching factor for stress (or force) during reloading
<code>damage1</code> ( <code>float</code> )	damage due to ductility: $D1(m-1)$
<code>damage2</code> ( <code>float</code> )	damage due to energy: $D2(E_i/E_{ult})$
<code>beta</code> ( <code>float</code> )	power used to determine the degraded unloading stiffness based on ductility, $m-b$ (optional, default=0.0)
<code>curveTag</code> ( <code>int</code> )	an integer tag for the Limit Curve defining the limit surface
<code>curveType</code> ( <code>int</code> )	an integer defining the type of LimitCurve (0 = no curve, 1 = axial curve, all other curves can be any other integer)

---

**Note:**

- negative backbone points should be entered as negative numeric values
- 

**See also:**[Notes](#)**MinMax Material****uniaxialMaterial ('MinMax', matTag, otherTag, '-min', minStrain=1e-16, '-max', maxStrain=1e16)**

This command is used to construct a MinMax material object. This stress-strain behaviour for this material is provided by another material. If however the strain ever falls below or above certain threshold values, the other material is assumed to have failed. From that point on, values of 0.0 are returned for the tangent and stress.

matTag (int)	integer tag identifying material
otherTag (float)	tag of the other material
minStrain (float)	minimum value of strain. optional default = -1.0e16.
maxStrain (float)	max value of strain. optional default = 1.0e16.

**See also:**[Notes](#)**ElasticBilin Material****uniaxialMaterial ('ElasticBilin', matTag, EP1, EP2, epsP2, EN1=EP1, EN2=EP2, epsN2=-epsP2)**

This command is used to construct an elastic bilinear uniaxial material object. Unlike all other bilinear materials, the unloading curve follows the loading curve exactly.

matTag (int)	integer tag identifying material
EP1 (float)	tangent in tension for strains: 0 <= strains <= epsP2
EP2 (float)	tangent when material in tension with strains > epsP2
epsP2 (float)	strain at which material changes tangent in tension.
EN1 (float)	optional, default = EP1. tangent in compression for strains: 0 < strains <= epsN2
EN2 (float)	optional, default = EP2. tangent in compression with strains < epsN2
epsN2 (float)	optional, default = -epsP2. strain at which material changes tangent in compression.

---

**Note:** eps0 can not be controlled. It is always zero.**See also:**[Notes](#)**ElasticMultiLinear Material****uniaxialMaterial ('ElasticMultiLinear', matTag, eta=0.0, '-strain', \*strainPoints, '-stress', \*stressPoints)**

This command is used to construct a multi-linear elastic uniaxial material object. The nonlinear stress-strain

relationship is given by a multi-linear curve that is define by a set of points. The behavior is nonlinear but it is elastic. This means that the material loads and unloads along the same curve, and no energy is dissipated. The slope given by the last two specified points on the positive strain axis is extrapolated to infinite positive strain. Similarly, the slope given by the last two specified points on the negative strain axis is extrapolated to infinite negative strain. The number of provided strain points needs to be equal to the number of provided stress points.

matTag ( <a href="#">int</a> )	integer tag identifying material
eta ( <a href="#">float</a> )	damping tangent (optional, default=0.0)
strainPoints ( <a href="#">list (float)</a> )	list of strain points along stress-strain curve
stressPoints ( <a href="#">list (float)</a> )	list of stress points along stress-strain curve

#### See also:

[Notes](#)

### **MultiLinear**

**uniaxialMaterial ('MultiLinear', matTag, \*pts)**

This command is used to construct a uniaxial multilinear material object.

matTag ( <a href="#">int</a> )	integer tag identifying material
pts ( <a href="#">list (float)</a> )	a list of strain and stress points pts = [strain1, stress1, strain2, stress2, ... , ]

#### See also:

[Notes](#)

### **Initial Strain Material**

**uniaxialMaterial ('InitStrainMaterial', matTag, otherTag, initStrain)**

This command is used to construct an Initial Strain material object. The stress-strain behaviour for this material is defined by another material. Initial Strain Material enables definition of initial strains for the material under consideration. The stress that corresponds to the initial strain will be calculated from the other material.

matTag ( <a href="#">int</a> )	integer tag identifying material
otherTag ( <a href="#">int</a> )	tag of the other material
initStrain ( <a href="#">float</a> )	initial strain

#### See also:

[Notes](#)

### **Initial Stress Material**

**uniaxialMaterial ('InitStressMaterial', matTag, otherTag, initStress)**

This command is used to construct an Initial Stress material object. The stress-strain behaviour for this material is defined by another material. Initial Stress Material enables definition of initial stress for the material under consideration. The stress that corresponds to the initial stress will be calculated from the other material.

matTag ( <a href="#">int</a> )	integer tag identifying material
otherTag ( <a href="#">float</a> )	tag of the other material
initStress ( <a href="#">float</a> )	initial stress

See also:

[Notes](#)

## PathIndependent Material

**uniaxialMaterial** ('PathIndependent', matTag, tag)

This command is to create a PathIndependent material

matTag ( <a href="#">int</a> )	integer tag identifying material
tag ( <a href="#">int</a> )	a pre-defined material

## Pinching4 Material

**uniaxialMaterial** ('Pinching4', matTag, ePf1, ePd1, ePf2, ePd2, ePf3, ePd3, ePf4, ePd4[, eNf1, eNd1, eNf2, eNd2, eNf3, eNd3, eNf4, eNd4], rDispP, rForceP, uForceP[, rDispN, rForceN, uForceN], gK1, gK2, gK3, gK4, gKLim, gD1, gD2, gD3, gD4, gDLim, gF1, gF2, gF3, gF4, gFLim, gE, dmgType)

This command is used to construct a uniaxial material that represents a ‘pinched’ load-deformation response and exhibits degradation under cyclic loading. Cyclic degradation of strength and stiffness occurs in three ways: unloading stiffness degradation, reloading stiffness degradation, strength degradation.

<code>matTag (int)</code>	integer tag identifying material
<code>ePf1 ePf2 ePf3 ePf4 (float)</code>	floating point values defining force points on the positive response envelope
<code>ePd1 ePd2 ePd3 ePd4 (float)</code>	floating point values defining deformation points on the positive response envelope
<code>eNf1 eNf2 eNf3 eNf4 (float)</code>	floating point values defining force points on the negative response envelope
<code>eNd1 eNd2 eNd3 eNd4 (float)</code>	floating point values defining deformation points on the negative response envelope
<code>rDispP (float)</code>	floating point value defining the ratio of the deformation at which reloading occurs to the maximum historic deformation demand
<code>fFoceP (float)</code>	floating point value defining the ratio of the force at which reloading begins to force corresponding to the maximum historic deformation demand
<code>uForceP (float)</code>	floating point value defining the ratio of strength developed upon unloading from negative load to the maximum strength developed under monotonic loading
<code>rDispN (float)</code>	floating point value defining the ratio of the deformation at which reloading occurs to the minimum historic deformation demand
<code>fFoceN (float)</code>	floating point value defining the ratio of the force at which reloading begins to force corresponding to the minimum historic deformation demand
<code>uForceN (float)</code>	floating point value defining the ratio of strength developed upon unloading from negative load to the minimum strength developed under monotonic loading
<code>gK1 gK2 gK3 gK4 gKLim (float)</code>	floating point values controlling cyclic degradation model for unloading stiffness degradation
<code>gD1 gD2 gD3 gD4 gDLim (float)</code>	floating point values controlling cyclic degradation model for reloading stiffness degradation
<code>gF1 gF2 gF3 gF4 gFLim (float)</code>	floating point values controlling cyclic degradation model for strength degradation
<code>gE (float)</code>	floating point value used to define maximum energy dissipation under cyclic loading. Total energy dissipation capacity is defined as this factor multiplied by the energy dissipated under monotonic loading.
<code>dmgType (str)</code>	string to indicate type of damage (option: 'cycle', 'energy')

See also:

Notes

## Engineered Cementitious Composites Material

`uniaxialMaterial ('ECC01', matTag, sigt0, epsi0, sigt1, epst1, epst2, sigc0, epsc0, epsc1, alphaT1, alphaT2, alphaC, alphaCU, betaT, betaC)`

This command is used to construct a uniaxial Engineered Cementitious Composites (ECC)material object based on the ECC material model of Han, et al. (see references). Reloading in tension and compression is linear.

matTag (int)	integer tag identifying material
sigt0 (float)	tensile cracking stress
epst0 (float)	strain at tensile cracking stress
sigt1 (float)	peak tensile stress
epst1 (float)	strain at peak tensile stress
sigt2 (float)	ultimate tensile strain
sigc0 (float)	compressive strength (see NOTES)
epsc0 (float)	strain at compressive strength (see NOTES)
epsc1 (float)	ultimate compressive strain (see NOTES)
alphaT1 (float)	exponent of the unloading curve in tensile strain hardening region
alphaT2 (float)	exponent of the unloading curve in tensile softening region
alphaC (float)	exponent of the unloading curve in the compressive softening
alphaCU (float)	exponent of the compressive softening curve (use 1 for linear softening)
betaT (float)	parameter to determine permanent strain in tension
betaC (float)	parameter to determine permanent strain in compression

**See also:**

Notes

**SelfCentering Material****uniaxialMaterial ('SelfCentering', matTag, k1, k2, sigAct, beta[, epsSlip, epsBear, rBear ])**

This command is used to construct a uniaxial self-centering (flag-shaped) material object with optional non-recoverable slip behaviour and an optional stiffness increase at high strains (bearing behaviour).

matTag (int)	integer tag identifying material
k1 (float)	Initial Stiffness
k2 (float)	Post-Activation Stiffness ( $0 < k2 < k1$ )
sigAct (float)	Forward Activation Stress/Force
beta (float)	Ratio of Forward to Reverse Activation Stress/Force
epsSlip (float)	slip Strain/Deformation (if <code>epsSlip = 0</code> , there will be no slippage)
epsBear (float)	Bearing Strain/Deformation (if <code>epsBear = 0</code> , there will be no bearing)
rBear (float)	Ratio of Bearing Stiffness to Initial Stiffness $k1$

**See also:**

Notes

**Viscous Material****uniaxialMaterial ('Viscous', matTag)**

This command is used to construct a uniaxial viscous material object. stress = $C(\text{strain-rate})^{\alpha}$

matTag (int)	integer tag identifying material
C (float)	damping coefficient
alpha (float)	power factor (=1 means linear damping)

**Note:**

1. This material can only be assigned to truss and zeroLength elements.
  2. This material can not be combined in parallel/series with other materials. When defined in parallel with other materials it is ignored.
- 

**See also:**[Notes](#)**BoucWen Material****uniaxialMaterial ('BoucWen', matTag, alpha, ko, n, gamma, beta, Ao, deltaA, deltaNu, deltaEta)**

This command is used to construct a uniaxial Bouc-Wen smooth hysteretic material object. This material model is an extension of the original Bouc-Wen model that includes stiffness and strength degradation (Baber and Noori (1985)).

matTag (int)	integer tag identifying material
alpha (float)	ratio of post-yield stiffness to the initial elastic stiffness ( $0 < \alpha < 1$ )
ko (float)	initial elastic stiffness
n (float)	parameter that controls transition from linear to nonlinear range (as n increases the transition becomes sharper; n is usually greater or equal to 1)
gamma beta (float)	parameters that control shape of hysteresis loop; depending on the values of $\gamma$ and $\beta$ softening, hardening or quasi-linearity can be simulated (look at the NOTES)
Ao deltaA (float)	parameters that control tangent stiffness
deltaNu deltaEta (float)	parameters that control material degradation

**See also:**[Notes](#)**BWBN Material****uniaxialMaterial ('BWBN', matTag, alpha, ko, n, gamma, beta, Ao, q, zetas, p, Shi, deltaShi, lambda, tol, maxIter)**

This command is used to construct a uniaxial Bouc-Wen pinching hysteretic material object. This material model is an extension of the original Bouc-Wen model that includes pinching (Baber and Noori (1986) and Foliente (1995)).

<code>matTag (int)</code>	integer tag identifying material
<code>alpha (float)</code>	ratio of post-yield stiffness to the initial elastic stiffness ( $0 < \alpha < 1$ )
<code>ko (float)</code>	initial elastic stiffness
<code>n (float)</code>	parameter that controls transition from linear to nonlinear range (as n increases the transition becomes sharper; n is usually greater or equal to 1)
<code>gamma beta (float)</code>	parameters that control shape of hysteresis loop; depending on the values of $\gamma$ and $\beta$ softening, hardening or quasi-linearity can be simulated (look at the BoucWen Material)
<code>Ao (float)</code>	parameter that controls tangent stiffness
<code>q zetas p Shi deltaShi lambda (float)</code>	parameters that control pinching
<code>tol (float)</code>	tolerance
<code>maxIter (float)</code>	maximum iterations

See also:

Notes

### KikuchiAikenHDR Material

`uniaxialMaterial ('KikuchiAikenHDR', matTag, tp, ar, hr[, '-coGHU', cg, ch, cu ][, '-coMSS', rs, rf ])`

This command is used to construct a uniaxial KikuchiAikenHDR material object. This material model produces nonlinear hysteretic curves of high damping rubber bearings (HDRs).

<code>matTag (int)</code>	integer tag identifying material
<code>tp (str)</code>	rubber type (see note 1)
<code>ar (float)</code>	area of rubber [unit: m <sup>2</sup> ] (see note 2)
<code>hr (float)</code>	total thickness of rubber [unit: m] (see note 2)
<code>cg ch cu (float)</code>	correction coefficients for equivalent shear modulus (cg), equivalent viscous damping ratio (ch), ratio of shear force at zero displacement (cu).
<code>rs rf (float)</code>	reduction rate for stiffness (rs) and force (rf) (see note 3)

---

Note:

- 1) Following rubber types for `tp` are available:
  - 'X0 . 6' Bridgestone X0.6, standard compressive stress, up to 400% shear strain
  - 'X0 . 6-0MPa' Bridgestone X0.6, zero compressive stress, up to 400% shear strain
  - 'X0 . 4' Bridgestone X0.4, standard compressive stress, up to 400% shear strain
  - 'X0 . 4-0MPa' Bridgestone X0.4, zero compressive stress, up to 400% shear strain
  - 'X0 . 3' Bridgestone X0.3, standard compressive stress, up to 400% shear strain
  - 'X0 . 3-0MPa' Bridgestone X0.3, zero compressive stress, up to 400% shear strain
- 2) This material uses SI unit in calculation formula. `ar` and `hr` must be converted into [m<sup>2</sup>] and [m], respectively.

- 3) `rs` and `rf` are available if this material is applied to multipleShearSpring (MSS) element. Recommended values are  $rs = \frac{1}{\sum_{i=0}^{n-1} \sin(\pi*i/n)^2}$  and  $rf = \frac{1}{\sum_{i=0}^{n-1} n-1 \sin(\pi*i/n)}$ , where  $n$  is the number of springs in the MSS. For example, when  $n=8$ ,  $rs = 0.2500$ ,  $rf = 0.1989$ .
- 

**See also:**[Notes](#)**KikuchiAikenLRB Material**

**uniaxialMaterial** ('KikuchiAikenLRB', `matTag`, `type`, `ar`, `hr`, `gr`, `ap`, `tp`, `alph`, `beta`[, '-T', `temp`][, '-coKQ', `rk`, `rq`][, '-coMSS', `rs`, `rf`])

This command is used to construct a uniaxial KikuchiAikenLRB material object. This material model produces nonlinear hysteretic curves of lead-rubber bearings.

<code>matTag</code> ( <code>int</code> )	integer tag identifying material
<code>type</code> ( <code>int</code> )	rubber type (see note 1)
<code>ar</code> ( <code>float</code> )	area of rubber [unit: m <sup>2</sup> ]
<code>hr</code> ( <code>float</code> )	total thickness of rubber [unit: m]
<code>gr</code> ( <code>float</code> )	shear modulus of rubber [unit: N/m <sup>2</sup> ]
<code>ap</code> ( <code>float</code> )	area of lead plug [unit: m <sup>2</sup> ]
<code>tp</code> ( <code>float</code> )	yield stress of lead plug [unit: N/m <sup>2</sup> ]
<code>alph</code> ( <code>float</code> )	shear modulus of lead plug [unit: N/m <sup>2</sup> ]
<code>beta</code> ( <code>float</code> )	ratio of initial stiffness to yielding stiffness
<code>temp</code> ( <code>float</code> )	temperature [unit: °C]
<code>rk rq</code> ( <code>float</code> )	reduction rate for yielding stiffness ( <code>rk</code> ) and force at zero displacement ( <code>rq</code> )
<code>rs rf</code> ( <code>float</code> )	reduction rate for stiffness ( <code>rs</code> ) and force ( <code>rf</code> ) (see note 3)

---

**Note:**

- 1) Following rubber types for `type` are available:
    - 1 lead-rubber bearing, up to 400% shear strain [Kikuchi et al., 2010 & 2012]
  - 2) This material uses SI unit in calculation formula. Input arguments must be converted into [m], [m<sup>2</sup>], [N/m<sup>2</sup>].
  - 3) `rs` and `rf` are available if this material is applied to multipleShearSpring (MSS) element. Recommended values are  $rs = \frac{1}{\sum_{i=0}^{n-1} \sin(\pi*i/n)^2}$  and  $rf = \frac{1}{\sum_{i=0}^{n-1} n-1 \sin(\pi*i/n)}$ , where  $n$  is the number of springs in the MSS. For example, when  $n=8$ ,  $rs = 0.2500$  and  $rf = 0.1989$ .
- 

**See also:**[Notes](#)**AxialSp Material**

**uniaxialMaterial** ('AxialSp', `matTag`, `sce`, `fty`, `fcy`[, `bte`, `bty`, `bcy`, `fcr`])

This command is used to construct a uniaxial AxialSp material object. This material model produces axial stress-strain curve of elastomeric bearings.

<code>matTag (int)</code>	integer tag identifying material
<code>sce (float)</code>	compressive modulus
<code>fty fcy (float)</code>	yield stress under tension ( <code>fty</code> ) and compression ( <code>fcy</code> ) (see note 1)
<code>bte bty bcy (float)</code>	reduction rate for tensile elastic range ( <code>bte</code> ), tensile yielding ( <code>bty</code> ) and compressive yielding ( <code>bcy</code> ) (see note 1)
<code>fcr (float)</code>	target point stress (see note 1)

---

**Note:**

1. Input parameters are required to satisfy followings.

$\text{fcy} < 0.0 < \text{fty}$

$0.0 \leq \text{bty} < \text{bte} \leq 1.0$

$0.0 \leq \text{bcy} \leq 1.0$

$\text{fcy} \leq \text{fcr} \leq 0.0$

---

**See also:**

[Notes](#)

**AxialSpHD Material**

**`uniaxialMaterial ('AxialSpHD', matTag, sce, fty, fcy[, bte, bty, bth, bcy, fcr, ath])`**

This command is used to construct a uniaxial AxialSpHD material object. This material model produces axial stress-strain curve of elastomeric bearings including hardening behavior.

<code>matTag (int)</code>	integer tag identifying material
<code>sce (float)</code>	compressive modulus
<code>fty1 fcy (float)</code>	yield stress under tension ( <code>fty</code> ) and compression ( <code>fcy</code> ) (see note 1)
<code>bte bty bth bcy (float)</code>	reduction rate for tensile elastic range ( <code>bte</code> ), tensile yielding ( <code>bty</code> ), tensile hardening ( <code>bth</code> ) and compressive yielding ( <code>bcy</code> ) (see note 1)
<code>fcr (float)</code>	target point stress (see note 1)

---

**Note:**

1. Input parameters are required to satisfy followings.

$\text{fcy} < 0.0 < \text{fty}$

$0.0 \leq \text{bty} < \text{bth} < \text{bte} \leq 1.0$

$0.0 \leq \text{bcy} \leq 1.0$

$\text{fcy} \leq \text{fcr} \leq 0.0$

$1.0 \leq \text{ath}$

---

**See also:**

[Notes](#)

## Pinching Limit State Material

This command is used to construct a uniaxial material that simulates a pinched load-deformation response and exhibits degradation under cyclic loading. This material works with the RotationShearCurve limit surface that can monitor a key deformation and/or a key force in an associated frame element and trigger a degrading behavior in this material when a limiting value of the deformation and/or force are reached. The material can be used in two modes: 1) direct input mode, where pinching and damage parameters are directly input; and 2) calibrated mode for shear-critical concrete columns, where only key column properties are input for model to fully define pinching and damage parameters.

```
uniaxialMaterial ('PinchingLimitStateMaterial', matTag, nodeT, nodeB, driftAxis, Kelas, crvTyp, crv-  
Tag, YpinchUPN, YpinchRPN, XpinchRPN, YpinchUNP, YpinchRNP, XpinchRNP,  
dmgStrsLimE, dmgDispMax, dmgE1, dmgE2, dmgE3, dmgE4, dmgELim, dmgR1,  
dmgR2, dmgR3, dmgR4, dmgRLim, dmgRCyc, dmgS1, dmgS2, dmgS3, dmgS4,  
dmgSLim, dmgSCyc)
```

MODE 1: Direct Input

matTag	integer tag identifying material (int)	
nodeL1	integer node tag to define the first node at the extreme end of the associated flexural frame member (int) (L3 or D5 in Figure)	
nodeL2	integer node tag to define the last node at the extreme end of the associated flexural frame member (int) (L2 or D2 in Figure)	
driftAxis	integer to indicate the drift axis in which lateral-strength degradation will occur. This axis should be orthogonal to the axis of measured rotation (see rotAxis in Rotation Shear Curve definition) (int) driftAxis = 1 – Drift along the x-axis driftAxis = 2 – Drift along the y-axis driftAxis = 3 – Drift along the z-axis	
Kelas	floating point value to define the initial material elastic stiffness (Kelastic); Kelas > 0 (float)	
crvTyp	integer flag to indicate the type of limit curve associated with this material. (int) crvTyp = 0 – No limit curve crvTyp = 1 – axial limit curve crvTyp = 2 – RotationShearCurve	
crvTag	integer tag for the unique limit curve object associated with this material (int)	
Ypinc	floating point unloading force pinching factor for loading in the negative direction. <b>Note: This value must be between zero and unity</b> (float)	
Ypinc	floating point reloading force pinching factor for loading in the negative direction. <b>Note: This value must be between negative one and unity</b> (float)	
Xpinc	floating point reloading displacement pinching factor for loading in the negative direction. <b>Note: This value must be between negative one and unity</b> (float)	
Ypinc	floating point unloading force pinching factor for loading in the positive direction. <b>Note: This value must be between zero and unity</b> (float)	
Ypinc	floating point reloading force pinching factor for loading in the positive direction. <b>Note: This value must be between negative one and unity</b> (float)	
Xpinc	floating point reloading displacement pinching factor for loading in the positive direction. <b>Note: This value must be between negative one and unity</b> (float)	
dmgSt	floating point force limit for elastic stiffness damage (typically defined as the lowest of shear strength or shear at flexural yielding). This value is used to compute the maximum deformation at flexural yield ( $\delta_{max}$ Eq. 1) and using the initial elastic stiffness (Kelastic) the monotonic energy (Emono Eq. 1) to yield. Input 1 if this type of damage is not required and set dmgE1, dmgE2, dmgE3, dmgE4, and dmgELim to zero (float)	
dmgD	floating point for ultimate drift at failure ( $\delta_{max}$ Eq. 1) and is used for strength and stiffness damage. (float) This value is used to compute the monotonic energy at axial failure (Emono Eq. 2) by computing the area under the backbone in the positive loading direction up to $\delta_{max}$ . Input 1 if this type of damage is not required and set dmgR1, dmgR2, dmgR3, dmgr4, and dmgrLim to zero for reloading stiffness damage. Similarly set dmgS1, dmgS2, dmgS3, dmgs4, and dmgsLim to zero if reloading strength damage is not required	
dmgE1		
dmgE2		
(float)		
dmgE3	floating point elastic stiffness damage factors $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ shown in Eq. 1 (float)	
dmgE4		
(float)		
dmgEL	floating point elastic stiffness damage limit Dlim shown in Eq. 1; <b>Note: This value must be between zero and unity</b> (float)	
dmgR1		
dmgR2		
(float)		
dmgR3	floating point reloading stiffness damage factors $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ shown in Eq. 1 (float)	
dmgR4		
(float)		
dmgRL	floating point reloading stiffness damage limit Dlim shown in Eq. 1; <b>Note: This value must be between zero and unity</b> (float)	
dmgRC	floating point cyclic reloading stiffness damage index; <b>Note: This value must be between zero and unity</b> (float)	151
dmgS1		
dmgS2		
(float)		
dmgS3	floating point backbone strength damage factors $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ shown in Eq. 1 (float)	

## 1.4. Model Commands

```
uniaxialMaterial ('PinchingLimitStateMaterial', matTag, dnodeT, nodeB, driftAxis, Kelas, crvTyp, crv-
    Tag, eleTag, b, d, h, a, st, As, Acc, ld, db, rho, fc, fy, fyt)
MODE 2: Calibrated Model for Shear-Critical Concrete Columns
```

mat	integer tag identifying material (int)	
node	integer node tag to define the first node at the extreme end of the associated flexural frame member (int) (L3 or D5 in Figure)	
node	integer node tag to define the last node at the extreme end of the associated flexural frame member (int) (L2 or D2 in Figure)	
drift	integer to indicate the drift axis in which lateral-strength degradation will occur. This axis should be orthogonal to the axis of measured rotation (see <code>rotAxis</code> in Rotation Shear Curve definition) (int) driftAxis = 1 – Drift along the x-axis driftAxis = 2 – Drift along the y-axis driftAxis = 3 – Drift along the z-axis	
Kelas	floating point value to define the shear stiffness (Kelastic) of the shear spring prior to shear failure (float) Kelas = -4 – Shear stiffness calculated assuming double curvature and shear springs at both column element ends Kelas = -3 – Shear stiffness calculated assuming double curvature and a shear spring at one column element end Kelas = -2 – Shear stiffness calculated assuming single curvature and shear springs at both column element ends Kelas = -1 – Shear stiffness calculated assuming single curvature and a shear spring at one column element end Kelas > 0 – Shear stiffness is the input value  Note: integer inputs allow the model to know whether column height equals the shear span (cantilever) or twice the shear span (double curvature). For columns in frames, input the value for the case that best approximates column end conditions or manually input shear stiffness (typically double curvature better estimates framed column behavior)	
crv	integer tag for the unique limit curve object associated with this material (int)	
ele	integer element tag to define the associated beam-column element used to extract axial load (int)	
b	floating point column width (inches) (float)	
d	floating point column depth (inches) (float)	
h	floating point column height (inches) (float)	
a	floating point shear span length (inches) (float)	
st	floating point transverse reinforcement spacing (inches) along column height (float)	
As	floating point total area (inches squared) of longitudinal steel bars in section (float)	
Acc	floating point gross confined concrete area (inches squared) bounded by the transverse reinforcement in column section (float)	
ld	floating point development length (inches) of longitudinal bars using ACI 318-11 Eq. 12-1 and Eq. 12-2 (float)	
db	floating point diameter (inches) of longitudinal bars in column section (float)	
rhot	floating point transverse reinforcement ratio ( $A_{st}/st.db$ ) (float)	
f'c	floating point concrete compressive strength (ksi) (float)	
f_y	floating point longitudinal steel yield strength (ksi) (float)	
f_yt	floating point transverse steel yield strength (ksi) (float)	

**See also:**[Notes](#)**CFSWSWP Wood-Sheathed Cold-Formed Steel Shear Wall Panel**

**uniaxialMaterial** ('CFSWSWP', *matTag*, *height*, *width*, *fut*, *tf*, *Ife*, *Ifi*, *ts*, *np*, *ds*, *Vs*, *sc*, *nc*, *type*, *openingArea*, *openingLength*)

This command is used to construct a uniaxialMaterial model that simulates the hysteresis response (Shear strength-Lateral displacement) of a wood-sheathed cold-formed steel shear wall panel (CFS-SWP). The hysteresis model has smooth curves and takes into account the strength and stiffness degradation, as well as pinching effect.

This uniaxialMaterial gives results in Newton and Meter units, for strength and displacement, respectively.

<i>matTag</i> (int)	integer tag identifying material
<i>height</i> (float)	SWP's height (mm)
<i>width</i> (float)	SWP's width (mm)
<i>fuf</i> (float)	Tensile strength of framing members (MPa)
<i>tf</i> (float)	Framing thickness (mm)
<i>Ife</i> (float)	Moment of inertia of the double end-stud (mm <sup>4</sup> )
<i>Ifi</i> (float)	Moment of inertia of the intermediate stud (mm <sup>4</sup> )
<i>ts</i> (float)	Sheathing thickness (mm)
<i>np</i> (float)	Sheathing number (one or two sides sheathed)
<i>ds</i> (float)	Screws diameter (mm)
<i>Vs</i> (float)	Screws shear strength (N)
<i>sc</i> (float)	Screw spacing on the SWP perimeter (mm)
<i>nc</i> (float)	Total number of screws located on the SWP perimeter
<i>type</i> (int)	Integer identifier used to define wood sheathing type (DFP=1, OSB=2, CSP=3)
<i>openingArea</i> (float)	Total area of openings (mm <sup>2</sup> )
<i>openingLength</i> (float)	Cumulative length of openings (mm)

**See also:**[Notes](#)**CFSSSWP Steel-Sheathed Cold-formed Steel Shear Wall Panel**

**uniaxialMaterial** ('CFSSSWP', *matTag*, *height*, *width*, *fuf*, *fjf*, *tf*, *Af*, *fus*, *fys*, *ts*, *np*, *ds*, *Vs*, *sc*, *dt*, *openingArea*, *openingLength*)

This command is used to construct a uniaxialMaterial model that simulates the hysteresis response (Shear strength-lateral Displacement) of a Steel-Sheathed Cold-Formed Steel Shear Wall Panel (CFS-SWP). The hysteresis model has smooth curves and takes into account the strength and stiffness degradation, as well as pinching effect.

This uniaxialMaterial gives results in Newton and Meter units, for strength and displacement, respectively.

matTag (int)	integer tag identifying material
height (float)	SWP's height (mm)
width (float)	SWP's width (mm)
fuf (float)	Tensile strength of framing members (MPa)
f_yf (float)	Yield strength of framing members (MPa)
t_f (float)	Framing thickness (mm)
A_f (float)	Framing cross section area (mm <sup>2</sup> )
f_us (float)	Tensile strength of steel sheet sheathing (MPa)
f_ys (float)	Yield strength of steel sheet sheathing (MPa)
t_s (float)	Sheathing thickness (mm)
np (float)	Sheathing number (one or two sides sheathed)
ds (float)	Screws diameter (mm)
Vs (float)	Screws shear strength (N)
sc (float)	Screw spacing on the SWP perimeter (mm)
dt (float)	Anchor bolt's diameter (mm)
openingArea (float)	Total area of openings (mm <sup>2</sup> )
openingLength (float)	Cumulative length of openings (mm)

See also:

Notes

### 1.4.14 nDMaterial commands

**nDMaterial** (*matType*, *matTag*, \**matArgs*)

This command is used to construct an NDMaterial object which represents the stress-strain relationship at the gauss-point of a continuum element.

matType (str)	material type
matTag (int)	material tag.
matArgs (list)	a list of material arguments, must be preceded with *.

For example,

```
matType = 'ElasticIsotropic'
matTag = 1
matArgs = [E, v]
nDMaterial(matType, matTag, *matArgs)
```

The following contain information about available *matType*:

1. *ElasticIsotropic*
2. *ElasticOrthotropic*
3. *J2Plasticity*
4. *DrukerPrager*
5. *Damage2p*
6. *PlaneStress*
7. *PlaneStrain*
8. *MultiaxialCyclicPlasticity*

9. *BoundingCamClay*
10. *PlateFiber*
11. *FSAM*
12. *ManzariDafalias*
13. *PM4Sand*
14. *StressDensityModel*
15. *AcousticMedium*

## ElasticIsotropic

**nDMaterial** ('ElasticIsotropic', matTag, E, v, rho=0.0)

This command is used to construct an ElasticIsotropic material object.

matTag ( <a href="#">int</a> )	integer tag identifying material
E ( <a href="#">float</a> )	elastic modulus
v ( <a href="#">float</a> )	Poisson's ratio
rho ( <a href="#">float</a> )	mass density (optional)

The material formulations for the ElasticIsotropic object are:

- 'ThreeDimensional'
- 'PlaneStrain'
- 'Plane Stress'
- 'Axisymmetric'
- 'PlateFiber'

## ElasticOrthotropic

**nDMaterial** ('ElasticOrthotropic', matTag, Ex, Ey, Ez, vxy, vyz, vxz, Gxy, Gyz, Gzx, rho=0.0)

This command is used to construct an ElasticOrthotropic material object.

matTag ( <a href="#">int</a> )	integer tag identifying material
Ex ( <a href="#">float</a> )	elastic modulus in x direction
Ey ( <a href="#">float</a> )	elastic modulus in y direction
Ez ( <a href="#">float</a> )	elastic modulus in z direction
vxy ( <a href="#">float</a> )	Poisson's ratios in x and y plane
vyz ( <a href="#">float</a> )	Poisson's ratios in y and z plane
vxz ( <a href="#">float</a> )	Poisson's ratios in z and x plane
Gxy ( <a href="#">float</a> )	shear moduli in x and y plane
Gyz ( <a href="#">float</a> )	shear moduli in y and z plane
Gzx ( <a href="#">float</a> )	shear moduli in z and x plane
rho ( <a href="#">float</a> )	mass density (optional)

The material formulations for the ElasticOrthotropic object are:

- 'ThreeDimensional'
- 'PlaneStrain'

- 'Plane Stress'
- 'AxiSymmetric'
- 'BeamFiber'
- 'PlateFiber'

## J2Plasticity

**nDMaterial** ('J2Plasticity', matTag, K, G, sig0, sigInf, delta, H)

This command is used to construct an multi dimensional material object that has a von Mises (J2) yield criterium and isotropic hardening.

matTag (int)	integer tag identifying material
K (float)	bulk modulus
G (float)	shear modulus
sig0 (float)	initial yield stress
sigInf (float)	final saturation yield stress
delta (float)	exponential hardening parameter
H (float)	linear hardening parameter

The material formulations for the J2Plasticity object are:

- 'ThreeDimensional'
- 'PlaneStrain'
- 'Plane Stress'
- 'AxiSymmetric'
- 'PlateFiber'

J2 isotropic hardening material class

Elastic Model

$$\sigma = K * \text{trace}(\epsilon_e) + (2 * G) * \text{dev}(\epsilon_e)$$

Yield Function

$$\phi(\sigma, q) = ||\text{dev}(\sigma)|| - \sqrt{\left(\frac{2}{3} * q(x_i)\right)}$$

Saturation Isotropic Hardening with linear term

$$q(x_i) = \sigma_0 + (\sigma_\infty - \sigma_0) * \exp(-\text{delta} * \xi) + H * \xi$$

Flow Rules

$$\begin{aligned}\dot{\epsilon}_p &= \gamma * \frac{\partial \phi}{\partial \sigma} \\ \dot{\xi} &= -\gamma * \frac{\partial \phi}{\partial q}\end{aligned}$$

Linear Viscosity

$$\gamma = \frac{\phi}{\eta} (if \phi > 0)$$

Backward Euler Integration Routine Yield condition enforced at time n+1

set  $\eta = 0$  for rate independent case

## DrukerPrager

**nDMaterial** ('DrukerPrager', matTag, K, G, sigmaY, rho, rhoBar, Kinf, Ko, delta1, delta2, H, theta, density, atmPressure=101e3)

This command is used to construct an multi dimensional material object that has a Drucker-Prager yield criterium.

matTag (int)	integer tag identifying material
K (float)	bulk modulus
G (float)	shear modulus
sigmaY (float)	yield stress
rho (float)	frictional strength parameter
rhoBar (float)	controls evolution of plastic volume change, $0 \leq \text{rhoBar} \leq \text{rho}$ .
Kinf (float)	nonlinear isotropic strain hardening parameter, $\text{Kinf} \geq 0$ .
Ko (float)	nonlinear isotropic strain hardening parameter, $\text{Ko} \geq 0$ .
delta1 (float)	nonlinear isotropic strain hardening parameter, $\text{delta1} \geq 0$ .
delta2 (float)	tension softening parameter, $\text{delta2} \geq 0$ .
H (float)	linear hardening parameter, $H \geq 0$ .
theta (float)	controls relative proportions of isotropic and kinematic hardening, $0 \leq \text{theta} \leq 1$ .
density (float)	mass density of the material
atmPressure (float)	optional atmospheric pressure for update of elastic bulk and shear moduli

The material formulations for the DrukerPrager object are:

- 'ThreeDimensional'
- 'PlaneStrain'

See [theory](#).

## Damage2p

**nDMaterial** ('Damage2p', matTag, fcc, '-fct', fct, '-E', E, '-ni', ni, '-Gt', Gt, '-Gc', Gc, '-rho\_bar', rho\_bar, '-H', H, '-theta', theta, '-tangent', tangent)

This command is used to construct a three-dimensional material object that has a Drucker-Prager plasticity model coupled with a two-parameter damage model.

<code>matTag</code>	integer tag identifying material <b>(int)</b>
<code>fcc</code>	concrete compressive strength, negative real value (positive input is changed in sign automatically) <b>(float)</b>
<code>fct</code>	optional concrete tensile strength, positive real value (for concrete like materials is less than fcc), $0.1 * abs(fcc) = 4750 * sqrt(abs(fcc))$ if $abs(fcc) < 2000$ because fcc is assumed in MPa (see ACI 318) <b>(float)</b>
<code>E</code>	optional Young modulus, $57000 * sqrt(abs(fcc))$ if $abs(fcc) > 2000$ because fcc is assumed in psi (see ACI 318) <b>(float)</b>
<code>ni</code>	optional Poisson coefficient, 0.15 (from comparison with tests by Kupfer Hilsdorf Rusch 1969) <b>(float)</b>
<code>Gt</code>	optional tension fracture energy density, positive real value (integral of the stress-strain envelope in tension), $1840 * fct * fct/E$ (from comparison with tests by Gopalaratnam and Shah 1985) <b>(float)</b>
<code>Gc</code>	optional compression fracture energy density, positive real value (integral of the stress-strain envelope after the peak in compression), $:math:6250*fcc*fcc/E$ (from comparison with tests by Karsan and Jirsa 1969) <b>(float)</b>
<code>rho_bar</code>	optional parameter of plastic volume change, positive real value $0 = rhoBar < sqrt(2/3)$ , 0.2 (from comparison with tests by Kupfer Hilsdorf Rusch 1969) <b>(float)</b>
<code>H</code>	optional linear hardening parameter for plasticity, positive real value (usually less than E), $0.25 * E$ (from comparison with tests by Karsan and Jirsa 1969 and Gopalaratnam and Shah 1985) <b>(float)</b>
<code>theta</code>	optional ratio between isotropic and kinematic hardening, positive real value $0 = theta = 1$ (with: 0 hardening kinematic only and 1 hardening isotropic only, 0.5 (from comparison with tests by Karsan and Jirsa 1969 and Gopalaratnam and Shah 1985) <b>(float)</b>
<code>tangent</code>	optional integer to choose the computational stiffness matrix, 0: computational tangent; 1: damaged secant stiffness (hint: in case of strong nonlinearities use it with Krylov-Newton algorithm) <b>(float)</b>

The material formulations for the Damage2p object are:

- 'ThreeDimensional'
- 'PlaneStrain'
- 'Plane Stress'
- 'Axisymmetric'
- 'PlateFiber'

See also [here](#)

## PlaneStress

`nDMaterial ('PlaneStress', matTag, threeDtag)`

This command is used to construct a plane-stress material wrapper which converts any three-dimensional material into a plane stress material via static condensation.

<code>matTag</code> <b>(int)</b>	integer tag identifying material
<code>threeDtag</code> <b>(int)</b>	tag of previously defined 3d ndMaterial material

The material formulations for the PlaneStress object are:

- 'Plane Stress'

## PlaneStrain

**nDMaterial** ('PlaneStrain', matTag, threeDtag)

This command is used to construct a plane-stress material wrapper which converts any three-dimensional material into a plane strain material by imposing plain strain conditions on the three-dimensional material.

matTag (int)	integer tag identifying material
threeDtag (int)	integer tag of previously defined 3d ndMaterial material

The material formulations for the PlaneStrain object are:

- 'PlaneStrain'

## MultiaxialCyclicPlasticity

**nDMaterial** ('MultiaxialCyclicPlasticity', matTag, rho, K, G, Su, Ho, h, m, beta, KCoeff)

This command is used to construct an multiaxial Cyclic Plasticity model for clays

matTag (int)	integer tag identifying material
rho (float)	density
K (float)	buck modulus
G (float)	maximum (small strain) shear modulus
Su (float)	undrained shear strength, size of bounding surface $R = \sqrt{8/3 * Su}$
Ho (float)	linear kinematic hardening modulus of bounding surface
h (float)	hardening parameter
m (float)	hardening parameter
beta (float)	integration parameter, usually beta=0.5
KCoeff (float)	coefficient of earth pressure, K0

## BoundingCamClay

**nDMaterial** ('BoundingCamClay', matTag, massDensity, C, bulkMod, OCR, mu\_o, alpha, lambda, h, m)

This command is used to construct a multi-dimensional bounding surface Cam Clay material object after Borja et al. (2001).

matTag (int)	integer tag identifying material
massDensity (float)	mass density
C (float)	ellipsoidal axis ratio (defines shape of ellipsoidal loading/bounding surfaces)
bulkMod (float)	initial bulk modulus
OCR (float)	overconsolidation ratio
mu_o (float)	initial shear modulus
alpha (float)	pressure-dependency parameter for modulii (greater than or equal to zero)
lambda (float)	soil compressibility index for virgin loading
h (float)	hardening parameter for plastic response inside of bounding surface (if h = 0, no hardening)
m (float)	hardening parameter (exponent) for plastic response inside of bounding surface (if m = 0, only linear hardening)

The material formulations for the BoundingCamClay object are:

- 'ThreeDimensional'
- 'PlaneStrain'

See also for information

## PlateFiber

### **nDMaterial ('PlateFiber', matTag, threeDTag)**

This command is used to construct a plate-fiber material wrapper which converts any three-dimensional material into a plate fiber material (by static condensation) appropriate for shell analysis.

matTag (int)	integer tag identifying material
threeDTag (float)	material tag for a previously-defined three-dimensional material

## FSAM

### **nDMaterial ('FSAM', matTag, rho, sX, sY, conc, rouX, rouY, nu, alfadow)**

This command is used to construct a nDMaterial FSAM (Fixed-Strut-Angle-Model, Figure 1, Kolozvari et al., 2015), which is a plane-stress constitutive model for simulating the behavior of RC panel elements under generalized, in-plane, reversed-cyclic loading conditions (Ulugtekin, 2010; Orakcal et al., 2012). In the FSAM constitutive model, the strain fields acting on concrete and reinforcing steel components of a RC panel are assumed to be equal to each other, implying perfect bond assumption between concrete and reinforcing steel bars. While the reinforcing steel bars develop uniaxial stresses under strains in their longitudinal direction, the behavior of concrete is defined using stress-strain relationships in biaxial directions, the orientation of which is governed by the state of cracking in concrete. Although the concrete stress-strain relationship used in the FSAM is fundamentally uniaxial in nature, it also incorporates biaxial softening effects including compression softening and biaxial damage. For transfer of shear stresses across the cracks, a friction-based elasto-plastic shear aggregate interlock model is adopted, together with a linear elastic model for representing dowel action on the reinforcing steel bars (Kolozvari, 2013). Note that FSAM constitutive model is implemented to be used with Shear-Flexure Interaction model for RC walls (SFI\_MVLEM), but it could be also used elsewhere.

matTag (int)	integer tag identifying material
rho (float)	Material density
sX (float)	Tag of uniaxialMaterial simulating horizontal (x) reinforcement
sY (float)	Tag of uniaxialMaterial simulating vertical (y) reinforcement
conc (float)	Tag of uniaxialMaterial simulating concrete, shall be used with uniaxialMaterial ConcreteCM
rouX (float)	Reinforcing ratio in horizontal (x) direction ( $rouX =_{s,x} / A_{gross,x}$ )
rouY (float)	Reinforcing ratio in vertical (x) direction ( $rouY =_{s,y} / A_{gross,y}$ )
nu (float)	Concrete friction coefficient ( $0.0 < \nu < 1.5$ )
alfadow (float)	Stiffness coefficient of reinforcement dowel action ( $0.0 < alfadow < 0.05$ )

See also [here](#)

References:

- 1) Kolozvari K., Orakcal K., and Wallace J. W. (2015). "Shear-Flexure Interaction Modeling of reinforced Concrete Structural Walls and Columns under Reversed Cyclic Loading", Pacific Earthquake Engineering Research Center, University of California, Berkeley, PEER Report No. 2015/12
- 2) Kolozvari K. (2013). "Analytical Modeling of Cyclic Shear-Flexure Interaction in Reinforced Concrete Structural Walls", PhD Dissertation, University of California, Los Angeles.

- 3) Orakcal K., Massone L.M., and Ulugtekin D. (2012). “Constitutive Modeling of Reinforced Concrete Panel Behavior under Cyclic Loading”, Proceedings, 15th World Conference on Earthquake Engineering, Lisbon, Portugal.
- 4) Ulugtekin D. (2010). “Analytical Modeling of Reinforced Concrete Panel Elements under Reversed Cyclic Loadings”, M.S. Thesis, Bogazici University, Istanbul, Turkey.

## ManzariDafalias

**nDMaterial** ('ManzariDafalias', matTag, G0, nu, e\_init, Mc, c, lambda\_c, e0, ksi, P\_atm, m, h0, ch, nb, A0, nd, z\_max, cz, Den)

This command is used to construct a multi-dimensional Manzari-Dafalias(2004) material.

matTag (int)	integer tag identifying material
G0 (float)	shear modulus constant
nu (float)	poisson ratio
e_init (float)	initial void ratio
Mc (float)	critical state stress ratio
c (float)	ratio of critical state stress ratio in extension and compression
lambda_c (float)	critical state line constant
e0 (float)	critical void ratio at p = 0
ksi (float)	critical state line constant
P_atm (float)	atmospheric pressure
m (float)	yield surface constant (radius of yield surface in stress ratio space)
h0 (float)	constant parameter
ch (float)	constant parameter
nb (float)	bounding surface parameter, $nb \geq 0$
A0 (float)	dilatancy parameter
nd (float)	dilatancy surface parameter $nd \geq 0$
z_max (float)	fabric-dilatancy tensor parameter
cz (float)	fabric-dilatancy tensor parameter
Den (float)	mass density of the material

The material formulations for the ManzariDafalias object are:

- 'ThreeDimensional'
- 'PlaneStrain'

See also [here](#)

## References

Dafalias YF, Manzari MT. “Simple plasticity sand model accounting for fabric change effects”. Journal of Engineering Mechanics 2004

## PM4Sand

**nDMaterial** ('PM4Sand', matTag, Dr, G0, hpo, Den, patm, h0, emax, emin, nb, nd, Ado, zmax, cz, ce, phic, nu, cgd, cdr, ckaf, Q, R, m, Fsed\_min, p\_sedo)

This command is used to construct a 2-dimensional PM4Sand material.

<code>matTag (int)</code>	integer tag identifying material
<code>Dr (float)</code>	Relative density, in fraction
<code>G0 (float)</code>	Shear modulus constant
<code>hpo (float)</code>	Contraction rate parameter
<code>Den (float)</code>	Mass density of the material
<code>P_atm (float)</code>	Optional, Atmospheric pressure
<code>h0 (float)</code>	Optional, Variable that adjusts the ratio of plastic modulus to elastic modulus
<code>emax (float)</code>	Optional, Maximum and minimum void ratios
<code>emin (float)</code>	Optional, Maximum and minimum void ratios
<code>nb (float)</code>	Optional, Bounding surface parameter, $nb \geq 0$
<code>nd (float)</code>	Optional, Dilatancy surface parameter $nd \geq 0$
<code>Ado (float)</code>	Optional, Dilatancy parameter, will be computed at the time of initialization if input value is negative
<code>z_max (float)</code>	Optional, Fabric-dilatancy tensor parameter
<code>cz (float)</code>	Optional, Fabric-dilatancy tensor parameter
<code>ce (float)</code>	Optional, Variable that adjusts the rate of strain accumulation in cyclic loading
<code>phic (float)</code>	Optional, Critical state effective friction angle
<code>nu (float)</code>	Optional, Poisson's ratio
<code>cgd (float)</code>	Optional, Variable that adjusts degradation of elastic modulus with accumulation of fabric
<code>cdr (float)</code>	Optional, Variable that controls the rotated dilatancy surface
<code>ckaf (float)</code>	Optional, Variable that controls the effect that sustained static shear stresses have on plastic modulus
<code>Q (float)</code>	Optional, Critical state line parameter
<code>R (float)</code>	Optional, Critical state line parameter
<code>m (float)</code>	Optional, Yield surface constant (radius of yield surface in stress ratio space)
<code>Fsed_min (float)</code>	Optional, Variable that controls the minimum value the reduction factor of the elastic moduli can get during reconsolidation
<code>p_sedo (float)</code>	Optional, Mean effective stress up to which reconsolidation strains are enhanced

The material formulations for the PM4Sand object are:

- 'PlaneStrain'

See also [here](#)

References

R.W.Boulanger, K.Ziotopoulou. "PM4Sand(Version 3.1): A Sand Plasticity Model for Earthquake Engineering Applications". Report No. UCD/CGM-17/01 2017

## StressDensityModel

```
nDMaterial ('StressDensityModel', matTag, mDen, eNot, A, n, nu, a1, b1, a2, b2, a3, b3, fd, muNot, muCyc,
sc, M, patm, ssl1, ssl2, ssl3, ssl4, ssl5, ssl6, ssl7, ssl8, ssl9, ssl10, hsl, p1, p2, p3, p4, p5, p6, p7,
p8, p9, p10)
```

This command is used to construct a multi-dimensional stress density material object for modeling sand behaviour following the work of Cubrinovski and Ishihara (1998a,b).

<code>matTag (int)</code>	integer tag identifying material
---------------------------	----------------------------------

Continued on next page

Table 1 – continued from previous page

mDen (float)	mass density
eNot (float)	initial void ratio
A (float)	constant for elastic shear modulus
n (float)	pressure dependency exponent for elastic shear modulus
nu (float)	Poisson's ratio
a1 (float)	peak stress ratio coefficient ( $\eta_{Max} = a1 + b1 * Is$ )
b1 (float)	peak stress ratio coefficient ( $\eta_{Max} = a1 + b1 * Is$ )
a2 (float)	max shear modulus coefficient ( $G_{n_{Max}} = a2 + b2 * Is$ )
b2 (float)	max shear modulus coefficient ( $G_{n_{Max}} = a2 + b2 * Is$ )
a3 (float)	min shear modulus coefficient ( $G_{n_{Min}} = a3 + b3 * Is$ )
b3 (float)	min shear modulus coefficient ( $G_{n_{Min}} = a3 + b3 * Is$ )
fd (float)	degradation constant
muNot (float)	dilatancy coefficient (monotonic loading)
muCyc (float)	dilatancy coefficient (cyclic loading)
sc (float)	dilatancy strain
M (float)	critical state stress ratio
patm (float)	atmospheric pressure (in appropriate units)
ss11 (float)	void ratio of quasi steady state (QSS-line) at pressure p1 (default = 0.877)
ss12 (float)	void ratio of quasi steady state (QSS-line) at pressure p2 (default = 0.877)
ss13 (float)	void ratio of quasi steady state (QSS-line) at pressure p3 (default = 0.873)
ss14 (float)	void ratio of quasi steady state (QSS-line) at pressure p4 (default = 0.870)
ss15 (float)	void ratio of quasi steady state (QSS-line) at pressure p5 (default = 0.860)
ss16 (float)	void ratio of quasi steady state (QSS-line) at pressure p6 (default = 0.850)
ss17 (float)	void ratio of quasi steady state (QSS-line) at pressure p7 (default = 0.833)
ss18 (float)	void ratio of quasi steady state (QSS-line) at pressure p8 (default = 0.833)
ss19 (float)	void ratio of quasi steady state (QSS-line) at pressure p9 (default = 0.833)
ss110 (float)	void ratio of quasi steady state (QSS-line) at pressure p10 (default = 0.833)
hs1 (float)	void ratio of upper reference state (UR-line) for all pressures (default = 0.895)
p1 (float)	pressure corresponding to ss11 (default = 1.0 kPa)
p2 (float)	pressure corresponding to ss11 (default = 10.0 kPa)
p3 (float)	pressure corresponding to ss11 (default = 30.0 kPa)
p4 (float)	pressure corresponding to ss11 (default = 50.0 kPa)
p5 (float)	pressure corresponding to ss11 (default = 100.0 kPa)
p6 (float)	pressure corresponding to ss11 (default = 200.0 kPa)
p7 (float)	pressure corresponding to ss11 (default = 400.0 kPa)
p8 (float)	pressure corresponding to ss11 (default = 400.0 kPa)
p9 (float)	pressure corresponding to ss11 (default = 400.0 kPa)
p10 (float)	pressure corresponding to ss11 (default = 400.0 kPa)

The material formulations for the StressDensityModel object are:

- 'ThreeDimensional'
- 'PlaneStrain'

#### References

Cubrinovski, M. and Ishihara K. (1998a) 'Modelling of sand behaviour based on state concept,' Soils and Foundations, 38(3), 115-127.

Cubrinovski, M. and Ishihara K. (1998b) 'State concept and modified elastoplasticity for sand modelling,' Soils and Foundations, 38(4), 213-225.

Das, S. (2014) Three Dimensional Formulation for the Stress-Strain-Dilatancy Elasto-Plastic Constitutive Model for Sand Under Cyclic Behaviour, Master's Thesis, University of Canterbury.

## AcousticMedium

**nDMaterial** ('AcousticMedium', *matTag*, *K*, *rho*)

This command is used to construct an acoustic medium NDMaterial object.

<i>matTag</i> ( <b>int</b> )	integer tag identifying material
<i>K</i> ( <b>float</b> )	bulk module of the acoustic medium
<i>rho</i> ( <b>float</b> )	mass density of the acoustic medium

## Tsinghua Sand Models

1. *CycLiqCP*
2. *CycLiqCPSP*

### CycLiqCP

**nDMaterial** ('CycLiqCP', *matTag*, *G0*, *kappa*, *h*, *Mfc*, *dre1*, *Mdc*, *dre2*, *rdr*, *alpha*, *dir*, *ein*, *rho*)

This command is used to construct a multi-dimensional material object that follows the constitutive behavior of a cyclic elastoplasticity model for large post- liquefaction deformation.

CycLiqCP material is a cyclic elastoplasticity model for large post-liquefaction deformation, and is implemented using a cutting plane algorithm. The model is capable of reproducing small to large deformation in the pre- to post-liquefaction regime. The elastic moduli of the model are pressure dependent. The plasticity in the model is developed within the framework of bounding surface plasticity, with special consideration to the formulation of reversible and irreversible dilatancy.

The model does not take into consideration of the state of sand, and requires different parameters for sand under different densities and confining pressures. The surfaces (i.e. failure and maximum pre-stress) are considered as circles in the pi plane.

The model has been validated against VELACS centrifuge model tests and has used on numerous simulations of liquefaction related problems.

When this material is employed in regular solid elements (e.g., FourNodeQuad, Brick), it simulates drained soil response. When solid-fluid coupled elements (u-p elements and SSP u-p elements) are used, the model is able to simulate undrained and partially drained behavior of soil.

<i>matTag</i> ( <b>int</b> )	integer tag identifying material
<i>G0</i> ( <b>float</b> )	A constant related to elastic shear modulus
<i>kappa</i> ( <b>float</b> )	bulk modulus
<i>h</i> ( <b>float</b> )	Model parameter for plastic modulus
<i>Mfc</i> ( <b>float</b> )	Stress ratio at failure in triaxial compression
<i>dre1</i> ( <b>float</b> )	Coefficient for reversible dilatancy generation
<i>Mdc</i> ( <b>float</b> )	Stress ratio at which the reversible dilatancy sign changes
<i>dre2</i> ( <b>float</b> )	Coefficient for reversible dilatancy release
<i>rdr</i> ( <b>float</b> )	Reference shear strain length
<i>alpha</i> ( <b>float</b> )	Parameter controlling the decrease rate of irreversible dilatancy
<i>dir</i> ( <b>float</b> )	Coefficient for irreversible dilatancy potential
<i>ein</i> ( <b>float</b> )	Initial void ratio
<i>rho</i> ( <b>float</b> )	Saturated mass density

The material formulations for the CycLiqCP object are:

- 'ThreeDimensional'
- 'PlaneStrain'

See also [here](#)

## CycLiqCPSP

**nDMaterial** ('CycLiqCPSP', matTag, G0, kappa, h, M, dre1, dre2, rdr, alpha, dir, lambdaC, ksi, e0, np, nd, ein, rho)

This command is used to construct a multi-dimensional material object that follows the constitutive behavior of a cyclic elastoplasticity model for large post- liquefaction deformation.

CycLiqCPSP material is a constitutive model for sand with special considerations for cyclic behaviour and accumulation of large post-liquefaction shear deformation, and is implemented using a cutting plane algorithm. The model: (1) achieves the simulation of post-liquefaction shear deformation based on its physics, allowing the unified description of pre- and post-liquefaction behavior of sand; (2) directly links the cyclic mobility of sand with reversible and irreversible dilatancy, enabling the unified description of monotonic and cyclic loading; (3) introduces critical state soil mechanics concepts to achieve unified modelling of sand under different states.

The critical, maximum stress ratio and reversible dilatancy surfaces follow a rounded triangle in the pi plane similar to the Matsuoka-Nakai criterion.

When this material is employed in regular solid elements (e.g., FourNodeQuad, Brick), it simulates drained soil response. When solid-fluid coupled elements (u-p elements and SSP u-p elements) are used, the model is able to simulate undrained and partially drained behavior of soil.

matTag (int)	integer tag identifying material
G0 (float)	A constant related to elastic shear modulus
kappa (float)	bulk modulus
h (float)	Model parameter for plastic modulus
M (float)	Critical state stress ratio
dre1 (float)	Coefficient for reversible dilatancy generation
dre2 (float)	Coefficient for reversible dilatancy release
rdr (float)	Reference shear strain length
alpha (float)	Parameter controlling the decrease rate of irreversible dilatancy
dir (float)	Coefficient for irreversible dilatancy potential
lambdaC (float)	Critical state constant
ksi (float)	Critical state constant
e0 (float)	Void ratio at pc=0
np (float)	Material constant for peak mobilized stress ratio
nd (float)	Material constant for reversible dilatancy generation stress ratio
ein (float)	Initial void ratio
rho (float)	Saturated mass density

The material formulations for the CycLiqCP object are:

- 'ThreeDimensional'
- 'PlaneStrain'

See also [here](#)

REFERENCES: Wang R., Zhang J.M., Wang G., 2014. A unified plasticity model for large post-liquefaction shear deformation of sand. Computers and Geotechnics. 59, 54-66.

## Materials for Modeling Concrete Walls

1. *PlaneStressUserMaterial*
2. *PlateFromPlaneStress*
3. *PlateRebar*

### PlaneStressUserMaterial

**nDMaterial** ('PlaneStressUserMaterial', *matTag*, *fc*, *ft*, *fcu*, *epsc0*, *epscu*, *epstu*, *stc*)

This command is used to create the multi-dimensional concrete material model that is based on the damage mechanism and smeared crack model.

<i>matTag</i> ( <b>int</b> )	integer tag identifying material
<i>fc</i> ( <b>float</b> )	concrete compressive strength at 28 days (positive)
<i>ft</i> ( <b>float</b> )	concrete tensile strength (positive)
<i>fcu</i> ( <b>float</b> )	concrete crushing strength (negative)
<i>epsc0</i> ( <b>float</b> )	concrete strain at maximum strength (negative)
<i>epscu</i> ( <b>float</b> )	concrete strain at crushing strength (negative)
<i>epstu</i> ( <b>float</b> )	ultimate tensile strain (positive)
<i>stc</i> ( <b>float</b> )	shear retention factor

### PlateFromPlaneStress

**nDMaterial** ('PlateFromPlaneStress', *matTag*, *newmatTag*, *matTag*, *OutofPlaneModulus*)

This command is used to create the multi-dimensional concrete material model that is based on the damage mechanism and smeared crack model.

<i>matTag</i> ( <b>int</b> )	integer tag identifying material
<i>newmatTag</i> ( <b>int</b> )	new integer tag identifying material deriving from pre-defined PlaneStressUserMaterial
<i>matTag</i> ( <b>int</b> )	integer tag identifying PlaneStressUserMaterial
<i>OutofPlaneModulus</i> ( <b>float</b> )	shear modulus of out plane

### PlateRebar

**nDMaterial** ('PlateRebar', *matTag*, *newmatTag*, *matTag*, *sita*)

This command is used to create the multi-dimensional reinforcement material.

<i>matTag</i> ( <b>int</b> )	integer tag identifying material
<i>newmatTag</i> ( <b>int</b> )	new integer tag identifying material deriving from pre-defined uniaxial steel material
<i>matTag</i> ( <b>int</b> )	integer tag identifying uniaxial steel material
<i>sita</i> ( <b>float</b> )	define the angle of steel layer, 90 (longitudinal steel), 0 (transverse steel)

## Contact Materials for 2D and 3D

1. *ContactMaterial2D*

## 2. *ContactMaterial3D*

### ContactMaterial2D

**nDMaterial** ('ContactMaterial2D', matTag, mu, G, c, t)

This command is used to construct a ContactMaterial2D nDMaterial object.

matTag (int)	integer tag identifying material
mu (float)	interface frictional coefficient
G (float)	interface stiffness parameter
c (float)	interface cohesive intercept
t (float)	interface tensile strength

The ContactMaterial2D nDMaterial defines the constitutive behavior of a frictional interface between two bodies in contact. The interface defined by this material object allows for sticking, frictional slip, and separation between the two bodies in a two-dimensional analysis. A regularized Coulomb frictional law is assumed. Information on the theory behind this material can be found in, e.g. Wriggers (2002).

---

#### Note:

1. The ContactMaterial2D nDMaterial has been written to work with the SimpleContact2D and BeamContact2D element objects.
  2. There are no valid recorder queries for this material other than those which are listed with those elements
- 

### References:

Wriggers, P. (2002). Computational Contact Mechanics. John Wiley & Sons, Ltd, West Sussex, England.

### ContactMaterial3D

**nDMaterial** ('ContactMaterial3D', matTag, mu, G, c, t)

This command is used to construct a ContactMaterial3D nDMaterial object.

matTag (int)	integer tag identifying material
mu (float)	interface frictional coefficient
G (float)	interface stiffness parameter
c (float)	interface cohesive intercept
t (float)	interface tensile strength

The ContactMaterial3D nDMaterial defines the constitutive behavior of a frictional interface between two bodies in contact. The interface defined by this material object allows for sticking, frictional slip, and separation between the two bodies in a three-dimensional analysis. A regularized Coulomb frictional law is assumed. Information on the theory behind this material can be found in, e.g. Wriggers (2002).

---

#### Note:

1. The ContactMaterial3D nDMaterial has been written to work with the SimpleContact3D and BeamContact3D element objects.
  2. There are no valid recorder queries for this material other than those which are listed with those elements.
-

---

## References:

Wriggers, P. (2002). Computational Contact Mechanics. John Wiley & Sons, Ltd, West Sussex, England.

## Wrapper material for Initial State Analysis

1. *InitialStateAnalysisWrapper*

### InitialStateAnalysisWrapper

**nDMaterial** ('*InitialStateAnalysisWrapper*', *matTag*, *nDMatTag*, *nDim*)

The InitialStateAnalysisWrapper nDMaterial allows for the use of the InitialStateAnalysis command for setting initial conditions. The InitialStateAnalysisWrapper can be used with any nDMaterial. This material wrapper allows for the development of an initial stress field while maintaining the original geometry of the problem. An example analysis is provided below to demonstrate the use of this material wrapper object.

<i>matTag</i> (int)	integer tag identifying material
<i>nDMatTag</i> (int)	the tag of the associated nDMaterial object
<i>nDim</i> (int)	number of dimensions (2 for 2D, 3 for 3D)

---

#### Note:

1. There are no valid recorder queries for the InitialStateAnalysisWrapper.
  2. The InitialStateAnalysis off command removes all previously defined recorders. Two sets of recorders are needed if the results before and after this command are desired. See the example below for more.
  3. The InitialStateAnalysisWrapper material is somewhat tricky to use in dynamic analysis. Sometimes setting the displacement to zero appears to be interpreted as an initial displacement in subsequent steps, resulting in undesirable vibrations.
- 

## UC San Diego soil models

1. *PressureIndependMultiYield*
2. *PressureDependMultiYield*
3. *PressureDependMultiYield02*

### PressureIndependMultiYield

**nDMaterial** ('*PressureIndependMultiYield*', *matTag*, *nd*, *rho*, *refShearModul*, *refBulkModul*, *cohesi*, *peakShearStra*, *frictionAng=0.*, *refPress=100.*, *pressDependCoe=0.*, *noYieldSurf=20*, \**yieldSurf*)

PressureIndependMultiYield material is an elastic-plastic material in which plasticity exhibits only in the deviatoric stress-strain response. The volumetric stress-strain response is linear-elastic and is independent of the deviatoric response. This material is implemented to simulate monotonic or cyclic response of materials whose shear behavior is insensitive to the confinement change. Such materials include, for example, organic soils or clay under fast (undrained) loading conditions.

matTag (int)	integer tag identifying material
nd (float)	Number of dimensions, 2 for plane-strain, and 3 for 3D analysis.
rho (float)	Saturated soil mass density.
refShearModul (float)	Reference low-strain shear modulus, specified at a reference mean effective confining pressure refPress of $p'_r$ (see below).
refBulkModul (float)	Reference bulk modulus, specified at a reference mean effective confining pressure refPress of $p'_r$ (see below).
cohesi (float)	(c) Apparent cohesion at zero effective confinement.
peakShearStrain (float)	An octahedral shear strain at which the maximum shear strength is reached, specified at a reference mean effective confining pressure refPress of $p'_r$ (see below).
frictionAng (float)	Friction angle at peak shear strength in degrees, optional (default is 0.0).
refPress (float)	Reference mean effective confining pressure at which $G_r$ , $B_r$ , and $\gamma_{max}$ are defined, optional (default is 100. kPa).
pressDependCoe (float)	A positive constant defining variations of $G$ and $B$ as a function of instantaneous effective confinement $p'$ (default is 0.0) $G = G_r \left( \frac{p'}{p'_r} \right)^d$ $B = B_r \left( \frac{p'}{p'_r} \right)^d$ If $\phi = 0$ , $d$ is reset to 0.0.
noYieldSurf (float)	Number of yield surfaces, optional (must be less than 40, default is 20). The surfaces are generated based on the hyperbolic relation defined in Note 2 below.
yieldSurf (list (float))	Instead of automatic surfaces generation (Note 2), you can define yield surfaces directly based on desired shear modulus reduction curve. To do so, add a minus sign in front of noYieldSurf, then provide noYieldSurf pairs of shear strain ( $r$ ) and modulus ratio ( $G_s$ ) values. For example, to define 10 surfaces: yieldSurf = [r1, Gs1, ..., r10, Gs10]

See also notes

## PressureDependMultiYield

```
nDMaterial ('PressureDependMultiYield', matTag, nd, rho, refShearModul, refBulkModul, frictionAng,
peakShearStra, refPress, pressDependCoe, PTAng, contrac, *dilat, *liquefac, noYield-
Surf=20.0, *yieldSurf=[], e=0.6, *params=[0.9, 0.02, 0.7, 101.0], c=0.3)
```

PressureDependMultiYield material is an elastic-plastic material for simulating the essential response characteristics of pressure sensitive soil materials under general loading conditions. Such characteristics include dilatancy (shear-induced volume contraction or dilation) and non-flow liquefaction (cyclic mobility), typically exhibited in sands or silts during monotonic or cyclic loading.

mat	Integer tag identifying material
(int)	
nd	Number of dimensions, 2 for plane-strain, and 3 for 3D analysis.
(float)	
rho	Saturated soil mass density.
(float)	
refSh	Reference low-strain shear modulus, specified at a reference mean effective confining pressure refPress of p'r (see below).
(float)	
refBu	Reference bulk modulus, specified at a reference mean effective confining pressure refPress of p'r (see below).
(float)	
frict	Friction angle at peak shear strength in degrees, optional (default is 0.0).
(float)	
peakShear	An octahedral shear strain at which the maximum shear strength is reached, specified at a reference mean effective confining pressure refPress of p'r (see below).
(float)	
refP	Reference mean effective confining pressure at which $G_r$ , $B_r$ , and $\gamma_{max}$ are defined, optional (default is 100. kPa).
(float)	
pressD	A positive constant defining variations of $G$ and $B$ as a function of instantaneous effective confining pressure $p'$ (default is 0.0)
(float)	
	$G = G_r \left( \frac{p'}{p'_r} \right)^d$
	$B = B_r \left( \frac{p'}{p'_r} \right)^d$
	If $\phi = 0$ , $d$ is reset to 0.0.
PTAng	Phase transformation angle, in degrees.
(float)	
contr	A non-negative constant defining the rate of shear-induced volume decrease (contraction) or pore pressure buildup. A larger value corresponds to faster contraction rate.
(float)	
dilat	Non-negative constants defining the rate of shear-induced volume increase (dilation). Larger values correspond to stronger dilation rate. dilat = [dilat1, dilat2].
(list)	
liquefac	Parameters controlling the mechanism of liquefaction-induced perfectly plastic shear strain accumulation, i.e., cyclic mobility. Set liquefac[0] = 0 to deactivate this mechanism altogether. liquefac[0] defines the effective confining pressure (e.g., 10 kPa in SI units or 1.45 psi in English units) below which the mechanism is in effect. Smaller values should be assigned to denser sands. Liquefac[1] defines the maximum amount of perfectly plastic shear strain developed at zero effective confinement during each loading phase. Smaller values should be assigned to denser sands. Liquefac[2] defines the maximum amount of biased perfectly plastic shear strain $\gamma_b$ accumulated at each loading phase under biased shear loading conditions, as $\gamma_b = liquefac[1] \times liquefac[2]$ . Typically, liquefac[2] takes a value between 0.0 and 3.0. Smaller values should be assigned to denser sands. See the references listed at the end of this chapter for more information.
noYieldSurf	Number of yield surfaces, optional (must be less than 40, default is 20). The surfaces are generated based on the hyperbolic relation defined in Note 2 below.
yieldSurf	If noYieldSurf < 0 && >-100, the user defined yield surface is used. You have to provide a list of 2 * (-noYieldSurf), otherwise, the arguments will be messed up. Also don't provide user defined yield surface if noYieldSurf > 0, it will mess up the argument list too. Instead of automatic surfaces generation (Note 2), you can define yield surfaces directly based on desired shear modulus reduction curve. To do so, add a minus sign in front of noYieldSurf, then provide noYieldSurf pairs of shear strain (r) and modulus ratio (Gs) values. For example, to define 10 surfaces: yieldSurf = [r1, Gs1, ..., r10, Gs10]
e	Initial void ratio, optional (default is 0.6).
(float)	
params	params=[cs1, cs2, cs3, pa] defining a straight critical-state line ec in e-p' space.
(list)	If cs3=0,
(float)	ec = cs1-cs2 log(p'/pa)
	else (Li and Wang, JGGE, 124(12)),
	ec = cs1-cs2(p'/pa)cs3
1.4. Model Commands	atmospheric pressure for normalization (typically 101 kPa in SI units, or 14.65 psi <b>171</b> English units). All four constants are optional
c	Numerical constant (default value = 0.3 kPa)
(float)	

See also [notes](#)

## PressureDependMultiYield02

**nDMaterial** ('PressureDependMultiYield02', matTag, nd, rho, refShearModul, refBulkModul, frictionAng, peakShearStra, refPress, pressDependCoe, PTAng, contrac[0], contrac[2], dilat[0], dilat[2], noYieldSurf=20.0, \*yieldSurf=[], contrac[1]=5.0, dilat[1]=3.0, \*liquefac=[1.0,0.0],e=0.6, \*params=[0.9, 0.02, 0.7, 101.0], c=0.1)

PressureDependMultiYield02 material is modified from PressureDependMultiYield material, with:

1. additional parameters (contrac [2] and dilat [2]) to account for  $K_\sigma$  effect,
2. a parameter to account for the influence of previous dilation history on subsequent contraction phase (contrac [1]), and
3. modified logic related to permanent shear strain accumulation (liquefac [0] and liquefac [1]).

matTag (int)	integer tag identifying material
contrac[2] (float)	A non-negative constant reflecting $K_\sigma$ effect.
dilat[2] (float)	A non-negative constant reflecting $K_\sigma$ effect.
contrac[1] (float)	A non-negative constant reflecting dilation history on contraction tendency.
liquefac[0] (float)	Damage parameter to define accumulated permanent shear strain as a function of dilation history. (Redefined and different from PressureDependMultiYield material).
liquefac[1] (float)	Damage parameter to define biased accumulation of permanent shear strain as a function of load reversal history. (Redefined and different from PressureDependMultiYield material).
c (float)	Numerical constant (default value = 0.1 kPa)

See also [notes](#)

## UC San Diego Saturated Undrained soil

1. [FluidSolidPorousMaterial](#)

## FluidSolidPorousMaterial

**nDMaterial** ('FluidSolidPorousMaterial', matTag, nd, soilMatTag, combinedBulkModul, pa=101.0)

FluidSolidPorousMaterial couples the responses of two phases: fluid and solid. The fluid phase response is only volumetric and linear elastic. The solid phase can be any NDMaterial. This material is developed to simulate the response of saturated porous media under fully undrained condition.

matTag (int)	integer tag identifying material
nd (float)	Number of dimensions, 2 for plane-strain, and 3 for 3D analysis.
soilMatTag (int)	The material number for the solid phase material (previously defined).
combiBu (float)	Combined undrained bulk modulus $B_c$ relating changes in pore pressure and volumetric strain, may be approximated by: $B_c \approx B_f/n$ where $B_f$ is the bulk modulus of fluid phase ( $2.2 \times 10^6$ kPa (or $3.191 \times 10^5$ psi) for water), and $n$ the initial porosity.
pa (float)	Optional atmospheric pressure for normalization (typically 101 kPa in SI units, or 14.65 psi in English units)

See also [notes](#)

### 1.4.15 section commands

#### **section** (*secType*, *secTag*, \**secArgs*)

This command is used to construct a SectionForceDeformation object, hereto referred to as Section, which represents force-deformation (or resultant stress-strain) relationships at beam-column and plate sample points.

<i>secType</i> (str)	section type
<i>secTag</i> (int)	section tag.
<i>secArgs</i> (list)	a list of section arguments, must be preceded with *

For example,

```
secType = 'Elastic'
secTag = 1
secArgs = [E, A, Iz]
section(secType, secTag, *secArgs)
```

The following contain information about available *secType*:

1. [Elastic Section](#)
2. [Fiber Section](#)
3. [NDFiber Section](#)
4. [Wide Flange Section](#)
5. [RC Section](#)
6. [RCCircular Section](#)
7. [Parallel Section](#)
8. [Section Aggregator](#)
9. [Uniaxial Section](#)
10. [Elastic Membrane Plate Section](#)
11. [Plate Fiber Section](#)
12. [Bidirectional Section](#)

13. *Isolator2spring Section*

14. *LayeredShell*

## Elastic Section

**section ('Elastic', secTag, E, A, Iz, G=0.0, alphaY=0.0)**

**section ('Elastic', secTag, E, A, Iz, Iy, G, J, alphaY=0.0, alphaZ=0.0)**

This command allows the user to construct an ElasticSection. The inclusion of shear deformations is optional. The dofs for 2D elastic section are [P, Mz], for 3D are [P, Mz, My, T].

secTag (int)	unique section tag
E (float)	Young's Modulus
A (float)	cross-sectional area of section
Iz (float)	second moment of area about the local z-axis
Iy (float)	second moment of area about the local y-axis (required for 3D analysis)
G (float)	Shear Modulus (optional for 2D analysis, required for 3D analysis)
J (float)	torsional moment of inertia of section (required for 3D analysis)
alphaY (float)	shear shape factor along the local y-axis (optional)
alphaZ (float)	shear shape factor along the local z-axis (optional)

---

**Note:** The elastic section can be used in the nonlinear beam column elements, which is useful in the initial stages of developing a complex model.

---

## Fiber Section

**section ('Fiber', secTag, '-GJ', GJ=0.0)**

This command allows the user to construct a FiberSection object. Each FiberSection object is composed of Fibers, with each fiber containing a UniaxialMaterial, an area and a location (y,z). The dofs for 2D section are [P, Mz], for 3D are [P, Mz, My, T].

secTag (int)	unique section tag
GJ (float)	linear-elastic torsional stiffness assigned to the section (optional)

**section ('FiberThermal', secTag, '-GJ', GJ=0.0)**

This command creates a FiberSectionThermal object. The dofs for 2D section are [P, Mz], for 3D are [P, Mz, My].

---

### Note:

1. The commands below should be called after the section command to generate all the fibers in the section.
  2. The patch and layer commands can be used to generate multiple fibers in a single command.
- 

## Fiber Command

**fiber (yloc, zloc, A, matTag)**

This command allows the user to construct a single fiber and add it to the enclosing FiberSection or NDFiberSection.

yloc (float)	y coordinate of the fiber in the section (local coordinate system)
zloc (float)	z coordinate of the fiber in the section (local coordinate system)
A (float)	cross-sectional area of fiber
matTag (int)	material tag associated with this fiber (UniaxialMaterial tag for a FiberSection and NDMaterial tag for use in an NDFiberSection).

## Patch Command

**patch** (*type*, \**args*)

The patch command is used to generate a number of fibers over a cross-sectional area. Currently there are three types of cross-section that fibers can be generated: quadrilateral, rectangular and circular.

**patch** ('quad', *matTag*, *numSubdivIJ*, *numSubdivJK*, \**crdsI*, \**crdsJ*, \**crdsK*, \**crdsL*)

This is the command to generate a quadrilateral shaped patch (the geometry of the patch is defined by four vertices: I J K L. The coordinates of each of the four vertices is specified in COUNTER CLOCKWISE sequence)

matTag (int)	material tag associated with this fiber (UniaxialMaterial tag for a FiberSection and NDMaterial tag for use in an NDFiberSection).
numSubdivIJ (int)	number of subdivisions (fibers) in the IJ direction.
numSubdivJK (int)	number of subdivisions (fibers) in the JK direction.
crdsI (list (float))	y & z-coordinates of vertex I (local coordinate system)
crdsJ (list (float))	y & z-coordinates of vertex J (local coordinate system)
crdsK (list (float))	y & z-coordinates of vertex K (local coordinate system)
crdsL (list (float))	y & z-coordinates of vertex L (local coordinate system)

**patch** ('rect', *matTag*, *numSubdivY*, *numSubdivZ*, \**crdsI*, \**crdsJ*)

This is the command to generate a rectangular patch. The geometry of the patch is defined by coordinates of vertices: I and J. The first vertex, I, is the bottom-left point and the second vertex, J, is the top-right point, having as a reference the local y-z plane.

matTag (int)	material tag associated with this fiber (UniaxialMaterial tag for a FiberSection and NDMaterial tag for use in an NDFiberSection).
numSubdivY (int)	number of subdivisions (fibers) in local y direction.
numSubdivZ (int)	number of subdivisions (fibers) in local z direction.
crdsI (list (float))	y & z-coordinates of vertex I (local coordinate system)
crdsJ (list (float))	y & z-coordinates of vertex J (local coordinate system)

**patch** ('circ', *matTag*, *numSubdivCirc*, *numSubdivRad*, \**center*, \**rad*, \**ang*)

This is the command to generate a circular shaped patch

matTag (int)	material tag associated with this fiber (UniaxialMaterial tag for a FiberSection and NDMaterial tag for use in an NDFiberSection).
numSubdivCirc (int)	number of subdivisions (fibers) in the circumferential direction (number of wedges)
numSubdivRad (int)	number of subdivisions (fibers) in the radial direction (number of rings)
center (list (float))	y & z-coordinates of the center of the circle
rad (list (float))	internal & external radius
ang (list (float))	starting & ending-coordinates angles (degrees)

## Layer Command

**layer** (*type*, \*args)

The layer command is used to generate a number of fibers along a line or a circular arc.

**layer** ('straight', *matTag*, *numFiber*, *areaFiber*, \**start*, \**end*)

This command is used to construct a straight line of fibers

matTag (int)	material tag associated with this fiber (UniaxialMaterial tag for a FiberSection and NDMaterial tag for use in an NDFiberSection).
numFiber (int)	number of fibers along line
areaFiber (float)	area of each fiber
start (list (float))	y & z-coordinates of first fiber in line (local coordinate system)
end (list (float))	y & z-coordinates of last fiber in line (local coordinate system)

**layer** ('circ', *matTag*, *numFiber*, *areaFiber*, \**center*, *radius*, \**ang*=[0.0,360.0-360/*numFiber*])

This command is used to construct a line of fibers along a circular arc

matTag (int)	material tag associated with this fiber (UniaxialMaterial tag for a FiberSection and NDMaterial tag for use in an NDFiberSection).
numFiber (int)	number of fibers along line
areaFiber (float)	area of each fiber
center (list (float))	y & z-coordinates of center of circular arc
radius (float)	radius of circular arc
ang (list (float))	starting and ending angle (optional)

## NDFiber Section

**section** ('NDFiber', *secTag*)

This command allows the user to construct an NDFiberSection object. Each NDFiberSection object is composed of NDFibers, with each fiber containing an NDMaterial, an area and a location (y,z). The NDFiberSection

works for 2D and 3D frame elements and it queries the NDMaterial of each fiber for its axial and shear stresses. In 2D, stress components 11 and 12 are obtained from each fiber in order to provide stress resultants for axial force, bending moment, and shear [ $P$ ,  $M_z$ ,  $V_y$ ]. Stress components 11, 12, and 13 lead to all six stress resultants in 3D [ $P$ ,  $M_z$ ,  $V_y$ ,  $M_y$ ,  $V_z$ ,  $T$ ].

The NDFiberSection works with any NDMaterial via wrapper classes that perform static condensation of the stress vector down to the 11, 12, and 13 components, or via concrete NDMaterial subclasses that implement the appropriate fiber stress conditions.

secTag (int)	unique section tag
--------------	--------------------

---

#### Note:

1. The commands below should be called after the section command to generate all the fibers in the section.
2. The patch and layer commands can be used to generate multiple fibers in a single command.

- 
1. `fiber()`
  2. `patch()`
  3. `layer()`

### Wide Flange Section

**section ('WFSection2d', secTag, matTag, d, tw, bf, tf, Nfw, Nff)**

This command allows the user to construct a WFSection2d object, which is an encapsulated fiber representation of a wide flange steel section appropriate for plane frame analysis.

secTag (int)	unique section tag
matTag (int)	tag of uniaxialMaterial assigned to each fiber
d (float)	section depth
tw (float)	web thickness
bf (float)	flange width
tf (float)	flange thickness
Nfw (float)	number of fibers in the web
Nff (float)	number of fibers in each flange

---

**Note:** The section dimensions `d`, `tw`, `bf`, and `tf` can be found in the AISC steel manual.

---

### RC Section

**section ('RCSection2d', secTag, coreTag, coverTag, steelTag, d, b, cover, Atop, Abot, Aside, Nfcore, Nfcover, Nfs)**

This command allows the user to construct an RCSection2d object, which is an encapsulated fiber representation of a rectangular reinforced concrete section with core and confined regions of concrete and single top and bottom layers of reinforcement appropriate for plane frame analysis.

secTag (int)	unique section tag
coreTag (int)	tag of uniaxialMaterial assigned to each fiber in the core region
coverTag (int)	tag of uniaxialMaterial assigned to each fiber in the cover region
steelTag (int)	tag of uniaxialMaterial assigned to each reinforcing bar
d (float)	section depth
b (float)	section width
cover (float)	cover depth (assumed uniform around perimeter)
Atop (float)	area of reinforcing bars in top layer
Abot (float)	area of reinforcing bars in bottom layer
Aside (float)	area of reinforcing bars on intermediate layers
Nfcore (float)	number of fibers through the core depth
Nfcover (float)	number of fibers through the cover depth
Nfs (float)	number of bars on the top and bottom rows of reinforcement (Nfs-2 bars will be placed on the side rows)

---

**Note:** For more general reinforced concrete section definitions, use the Fiber Section command.

---

## RCCircular Section

**section ('RCCircularSection', secTag, coreTag, coverTag, steelTag, d, cover, As, NringsCore, NringsCover, Newedges, Nsteel)**

This command allows the user to construct an RCCircularSection object, which is an encapsulated fiber representation of a circular reinforced concrete section with core and confined regions of concrete.

secTag (int)	unique section tag
coreTag (int)	tag of uniaxialMaterial assigned to each fiber in the core region
coverTag (int)	tag of uniaxialMaterial assigned to each fiber in the cover region
steelTag (int)	tag of uniaxialMaterial assigned to each reinforcing bar
d (float)	section radius
cover (float)	cover depth (assumed uniform around perimeter)
As (float)	area of reinforcing bars
NringsCore (int)	number of fibers through the core depth
NringsCover (int)	number of fibers through the cover depth
Newedges (int)	number of fibers through the edges
Nsteel (int)	number of fibers through the steels

---

**Note:** For more general reinforced concrete section definitions, use the Fiber Section command.

---

## Parallel Section

**section ('Parallel', secTag, \*tags)**

Connect sections in parallel.

secTag (int)	unique section tag
tags (list (int))	tags of predefined sections.

## Section Aggregator

**section ('Aggregator', secTag, \*mats, '-section', sectionTag)**

This command is used to construct a SectionAggregator object which aggregates groups previously-defined UniaxialMaterial objects into a single section force-deformation model. Each UniaxialMaterial object represents the section force-deformation response for a particular section degree-of-freedom (dof). There is no interaction between responses in different dof directions. The aggregation can include one previously defined section.

secTag (int)	unique section tag
mats (list)	<p>list of tags and dofs of previously-defined UniaxialMaterial objects, mats = [matTag1, dof1, matTag2, dof2, ...]</p> <p>the force-deformation quantity to be modeled by this section object. One of the following section dof may be used:</p> <ul style="list-style-type: none"> <li>• 'P' Axial force-deformation</li> <li>• 'Mz' Moment-curvature about section local z-axis</li> <li>• 'Vy' Shear force-deformation along section local y-axis</li> <li>• 'My' Moment-curvature about section local y-axis</li> <li>• 'Vz' Shear force-deformation along section local z-axis</li> <li>• 'T' Torsion Force-Deformation</li> </ul>
sectionTag (int)	tag of previously-defined Section object to which the UniaxialMaterial objects are aggregated as additional force-deformation relationships (optional)

## Uniaxial Section

**section ('Uniaxial', secTag, matTag, quantity)**

This command is used to construct a UniaxialSection object which uses a previously-defined UniaxialMaterial object to represent a single section force-deformation response quantity.

secTag ( <a href="#">int</a> )	unique section tag
matTag ( <a href="#">int</a> )	tag of uniaxial material
quantity ( <a href="#">str</a> )	the force-deformation quantity to be modeled by this section object. One of the following section dof may be used: <ul style="list-style-type: none"><li>• 'P' Axial force-deformation</li><li>• 'Mz' Moment-curvature about section local z-axis</li><li>• 'Vy' Shear force-deformation along section local y-axis</li><li>• 'My' Moment-curvature about section local y-axis</li><li>• 'Vz' Shear force-deformation along section local z-axis</li><li>• 'T' Torsion Force-Deformation</li></ul>

## Elastic Membrane Plate Section

**section ('ElasticMembranePlateSection', secTag, E, nu, h, rho)**

This command allows the user to construct an ElasticMembranePlateSection object, which is an isotropic section appropriate for plate and shell analysis.

secTag ( <a href="#">int</a> )	unique section tag
E ( <a href="#">float</a> )	Young's Modulus
nu ( <a href="#">float</a> )	Poisson's Ratio
h ( <a href="#">float</a> )	depth of section
rho ( <a href="#">float</a> )	mass density

## Plate Fiber Section

**section ('PlateFiber', secTag, matTag, h)**

This command allows the user to construct a MembranePlateFiberSection object, which is a section that numerically integrates through the plate thickness with "fibers" and is appropriate for plate and shell analysis.

secTag ( <a href="#">int</a> )	unique section tag
matTag ( <a href="#">int</a> )	nDMaterial tag to be assigned to each fiber
h ( <a href="#">float</a> )	plate thickness

## Bidirectional Section

**section ('Bidirectional', secTag, E, Fy, Hiso, Hkin, code1='Vy', code2='P')**

This command allows the user to construct a Bidirectional section, which is a stress-resultant plasticity model of two coupled forces. The yield surface is circular and there is combined isotropic and kinematic hardening.

<code>sectag (int)</code>	unique section tag
<code>E (float)</code>	elastic modulus
<code>Fy (float)</code>	yield force
<code>Hiso (float)</code>	isotropic hardening modulus
<code>Hkin (float)</code>	kinematic hardening modulus
<code>code1 (str)</code>	section force code for direction 1 (optional)
<code>code2 (str)</code>	section force code for direction 2 (optional) One of the following section code may be used: <ul style="list-style-type: none"><li>• 'P' Axial force-deformation</li><li>• 'Mz' Moment-curvature about section local z-axis</li><li>• 'Vy' Shear force-deformation along section local y-axis</li><li>• 'My' Moment-curvature about section local y-axis</li><li>• 'Vz' Shear force-deformation along section local z-axis</li><li>• 'T' Torsion Force-Deformation</li></ul>

## Isolator2spring Section

**section ('Iso2spring', matTag, tol, k1, Fyo, k2o, kvo, hb, PE, Po=0.0)**

This command is used to construct an Isolator2spring section object, which represents the buckling behavior of an elastomeric bearing for two-dimensional analysis in the lateral and vertical plane. An Isolator2spring section represents the resultant force-deformation behavior of the bearing, and should be used with a zeroLengthSection element. The bearing should be constrained against rotation.

<code>sectag (int)</code>	unique section tag
<code>tol (float)</code>	tolerance for convergence of the element state. Suggested value: E-12 to E-10. OpenSees will warn if convergence is not achieved, however this usually does not prevent global convergence.
<code>k1 (float)</code>	initial stiffness for lateral force-deformation
<code>Fyo (float)</code>	nominal yield strength for lateral force-deformation
<code>k2o (float)</code>	nominal postyield stiffness for lateral force-deformation
<code>kvo (float)</code>	nominal stiffness in the vertical direction
<code>hb (float)</code>	total height of elastomeric bearing
<code>PE (float)</code>	Euler Buckling load for the bearing
<code>Po (float)</code>	axial load at which nominal yield strength is achieved (optional)

## LayeredShell

**nDMaterial ('LayeredShell', sectionTag, nLayers \*mats)**

This command will create the section of the multi-layer shell element, including the multi-dimensional concrete,

reinforcement material and the corresponding thickness.

sectionTag ( <a href="#">int</a> )	unique tag among sections
nLayers ( <a href="#">int</a> )	total numbers of layers
mats ( <a href="#">list</a> )	a list of material tags and thickenss, <code>[[mat1,thk1], ..., [mat2,thk2]]</code>

## 1.4.16 frictionModel commands

### **frictionModel** (*frnType*, *frnTag*, \**frnArgs*)

The frictionModel command is used to construct a friction model object, which specifies the behavior of the coefficient of friction in terms of the absolute sliding velocity and the pressure on the contact area. The command has at least one argument, the friction model type.

frnType ( <a href="#">str</a> )	frictionModel type
frnTag ( <a href="#">int</a> )	frictionModel tag.
frnArgs ( <a href="#">list</a> )	a list of frictionModel arguments, must be preceded with <code>*</code> .

For example,

```
frnType = 'Coulomb'  
frnTag = 1  
frnArgs = [mu]  
frictionModel(frnType, frnTag, *frnArgs)
```

The following contain information about available *frnType*:

1. [Coulomb](#)
2. [Velocity Dependent Friction](#)
3. [Velocity and Normal Force Dependent Friction](#)
4. [Velocity and Pressure Dependent Friction](#)
5. [Multi-Linear Velocity Dependent Friction](#)

### Coulomb

#### **frictionModel** ('Coulomb', *frnTag*, *mu*)

This command is used to construct a Coulomb friction model object. Coulomb's Law of Friction states that kinetic friction is independent of the sliding velocity.

frnTag ( <a href="#">int</a> )	unique friction model tag
mu ( <a href="#">float</a> )	coefficient of friction

### Velocity Dependent Friction

#### **frictionModel** ('VelDependent', *frnTag*, *muSlow*, *muFast*, *transRate*)

This command is used to construct a VelDependent friction model object. It is useful for modeling the behavior of PTFE or PTFE-like materials sliding on a stainless steel surface. For a detailed presentation on the velocity dependence of such interfaces please refer to Constantinou et al. (1999).

<code>frnTag (int)</code>	unique friction model tag
<code>muSlow (float)</code>	coefficient of friction at low velocity
<code>muFast (float)</code>	coefficient of friction at high velocity
<code>transRate (float)</code>	transition rate from low to high velocity

$$\mu = \mu_{fast} - (\mu_{fast} - \mu_{slow}) \cdot e^{-transRate \cdot |v|}$$

## REFERENCE:

Constantinou, M.C., Tsopelas, P., Kasalanati, A., and Wolff, E.D. (1999). "Property modification factors for seismic isolation bearings". Report MCEER-99-0012, Multidisciplinary Center for Earthquake Engineering Research, State University of New York.

**Velocity and Normal Force Dependent Friction**

**frictionModel ('VelNormalFrcDep',frnTag,aSlow,nSlow,aFast,nFast,alpha0,alpha1,alpha2,maxMuFact)**

This command is used to construct a VelNormalFrcDep friction model object.

<code>frnTag (int)</code>	unique friction model tag
<code>aSlow (float)</code>	constant for coefficient of friction at low velocity
<code>nSlow (float)</code>	exponent for coefficient of friction at low velocity
<code>aFast (float)</code>	constant for coefficient of friction at high velocity
<code>nFast (float)</code>	exponent for coefficient of friction at high velocity
<code>alpha0 (float)</code>	constant rate parameter coefficient
<code>alpha1 (float)</code>	linear rate parameter coefficient
<code>alpha2 (float)</code>	quadratic rate parameter coefficient
<code>maxMuFact (float)</code>	factor for determining the maximum coefficient of friction. This value prevents the friction coefficient from exceeding an unrealistic maximum value when the normal force becomes very small. The maximum friction coefficient is determined from $\mu_{Fast}$ , for example $\mu \leq maxMuFac * Fast$ .

**Velocity and Pressure Dependent Friction**

**frictionModel ('VelPressureDep',frnTag,muSlow,muFast0,A,deltaMu,alpha,transRate)**

This command is used to construct a VelPressureDep friction model object.

frnTag (int)	unique friction model tag
muSlow (float)	coefficient of friction at low velocity
muFast0 (float)	initial coefficient of friction at high velocity
A (float)	nominal contact area
deltaMu (float)	pressure parameter calibrated from experimental data
alpha (float)	pressure parameter calibrated from experimental data
transRate (float)	transition rate from low to high velocity

## Multi-Linear Velocity Dependent Friction

**frictionModel** ('VelDepMultiLinear',frnTag, '-vel', \*velocityPoints, '-frn', \*frictionPoints)

This command is used to construct a VelDepMultiLinear friction model object. The friction-velocity relationship is given by a multi-linear curve that is defined by a set of points. The slope given by the last two specified points on the positive velocity axis is extrapolated to infinite positive velocities. Velocity and friction points need to be equal or larger than zero (no negative values should be defined). The number of provided velocity points needs to be equal to the number of provided friction points.

frnTag (int)	unique friction model tag
velocityPoints (list (float))	list of velocity points along friction-velocity curve
frictionPoints (list (float))	list of friction points along friction-velocity curve

## 1.4.17 geomTransf commands

**geomTransf** (transfType, transfTag, \*transfArgs)

The geometric-transformation command is used to construct a coordinate-transformation (CrdTransf) object, which transforms beam element stiffness and resisting force from the basic system to the global-coordinate system. The command has at least one argument, the transformation type.

transfType (str)	geomTransf type
transfTag (int)	geomTransf tag.
transfArgs (list)	a list of geomTransf arguments, must be preceded with *.

For example,

```
transfType = 'Linear'
transfTag = 1
transfArgs = []
geomTransf(transfType, transfTag, *transfArgs)
```

The following contain information about available transfType:

1. *Linear Transformation*
2. *PDelta Transformation*
3. *Corotational Transformation*

### Linear Transformation

**geomTransf** ('Linear', transfTag, '-jntOffset', \*dI, \*dJ)

**geomTransf ('Linear', transfTag, \*vecxz, '-jntOffset', \*dI, \*dJ)**

This command is used to construct a linear coordinate transformation (LinearCrdTransf) object, which performs a linear geometric transformation of beam stiffness and resisting force from the basic system to the global-coordinate system.

<code>trans</code>	integer tag identifying transformation (int)
<code>vecxz</code>	X, Y, and Z components of vecxz, the vector used to define the local x-z plane of the local-coordinate system. The local y-axis is defined by taking the cross product of the vecxz vector and the x-axis. (list (float))
<code>dI</code>	These components are specified in the global-coordinate system X,Y,Z and define a vector that is in a plane parallel to the x-z plane of the local-coordinate system. These items need to be specified for the three-dimensional problem. (list (float))
<code>dJ</code>	joint offset values – offsets specified with respect to the global coordinate system for element-end node i (the number of arguments depends on the dimensions of the current model). (list (float))
<code>dJ</code>	joint offset values – offsets specified with respect to the global coordinate system for element-end node j (the number of arguments depends on the dimensions of the current model). (list (float))

**PDelta Transformation****geomTransf ('PDelta', transfTag, '-jntOffset', \*dI, \*dJ)****geomTransf ('PDelta', transfTag, \*vecxz, '-jntOffset', \*dI, \*dJ)**

This command is used to construct the P-Delta Coordinate Transformation (PDeltaCrdTransf) object, which performs a linear geometric transformation of beam stiffness and resisting force from the basic system to the global coordinate system, considering second-order P-Delta effects.

<code>trans</code>	integer tag identifying transformation (int)
<code>vecxz</code>	X, Y, and Z components of vecxz, the vector used to define the local x-z plane of the local-coordinate system. The local y-axis is defined by taking the cross product of the vecxz vector and the x-axis. (list (float))
<code>dI</code>	These components are specified in the global-coordinate system X,Y,Z and define a vector that is in a plane parallel to the x-z plane of the local-coordinate system. These items need to be specified for the three-dimensional problem. (list (float))
<code>dJ</code>	joint offset values – offsets specified with respect to the global coordinate system for element-end node i (the number of arguments depends on the dimensions of the current model). (list (float))
<code>dJ</code>	joint offset values – offsets specified with respect to the global coordinate system for element-end node j (the number of arguments depends on the dimensions of the current model). (list (float))

---

**Note:** P LARGE Delta effects do not include P small delta effects.

---

**Corotational Transformation****geomTransf ('Corotational', transTag, '-jntOffset', \*dI, \*dJ)****geomTransf ('Corotational', transTag, \*vecxz)**

This command is used to construct the Corotational Coordinate Transformation (CorotCrdTransf) object. Corotational transformation can be used in large displacement-small strain problems.

trans	integer tag identifying transformation (int)
vecxz	X, Y, and Z components of vecxz, the vector used to define the local x-z plane of the local-coordinate system. The local y-axis is defined by taking the cross product of the vecxz vector and the x-axis. (list (float)) These components are specified in the global-coordinate system X,Y,Z and define a vector that is in a plane parallel to the x-z plane of the local-coordinate system. These items need to be specified for the three-dimensional problem.
dI	joint offset values – offsets specified with respect to the global coordinate system for element-end node i (the number of arguments depends on the dimensions of the current model). (list (float))
dJ	joint offset values – offsets specified with respect to the global coordinate system for element-end node j (the number of arguments depends on the dimensions of the current model). (list (float))

---

**Note:** Currently the transformation does not deal with element loads and will ignore any that are applied to the element.

---

## 1.5 Analysis Commands

In OpenSees, an analysis is an object which is composed by the aggregation of component objects. It is the component objects which define the type of analysis that is performed on the model. The component classes, as shown in the figure below, consist of the following:

1. ConstraintHandler – determines how the constraint equations are enforced in the analysis – how it handles the boundary conditions/imposed displacements
2. DOF\_Numberer – determines the mapping between equation numbers and degrees-of-freedom
3. Integrator – determines the predictive step for time  $t+dt$
4. SolutionAlgorithm – determines the sequence of steps taken to solve the non-linear equation at the current time step
5. SystemOfEqn/Solver – within the solution algorithm, it specifies how to store and solve the system of equations in the analysis
6. Convergence Test – determines when convergence has been achieved.

Analysis commands

1. *constraints commands*
2. *numberer commands*
3. *system commands*
4. *test commands*
5. *algorithm commands*
6. *integrator commands*
7. *analysis command*
8. *eigen command*
9. *analyze command*

### 1.5.1 constraints commands

#### **constraints** (*constraintType*, \**constraintArgs*)

This command is used to construct the ConstraintHandler object. The ConstraintHandler object determines how the constraint equations are enforced in the analysis. Constraint equations enforce a specified value for a DOF, or a relationship between DOFs.

<i>constraintType</i> ( <i>str</i> )	constraints type
<i>constraintArgs</i> ( <i>list</i> )	a list of constraints arguments

The following contain information about available *constraintType*:

1. *Plain Constraints*
2. *Lagrange Multipliers*
3. *Penalty Method*
4. *Transformation Method*

#### Plain Constraints

##### **constraints** ('Plain')

This command is used to construct a Plain constraint handler. A plain constraint handler can only enforce homogeneous single point constraints (fix command) and multi-point constraints constructed where the constraint matrix is equal to the identity (equalDOF command). The following is the command to construct a plain constraint handler:

---

**Note:** As mentioned, this constraint handler can only enforce homogeneous single point constraints (fix command) and multi-point constraints where the constraint matrix is equal to the identity (equalDOF command).

---

#### Lagrange Multipliers

##### **constraints** ('Lagrange', *alphaS*=1.0, *alphaM*=1.0)

This command is used to construct a LagrangeMultiplier constraint handler, which enforces the constraints by introducing Lagrange multiplies to the system of equation. The following is the command to construct a plain constraint handler:

<i>alphaS</i> ( <i>float</i> )	$\alpha_S$ factor on single points.
<i>alphaM</i> ( <i>float</i> )	$\alpha_M$ factor on multi-points.

---

**Note:** The Lagrange multiplier method introduces new unknowns to the system of equations. The diagonal part of the system corresponding to these new unknowns is 0.0. This ensure that the system IS NOT symmetric positive definite.

---

#### Penalty Method

##### **constraints** ('Penalty', *alphaS*=1.0, *alphaM*=1.0)

This command is used to construct a Penalty constraint handler, which enforces the constraints using the penalty method. The following is the command to construct a penalty constraint handler:

alphaS (float)	$\alpha_S$ factor on single points.
alphaM (float)	$\alpha_M$ factor on multi-points.

---

**Note:** The degree to which the constraints are enforced is dependent on the penalty values chosen. Problems can arise if these values are too small (constraint not enforced strongly enough) or too large (problems associated with conditioning of the system of equations).

---

## Transformation Method

### **constraints ('Transformation')**

This command is used to construct a transformation constraint handler, which enforces the constraints using the transformation method. The following is the command to construct a transformation constraint handler

---

#### Note:

- The single-point constraints when using the transformation method are done directly. The matrix equation is not manipulated to enforce them, rather the trial displacements are set directly at the nodes at the start of each analysis step.
- Great care must be taken when multiple constraints are being enforced as the transformation method does not follow constraints:
  1. If a node is fixed, constrain it with the fix command and not equalDOF or other type of constraint.
  2. If multiple nodes are constrained, make sure that the retained node is not constrained in any other constraint.

And remember if a node is constrained to multiple nodes in your model it probably means you have messed up.

---

## 1.5.2 numberer commands

### **numberer (numbererType, \*numbererArgs)**

This command is used to construct the DOF\_Numberer object. The DOF\_Numberer object determines the mapping between equation numbers and degrees-of-freedom – how degrees-of-freedom are numbered.

numbererType (str)	numberer type
numbererArgs (list)	a list of numberer arguments

The following contain information about available numbererType:

1. *Plain Numberer*
2. *RCM Numberer*
3. *AMD Numberer*
4. *Parallel Plain Numberer*
5. *Parallel RCM Numberer*

## Plain Numberer

**numberer ('Plain')**

This command is used to construct a Plain degree-of-freedom numbering object to provide the mapping between the degrees-of-freedom at the nodes and the equation numbers. A Plain numberer just takes whatever order the domain gives it nodes and numbers them, this ordering is both dependent on node numbering and size of the model.

---

**Note:** For very small problems and for the sparse matrix solvers which provide their own numbering scheme, order is not really important so plain numberer is just fine. For large models and analysis using solver types other than the sparse solvers, the order will have a major impact on performance of the solver and the plain handler is a poor choice.

---

## RCM Numberer

**numberer ('RCM')**

This command is used to construct an RCM degree-of-freedom numbering object to provide the mapping between the degrees-of-freedom at the nodes and the equation numbers. An RCM numberer uses the reverse Cuthill-McKee scheme to order the matrix equations.

## AMD Numberer

**numberer ('AMD')**

This command is used to construct an AMD degree-of-freedom numbering object to provide the mapping between the degrees-of-freedom at the nodes and the equation numbers. An AMD numberer uses the approximate minimum degree scheme to order the matrix equations.

## Parallel Plain Numberer

**numberer ('ParallelPlain')**

This command is used to construct a parallel version of Plain degree-of-freedom numbering object to provide the mapping between the degrees-of-freedom at the nodes and the equation numbers. A Plain numberer just takes whatever order the domain gives it nodes and numbers them, this ordering is both dependent on node numbering and size of the model.

Use this command only for parallel model.

**Warning:** Don't use this command if model is not parallel, for example, parametric study.

## Parallel RCM Numberer

**numberer ('ParallelRCM')**

This command is used to construct a parallel version of RCM degree-of-freedom numbering object to provide the mapping between the degrees-of-freedom at the nodes and the equation numbers. A Plain numberer just takes whatever order the domain gives it nodes and numbers them, this ordering is both dependent on node numbering and size of the model.

Use this command only for parallel model.

**Warning:** Don't use this command if model is not parallel, for example, parametric study.

### 1.5.3 system commands

**system**(*systemType*, \**systemArgs*)

This command is used to construct the LinearSOE and LinearSolver objects to store and solve the system of equations in the analysis.

<i>systemType</i> ( <a href="#">str</a> )	system type
<i>systemArgs</i> ( <a href="#">list</a> )	a list of system arguments

The following contain information about available *systemType*:

1. [BandGeneral SOE](#)
2. [BandSPD SOE](#)
3. [ProfileSPD SOE](#)
4. [SuperLU SOE](#)
5. [UmfPack SOE](#)
6. [FullGeneral SOE](#)
7. [SparseSYM SOE](#)
8. [PFEM SOE](#)
9. [MUMPS Solver](#)

#### BandGeneral SOE

**system**('BandGen')

This command is used to construct a BandGeneralSOE linear system of equation object. As the name implies, this class is used for matrix systems which have a banded profile. The matrix is stored as shown below in a 1dimensional array of size equal to the bandwidth times the number of unknowns. When a solution is required, the Lapack routines DGBSV and SGBTRS are used.

#### BandSPD SOE

**system**('BandSPD')

This command is used to construct a BandSPDSOE linear system of equation object. As the name implies, this class is used for symmetric positive definite matrix systems which have a banded profile. The matrix is stored as shown below in a 1 dimensional array of size equal to the (bandwidth/2) times the number of unknowns. When a solution is required, the Lapack routines DPBSV and DPBTRS are used.

#### ProfileSPD SOE

**system**('ProfileSPD')

This command is used to construct a profileSPDSOE linear system of equation object. As the name implies, this class is used for symmetric positive definite matrix systems. The matrix is stored as shown below in a 1 dimensional array with only those values below the first non-zero row in any column being stored. This is sometimes also referred to as a skyline storage scheme.

## SuperLU SOE

**system**('SuperLU')

This command is used to construct a SparseGEN linear system of equation object. As the name implies, this class is used for sparse matrix systems. The solution of the sparse matrix is carried out using SuperLU.

## UmfPack SOE

**system**('UmfPack')

This command is used to construct a sparse system of equations which uses the [UmfPack](#) solver.

## FullGeneral SOE

**system**('FullGeneral')

This command is used to construct a Full General linear system of equation object. As the name implies, the class utilizes NO space saving techniques to cut down on the amount of memory used. If the matrix is of size, nxn, then storage for an nxn array is sought from memory when the program runs. When a solution is required, the Lapack routines DGESV and DGETRS are used.

---

**Note:** This type of system should almost never be used! This is because it requires a lot more memory than every other solver and takes more time in the actual solving operation than any other solver. It is required if the user is interested in looking at the global system matrix.

---

## SparseSYM SOE

**system**('SparseSYM')

This command is used to construct a sparse symmetric system of equations which uses a row-oriented solution method in the solution phase.

## MUMPS Solver

**system**('Mumps', '-ICNTL14', icntl14=20.0, '-ICNTL7', icntl7=7)

Create a system of equations using the Mumps solver

icntl14	controls the percentage increase in the estimated working space (optional)
icntl7	computes a symmetric permutation (ordering) to determine the pivot order to be used for the factorization in case of sequential analysis (optional) <ul style="list-style-type: none"><li>• 0: AMD</li><li>• 1: set by user</li><li>• 2: AMF</li><li>• 3: SCOTCH</li><li>• 4: PORD</li><li>• 5: Metis</li><li>• 6: AMD with QADM</li><li>• 7: automatic</li></ul>

Use this command only for parallel model.

**Warning:** Don't use this command if model is not parallel, for example, parametric study.

#### 1.5.4 test commands

##### **test** (*testType*, \**testArgs*)

This command is used to construct the LinearSOE and LinearSolver objects to store and solve the test of equations in the analysis.

<i>testType</i> ( <a href="#">str</a> )	test type
<i>testArgs</i> ( <a href="#">list</a> )	a list of test arguments

The following contain information about available *testType*:

1. [\*NormUnbalance\*](#)
2. [\*NormDispIncr\*](#)
3. [\*energyIncr\*](#)
4. [\*RelativeNormUnbalance\*](#)
5. [\*RelativeNormDispIncr\*](#)
6. [\*RelativeTotalNormDispIncr\*](#)
7. [\*RelativeEnergyIncr\*](#)
8. [\*FixedNumIter\*](#)
9. [\*NormDispAndUnbalance\*](#)
10. [\*NormDispOrUnbalance\*](#)
11. [\*PFEM test\*](#)

##### **NormUnbalance**

###### **test** ('*NormUnbalance*', *tol*, *iter*, *pFlag=0*, *nType=2*, *maxincr=-1*)

Create a NormUnbalance test, which uses the norm of the right hand side of the matrix equation to determine if convergence has been reached.

<code>tol (float)</code>	Tolerance criteria used to check for convergence.
<code>iter (int)</code>	Max number of iterations to check
<code>pFlag (int)</code>	Print flag (optional): <ul style="list-style-type: none"> <li>• 0 print nothing.</li> <li>• 1 print information on norms each time <code>test()</code> is invoked.</li> <li>• 2 print information on norms and number of iterations at end of successful test.</li> <li>• 4 at each step it will print the norms and also the <math>\Delta U</math> and <math>R(U)</math> vectors.</li> <li>• 5 if it fails to converge at end of <code>numIter</code> it will print an error message <b>but return a successfull test.</b></li> </ul>
<code>nType (int)</code>	Type of norm, (0 = max-norm, 1 = 1-norm, 2 = 2-norm). (optional)
<code>maxincr (int)</code>	Maximum times of error increasing. (optional)

When using the Penalty method additional large forces to enforce the penalty functions exist on the right hand side, making convergence using this test usually impossible (even though solution might have converged).

### NormDispIncr

`test ('NormDispIncr', tol, iter, pFlag=0, nType=2)`

Create a NormUnbalance test, which uses the norm of the left hand side solution vector of the matrix equation to determine if convergence has been reached.

<code>tol (float)</code>	Tolerance criteria used to check for convergence.
<code>iter (int)</code>	Max number of iterations to check
<code>pFlag (int)</code>	Print flag (optional): <ul style="list-style-type: none"> <li>• 0 print nothing.</li> <li>• 1 print information on norms each time <code>test()</code> is invoked.</li> <li>• 2 print information on norms and number of iterations at end of successful test.</li> <li>• 4 at each step it will print the norms and also the <math>\Delta U</math> and <math>R(U)</math> vectors.</li> <li>• 5 if it fails to converge at end of <code>numIter</code> it will print an error message <b>but return a successfull test.</b></li> </ul>
<code>nType (int)</code>	Type of norm, (0 = max-norm, 1 = 1-norm, 2 = 2-norm). (optional)

When using the Lagrange method to enforce the constraints, the Lagrange multipliers appear in the solution vector.

### energyIncr

`test ('EnergyIncr', tol, iter, pFlag=0, nType=2)`

Create a EnergyIncr test, which uses the dot product of the solution vector and norm of the right hand side of

the matrix equation to determine if convergence has been reached.

<code>tol</code> (float)	Tolerance criteria used to check for convergence.
<code>iter</code> (int)	Max number of iterations to check
<code>pFlag</code> (int)	Print flag (optional): <ul style="list-style-type: none"><li>• 0 print nothing.</li><li>• 1 print information on norms each time <code>test()</code> is invoked.</li><li>• 2 print information on norms and number of iterations at end of successful test.</li><li>• 4 at each step it will print the norms and also the <math>\Delta U</math> and <math>R(U)</math> vectors.</li><li>• 5 if it fails to converge at end of <code>numIter</code> it will print an error message <b>but return a successfull test.</b></li></ul>
<code>nType</code> (int)	Type of norm, (0 = max-norm, 1 = 1-norm, 2 = 2-norm). (optional)

- When using the Penalty method additional large forces to enforce the penalty functions exist on the right hand side, making convergence using this test usually impossible (even though solution might have converged).
- When using the Lagrange method to enforce the constraints, the Lagrange multipliers appear in the solution vector.

## RelativeNormUnbalance

`test ('RelativeNormUnbalance', tol, iter, pFlag=0, nType=2)`

Create a RelativeNormUnbalance test, which uses the relative norm of the right hand side of the matrix equation to determine if convergence has been reached.

<code>tol</code> (float)	Tolerance criteria used to check for convergence.
<code>iter</code> (int)	Max number of iterations to check
<code>pFlag</code> (int)	Print flag (optional): <ul style="list-style-type: none"><li>• 0 print nothing.</li><li>• 1 print information on norms each time <code>test()</code> is invoked.</li><li>• 2 print information on norms and number of iterations at end of successful test.</li><li>• 4 at each step it will print the norms and also the <math>\Delta U</math> and <math>R(U)</math> vectors.</li><li>• 5 if it fails to converge at end of <code>numIter</code> it will print an error message <b>but return a successfull test.</b></li></ul>
<code>nType</code> (int)	Type of norm, (0 = max-norm, 1 = 1-norm, 2 = 2-norm). (optional)

- When using the Penalty method additional large forces to enforce the penalty functions exist on the right hand side, making convergence using this test usually impossible (even though solution might have converged).

## RelativeNormDispIncr

**test** ('RelativeNormDispIncr', tol, iter, pFlag=0, nType=2)

Create a RelativeNormDispIncr test, which uses the relative of the solution vector of the matrix equation to determine if convergence has been reached.

tol (float)	Tolerance criteria used to check for convergence.
iter (int)	Max number of iterations to check
pFlag (int)	Print flag (optional): <ul style="list-style-type: none"> <li>• 0 print nothing.</li> <li>• 1 print information on norms each time <code>test()</code> is invoked.</li> <li>• 2 print information on norms and number of iterations at end of successful test.</li> <li>• 4 at each step it will print the norms and also the <math>\Delta U</math> and <math>R(U)</math> vectors.</li> <li>• 5 if it fails to converge at end of <code>numIter</code> it will print an error message <b>but return a successfull test.</b></li> </ul>
nType (int)	Type of norm, (0 = max-norm, 1 = 1-norm, 2 = 2-norm). (optional)

## RelativeTotalNormDispIncr

**test** ('relativeTotalNormDispIncr', tol, iter, pFlag=0, nType=2)

Create a RelativeTotalNormDispIncr test, which uses the ratio of the current norm to the total norm (the sum of all the norms since last convergence) of the solution vector.

tol (float)	Tolerance criteria used to check for convergence.
iter (int)	Max number of iterations to check
pFlag (int)	Print flag (optional): <ul style="list-style-type: none"> <li>• 0 print nothing.</li> <li>• 1 print information on norms each time <code>test()</code> is invoked.</li> <li>• 2 print information on norms and number of iterations at end of successful test.</li> <li>• 4 at each step it will print the norms and also the <math>\Delta U</math> and <math>R(U)</math> vectors.</li> <li>• 5 if it fails to converge at end of <code>numIter</code> it will print an error message <b>but return a successfull test.</b></li> </ul>
nType (int)	Type of norm, (0 = max-norm, 1 = 1-norm, 2 = 2-norm). (optional)

## RelativeEnergyIncr

**test** ('RelativeEnergyIncr', tol, iter, pFlag=0, nType=2)

Create a RelativeEnergyIncr test, which uses the relative dot product of the solution vector and norm of the right hand side of the matrix equation to determine if convergence has been reached.

tol (float)	Tolerance criteria used to check for convergence.
iter (int)	Max number of iterations to check
pFlag (int)	Print flag (optional): <ul style="list-style-type: none"><li>• 0 print nothing.</li><li>• 1 print information on norms each time <code>test()</code> is invoked.</li><li>• 2 print information on norms and number of iterations at end of successful test.</li><li>• 4 at each step it will print the norms and also the <math>\Delta U</math> and <math>R(U)</math> vectors.</li><li>• 5 if it fails to converge at end of <code>numIter</code> it will print an error message <b>but return a successfull test.</b></li></ul>
nType (int)	Type of norm, (0 = max-norm, 1 = 1-norm, 2 = 2-norm). (optional)

## FixedNumIter

**test ('FixedNumIter', iter, pFlag=0, nType=2)**

Create a FixedNumIter test, that performs a fixed number of iterations without testing for convergence.

tol (float)	Tolerance criteria used to check for convergence.
iter (int)	Max number of iterations to check
pFlag (int)	Print flag (optional): <ul style="list-style-type: none"><li>• 0 print nothing.</li><li>• 1 print information on norms each time <code>test()</code> is invoked.</li><li>• 2 print information on norms and number of iterations at end of successful test.</li><li>• 4 at each step it will print the norms and also the <math>\Delta U</math> and <math>R(U)</math> vectors.</li><li>• 5 if it fails to converge at end of <code>numIter</code> it will print an error message <b>but return a successfull test.</b></li></ul>
nType (int)	Type of norm, (0 = max-norm, 1 = 1-norm, 2 = 2-norm). (optional)

## NormDispAndUnbalance

**test ('NormDispAndUnbalance', tolIncr, tolR, iter, pFlag=0, nType=2, maxincr=-1)**

Create a NormDispAndUnbalance test, which check if both 'NormUnbalance' and 'NormDispIncr' are converged.

<code>tolIncr (float)</code>	Tolerance for left hand solution increments
<code>tolIncr (float)</code>	Tolerance for right hand residual
<code>iter (int)</code>	Max number of iterations to check
<code>pFlag (int)</code>	Print flag (optional): <ul style="list-style-type: none"> <li>• 0 print nothing.</li> <li>• 1 print information on norms each time <code>test()</code> is invoked.</li> <li>• 2 print information on norms and number of iterations at end of successful test.</li> <li>• 4 at each step it will print the norms and also the <math>\Delta U</math> and <math>R(U)</math> vectors.</li> <li>• 5 if it fails to converge at end of <code>numIter</code> it will print an error message <b>but return a successfull test.</b></li> </ul>
<code>nType (int)</code>	Type of norm, (0 = max-norm, 1 = 1-norm, 2 = 2-norm). (optional)
<code>maxincr (int)</code>	Maximum times of error increasing. (optional)

### NormDispOrUnbalance

`test ('NormDispOrUnbalance', tolIncr, tolR, iter, pFlag=0, nType=2, maxincr=-1)`

Create a NormDispOrUnbalance test, which check if both '`NormUnbalance`' and '`normDispIncr`' are converged.

<code>tolIncr (float)</code>	Tolerance for left hand solution increments
<code>tolIncr (float)</code>	Tolerance for right hand residual
<code>iter (int)</code>	Max number of iterations to check
<code>pFlag (int)</code>	Print flag (optional): <ul style="list-style-type: none"> <li>• 0 print nothing.</li> <li>• 1 print information on norms each time <code>test()</code> is invoked.</li> <li>• 2 print information on norms and number of iterations at end of successful test.</li> <li>• 4 at each step it will print the norms and also the <math>\Delta U</math> and <math>R(U)</math> vectors.</li> <li>• 5 if it fails to converge at end of <code>numIter</code> it will print an error message <b>but return a successfull test.</b></li> </ul>
<code>nType (int)</code>	Type of norm, (0 = max-norm, 1 = 1-norm, 2 = 2-norm). (optional)
<code>maxincr (int)</code>	Maximum times of error increasing. (optional)

### 1.5.5 algorithm commands

`algorithm (algoType, *algoArgs)`

This command is used to construct a `SolutionAlgorithm` object, which determines the sequence of steps taken to solve the non-linear equation.

algoType (str)	algorithm type
algoArgs (list)	a list of algorithm arguments

The following contain information about available algoType:

1. [Linear Algorithm](#)
2. [Newton Algorithm](#)
3. [Newton with Line Search](#)
4. [Modified Newton Algorithm](#)
5. [Krylov-Newton Algorithm](#)
6. [SecantNewton Algorithm](#)
7. [RaphsonNewton Algorithm](#)
8. [PeriodicNewton Algorithm](#)
9. [BFGS Algorithm](#)
10. [Broyden Algorithm](#)

## Linear Algorithm

**algorithm ('Linear', secant=False, initial=False, factorOnce=False)**

Create a Linear algorithm which takes one iteration to solve the system of equations.

secant (bool)	Flag to indicate to use secant stiffness. (optional)
initial (bool)	Flag to indicate to use initial stiffness. (optional)
factorOnce (bool)	Flag to indicate to only set up and factor matrix once. (optional)

---

**Note:** As the tangent matrix typically will not change during the analysis in case of an elastic system it is highly advantageous to use the -factorOnce option. Do not use this option if you have a nonlinear system and you want the tangent used to be actual tangent at time of the analysis step.

---

## Newton Algorithm

**algorithm ('Newton', secant=False, initial=False, initialThenCurrent=False)**

Create a Newton-Raphson algorithm. The Newton-Raphson method is the most widely used and most robust method for solving nonlinear algebraic equations.

secant (bool)	Flag to indicate to use secant stiffness. (optional)
initial (bool)	Flag to indicate to use initial stiffness.(optional)
initialThenCurrent (bool)	Flag to indicate to use initial stiffness on first step, then use current stiffness for subsequent steps. (optional)

## Newton with Line Search

```
algorithm('NewtonLineSearch', Bisection=False, Secant=False, RegulaFalsi=False, InitialInterpolated=False, tol=0.8, maxIter=10, minEta=0.1, maxEta=10.0)
```

Create a NewtonLineSearch algorithm. Introduces line search to the Newton algorithm to solve the nonlinear residual equation.

<i>Bisection</i> ( <i>bool</i> )	Flag to use Bisection line search. (optional)
<i>Secant</i> ( <i>bool</i> )	Flag to use Secant line search. (optional)
<i>RegulaFalsi</i> ( <i>bool</i> )	Flag to use RegulaFalsi line search. (optional)
<i>InitialInterpolated</i> ( <i>bool</i> )	Flag to use InitialInterpolated line search.(optional)
<i>tol</i> ( <i>float</i> )	Tolerance for search. (optional)
<i>maxIter</i> ( <i>float</i> )	Max num of iterations to try. (optional)
<i>minEta</i> ( <i>float</i> )	Min $\eta$ value. (optional)
<i>maxEta</i> ( <i>float</i> )	Max $\eta$ value. (optional)

## Modified Newton Algorithm

```
algorithm('ModifiedNewton', secant=False, initial=False)
```

Create a ModifiedNewton algorithm. The difference to Newton is that the tangent at the initial guess is used in the iterations, instead of the current tangent.

<i>secant</i> ( <i>bool</i> )	Flag to indicate to use secant stiffness. (optional)
<i>initial</i> ( <i>bool</i> )	Flag to indicate to use initial stiffness.(optional)

## Krylov-Newton Algorithm

```
algorithm('KrylovNewton', iterate='current', increment='current', maxDim=3)
```

Create a KrylovNewton algorithm which uses a Krylov subspace accelerator to accelerate the convergence of the ModifiedNewton.

<i>iterate</i> ( <i>str</i> )	Tangent to iterate on, 'current', 'initial', 'noTangent' (optional)
<i>increment</i> ( <i>str</i> )	Tangent to increment on, 'current', 'initial', 'noTangent' (optional)
<i>maxDim</i> ( <i>int</i> )	Max number of iterations until the tangent is reformed and the acceleration restarts. (optional)

## SecantNewton Algorithm

```
algorithm('SecantNewton', iterate='current', increment='current', maxDim=3)
```

Create a SecantNewton algorithm which uses the two-term update to accelerate the convergence of the ModifiedNewton.

The default “cut-out” values recommended by Crisfield (R1=3.5, R2=0.3) are used.

<i>iterate</i> ( <i>str</i> )	Tangent to iterate on, 'current', 'initial', 'noTangent' (optional)
<i>increment</i> ( <i>str</i> )	Tangent to increment on, 'current', 'initial', 'noTangent' (optional)
<i>maxDim</i> ( <i>int</i> )	Max number of iterations until the tangent is reformed and the acceleration restarts. (optional)

## RaphsonNewton Algorithm

```
algorithm('RaphsonNewton', iterate='current', increment='current')  
Create a RaphsonNewton algorithm which uses Raphson accelerator.
```

iterate (str)	Tangent to iterate on, 'current', 'initial', 'noTangent' (optional)
increment (str)	Tangent to increment on, 'current', 'initial', 'noTangent' (optional)

## PeriodicNewton Algorithm

```
algorithm('PeriodicNewton', iterate='current', increment='current', maxDim=3)  
Create a PeriodicNewton algorithm using periodic accelerator.
```

iterate (str)	Tangent to iterate on, 'current', 'initial', 'noTangent' (optional)
increment (str)	Tangent to increment on, 'current', 'initial', 'noTangent' (optional)
maxDim (int)	Max number of iterations until the tangent is reformed and the acceleration restarts. (optional)

## BFGS Algorithm

```
algorithm('BFGS', secant=False, initial=False, count=10)
```

Create a BFGS algorithm. The BFGS method is one of the most effective matrix-update or quasi Newton methods for iteration on a nonlinear system of equations. The method computes new search directions at each iteration step based on the initial jacobian, and subsequent trial solutions. The unlike regular Newton does not require the tangent matrix be reformulated and refactored at every iteration, however unlike ModifiedNewton it does not rely on the tangent matrix from a previous iteration.

secant (bool)	Flag to indicate to use secant stiffness. (optional)
initial (bool)	Flag to indicate to use initial stiffness.(optional)
count (int)	Number of iterations. (optional)

## Broyden Algorithm

```
algorithm('Broyden', secant=False, initial=False, count=10)
```

Create a Broyden algorithm for general unsymmetric systems which performs successive rank-one updates of the tangent at the first iteration of the current time step.

secant (bool)	Flag to indicate to use secant stiffness. (optional)
initial (bool)	Flag to indicate to use initial stiffness.(optional)
count (int)	Number of iterations. (optional)

## 1.5.6 integrator commands

```
integrator(intType, *intArgs)
```

This command is used to construct the Integrator object. The Integrator object determines the meaning of the terms in the system of equation object Ax=B.

The Integrator object is used for the following:

- determine the predictive step for time  $t+dt$
- specify the tangent matrix and residual vector at any iteration
- determine the corrective step based on the displacement increment  $dU$

<code>intType</code> ( <a href="#">str</a> )	integrator type
<code>intArgs</code> ( <a href="#">list</a> )	a list of integrator arguments

The following contain information about available `intType`:

## Static integrator objects

1. *LoadControl*
2. *DisplacementControl*
3. *Parallel DisplacementControl*
4. *Minimum Unbalanced Displacement Norm*
5. *Arc-Length Control*

### LoadControl

**integrator** ('LoadControl', `incr`, `numIter=1`, `minIncr=incr`, `maxIncr=incr`)

Create a OpenSees LoadControl integrator object.

<code>incr</code> ( <a href="#">float</a> )	Load factor increment $\lambda$ .
<code>numIter</code> ( <a href="#">int</a> )	Number of iterations the user would like to occur in the solution algorithm. (optional)
<code>minIncr</code> ( <a href="#">float</a> )	Min stepsize the user will allow $\lambda_{min}$ . (optional)
<code>maxIncr</code> ( <a href="#">float</a> )	Max stepsize the user will allow $\lambda_{max}$ . (optional)

1. The change in applied loads that this causes depends on the active load pattern (those load pattern not set constant) and the loads in the load pattern. If the only active load acting on the Domain are in load pattern with a Linear time series with a factor of 1.0, this integrator is the same as the classical load control method.
2. The optional arguments are supplied to speed up the step size in cases where convergence is too fast and slow down the step size in cases where convergence is too slow.

### DisplacementControl

**integrator** ('DisplacementControl', `nd`, `dof`, `incr`, `numIter=1`, `dUmin=incr`, `dUmax=incr`)

Create a DisplacementControl integrator. In an analysis step with Displacement Control we seek to determine the time step that will result in a displacement increment for a particular degree-of-freedom at a node to be a prescribed value.

<code>nd</code> ( <a href="#">int</a> )	tag of node whose response controls solution
<code>dof</code> ( <a href="#">int</a> )	Degree of freedom at the node, 1 through <code>ndf</code> .
<code>incr</code> ( <a href="#">float</a> )	First displacement increment $\Delta U_{dof}$ .
<code>numIter</code> ( <a href="#">int</a> )	Number of iterations the user would like to occur in the solution algorithm. (optional)
<code>minIncr</code> ( <a href="#">float</a> )	Min stepsize the user will allow $\Delta U_{min}$ . (optional)
<code>maxIncr</code> ( <a href="#">float</a> )	Max stepsize the user will allow $\Delta U_{max}$ . (optional)

## Parallel DisplacementControl

**integrator** ('ParallelDisplacementControl', *nd*, *dof*, *incr*, *numIter=1*, *dUmin=incr*, *dUmax=incr*)

Create a Parallel version of DisplacementControl integrator. In an analysis step with Displacement Control we seek to determine the time step that will result in a displacement increment for a particular degree-of-freedom at a node to be a prescribed value.

<i>nd</i> (int)	tag of node whose response controls solution
<i>dof</i> (int)	Degree of freedom at the node, 1 through <i>ndf</i> .
<i>incr</i> (float)	First displacement increment $\Delta U_{dof}$ .
<i>numIter</i> (int)	Number of iterations the user would like to occur in the solution algorithm. (optional)
<i>minIncr</i> (float)	Min stepsize the user will allow $\Delta U_{min}$ . (optional)
<i>maxIncr</i> (float)	Max stepsize the user will allow $\Delta U_{max}$ . (optional)

Use this command only for parallel model.

**Warning:** Don't use this command if model is not parallel, for example, parametric study.

## Minimum Unbalanced Displacement Norm

**integrator** ('MinUnbalDispNorm', *dlambda1*, *Jd=1*, *minLambda=dlambda1*, *maxLambda=dlambda1*, *det=False*)

Create a MinUnbalDispNorm integrator.

<i>dlambda1</i> (float)	First load increment (pseudo-time step) at the first iteration in the next invocation of the analysis command.
<i>Jd</i> (int)	Factor relating first load increment at subsequent time steps. (optional)
<i>minLambda</i> (float)	Min load increment. (optional)
<i>maxLambda</i> (float)	Max load increment. (optional)

## Arc-Length Control

**integrator** ('ArcLength', *s*, *alpha*)

Create a ArcLength integrator. In an analysis step with ArcLength we seek to determine the time step that will result in our constraint equation being satisfied.

<i>s</i> (float)	The arcLength.
<i>alpha</i> (float)	$\alpha$ a scaling factor on the reference loads.

## Transient integrator objects

1. *Central Difference*
2. *Explicit Difference*
3. *Newmark Method*

4. *Hilber-Hughes-Taylor Method*
5. *Generalized Alpha Method*
6. *TRBDF2*
7. *Explicit Difference*
8. *PFEM integrator*

## Central Difference

**integrator ('CentralDifference')**

Create a centralDifference integrator.

1. The calculation of  $U_t + \Delta t$ , is based on using the equilibrium equation at time t. For this reason the method is called an explicit integration method.
2. If there is no rayleigh damping and the C matrix is 0, for a diagonal mass matrix a diagonal solver may and should be used.
3. For stability,  $\frac{\Delta t}{T_n} < \frac{1}{\pi}$

## Explicit Difference

**integrator ('ExplicitDifference')**

Create a ExplicitDifference integrator.

## Newmark Method

**integrator ('Newmark', gamma, beta, '-formD', form)**

Create a Newmark integrator.

gamma (float)	$\gamma$ factor.
beta (float)	$\beta$ factor.
form (str)	Flag to indicate which variable to be used as primary variable (optional) <ul style="list-style-type: none"><li>• 'D' – displacement (default)</li><li>• 'V' – velocity</li><li>• 'A' – acceleration</li></ul>

1. If the accelerations are chosen as the unknowns and  $\beta$  is chosen as 0, the formulation results in the fast but conditionally stable explicit Central Difference method. Otherwise the method is implicit and requires an iterative solution process.
2. Two common sets of choices are
  1. Average Acceleration Method ( $\gamma = \frac{1}{2}, \beta = \frac{1}{4}$ )
  2. Linear Acceleration Method ( $\gamma = \frac{1}{2}, \beta = \frac{1}{6}$ )
  3.  $\gamma > \frac{1}{2}$  results in numerical damping proportional to  $\gamma - \frac{1}{2}$
  4. The method is second order accurate if and only if  $\gamma = \frac{1}{2}$
  5. The method is unconditionally stable for  $\beta \geq \frac{\gamma}{2} \geq \frac{1}{4}$

## Hilber-Hughes-Taylor Method

**integrator** ('HHT', *alpha*, *gamma*=1.5-*alpha*, *beta*=(2-*alpha*)^2/4)

Create a Hilber-Hughes-Taylor (HHT) integrator. This is an implicit method that allows for energy dissipation and second order accuracy (which is not possible with the regular Newmark object). Depending on choices of input parameters, the method can be unconditionally stable.

<i>alpha</i> ( <b>float</b> )	$\alpha$ factor.
<i>gamma</i> ( <b>float</b> )	$\gamma$ factor. (optional)
<i>beta</i> ( <b>float</b> )	$\beta$ factor. (optional)

1. Like Newmark and all the implicit schemes, the unconditional stability of this method applies to linear problems. There are no results showing stability of this method over the wide range of nonlinear problems that potentially exist. Experience indicates that the time step for implicit schemes in nonlinear situations can be much greater than those for explicit schemes.
2.  $\alpha = 1.0$  corresponds to the Newmark method.
3.  $\alpha$  should be between 0.67 and 1.0. The smaller the  $\alpha$  the greater the numerical damping.
4.  $\gamma$  and  $\beta$  are optional. The default values ensure the method is second order accurate and unconditionally stable when  $\alpha$  is  $\frac{2}{3} \leq \alpha \leq 1.0$ . The defaults are:

$$\beta = \frac{(2-\alpha)^2}{4}$$

and

$$\gamma = \frac{3}{2} - \alpha$$

## Generalized Alpha Method

**integrator** ('GeneralizedAlpha', *alphaM*, *alphaF*, *gamma*=0.5+*alphaM*-*alphaF*, *beta*=(1+*alphaM*-*alphaF*)^2/4)

Create a GeneralizedAlpha integrator. This is an implicit method that like the HHT method allows for high frequency energy dissipation and second order accuracy, i.e.  $\Delta t^2$ . Depending on choices of input parameters, the method can be unconditionally stable.

<i>alphaM</i> ( <b>float</b> )	$\alpha_M$ factor.
<i>alphaF</i> ( <b>float</b> )	$\alpha_F$ factor.
<i>gamma</i> ( <b>float</b> )	$\gamma$ factor. (optional)
<i>beta</i> ( <b>float</b> )	$\beta$ factor. (optional)

1. Like Newmark and all the implicit schemes, the unconditional stability of this method applies to linear problems. There are no results showing stability of this method over the wide range of nonlinear problems that potentially exist. Experience indicates that the time step for implicit schemes in nonlinear situations can be much greater than those for explicit schemes.
2.  $\alpha_M = 1.0, \alpha_F = 1.0$  produces the Newmark Method.
3.  $\alpha_M = 1.0$  corresponds to the `integrator.HHT()` method.
4. The method is second-order accurate provided  $\gamma = \frac{1}{2} + \alpha_M - \alpha_F$
5. The method is unconditionally stable provided  $\alpha_M \geq \alpha_F \geq \frac{1}{2}, \beta \geq \frac{1}{4} + \frac{1}{2}(\gamma_M - \gamma_F)$

6.  $\gamma$  and  $\beta$  are optional. The default values ensure the method is unconditionally stable, second order accurate and high frequency dissipation is maximized.

The defaults are:

$$\gamma = \frac{1}{2} + \alpha_M - \alpha_F$$

and

$$\beta = \frac{1}{4}(1 + \alpha_M - \alpha_F)^2$$

## TRBDF2

**integrator ('TRBDF2')**

Create a TRBDF2 integrator. The TRBDF2 integrator is a composite scheme that alternates between the Trapezoidal scheme and a 3 point backward Euler scheme. It does this in an attempt to conserve energy and momentum, something Newmark does not always do.

As opposed to dividing the time-step in 2 as outlined in the [Bathe2007](#), we just switch alternate between the 2 integration strategies,i.e. the time step in our implementation is double that described in the [Bathe2007](#).

## Explicit Difference

**integrator ('ExplicitDifference')**

Create a ExplicitDifference integrator.

1. When using Rayleigh damping, the damping ratio of high vibration modes is overrated, and the critical time step size will be much smaller. Hence Modal damping is more suitable for this method.
2. There should be no zero element on the diagonal of the mass matrix when using this method.
3. Diagonal solver should be used when lumped mass matrix is used because the equations are uncoupled.
4. For stability,  $\Delta t \leq \left( \sqrt{\zeta^2 + 1} - \zeta \right) \frac{2}{\omega}$

## 1.5.7 analysis command

**analysis (analysisType)**

This command is used to construct the Analysis object, which defines what type of analysis is to be performed.

- determine the predictive step for time  $t+dt$
- specify the tangent matrix and residual vector at any iteration
- determine the corrective step based on the displacement increment  $dU$

analysisType (str)	char string identifying type of analysis object to be constructed. Currently 3 valid options: 1. 'Static' - for static analysis 2. 'Transient' - for transient analysis constant time step 3. 'VariableTransient' - for transient analysis with variable time step 4. 'PFEM' - for <a href="#">PFEM analysis</a> .
--------------------	--

---

**Note:** If the component objects are not defined before hand, the command automatically creates default component objects and issues warning messages to this effect. The number of warning messages depends on the number of component objects that are undefined.

---

### 1.5.8 eigen command

**eigen** (*solver=-genBandArpack*', *numEigenvalues*)

Eigen value analysis. Return a list of eigen values.

<b>numEigenvalues</b> ( <i>int</i> )	number of eigenvalues required
<b>solver</b> ( <i>str</i> )	optional string detailing type of solver: '-genBandArpack', '-symmBandLapack', '-fullGenLapack', (optional)

---

**Note:**

1. The eigenvectors are stored at the nodes and can be printed out using a Node Recorder, the nodeEigenvector command, or the Print command.
  2. The default eigensolver is able to solve only for N-1 eigenvalues, where N is the number of inertial DOFs. When running into this limitation the -fullGenLapack solver can be used instead of the default Arpack solver.
- 

### 1.5.9 analyze command

**analyze** (*numIncr=1*, *dt=0.0*, *dtMin=0.0*, *dtMax=0.0*, *Jd=0*)

Perform the analysis. Return 0 if successful, <0 if NOT successful

<b>numIncr</b> ( <i>int</i> )	Number of analysis steps to perform. (required except for <i>PFEM analysis</i> )
<b>dt</b> ( <i>float</i> )	Time-step increment. (required for Transient analysis and VariableTransient analysis.)
<b>dtMin</b> ( <i>float</i> )	Minimum time steps. (required for VariableTransient analysis)
<b>dtMax</b> ( <i>float</i> )	Maximum time steps (required for VariableTransient analysis)
<b>Jd</b> ( <i>float</i> )	Number of iterations user would like performed at each step. The variable transient analysis will change current time step if last analysis step took more or less iterations than this to converge (required for VariableTransient analysis)

## 1.6 Output Commands

Get outputs from OpenSees. These commands don't change internal states of OpenSees.

1. *basicDeformation command*
2. *basicForce command*
3. *basicStiffness command*

4. *eleDynamicalForce command*
5. *eleForce command*
6. *eleNodes command*
7. *eleResponse command*
8. *getEleTags command*
9. *getLoadFactor command*
10. *getNodeTags command*
11. *getTime command*
12. *nodeAccel command*
13. *nodeBounds command*
14. *nodeCoord command*
15. *nodeDisp command*
16. *nodeEigenvector command*
17. *nodeDOFs command*
18. *nodeMass command*
19. *nodePressure command*
20. *nodeReaction command*
21. *nodeResponse command*
22. *nodeVel command*
23. *nodeUnbalance command*
24. *numFact command*
25. *numIter command*
26. *printA command*
27. *printB command*
28. *printGID command*
29. *printModel command*
30. *record command*
31. *recorder command*
32. *sectionForce command*
33. *sectionDeformation command*
34. *sectionStiffness command*
35. *sectionFlexibility command*
36. *sectionLocation command*
37. *sectionWeight command*
38. *systemSize command*
39. *testIter command*

40. *testNorm command*
41. *version command*
42. *logFile command*

### 1.6.1 basicDeformation command

**basicDeformation** (*eleTag*)

Returns the deformation of the basic system for a beam-column element.

eleTag (int)	element tag.
--------------	--------------

### 1.6.2 basicForce command

**basicForce** (*eleTag*)

Returns the forces of the basic system for a beam-column element.

eleTag (int)	element tag.
--------------	--------------

### 1.6.3 basicStiffness command

**basicStiffness** (*eleTag*)

Returns the stiffness of the basic system for a beam-column element. A list of values in row order will be returned.

eleTag (int)	element tag.
--------------	--------------

### 1.6.4 eleDynamicalForce command

**eleDynamicalForce** (*eleTag, dof=-1*)

Returns the elemental dynamic force.

eleTag (int)	element tag.
dof (int)	specific dof at the element, (optional), if no dof is provided, a list of values for all dofs is returned.

### 1.6.5 eleForce command

**eleForce** (*eleTag, dof=-1*)

Returns the elemental resisting force.

eleTag (int)	element tag.
dof (int)	specific dof at the element, (optional), if no dof is provided, a list of values for all dofs is returned.

## 1.6.6 eleNodes command

**eleNodes** (*eleTag*)

Get nodes in an element

eletag (int)	element tag.
--------------	--------------

## 1.6.7 eleResponse command

**eleResponse** (*eleTag*, \**args*)

This command is used to obtain the same element quantities as those obtained from the element recorder at a particular time step.

eletag (int)	element tag.
args (list)	same arguments as those specified in element recorder. These arguments are specific to the type of element being used.

## 1.6.8 getEleTags command

**getEleTags** ('-mesh', *mtag*)

Get all elements in the domain or in a mesh.

mtag (int)	mesh tag. (optional)
------------	----------------------

## 1.6.9 getLoadFactor command

**getLoadFactor** (*patternTag*)

Returns the load factor  $\lambda$  for the pattern

patternTag (int)	pattern tag.
------------------	--------------

## 1.6.10 getNodeTags command

**getNodeTags** ('-mesh', *mtag*)

Get all nodes in the domain or in a mesh.

mtag (int)	mesh tag. (optional)
------------	----------------------

## 1.6.11 getTime command

**getTime** ()

Returns the current time in the domain.

### 1.6.12 nodeAccel command

**nodeAccel** (*nodeTag, dof=-1*)

Returns the current acceleration at a specified node.

nodeTag (int)	node tag.
dof (int)	specific dof at the node (1 through ndf), (optional), if no dof is provided, a list of values for all dofs is returned.

### 1.6.13 nodeBounds command

**nodeBounds** ()

Get the boundary of all nodes. Return a list of boundary values.

### 1.6.14 nodeCoord command

**nodeCoord** (*nodeTag, dim=-1*)

Returns the coordinates of a specified node.

nodeTag (int)	node tag.
dof (int)	specific dimension at the node (1 through ndf), (optional), if no dim is provided, a list of values for all dimensions is returned.

### 1.6.15 nodeDisp command

**nodeDisp** (*nodeTag, dof=-1*)

Returns the current displacement at a specified node.

nodeTag (int)	node tag.
dof (int)	specific dof at the node (1 through ndf), (optional), if no dof is provided, a list of values for all dofs is returned.

### 1.6.16 nodeEigenvector command

**nodeEigenvector** (*eigenvector, dof=-1*)

Returns the eigenvector at a specified node.

nodeTag (int)	node tag.
eigenvector (int)	mode number of eigenvector to be returned
dof (int)	specific dof at the node (1 through ndf), (optional), if no dof is provided, a list of values for all dofs is returned.

### 1.6.17 nodeDOFs command

**nodeDOFs** (*nodeTag*)

Returns the DOF numbering of a node.

nodeTag (int)	node tag.
---------------	-----------

### 1.6.18 nodeMass command

**nodeMass** (*nodeTag, dof=-1*)

Returns the mass at a specified node.

nodeTag (int)	node tag.
dof (int)	specific dof at the node (1 through ndf), (optional), if no dof is provided, a list of values for all dofs is returned.

### 1.6.19 nodePressure command

**nodePressure** (*nodeTag*)

Returns the fluid pressures at a specified node if this is a fluid node.

nodeTag (int)	node tag.
---------------	-----------

### 1.6.20 nodeReaction command

**nodeReaction** (*nodeTag, dof=-1*)

Returns the reactions at a specified node. Must call `reactions()` command before this command.

nodeTag (int)	node tag.
dof (int)	specific dof at the node (1 through ndf), (optional), if no dof is provided, a list of values for all dofs is returned.

### 1.6.21 nodeResponse command

**nodeResponse** (*nodeTag, dof, responseID*)

Returns the responses at a specified node. Must call `responses` command before this command.

nodeTag ( <a href="#">int</a> )	node tag.
dof ( <a href="#">int</a> )	specific dof of the response
responseID ( <a href="#">int</a> )	the id of responses: <ul style="list-style-type: none"><li>• Disp = 1</li><li>• Vel = 2</li><li>• Accel = 3</li><li>• IncrDisp = 4</li><li>• IncrDeltaDisp = 5</li><li>• Reaction = 6</li><li>• Unbalance = 7</li><li>• RayleighForces = 8</li></ul>

## 1.6.22 nodeVel command

**nodeVel** (*nodeTag*, *dof=-1*)

Returns the current velocity at a specified node.

nodeTag ( <a href="#">int</a> )	node tag.
dof ( <a href="#">int</a> )	specific dof at the node (1 through ndf), (optional), if no <code>dof</code> is provided, a list of values for all dofs is returned.

## 1.6.23 nodeUnbalance command

**nodeUnbalance** (*nodeTag*, *dof=-1*)

Returns the unbalanced force at a specified node.

nodeTag ( <a href="#">int</a> )	node tag.
dof ( <a href="#">int</a> )	specific dof at the node (1 through ndf), (optional), if no <code>dof</code> is provided, a list of values for all dofs is returned.

## 1.6.24 numFact command

**numFact** ()

Return the number of factorizations.

## 1.6.25 numIter command

**numIter** ()

Return the number of iterations.

## 1.6.26 printA command

**printA** ('-file', *filename*, '-ret')

print the contents of a FullGeneral system that the integrator creates to the screen or a file if the '`-file`'

option is used. If using a static integrator, the resulting matrix is the stiffness matrix. If a transient integrator, it will be some combination of mass and stiffness matrices.

<code>filename (str)</code>	name of file to which output is sent, by default, print to the screen. (optional)
<code>'-ret' (str)</code>	return the A matrix as a list. (optional)

## 1.6.27 `printB` command

`printB ('-file', filename, '-ret')`

print the right hand side of a FullGeneral system that the integrator creates to the screen or a file if the '`-file`' option is used.

<code>filename (str)</code>	name of file to which output is sent, by default, print to the screen. (optional)
<code>'-ret' (str)</code>	return the B vector as a list. (optional)

## 1.6.28 `printGID` command

`printGID (filename, '-append', '-eleRange', startEle, endEle)`

Print in GID format.

<code>filename (str)</code>	output file name.
<code>'-append' (str)</code>	append to existing file. (optional)
<code>startEle (int)</code>	start element tag. (optional)
<code>endEle (int)</code>	end element tag. (optional)

## 1.6.29 `printModel` command

`printModel ('-file', filename, '-JSON', '-node', '-flag', flag, *nodes=[], *eles=[])`

This command is used to print output to screen or file.

<code>filename (str)</code>	name of file to which output is sent, by default, print to the screen. (optional)
<code>'-JSON' (str)</code>	print to a JSON file. (optional)
<code>'-node' (str)</code>	print node information. (optional)
<code>flag (int)</code>	integer flag to be sent to the print() method, depending on the node and element type (optional)
<code>nodes (list (int))</code>	a list of nodes tags to be printed, default is to print all, (optional)
<code>eles (list (int))</code>	a list of element tags to be printed, default is to print all, (optional)

---

**Note:** This command was called `print` in Tcl. Since `print` is a built-in function in Python, it is renamed to `printModel`.

---

## 1.6.30 record command

### **record()**

This command is used to cause all the recorders to do a record on the current state of the model.

---

**Note:** A record is issued after every successfull static or transient analysis step. Sometimes the user may need the record to be issued on more occasions than this, for example if the user is just looking to record the eigenvectors after an eigen command or for example the user wishes to include the state of the model at time 0.0 before any analysis has been completed.

---

## 1.6.31 recorder command

### **recorder (recorderType, \*recorderArgs)**

This command is used to generate a recorder object which is to monitor what is happening during the analysis and generate output for the user.

Return:

- >0 an integer tag that can be used as a handle on the recorder for the remove recorder command.
- -1 recorder command failed if integer -1 returned.

recorderType (str)	recorder type
recorderArgs (list)	a list of recorder arguments

The following contain information about available recorderType:

### **node recorder command**

```
recorder ('Node', '-file', filename, '-xml', filename, '-binary', filename, '-tcp', inetAddress, port, '-precision', nSD=6, '-timeSeries', tsTag, '-time', '-dT', deltaT=0.0, '-closeOnWrite', '-node', *nodeTags=[], '-nodeRange', startNode, endNode, '-region', regionTag, '-dof', *dofs=[], respType)
```

The Node recorder type records the response of a number of nodes at every converged step.

<code>filename (str)</code>	name of file to which output is sent. file output is either in xml format ('-xml' option), textual ('-file' option) or binary ('-binary' option) which must pre-exist.
<code>inetAddr (str)</code>	ip address, "xx.xx.xx.xx", of remote machine to which data is sent. (optional)
<code>port (int)</code>	port on remote machine awaiting tcp. (optional)
<code>nSD (int)</code>	number of significant digits (optional)
<code>'-time' (str)</code>	using this option places domain time in first entry of each data line, default is to have time ommitted, (optional)
<code>'-closeOnWrite' (str)</code>	using this option will instruct the recorder to invoke a close on the data handler after every timestep. If this is a file it will close the file on every step and then re-open it for the next step. Note, this greatly slows the execution time, but is useful if you need to monitor the data during the analysis. (optional)
<code>deltaT (float)</code>	time interval for recording. will record when next step is <code>deltaT</code> greater than last recorder step. (optional, default: records at every time step)
<code>tsTag (int)</code>	the tag of a previously constructed TimeSeries, results from node at each time step are added to load factor from series (optional)
<code>nodeTags (list (int))</code>	list of tags of nodes whose response is being recorded (optional)
<code>startNode (int)</code>	tag for start node whose response is being recorded (optional)
<code>endNode (int)</code>	tag for end node whose response is being recorded (optional)
<code>regionTag (int)</code>	a region tag; to specify all nodes in the previously defined region. (optional)
<code>dofs (list (int))</code>	the specified dof at the nodes whose response is requested.
<code>resType (list (str))</code>	<p>a string indicating response required. Response types are given in table below</p> <ul style="list-style-type: none"> <li>• 'disp' displacement</li> <li>• 'vel' velocity</li> <li>• 'accel' acceleration</li> <li>• 'incrDisp' incremental displacement</li> <li>• 'reaction' nodal reaction</li> <li>• 'eigen i' eigenvector for mode i</li> <li>• 'rayleighForces' damping forces</li> </ul>

---

**Note:** Only one of '-file', '-xml', '-binary', '-tcp' will be used. If multiple specified last option is used.

---

**node envelope recorder command**

```
recorder ('EnvelopeNode', '-file', filename, '-xml', filename, '-precision', nSD=6, '-timeSeries', tsTag, '-time', '-dT', deltaT=0.0, '-closeOnWrite', '-node', *nodeTags=[], '-nodeRange', startNode, endNode, '-region', regionTag, '-dof', *dofs=[], respType)
```

The EnvelopeNode recorder type records the min, max and absolute max of a number of nodal response quantities.

filename (str)	name of file to which output is sent. file output is either in xml format ('-xml' option), or textual ('-file' option) which must pre-exist
nSD (int)	number of significant digits (optional)
'-time' (str)	using this option places domain time in first entry of each data line, default is to have time omitted, (optional)
'-closeOnWrite' (str)	using this option will instruct the recorder to invoke a close on the data handler after every timestep. If this is a file it will close the file on every step and then re-open it for the next step. Note, this greatly slows the execution time, but is useful if you need to monitor the data during the analysis. (optional)
deltaT (float)	time interval for recording. will record when next step is deltaT greater than last recorder step. (optional, default: records at every time step)
tsTag (int)	the tag of a previously constructed TimeSeries, results from node at each time step are added to load factor from series (optional)
nodeTags (list (int))	list of tags of nodes whose response is being recorded (optional)
startNode (int)	tag for start node whose response is being recorded (optional)
endNode (int)	tag for end node whose response is being recorded (optional)
regionTag (int)	a region tag; to specify all nodes in the previously defined region. (optional)
dofs (list (int))	the specified dof at the nodes whose response is requested.
respType (list (str))	a string indicating response required. Response types are given in table below <ul style="list-style-type: none"><li>• 'disp' displacement</li><li>• 'vel' velocity</li><li>• 'accel' acceleration</li><li>• 'incrDisp' incremental displacement</li><li>• 'reaction' nodal reaction</li><li>• 'eigen i' eigenvector for mode i</li></ul>

**element recorder command**

```
recorder ('Element', '-file', filename, '-xml', filename, '-binary', filename, '-precision', nSD=6, '-timeSeries', tsTag, '-time', '-dT', deltaT=0.0, '-closeOnWrite', '-ele', *eleTags=[], '-eleRange', startEle, endEle, '-region', regionTag, *args)
```

The Element recorder type records the response of a number of elements at every converged step. The response

recorded is element-dependent and also depends on the arguments which are passed to the setResponse() element method.

<code>filename</code> (str)	name of file to which output is sent. file output is either in xml format ('-xml' option), textual ('-file' option) or binary ('-binary' option) which must pre-exist.
<code>nSD</code> (int)	number of significant digits (optional)
<code>'-time'</code> (str)	using this option places domain time in first entry of each data line, default is to have time omitted, (optional)
<code>'-closeOnWrite'</code> (str)	using this option will instruct the recorder to invoke a close on the data handler after every timestep. If this is a file it will close the file on every step and then re-open it for the next step. Note, this greatly slows the execution time, but is useful if you need to monitor the data during the analysis. (optional)
<code>deltaT</code> (float)	time interval for recording. will record when next step is <code>deltaT</code> greater than last recorder step. (optional, default: records at every time step)
<code>tsTag</code> (int)	the tag of a previously constructed TimeSeries, results from node at each time step are added to load factor from series (optional)
<code>eleTags</code> (list (int))	list of tags of elements whose response is being recorded (optional)
<code>startEle</code> (int)	tag for start node whose response is being recorded (optional)
<code>endEle</code> (int)	tag for end node whose response is being recorded (optional)
<code>regionTag</code> (int)	a region tag; to specify all nodes in the previously defined region. (optional)
<code>args</code> (list)	arguments which are passed to the setResponse() element method, all arguments must be in string format even for double and integer numbers because internally the setResponse() element method only accepts strings.

---

**Note:** The setResponse() element method is dependent on the element type, and is described with the [element \(\) Command](#).

---

### element envelope recorder command

```
recorder ('EnvelopeElement', '-file', filename, '-xml', filename, '-binary', filename, '-precision', nSD=6, '-timeSeries', tsTag, '-time', '-dT', deltaT=0.0, '-closeOnWrite', '-ele', *eleTags=[], '-eleRange', startEle, endEle, '-region', regionTag, *args)
```

The Envelope Element recorder type records the response of a number of elements at every converged step. The response recorded is element-dependent and also depends on the arguments which are passed to the setResponse() element method. When the object is terminated, through the use of a wipe, exit, or remove the object will output the min, max and absolute max values on 3 separate lines of the output file for each quantity.

filename (str)	name of file to which output is sent. file output is either in xml format ('-xml' option), textual ('-file' option) or binary ('-binary' option) which must pre-exist.
nSD (int)	number of significant digits (optional)
'-time' (str)	using this option places domain time in first entry of each data line, default is to have time omitted, (optional)
'-close' (str)	Using this option will instruct the recorder to invoke a close on the data handler after every timestep. If this is a file it will close the file on every step and then re-open it for the next step. Note, this greatly slows the execution time, but is useful if you need to monitor the data during the analysis. (optional)
deltaT (float)	time interval for recording. will record when next step is deltaT greater than last recorder step. (optional, default: records at every time step)
tsTag (int)	the tag of a previously constructed TimeSeries, results from node at each time step are added to load factor from series (optional)
eleTags (list (int))	list of tags of elements whose response is being recorded (optional)
startEle (int)	tag for start node whose response is being recorded (optional)
endEle (int)	tag for end node whose response is being recorded (optional)
regionTag (int)	region tag; to specify all nodes in the previously defined region. (optional)
args (list)	arguments which are passed to the setResponse() element method

---

**Note:** The setResponse() element method is dependent on the element type, and is described with the [element \(\)](#) Command.

---

## pvd recorder command

**recorder** ('PVD', *filename*, '-precision', *precision*=10, '-dT', *dT*=0.0, \**res*)  
Create a PVD recorder.

filename (str)	the name for <i>filename</i> .pvd and <i>filename</i> / directory, which must pre-exist.
precision (int)	the precision of data. (optional)
dT (float)	the time interval for recording. (optional)
res (list (str))	a list of (str) of responses to be recorded, (optional) <ul style="list-style-type: none"><li>• 'disp'</li><li>• 'vel'</li><li>• 'accel'</li><li>• 'incrDisp'</li><li>• 'reaction'</li><li>• 'pressure'</li><li>• 'unbalancedLoad'</li><li>• 'mass'</li><li>• 'eigen'</li></ul>

## background recorder command

**recorder** ('BgPVD', *filename*, '-precision', *precision*=10, '-dT', *dT*=0.0, \**res*)

Create a PVD recorder for background mesh. This recorder is same as the PVD recorder, but will be automatically called in background mesh and is able to record wave height and velocity.

<i>filename</i> ( <i>str</i> )	the name for <i>filename.pvd</i> and <i>filename/</i> directory, which must pre-exist.
<i>precision</i> ( <i>int</i> )	the precision of data. (optional)
<i>dT</i> ( <i>float</i> )	the time interval for recording. (optional)
<i>res</i> ( <i>list(str)</i> )	a list of ( <i>str</i> ) of responses to be recorded, (optional) • 'disp' • 'vel' • 'accel' • 'incrDisp' • 'reaction' • 'pressure' • 'unbalancedLoad' • 'mass' • 'eigen'

## 1.6.32 sectionForce command

**sectionForce** (*eleTag*, *secNum*, *dof*)

Returns the section force for a beam-column element. The dof of the section depends on the section type. Please check with the section manual.

<i>eleTag</i> ( <i>int</i> )	element tag.
<i>secNum</i> ( <i>int</i> )	section number, i.e. the Gauss integratio number
<i>dof</i> ( <i>int</i> )	the dof of the section

## 1.6.33 sectionDeformation command

**sectionDeformation** (*eleTag*, *secNum*, *dof*)

Returns the section deformation for a beam-column element. The dof of the section depends on the section type. Please check with the section manual.

<i>eleTag</i> ( <i>int</i> )	element tag.
<i>secNum</i> ( <i>int</i> )	section number, i.e. the Gauss integratio number
<i>dof</i> ( <i>int</i> )	the dof of the section

## 1.6.34 sectionStiffness command

**sectionStiffness** (*eleTag*, *secNum*, *dof*)

Returns the section stiffness matrix for a beam-column element. A list of values in the row order will be returned.

eleTag (int)	element tag.
secNum (int)	section number, i.e. the Gauss integratio number
dof (int)	the dof of the section

### 1.6.35 sectionFlexibility command

**sectionFlexibility** (*eleTag*, *secNum*, *dof*)

Returns the section flexibility matrix for a beam-column element. A list of values in the row order will be returned.

eleTag (int)	element tag.
secNum (int)	section number, i.e. the Gauss integratio number
dof (int)	the dof of the section

### 1.6.36 sectionLocation command

**sectionLocation** (*eleTag*, *secNum*)

Returns the locations of integration points of a section for a beam-column element.

eleTag (int)	element tag.
secNum (int)	section number, i.e. the Gauss integration number

### 1.6.37 sectionWeight command

**sectionWeight** (*eleTag*, *secNum*)

Returns the weights of integration points of a section for a beam-column element.

eleTag (int)	element tag.
secNum (int)	section number, i.e. the Gauss integration number

### 1.6.38 systemSize command

**systemSize** ()

Return the size of the system.

### 1.6.39 testIter command

**testIter** ()

Returns the number of iterations the convergence test took in the last analysis step

### 1.6.40 testNorm command

**testNorm** ()

Returns the norms from the convergence test for the last analysis step.

---

**Note:** The size of norms will be equal to the max number of iterations specified. The first `testIter` of these will be non-zero, the remaining ones will be zero.

---

### 1.6.41 `version` command

`version()`  
Return the current OpenSees version.

### 1.6.42 `logFile` command

`logFile(filename, '-append', '-noEcho')`  
Log all messages and errors in a file. By default, all messages and errors print to terminal or Jupyter Notebook depending on how Python was run.

<code>filename</code> (str)	name of the log file
<code>'-append'</code> (str)	append to the file
<code>'-noEcho'</code> (str)	do not print to terminal or Jupyter Notebook

## 1.7 Utility Commands

These commands are used to monitor and change the state of the model.

1. *convertBinaryToText command*
2. *convertTextToBinary command*
3. *database command*
4. *InitialStateAnalysis command*
5. *loadConst command*
6. *modalDamping command*
7. *reactions command*
8. *remove command*
9. *reset command*
10. *restore command*
11. *save command*
12. *sdfResponse command*
13. *setTime command*
14. *setNodeCoord command*
15. *setNodeDisp command*
16. *setNodeVel command*
17. *setNodeAccel command*
18. *setPrecision command*

19. *setElementRayleighDampingFactors command*
20. *start command*
21. *stop command*
22. *stripXML command*
23. *updateElementDomain command*
24. *updateMaterialStage*
25. *wipe command*
26. *wipeAnalysis command*
27. *setNumthread command*
28. *getNumthread command*
29. *plot\_model command*
30. *plot\_modeshape command*

### 1.7.1 convertBinaryToText command

**convertBinaryToText** (*inputfile*, *outputfile*)

Convert binary file to text file

<code>inputfile (str)</code>	input file name.
<code>outputfile (str)</code>	output file name.

### 1.7.2 convertTextToBinary command

**convertTextToBinary** (*inputfile*, *outputfile*)

Convert text file to binary file

<code>inputfile (str)</code>	input file name.
<code>outputfile (str)</code>	output file name.

### 1.7.3 database command

**database** (*type*, *dbName*)

Create a database.

<code>type (str)</code>	database type: <ul style="list-style-type: none"><li>• 'File' - outputs database into a file</li><li>• 'MySQL' - creates a SQL database</li><li>• 'BerkeleyDB' - creates a BerkeleyDB database</li></ul>
<code>dbName (str)</code>	database name.

## 1.7.4 InitialStateAnalysis command

**InitialStateAnalysis** (*flag*)

Set the initial state analysis to 'on' or 'off'

flag ( <code>str</code> )	'on' or 'off'
---------------------------	---------------

## 1.7.5 loadConst command

**loadConst** ('-time', *pseudoTime*)

This command is used to set the loads constant in the domain and to also set the time in the domain. When setting the loads constant, the procedure will invoke `setLoadConst()` on all `LoadPattern` objects which exist in the domain at the time the command is called.

pseudoTime ( <code>float</code> )	Time domain is to be set to (optional)
-----------------------------------	--

---

**Note:** Load Patterns added after this command is invoked are not set to constant.

---

## 1.7.6 modalDamping command

**modalDamping** (*factor*)

Set modal damping factor. The `eigen()` must be called before.

factor ( <code>float</code> )	damping factor.
-------------------------------	-----------------

## 1.7.7 reactions command

**reactions** ('-dynamic', '-rayleigh')

Calculate the reactions. Call this command before the `nodeReaction()`.

'-dynamic' ( <code>str</code> )	Include dynamic effects.
'-rayleigh' ( <code>str</code> )	Include rayleigh damping.

## 1.7.8 remove command

**remove** (*type*, *tag*)

This command is used to remove components from the model.

<code>type</code> ( <code>str</code> )	type of the object, 'ele', 'loadPattern', 'parameter', 'node', 'timeSeries', 'sp', 'mp'.
<code>tag</code> ( <code>int</code> )	tag of the object

**remove** ('recorders')

Remove all recorder objects.

**remove** ('sp', nodeTag, dofTag, patternTag)

Remove a sp object based on node

nodeTag ( <a href="#">int</a> )	node tag
dof ( <a href="#">int</a> )	dof the sp constrains
patternTag ( <a href="#">int</a> )	pattern tag, (optional)

## 1.7.9 reset command

**reset** ()

This command is used to set the state of the domain to its original state.

---

**Note:** It iterates over all components of the domain telling them to set their state back to the initial state. This is not always the same as going back to the state of the model after initial model generation, e.g. if elements have been removed.

---

## 1.7.10 restore command

**restore** (commitTag)Restore data from database, which should be created through [database](#) () .

commitTag ( <a href="#">int</a> )	a tag identify the commit
-----------------------------------	---------------------------

## 1.7.11 save command

**save** (commitTag)Save current state to database, which should be created through [database](#) () .

commitTag ( <a href="#">int</a> )	a tag identify the commit
-----------------------------------	---------------------------

## 1.7.12 sdfResponse command

**sdfResponse** (m, zeta, k, Fy, alpha, dtF, filename, dt[, uresidual, umaxprev ])

It is a command that computes bilinear single degree of freedom response in C++, and is much quicker than using the OpenSees model builder. The command implements Newmark's method with an inner Newton loop.

m ( <a href="#">float</a> )	mass
zeta ( <a href="#">float</a> )	damping ratio
k ( <a href="#">float</a> )	stiffness
Fy ( <a href="#">float</a> )	yielding strength
alpha ( <a href="#">float</a> )	strain-hardening ratio
dtF ( <a href="#">float</a> )	time step for input data
filename ( <a href="#">str</a> )	input data file, one force per line
dt ( <a href="#">float</a> )	time step for analysis
uresidual ( <a href="#">float</a> )	residual displacement at the end of previous analysis (optional)
umaxprev ( <a href="#">float</a> )	previous displacement (optional)

### 1.7.13 setTime command

**setTime** (*pseudoTime*)

This command is used to set the time in the Domain.

pseudoTime ( <code>float</code> )	Time domain to be set
-----------------------------------	-----------------------

### 1.7.14 setNodeCoord command

**setNodeCoord** (*nodeTag*, *dim*, *value*)

set the nodal coordinate at the specified dimension.

nodeTag ( <code>int</code> )	node tag.
dim ( <code>int</code> )	the dimension of the coordinate to be set.
value ( <code>float</code> )	coordinate value

### 1.7.15 setNodeDisp command

**setNodeDisp** (*nodeTag*, *dim*, *value*, '-commit')

set the nodal displacement at the specified dimension.

nodeTag ( <code>int</code> )	node tag.
dim ( <code>int</code> )	the dimension of the disipinate to be set.
value ( <code>float</code> )	displacement value
'-commit' ( <code>str</code> )	commit nodal state. (optional)

### 1.7.16 setNodeVel command

**setNodeVel** (*nodeTag*, *dim*, *value*, '-commit')

set the nodal velocity at the specified dimension.

nodeTag ( <code>int</code> )	node tag.
dim ( <code>int</code> )	the dimension of the velinate to be set.
value ( <code>float</code> )	velocity value
'-commit' ( <code>str</code> )	commit nodal state. (optional)

### 1.7.17 setNodeAccel command

**setNodeAccel** (*nodeTag*, *dim*, *value*, '-commit')

set the nodal acceleration at the specified dimension.

nodeTag ( <code>int</code> )	node tag.
dim ( <code>int</code> )	the dimension of the accelinate to be set.
value ( <code>float</code> )	acceleration value
'-commit' ( <code>str</code> )	commit nodal state. (optional)

### 1.7.18 setPrecision command

**setPrecision** (*precision*)

Set the precision for screen output.

precision (int)	the precision number.
-----------------	-----------------------

### 1.7.19 setElementRayleighDampingFactors command

**setElementRayleighDampingFactors** (*eleTag, alphaM, betaK, betaK0, betaKc*)

Set the *rayleigh ()* damping for an element.

eleTag (int)	element tag
alphaM (float)	factor applied to elements or nodes mass matrix
betaK (float)	factor applied to elements current stiffness matrix.
betaK0 (float)	factor applied to elements initial stiffness matrix.
betaKc (float)	factor applied to elements committed stiffness matrix.

### 1.7.20 start command

**start ()**

Start the timer

### 1.7.21 stop command

**stop ()**

Stop the timer and print timing information.

### 1.7.22 stripXML command

**stripXML** (*inputxml, outputdata, outputxml*)

Strip a xml file to a data file and a descriptive file.

inputxml (str)	input xml file name.
outputdata (str)	output data file name.
outputxml (str)	output xml file name.

### 1.7.23 updateElementDomain command

**updateElementDomain ()**

Update elements in the domain.

### 1.7.24 updateMaterialStage

**updateMaterialStage** ('-material', *matTag*, '-stage', *value*, '-parameter', *paramTag*)

This function is used in geotechnical modeling to maintain elastic nDMaterial response during the application

of gravity loads. The material is then updated to allow for plastic strains during additional static loads or earthquakes.

matTag (int)	tag of nDMaterial
value (int)	stage value
paramTag (int)	tag of parameter (optional)

## 1.7.25 wipe command

### wipe()

This command is used to destroy all constructed objects, i.e. all components of the model, all components of the analysis and all recorders.

This command is used to start over without having to exit and restart the interpreter. It causes all elements, nodes, constraints, loads to be removed from the domain. In addition it deletes all recorders, analysis objects and all material objects created by the model builder.

## 1.7.26 wipeAnalysis command

### wipeAnalysis()

This command is used to destroy all components of the Analysis object, i.e. any objects created with system, numberer, constraints, integrator, algorithm, and analysis commands.

## 1.7.27 setNumthread command

### setNumThread(num)

set the number of threads to be used in the multi-threaded environment.

num (int)	number of threads
-----------	-------------------

## 1.7.28 getNumthread command

### getNumThread(num)

return the total number of threads available

## 1.7.29 plot\_model command

### plot\_model()

Once the model is built, it can be visualized using this command. Node and element numbers are displayed (default). Matplotlib and Numpy are required. No analysis is required in order to visualize the model.

## 1.7.30 plot\_modeshape command

### plot\_modeshape(mode\_number)

Any modeshape can be visualized using this command after an Eigen analysis is performed. Matplotlib and Numpy are required.

mode_number (int)	mode number to visualize (Integer). For example: plot_modeshape(3).
-------------------	---

## 1.8 FSI Commands

These commands are related to the Fluid-Structure Interaction analysis in OpenSees.

1. [mesh command](#)
2. [remesh command](#)
3. [PFEM integrator](#)
4. [PFEM SOE](#)
5. [PFEM test](#)
6. [PFEM analysis](#)

### 1.8.1 mesh command

**mesh** (*type, tag, \*args*)

Create a mesh object. See below for available mesh types.

#### line mesh

**mesh** ('line', *tag, numnodes, \*ndtags, id, ndf, meshsize, eleType=*', *\*eleArgs=[]*)

Create a line mesh object.

<b>tag</b> (int)	mesh tag.
<b>numnodes</b> (int)	number of nodes for defining consecutive lines.
<b>ndtags</b> (list (int))	the node tags
<b>id</b> (int)	mesh id. Meshes with same id are considered as same structure of fluid identity. <ul style="list-style-type: none"><li>• <i>id</i> = 0 : not in FSI</li><li>• <i>id</i> &gt; 0 : structure</li><li>• <i>id</i> &lt; 0 : fluid</li></ul>
<b>ndf</b> (int)	ndf for nodes to be created.
<b>meshsize</b> (float)	mesh size.
<b>eleType</b> (str)	the type of the element, (optional) <ul style="list-style-type: none"><li>• <i>Elastic Beam Column Element</i></li><li>• <i>forceBeamColumn</i></li><li>• <i>dispBeamColumn</i></li></ul> if no type is given, only nodes are created
<b>eleArgs</b> (list)	a list of element arguments. The arguments are same as in the element commands, but without element tag, and node tags. (optional) For example, <code>eleArgs = ['elasticBeamColumn', A, E, Iz, transfTag]</code>

#### triangular mesh

**mesh** ('tri', *tag, numlines, \*ltags, id, ndf, meshsize, eleType=*', *\*eleArgs=[]*)

Create a triangular mesh object.

<code>tag (int)</code>	mesh tag.
<code>numlines (int)</code>	number of lines ( <i>line mesh</i> ) for defining a polygon.
<code>ltags (list (int))</code>	the <i>line mesh</i> tags
<code>id (int)</code>	mesh id. Meshes with same id are considered as same structure of fluid identity. <ul style="list-style-type: none"><li>• <code>id = 0</code> : not in FSI</li><li>• <code>id &gt; 0</code> : structure</li><li>• <code>id &lt; 0</code> : fluid</li></ul>
<code>ndf (int)</code>	ndf for nodes to be created.
<code>meshsize (float)</code>	mesh size.
<code>eleType (str)</code>	the element type, (optional) <ul style="list-style-type: none"><li>• <i>PFEMElementBubble</i></li><li>• <i>PFEMElementCompressible</i></li><li>• <i>Tri31 Element</i></li></ul> if no type is given, only nodes are created
<code>eleArgs (list)</code>	a list of element arguments. The arguments are same as in the element commands, but without element tag, and node tags. (optional) For example, <code>eleArgs = ['PFEMElementBubble', rho, mu, b1, b2, thickness, kappa]</code>

## particle mesh

`mesh ('part', tag, type, *pArgs, eleType=”, *eleArgs=[], '-vel', *vel0, '-pressure', p0)`

Create a particle mesh which is used for background mesh.

<code>tag (int)</code>	mesh tag.
<code>type (str)</code>	type of the mesh
<code>pArgs (list (float))</code>	coordinates of points defining the mesh region <ul style="list-style-type: none"> <li>• 'quad' : [x1, y1, x2, y2, x3, y3, x4, y4, nx, ny] Coordinates of four corners in counter-clock wise order.</li> <li>• 'cube' [[x1, y1, z1, x2, y2, z2, x3, y3, z3, x4, y4, z4, x5, y5, z5, x6, y6, z6, x7, y7, z7, x8, y8, z8, nx, ny, nz] Coordinates of four corners at bottom and at top in counter-clock wise order</li> <li>• 'tri' : [x1, y1, x2, y2, x3, y3, nx, ny] Coordinates of three corners in counter-clock wise order</li> <li>• 'line' : [x1, y1, x2, y2, nx] Coordinates of two ends in counter-clock wise order</li> <li>• 'point' : [x1, y1] Coordinates of the point in counter-clock wise order</li> </ul> nx, ny, nz are number of particles in x, y, and z directions
<code>eleType (str)</code>	the element type, (optional) <ul style="list-style-type: none"> <li>• <i>PFEMElementBubble</i></li> <li>• <i>PFEMElementCompressible</i></li> <li>• <i>Tri31 Element</i></li> </ul> if no type is given, only nodes are created
<code>eleArgs (list)</code>	a list of element arguments. (optional, see <i>line mesh</i> and <i>triangular mesh</i> )
<code>vel0 (list (float))</code>	a list of initial velocities. (optional)
<code>p0 (float)</code>	initial pressure. (optional)

## background mesh

`mesh ('bg', basicsize, *lower, *upper, '-tol', tol, '-meshtol', meshtol, '-wave', wavefilename, numl, *locations, '-numsub', numsub, '-wall', *wlower, *wupper, '-structure', numsnodes, *sNodes)`  
Create a background mesh. Structural mesh must be created before the background mesh in order to be included.

<code>basicsize (float)</code>	basic mesh size
<code>lower (list (float))</code>	a list of coordinates of the lower point of the background region.
<code>upper (list (float))</code>	a list of coordinates of the upper point of the background region.
<code>tol (float)</code>	tolerance for intri check. (optional, default 1e-10)
<code>meshtol (float)</code>	tolerance for cell boundary check. (optional, default 0.1)
<code>wavefilename (str)</code>	a filename to record wave heights and velocities (optional)
<code>numl (int)</code>	number of locations to record wave (optional)
<code>locations (list (float))</code>	coordinates of the locations (optional)
<code>wlower (list (float))</code>	a list of coordinates of the lower point of the wall (optional)
<code>wupper (list (float))</code>	a list of coordinates of the upper point of the wall (optional)
<code>numsnodes (int)</code>	number of structural nodes (optional)
<code>sNodes (list (int))</code>	a list of structural nodes (optional)

## 1.8.2 remesh command

**remesh** (*alpha=-1.0*)

- $\alpha \geq 0$  for updating moving mesh.
- $\alpha < 0$  for updating background mesh.

alpha (float)	Parameter for the $\alpha$ method to construct a mesh from the node cloud of moving meshes. (optional) <ul style="list-style-type: none"> <li>• <math>\alpha = 0</math> : no elements are created</li> <li>• large <math>\alpha</math> : all elements in the convex hull are created</li> <li>• <math>1.0 &lt; \alpha &lt; 2.0</math> : usually gives a good shape</li> </ul>
---------------	---

## 1.8.3 PFEM integrator

**integrator** ('PFEM')

Create a PFEM Integrator.

## 1.8.4 PFEM SOE

**system** ('PFEM', '-compressible', '-mumps')

Create a incompressible PFEM system of equations using the Umfpack solver

-compressible	Solve using a quasi-incompressible formulation. (optional)
-mumps	Solve using the MUMPS solver. (optional, not supported on Windows)

## 1.8.5 PFEM test

**test** ('PFEM', *tolv*, *tolp*, *tolrv*, *tolrp*, *tolrelv*, *tolrelp*, *iter*, *maxincr*, *pFlag=0*, *nType=2*)

Create a PFEM test, which check both increments and residual for velocities and pressures.

<code>tolv (float)</code>	Tolerance for velocity increments
<code>tolp (float)</code>	Tolerance for pressure increments
<code>tolrv (float)</code>	Tolerance for velocity residual
<code>tolrp (float)</code>	Tolerance for pressure residual
<code>tolrv (float)</code>	Tolerance for relative velocity increments
<code>tolrp (float)</code>	Tolerance for relative pressure increments
<code>iter (int)</code>	Max number of iterations to check
<code>maxincr (int)</code>	Max times for error increasing
<code>pFlag (int)</code>	Print flag (optional): <ul style="list-style-type: none"><li>• 0 print nothing.</li><li>• 1 print information on norms each time <code>test()</code> is invoked.</li><li>• 2 print information on norms and number of iterations at end of successful test.</li><li>• 4 at each step it will print the norms and also the <math>\Delta U</math> and <math>R(U)</math> vectors.</li><li>• 5 if it fails to converge at end of <code>numIter</code> it will print an error message <b>but return a successfull test.</b></li></ul>
<code>nType (int)</code>	Type of norm, (0 = max-norm, 1 = 1-norm, 2 = 2-norm). (optional)

## 1.8.6 PFEM analysis

`analysis ('PFEM', dtmax, dtmin, gravity, ratio=0.5)`

Create a OpenSees PFEMAnalysis object.

<code>dtmax (float)</code>	Maximum time steps.
<code>dtmin (float)</code>	Mimimum time steps.
<code>gravity (float)</code>	Gravity acceleration used to move isolated particles.
<code>ratio (float)</code>	The ratio to reduce time steps if it was not converged. (optional)

## 1.9 Sensitivity Commands

These commands are for sensitivity analysis in OpenSees.

1. *parameter command*
2. *addToParameter command*
3. *updateParameter command*
4. *setParameter command*
5. *getParamTags command*
6. *getParamValue command*
7. *computeGradients command*
8. *sensitivityAlgorithm command*
9. *sensNodeDisp command*

10. *sensNodeVel command*
11. *sensNodeAccel command*
12. *sensLambda command*
13. *sensSectionForce command*
14. *sensNodePressure command*

### 1.9.1 parameter command

**parameter** (*tag*, <specific parameter args>)

In DDM-based FE response sensitivity analysis, the sensitivity parameters can be material, geometry or discrete loading parameters.

<code>tag (int)</code>	integer tag identifying the parameter.
<specific parameter args>	depend on the object in the FE model encapsulating the desired parameters.

---

**Note:** Each parameter must be unique in the FE domain, and all parameter tags must be numbered sequentially starting from 1.

---

#### Examples

1. To identify the elastic modulus, E, of the material 1 at section 3 of element 4, the <specific object arguments> string becomes:

```
parameter 1 element 4 section 3 material 1 E
```

2. To identify the elastic modulus, E, of elastic section 3 of element 4 (for elastic section, no specific material need to be defined), the <specific object arguments> string becomes:

```
parameter 1 element 4 section 3 E
```

3. To parameterize E for element 4 with material 1 (no section need to be defined), the <specific object arguments> string simplifies as:

```
parameter 1 element 4 material 1 E
```

---

**Note:** Notice that the format of the <specific object arguments> is different for each considered element/section/material. The specific set of parameters and the relative <specific object arguments> format will be added in the future.

---

### 1.9.2 addToParameter command

**addToParameter** (*tag*, <specific parameter args>)

In case that more objects (e.g., element, section) are mapped to an existing parameter, the command can be used to relate these additional objects to the specific parameter.

tag (int)	integer tag identifying the parameter.
<specific parameter args>	depend on the object in the FE model encapsulating the desired parameters.

### 1.9.3 updateParameter command

**updateParameter** (*tag*, *newValue*)

Once the parameters in FE model are defined, their value can be updated.

tag (int)	integer tag identifying the parameter.
newValue (float)	the updated value to which the parameter needs to be set.

---

**Note:** Scott M.H., Haukaas T. (2008). “Software framework for parameter updating and finite element response sensitivity analysis.” Journal of Computing in Civil Engineering, 22(5):281-291.

---

### 1.9.4 setParameter command

**setParameter** ('-val', *newValue*[, '-ele', \**eleTags* ][, '-eleRange', *start*, *end* ][, \**args* ])

set value for an element parameter

<i>newValue</i> (float)	the updated value to which the parameter needs to be set.
<i>eleTags</i> (list (int))	a list of element tags
<i>start</i> (int)	start element tag
<i>end</i> (int)	end element tag
<i>args</i> (list (str))	a list of strings for the element parameter

### 1.9.5 getParamTags command

**getParamTags** ()

Return a list of tags for all parameters

### 1.9.6 getParamValue command

**getParamValue** (*tag*)

Return the value of a parameter

tag (int)	integer tag identifying the parameter.
-----------	--

### 1.9.7 computeGradients command

**computeGradients** ()

This command is used to perform a sensitivity analysis. If the user wants to call this command, then the '-computeByCommand' should be set in the `sensitivityAlgorithm` command.

## 1.9.8 sensitivityAlgorithm command

**sensitivityAlgorithm** (*type*)

This command is used to create a sensitivity algorithm.

type (str)	the type of the sensitivity algorithm, <ul style="list-style-type: none"> <li>• '-computeAtEachStep' automatically compute at the end of each step</li> <li>• '-computeByCommand' compute by calling computeGradients.</li> </ul>
------------	--

## 1.9.9 sensNodeDisp command

**sensNodeDisp** (*nodeTag*, *dof*, *paramTag*)

Returns the current displacement sensitivity to a parameter at a specified node.

nodeTag (int)	node tag
dof (int)	specific dof at the node (1 through ndf)
paramTag (int)	parameter tag

## 1.9.10 sensNodeVel command

**sensNodeVel** (*nodeTag*, *dof*, *paramTag*)

Returns the current velocity sensitivity to a parameter at a specified node.

nodeTag (int)	node tag
dof (int)	specific dof at the node (1 through ndf)
paramTag (int)	parameter tag

## 1.9.11 sensNodeAccel command

**sensNodeAccel** (*nodeTag*, *dof*, *paramTag*)

Returns the current acceleration sensitivity to a parameter at a specified node.

nodeTag (int)	node tag
dof (int)	specific dof at the node (1 through ndf)
paramTag (int)	parameter tag

## 1.9.12 sensLambda command

**sensLambda** (*patternTag*, *paramTag*)

Returns the current load factor sensitivity to a parameter in a load pattern.

patternTag (int)	load pattern tag
paramTag (int)	parameter tag

### 1.9.13 sensSectionForce command

**sensSectionForce** (*eleTag*[, *secNum*], *dof*, *paramTag*)

Returns the current section force sensitivity to a parameter at a specified element and section.

<code>eleTag</code> (int)	element tag
<code>secNum</code> (int)	section number (optional)
<code>dof</code> (int)	specific dof at the element (1 through element force ndf)
<code>paramTag</code> (int)	parameter tag

### 1.9.14 sensNodePressure command

**sensNodePressure** (*nodeTag*, *paramTag*)

Returns the current pressure sensitivity to a parameter at a specified node.

<code>nodeTag</code> (int)	node tag
<code>paramTag</code> (int)	parameter tag

## 1.10 Reliability Commands

These commands are for reliability analysis in OpenSees.

1. *randomVariable command*

### 1.10.1 randomVariable command

**randomVariable** (*tag*, *dist*, '-mean', *mean*, '-stdv', *stdv*, '-startPoint', *startPoint*, '-parameters', \**params*)

Create a random variable with user specified distribution

<code>tag (int)</code>	random variable tag
<code>dist (str)</code>	random variable distribution <ul style="list-style-type: none"> <li>• 'normal'</li> <li>• 'lognormal'</li> <li>• 'gamma'</li> <li>• 'shiftedExponential'</li> <li>• 'shiftedRayleigh'</li> <li>• 'exponential'</li> <li>• 'rayleigh'</li> <li>• 'uniform'</li> <li>• 'beta'</li> <li>• 'type1LargestValue'</li> <li>• 'type1SmallestValue'</li> <li>• 'type2LargestValue'</li> <li>• 'type3SmallestValue'</li> <li>• 'chiSquare'</li> <li>• 'gumbel'</li> <li>• 'weibull'</li> <li>• 'laplace'</li> <li>• 'pareto'</li> </ul>
<code>mean (float)</code>	mean value
<code>stdv (float)</code>	standard deviation
<code>startPoint (float)</code>	starting point of the distribution
<code>params (list (int))</code>	a list of parameter tags

## 1.11 Parallel Commands

The parallel commands are currently only working in the Linux version. The parallel OpenSeesPy is similar to OpenSeesMP, which requires users to divide the model to distributed processors.

You can still run the single-processor version as before. To run the parallel version, you have to install a MPI implementation, such as `mpich`. Then call your python scripts in the command line

```
mpiexec -np np python filename.py
```

where `np` is the number of processors to be used, `python` is the python interpreter, and `filename.py` is the script name.

Inside the script, OpenSeesPy is still imported as

```
import openseespy.opensees as ops
```

Common problems:

1. Unmatch send/recv will cause deadlock.
2. Writing to the same files at the same from different processors will cause race conditions.
3. Poor model decomposition will cause load imbalance problem.

Following are commands related to parallel computing:

1. `getPID command`
2. `getNP command`

3. *barrier command*
4. *send command*
5. *recv command*
6. *Bcast command*
7. *setStartNodeTag command*
8. *domainChange command*
9. *Parallel Plain Numberer*
10. *Parallel RCM Numberer*
11. *MUMPS Solver*
12. *Parallel DisplacementControl*

### 1.11.1 getPID command

**getPID ()**  
Get the processor ID of the calling processor.

### 1.11.2 getNP command

**getNP ()**  
Get total number of processors.

### 1.11.3 barrier command

**barrier ()**  
Set a barrier for all processors, i.e., faster processors will pause here to wait for all processors to reach to this point.

### 1.11.4 send command

**send ('-pid', pid, \*data)**  
Send information to another processor.

<code>pid (int)</code>	ID of processor where data is sent to
<code>data (list (int))</code>	can be a list of integers
<code>data (list (float))</code>	can be a list of floats
<code>data (str)</code>	can be a string

---

**Note:** *send command* and *recv command* must match and the order of calling both commands matters.

---

### 1.11.5 recv command

**recv** ('-pid', pid)

Receive information from another processor.

pid (int)	ID of processor where data is received from
pid (str)	if pid is 'ANY', the processor can receive data from any processor.

---

**Note:** *send command* and *recv command* must match and the order of calling both commands matters.

---

### 1.11.6 Bcast command

**Bcast** (data)

Broadcast information from processor 0 to all processors.

data (list (int))	can be a list of integers
data (list (float))	can be a list of floats
data (str)	can be a string

### 1.11.7 setStartNodeTag command

**setStartNodeTag** (ndtag)

Set the starting node tag for the *mesh command*. The purpose of this command is to control the node tags generated by the *mesh command*. Some nodes are shared by processors, which must have same tags. Nodes which are unique to a processor must have uniques tags across all processors.

ndtag (int)	starting node tag for the next call of <i>mesh command</i>
-------------	--

### 1.11.8 domainChange command

**domainChange** ()

Mark the domain has changed manually. This is used to notify processors whose domain is not changed, but the domain in other processors have changed.

## 1.12 Examples

1. *Structural Examples*
2. *Earthquake Examples*
3. *Tsunami Examples*
4. *GeoTechnical Examples*
5. *Thermal Examples*
6. *Parallel Examples*
7. *Plotting Examples*

### 1.12.1 Structural Examples

1. *Elastic Truss Analysis*
2. *Nonlinear Truss Analysis*
3. *Portal Frame 2d Analysis*
4. *Moment Curvature Analysis*
5. *Reinforced Concrete Frame Gravity Analysis*
6. *Reinforced Concrete Frame Pushover Analysis*
7. *Three story steel building with rigid beam-column connections and W-section*

#### Elastic Truss Analysis

1. The source code is shown below, which can be downloaded [here](#).
2. Run the source code in your favorite Python program and should see Passed! in the results.

```
1 from openseespy.opensees import *
2
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 # -----
7 # Start of model generation
8 # -----
9
10 # remove existing model
11 wipe()
12
13 # set modelbuilder
14 model('basic', '-ndm', 2, '-ndf', 2)
15
16 # create nodes
17 node(1, 0.0, 0.0)
18 node(2, 144.0, 0.0)
19 node(3, 168.0, 0.0)
20 node(4, 72.0, 96.0)
21
22 # set boundary condition
23 fix(1, 1, 1)
24 fix(2, 1, 1)
25 fix(3, 1, 1)
26
27 # define materials
28 uniaxialMaterial("Elastic", 1, 3000.0)
29
30 # define elements
31 element("Truss", 1, 1, 4, 10.0, 1)
32 element("Truss", 2, 2, 4, 5.0, 1)
33 element("Truss", 3, 3, 4, 5.0, 1)
34
35 # create TimeSeries
36 timeSeries("Linear", 1)
```

(continues on next page)

(continued from previous page)

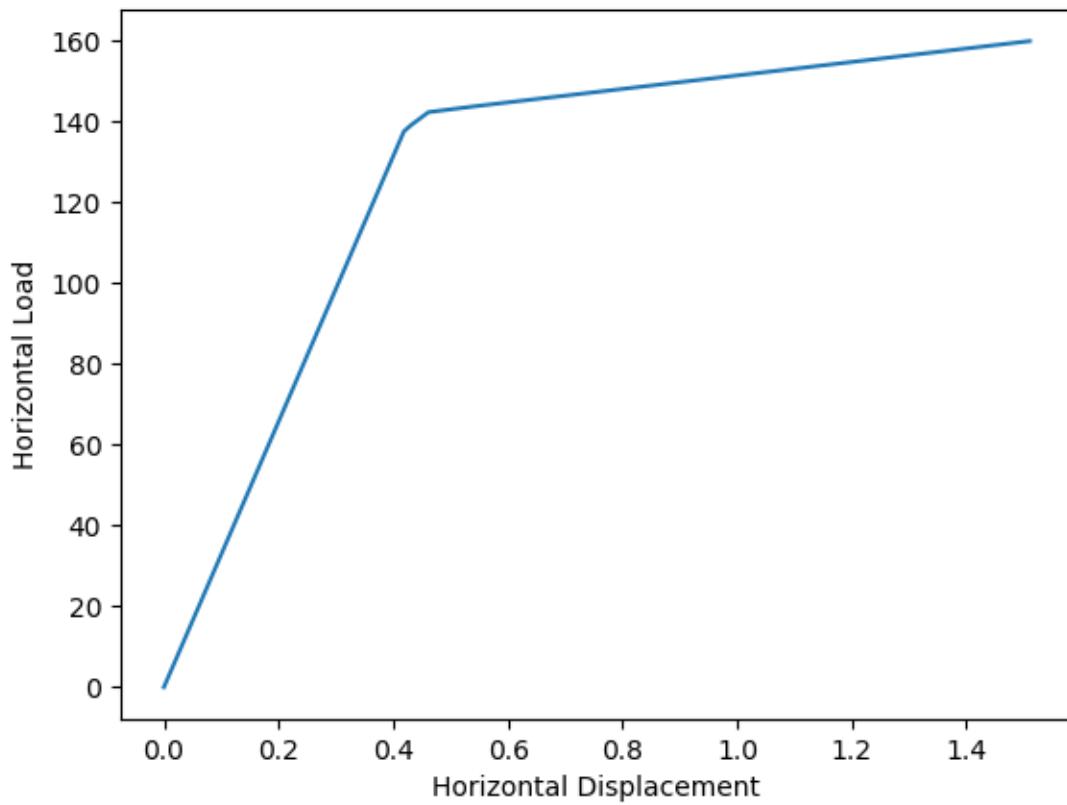
```

38 # create a plain load pattern
39 pattern("Plain", 1, 1)
40
41 # Create the nodal load - command: load nodeID xForce yForce
42 load(4, 100.0, -50.0)
43
44 # -----
45 # Start of analysis generation
46 # -----
47
48 # create SOE
49 system("BandSPD")
50
51 # create DOF number
52 numberer("RCM")
53
54 # create constraint handler
55 constraints("Plain")
56
57 # create integrator
58 integrator("LoadControl", 1.0)
59
60 # create algorithm
61 algorithm("Linear")
62
63 # create analysis object
64 analysis("Static")
65
66 # perform the analysis
67 analyze(1)
68
69 ux = nodeDisp(4,1)
70 uy = nodeDisp(4,2)
71 if abs(ux-0.53009277713228375450)<1e-12 and abs(uy+0.17789363846931768864)<1e-12:
72     print("Passed!")
73 else:
74     print("Failed!")

```

## Nonlinear Truss Analysis

1. The source code is shown below, which can be downloaded [here](#).
2. Make sure the `numpy` and `matplotlib` packages are installed in your Python distribution.
3. Run the source code in your favorite Python program and should see



```
1 from openseespy.opensees import *
2
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 # -----
7 # Start of model generation
8 # -----
9
10 # set modelbuilder
11 wipe()
12 model('basic', '-ndm', 2, '-ndf', 2)
13
14 # variables
15 A = 4.0
16 E = 29000.0
17 alpha = 0.05
18 sY = 36.0
19 udisp = 2.5
20 Nsteps = 1000
21 Px = 160.0
22 Py = 0.0
23
24 # create nodes
25 node(1, 0.0, 0.0)
```

(continues on next page)

(continued from previous page)

```

26 node(2, 72.0, 0.0)
27 node(3, 168.0, 0.0)
28 node(4, 48.0, 144.0)
29
30 # set boundary condition
31 fix(1, 1, 1)
32 fix(2, 1, 1)
33 fix(3, 1, 1)
34
35 # define materials
36 uniaxialMaterial("Hardening", 1, E, sY, 0.0, alpha/(1-alpha)*E)
37
38 # define elements
39 element("Truss", 1, 1, 4, A, 1)
40 element("Truss", 2, 2, 4, A, 1)
41 element("Truss", 3, 3, 4, A, 1)
42
43 # create TimeSeries
44 timeSeries("Linear", 1)
45
46 # create a plain load pattern
47 pattern("Plain", 1, 1)
48
49 # Create the nodal load
50 load(4, Px, Py)
51
52 # -----
53 # Start of analysis generation
54 # -----
55
56 # create SOE
57 system("ProfileSPD")
58
59 # create DOF number
60 numberer("Plain")
61
62 # create constraint handler
63 constraints("Plain")
64
65 # create integrator
66 integrator("LoadControl", 1.0/Nsteps)
67
68 # create algorithm
69 algorithm("Newton")
70
71 # create test
72 test('NormUnbalance', 1e-8, 10)
73
74 # create analysis object
75 analysis("Static")
76
77 # -----
78 # Finally perform the analysis
79 # -----
80
81 # perform the analysis
82 data = np.zeros((Nsteps+1, 2))

```

(continues on next page)

(continued from previous page)

```

83 for j in range(Nsteps):
84     analyze(1)
85     data[j+1,0] = nodeDisp(4,1)
86     data[j+1,1] = getLoadFactor(1)*Px
87
88 plt.plot(data[:,0], data[:,1])
89 plt.xlabel('Horizontal Displacement')
90 plt.ylabel('Horizontal Load')
91 plt.show()
92

```

## Portal Frame 2d Analysis

1. The source code is shown below, which can be downloaded [here](#).
2. Run the source code in your favorite Python program and should see results below

Period Comparisons:			
Period	OpenSees	SAP2000	SeismoStruct
1	1.27321	1.2732	1.2732
2	0.43128	0.4313	0.4313
3	0.24204	0.2420	0.2420
4	0.16018	0.1602	0.1602
5	0.11899	0.1190	0.1190
6	0.09506	0.0951	0.0951
7	0.07951	0.0795	0.0795

tStatic Analysis Result Comparisons:				
	Parameter	OpenSees	SAP2000	SeismoStruct
	Disp Top	1.451	1.45	1.45
Axial Force Bottom Left		69.987	69.99	70.01
Moment Bottom Left		2324.677	2324.68	2324.71

PASSED Verification Test PortalFrame2d.py

```

1  from openseespy.opensees import *
2
3  from math import asin, sqrt
4
5  # Two dimensional Frame: Eigenvalue & Static Loads
6
7
8  # REFERENCES:
9  # used in verification by SAP2000:
10 # SAP2000 Integrated Finite Element Analysis and Design of Structures, Verification Manual,
11 # Computers and Structures, 1997. Example 1.
12 # and seismo-struct (Example 10)
13 # SeismoStruct, Verification Report For Version 6, 2012. Example 11.
14
15
16 # set some properties
17 wipe()
18
19 model('Basic', '-ndm', 2)

```

(continues on next page)

(continued from previous page)

```

20
21 # properties
22
23 #     units kip, ft
24
25 numBay = 2
26 numFloor = 7
27
28 bayWidth = 360.0
29 storyHeights = [162.0, 162.0, 156.0, 156.0, 156.0, 156.0, 156.0]
30
31 E = 29500.0
32 massX = 0.49
33 M = 0.
34 coordTransf = "Linear" # Linear, PDelta, Corotational
35 massType = "-lMass" # -lMass, -cMass
36
37 beams = ['W24X160', 'W24X160', 'W24X130', 'W24X130', 'W24X110', 'W24X110', 'W24X110']
38 eColumn = ['W14X246', 'W14X246', 'W14X246', 'W14X211', 'W14X211', 'W14X176', 'W14X176'
39     ↪']
40 iColumn = ['W14X287', 'W14X287', 'W14X287', 'W14X246', 'W14X246', 'W14X211', 'W14X211'
41     ↪']
42 columns = [eColumn, iColumn, eColumn]
43
44 WSection = {
45     'W14X176': [51.7, 2150.],
46     'W14X211': [62.1, 2670.],
47     'W14X246': [72.3, 3230.],
48     'W14X287': [84.4, 3910.],
49     'W24X110': [32.5, 3330.],
50     'W24X130': [38.3, 4020.],
51     'W24X160': [47.1, 5120.]
52 }
53
54 nodeTag = 1
55
56 # procedure to read
57 def ElasticBeamColumn(eleTag, iNode, jNode, sectType, E, transfTag, M, massType):
58     found = 0
59
60     prop = WSection[sectType]
61
62     A = prop[0]
63     I = prop[1]
64     element('elasticBeamColumn', eleTag, iNode, jNode, A, E, I, transfTag, '-mass', M,
65             ↪ massType)
66
67 # add the nodes
68 # - floor at a time
69 yLoc = 0.
70 for j in range(0, numFloor + 1):
71
72     xLoc = 0.
73     for i in range(0, numBay + 1):
74         node(nodeTag, xLoc, yLoc)

```

(continues on next page)

(continued from previous page)

```
74     xLoc += bayWidth
75     nodeTag += 1
76
77     if j < numFloor:
78         storyHeight = storyHeights[j]
79
80     yLoc += storyHeight
81
82 # fix first floor
83 fix(1, 1, 1, 1)
84 fix(2, 1, 1, 1)
85 fix(3, 1, 1, 1)
86
87 # rigid floor constraint & masses
88 nodeTagR = 5
89 nodeTag = 4
90 for j in range(1, numFloor + 1):
91     for i in range(0, numBay + 1):
92
93         if nodeTag != nodeTagR:
94             equalDOF(nodeTagR, nodeTag, 1)
95         else:
96             mass(nodeTagR, massX, 1.0e-10, 1.0e-10)
97
98         nodeTag += 1
99
100    nodeTagR += numBay + 1
101
102 # add the columns
103 # add column element
104 geomTransf(coordTransf, 1)
105 eleTag = 1
106 for j in range(0, numBay + 1):
107
108     end1 = j + 1
109     end2 = end1 + numBay + 1
110     thisColumn = columns[j]
111
112     for i in range(0, numFloor):
113         secType = thisColumn[i]
114         ElasticBeamColumn(eleTag, end1, end2, secType, E, 1, M, massType)
115         end1 = end2
116         end2 += numBay + 1
117         eleTag += 1
118
119 # add beam elements
120 for j in range(1, numFloor + 1):
121     end1 = (numBay + 1) * j + 1
122     end2 = end1 + 1
123     secType = beams[j - 1]
124     for i in range(0, numBay):
125         ElasticBeamColumn(eleTag, end1, end2, secType, E, 1, M, massType)
126         end1 = end2
127         end2 = end1 + 1
128         eleTag += 1
129
130 # calculate eigenvalues & print results
```

(continues on next page)

(continued from previous page)

```

131 numEigen = 7
132 eigenValues = eigen(numEigen)
133 PI = 2 * asin(1.0)
134
135 #
136 # apply loads for static analysis & perform analysis
137 #
138
139 timeSeries('Linear', 1)
140 pattern('Plain', 1, 1)
141 load(22, 20.0, 0., 0.)
142 load(19, 15.0, 0., 0.)
143 load(16, 12.5, 0., 0.)
144 load(13, 10.0, 0., 0.)
145 load(10, 7.5, 0., 0.)
146 load(7, 5.0, 0., 0.)
147 load(4, 2.5, 0., 0.)
148
149 integrator('LoadControl', 1.0)
150 algorithm('Linear')
151 analysis('Static')
152 analyze(1)
153
154 # determine PASS/FAILURE of test
155 ok = 0
156
157 #
158 # print pretty output of comparisons
159 #
160
161 # SAP2000 SeismoStruct
162 comparisonResults = [[1.2732, 0.4313, 0.2420, 0.1602, 0.1190, 0.0951, 0.0795],
163 [1.2732, 0.4313, 0.2420, 0.1602, 0.1190, 0.0951, 0.0795]]
164 print("\n\nPeriod Comparisons:")
165 print('{:>10}{:>15}{:>15}{:>15}'.format('Period', 'OpenSees', 'SAP2000', 'SeismoStruct',
166     ↪))
167
168 # formatString %10s%15.5f%15.4f%15.4f
169 for i in range(0, numEigen):
170     lamb = eigenValues[i]
171     period = 2 * PI / sqrt(lamb)
172     print('{:>10}{:>15.5f}{:>15.4f}{:>15.4f}'.format(i + 1, period,
173         ↪comparisonResults[0][i], comparisonResults[1][i]))
174     resultOther = comparisonResults[0][i]
175     if abs(period - resultOther) > 9.99e-5:
176         ok -= 1
177
178 # print table of comparision
179 # Parameter SAP2000 SeismoStruct
180 comparisonResults = [["Disp Top", "Axial Force Bottom Left", "Moment Bottom Left"],
181 [1.45076, 69.99, 2324.68],
182 [1.451, 70.01, 2324.71]]
183 tolerances = [9.99e-6, 9.99e-3, 9.99e-3]
184
185 print("\n\nStatic Analysis Result Comparisons:")
186 print('{:>30}{:>15}{:>15}{:>15}'.format('Parameter', 'OpenSees', 'SAP2000',
187     ↪'SeismoStruct'))
```

(continues on next page)

(continued from previous page)

```
185 for i in range(3):
186     response = eleResponse(1, 'forces')
187     if i == 0:
188         result = nodeDisp(22, 1)
189     elif i == 1:
190         result = abs(response[1])
191     else:
192         result = response[2]
193
194     print('{:>30}{:>15.3f}{:>15.2f}{:>15.2f}'.format(comparisonResults[0][i],
195                                                 result,
196                                                 comparisonResults[1][i],
197                                                 comparisonResults[2][i]))
198
199     resultOther = comparisonResults[1][i]
200     tol = tolerances[i]
201     if abs(result - resultOther) > tol:
202         ok = 1
203         print("failed-> ", i, abs(result - resultOther), tol)
204
205 if ok == 0:
206     print("PASSED Verification Test PortalFrame2d.py \n\n")
207 else:
208     print("FAILED Verification Test PortalFrame2d.py \n\n")
```

## Moment Curvature Analysis

1. The source code is shown below, which can be downloaded [here](#).
2. Run the source code in your favorite Python program and should see results below

```
Start MomentCurvature.py example
Estimated yield curvature:  0.000126984126984127
Passed!
=====
```

```
1 from openseespy.opensees import *
2
3 def MomentCurvature(secTag, axialLoad, maxK, numIncr=100):
4
5     # Define two nodes at (0,0)
6     node(1, 0.0, 0.0)
7     node(2, 0.0, 0.0)
8
9     # Fix all degrees of freedom except axial and bending
10    fix(1, 1, 1, 1)
11    fix(2, 0, 1, 0)
12
13    # Define element
14    #                                     tag ndI ndJ secTag
15    element('zeroLengthSection', 1, 1, 2, secTag)
16
17    # Define constant axial load
18    timeSeries('Constant', 1)
19    pattern('Plain', 1, 1)
```

(continues on next page)

(continued from previous page)

```

20     load(2, axialLoad, 0.0, 0.0)
21
22     # Define analysis parameters
23     integrator('LoadControl', 0.0)
24     system('SparseGeneral', '-piv')
25     test('NormUnbalance', 1e-9, 10)
26     numberer('Plain')
27     constraints('Plain')
28     algorithm('Newton')
29     analysis('Static')
30
31     # Do one analysis for constant axial load
32     analyze(1)
33
34     # Define reference moment
35     timeSeries('Linear', 2)
36     pattern('Plain', 2, 2)
37     load(2, 0.0, 0.0, 1.0)
38
39     # Compute curvature increment
40     dK = maxK / numIncr
41
42     # Use displacement control at node 2 for section analysis
43     integrator('DisplacementControl', 2, 3, dK, 1, dK, dK)
44
45     # Do the section analysis
46     analyze(numIncr)
47
48
49 wipe()
50 print("Start MomentCurvature.py example")
51
52 # Define model builder
53 # -----
54 model('basic', '-ndm', 2, '-ndf', 3)
55
56 # Define materials for nonlinear columns
57 # -----
58 # CONCRETE          tag   f'c'        ec0    f'cu'       ecu
59 # Core concrete (confined)
60 uniaxialMaterial('Concrete01', 1, -6.0, -0.004, -5.0, -0.014)
61
62 # Cover concrete (unconfined)
63 uniaxialMaterial('Concrete01', 2, -5.0, -0.002, 0.0, -0.006)
64
65 # STEEL
66 # Reinforcing steel
67 fy = 60.0      # Yield stress
68 E = 30000.0    # Young's modulus
69
70 #           tag   fy E0      b
71 uniaxialMaterial('Steel01', 3, fy, E, 0.01)
72
73 # Define cross-section for nonlinear columns
74 # -----
75
76 # set some parameters

```

(continues on next page)

(continued from previous page)

```
77 colWidth = 15
78 colDepth = 24
79
80 cover = 1.5
81 As = 0.60;      # area of no. 7 bars
82
83 # some variables derived from the parameters
84 y1 = colDepth/2.0
85 z1 = colWidth/2.0
86
87
88 section('Fiber', 1)
89
90 # Create the concrete core fibers
91 patch('rect',1,10,1 ,cover-y1, cover-z1, y1-cover, z1-cover)
92
93 # Create the concrete cover fibers (top, bottom, left, right)
94 patch('rect',2,10,1 ,-y1, z1-cover, y1, z1)
95 patch('rect',2,10,1 ,-y1, -z1, y1, cover-z1)
96 patch('rect',2,2,1 ,-y1, cover-z1, cover-y1, z1-cover)
97 patch('rect',2,2,1 ,y1-cover, cover-z1, y1, z1-cover)
98
99 # Create the reinforcing fibers (left, middle, right)
100 layer('straight', 3, 3, As, y1-cover, z1-cover, y1-cover, cover-z1)
101 layer('straight', 3, 2, As, 0.0      , z1-cover, 0.0      , cover-z1)
102 layer('straight', 3, 3, As, cover-y1, z1-cover, cover-y1, cover-z1)
103
104 # Estimate yield curvature
105 # (Assuming no axial load and only top and bottom steel)
106 # d -- from cover to rebar
107 d = colDepth-cover
108 # steel yield strain
109 epsy = fy/E
110 Ky = epsy/(0.7*d)
111
112 # Print estimate to standard output
113 print("Estimated yield curvature: ", Ky)
114
115 # Set axial load
116 P = -180.0
117
118 # Target ductility for analysis
119 mu = 15.0
120
121 # Number of analysis increments
122 numIncr = 100
123
124 # Call the section analysis procedure
125 MomentCurvature(1, P, Ky*mu, numIncr)
126
127 results = open('results.out','a+')
128
129 u = nodeDisp(2,3)
130 if abs(u-0.00190476190476190541)<1e-12:
131     results.write('PASSED : MomentCurvature.py\n');
132     print("Passed!")
133 else:
```

(continues on next page)

(continued from previous page)

```

134     results.write('FAILED : MomentCurvature.py\n');
135     print("Failed!")
136
137 results.close()
138
139 print ("=====")
```

## Reinforced Concrete Frame Gravity Analysis

1. The source code is shown below, which can be downloaded [here](#).
2. Run the source code in your favorite Python program and should see Passed! in the results.

```

1 print ("=====")
2
3 from openseespy.opensees import *
4
5 print ("Starting RCFrameGravity example")
6
7 # Create ModelBuilder (with two-dimensions and 3 DOF/node)
8 model('basic', '-ndm', 2, '-ndf', 3)
9
10 # Create nodes
11 # -----
12
13 # Set parameters for overall model geometry
14 width = 360.0
15 height = 144.0
16
17 # Create nodes
18 #   tag, X, Y
19 node(1, 0.0, 0.0)
20 node(2, width, 0.0)
21 node(3, 0.0, height)
22 node(4, width, height)
23
24 # Fix supports at base of columns
25 #   tag, DX, DY, RZ
26 fix(1, 1, 1, 1)
27 fix(2, 1, 1, 1)
28
29 # Define materials for nonlinear columns
30 # -----
31 # CONCRETE           tag  f'c    ec0      f'cu    ecu
32 # Core concrete (confined)
33 uniaxialMaterial('Concrete01', 1, -6.0, -0.004, -5.0, -0.014)
34
35 # Cover concrete (unconfined)
36 uniaxialMaterial('Concrete01', 2, -5.0, -0.002, 0.0, -0.006)
37
38 # STEEL
39 # Reinforcing steel
40 fy = 60.0; # Yield stress
41 E = 30000.0; # Young's modulus
42 #           tag  fy  E0      b
```

(continues on next page)

(continued from previous page)

```
43 uniaxialMaterial('Steel01', 3, fy, E, 0.01)
44
45 # Define cross-section for nonlinear columns
46 # -----
47
48 # some parameters
49 colWidth = 15
50 colDepth = 24
51
52 cover = 1.5
53 As = 0.60 # area of no. 7 bars
54
55 # some variables derived from the parameters
56 y1 = colDepth / 2.0
57 z1 = colWidth / 2.0
58
59 section('Fiber', 1)
60
61 # Create the concrete core fibers
62 patch('rect', 1, 10, 1, cover - y1, cover - z1, y1 - cover, z1 - cover)
63
64 # Create the concrete cover fibers (top, bottom, left, right)
65 patch('rect', 2, 10, 1, -y1, z1 - cover, y1, z1)
66 patch('rect', 2, 10, 1, -y1, -z1, y1, cover - z1)
67 patch('rect', 2, 2, 1, -y1, cover - z1, cover - y1, z1 - cover)
68 patch('rect', 2, 2, 1, y1 - cover, cover - z1, y1, z1 - cover)
69
70 # Create the reinforcing fibers (left, middle, right)
71 layer('straight', 3, 3, As, y1 - cover, z1 - cover, y1 - cover, cover - z1)
72 layer('straight', 3, 2, As, 0.0, z1 - cover, 0.0, cover - z1)
73 layer('straight', 3, 3, As, cover - y1, z1 - cover, cover - y1, cover - z1)
74
75 # Define column elements
76 # -----
77
78 # Geometry of column elements
79 #           tag
80
81 geomTransf('PDelta', 1)
82
83 # Number of integration points along length of element
84 np = 5
85
86 # Lobatto integratoin
87 beamIntegration('Lobatto', 1, 1, np)
88
89 # Create the coulumns using Beam-column elements
90 #           e           tag ndI ndJ transfTag integrationTag
91 eleType = 'forceBeamColumn'
92 element(eleType, 1, 1, 3, 1, 1)
93 element(eleType, 2, 2, 4, 1, 1)
94
95 # Define beam elment
96 # -----
97
98 # Geometry of column elements
99 #           tag
```

(continues on next page)

(continued from previous page)

```

100 geomTransf('Linear', 2)

101
102 # Create the beam element
103 #           tag, ndI, ndJ, A,      E,      Iz, transfTag
104 element('elasticBeamColumn', 3, 3, 4, 360.0, 4030.0, 8640.0, 2)

105
106 # Define gravity loads
107 #
108
109 # a parameter for the axial load
110 P = 180.0; # 10% of axial capacity of columns

111
112 # Create a Plain load pattern with a Linear TimeSeries
113 timeSeries('Linear', 1)
114 pattern('Plain', 1, 1)

115
116 # Create nodal loads at nodes 3 & 4
117 #   nd FX, FY, MZ
118 load(3, 0.0, -P, 0.0)
119 load(4, 0.0, -P, 0.0)

120
121 # -----
122 # End of model generation
123 #
124
125
126 #
127 # Start of analysis generation
128 #

129
130 # Create the system of equation, a sparse solver with partial pivoting
131 system('BandGeneral')

132
133 # Create the constraint handler, the transformation method
134 constraints('Transformation')

135
136 # Create the DOF numberer, the reverse Cuthill-McKee algorithm
137 numberer('RCM')

138
139 # Create the convergence test, the norm of the residual with a tolerance of
140 # 1e-12 and a max number of iterations of 10
141 test('NormDispIncr', 1.0e-12, 10, 3)

142
143 # Create the solution algorithm, a Newton-Raphson algorithm
144 algorithm('Newton')

145
146 # Create the integration scheme, the LoadControl scheme using steps of 0.1
147 integrator('LoadControl', 0.1)

148
149 # Create the analysis object
150 analysis('Static')

151
152 #
153 # End of analysis generation
154 #
155
156

```

(continues on next page)

(continued from previous page)

```
157 # -----
158 # Finally perform the analysis
159 # -----
160
161 # perform the gravity load analysis, requires 10 steps to reach the load level
162 analyze(10)
163
164 # Print out the state of nodes 3 and 4
165 # print node 3 4
166
167 # Print out the state of element 1
168 # print ele 1
169
170 u3 = nodeDisp(3, 2)
171 u4 = nodeDisp(4, 2)
172
173 results = open('results.out', 'a+')
174
175 if abs(u3 + 0.0183736) < 1e-6 and abs(u4 + 0.0183736) < 1e-6:
176     results.write('PASSED : RCFrmeGravity.py\n')
177     print("Passed!")
178 else:
179     results.write('FAILED : RCFrmeGravity.py\n')
180     print("Failed!")
181
182 results.close()
183
184 print("=====")
```

## Reinforced Concrete Frame Pushover Analysis

1. The source code is shown below, which can be downloaded [here](#).
2. The file for gravity analysis is also needed [here](#).
3. Run the source code in your favorite Python program and should see Passed! in the results.

```
1 print("====")
2 print("Start RCFrmePushover Example")
3
4 # Units: kips, in, sec
5 #
6 # Written: GLF/MHS/fmk
7 # Date: January 2001
8 from openseespy.opensees import *
9
10 wipe()
11 #
12 # Start of Model Generation & Initial Gravity Analysis
13 #
14
15 # Do operations of Example3.1 by sourcing in the tcl file
16 import RCFrmeGravity
17 print("Gravity Analysis Completed")
18
19 # Set the gravity loads to be constant & reset the time in the domain
```

(continues on next page)

(continued from previous page)

```

20 loadConst('-time', 0.0)

21
22 # -----
23 # End of Model Generation & Initial Gravity Analysis
24 # -----
25
26
27 # -----
28 # Start of additional modelling for lateral loads
29 # -----
30
31 # Define lateral loads
32 # -----
33
34 # Set some parameters
35 H = 10.0 # Reference lateral load
36
37 # Set lateral load pattern with a Linear TimeSeries
38 pattern('Plain', 2, 1)
39
40 # Create nodal loads at nodes 3 & 4
41 #   nd   FX   FY   MZ
42 load(3, H, 0.0, 0.0)
43 load(4, H, 0.0, 0.0)
44
45 # -----
46 # End of additional modelling for lateral loads
47 # -----
48
49
50 # -----
51 # Start of modifications to analysis for push over
52 # -----
53
54 # Set some parameters
55 dU = 0.1 # Displacement increment
56
57 # Change the integration scheme to be displacement control
58 #           node dof init Jd min max
59 integrator('DisplacementControl', 3, 1, dU, 1, dU, dU)
60
61 # -----
62 # End of modifications to analysis for push over
63 # -----
64
65
66 # -----
67 # Start of recorder generation
68 # -----
69
70 # Stop the old recorders by destroying them
71 # remove recorders
72
73 # Create a recorder to monitor nodal displacements
74 # recorder Node -file node32.out -time -node 3 4 -dof 1 2 3 disp
75
76 # Create a recorder to monitor element forces in columns

```

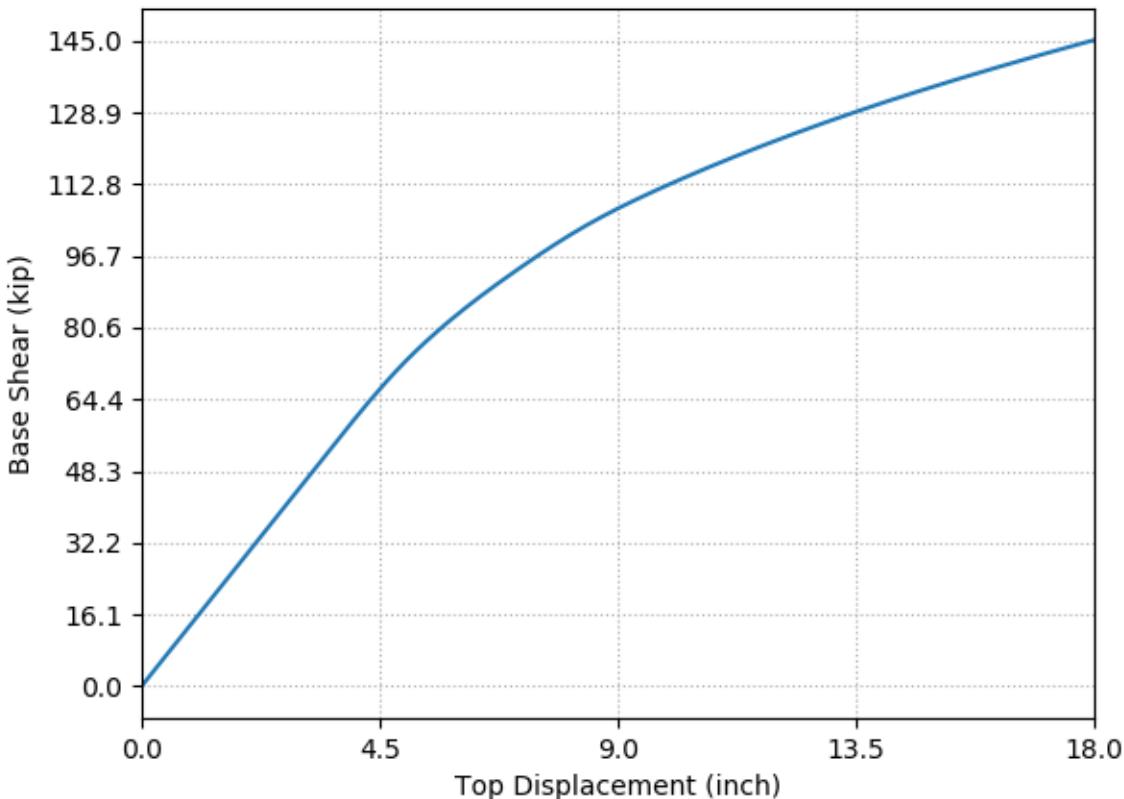
(continues on next page)

(continued from previous page)

```
77 # recorder EnvelopeElement -file ele32.out -time -ele 1 2 forces
78
79 # -----
80 # End of recorder generation
81 # -----
82
83
84 # -----
85 # Finally perform the analysis
86 # -----
87
88 # Set some parameters
89 maxU = 15.0 # Max displacement
90 currentDisp = 0.0
91 ok = 0
92
93 test('NormDispIncr', 1.0e-12, 1000)
94 algorithm('ModifiedNewton', '-initial')
95
96 while ok == 0 and currentDisp < maxU:
97
98     ok = analyze(1)
99
100    # if the analysis fails try initial tangent iteration
101    if ok != 0:
102        print("modified newton failed")
103        break
104    # print "regular newton failed .. lets try an initail stiffness for this step"
105    # test('NormDispIncr', 1.0e-12, 1000)
106    # # algorithm('ModifiedNewton', '-initial')
107    # ok = analyze(1)
108    # if ok == 0:
109    #     print "that worked .. back to regular newton"
110
111    # test('NormDispIncr', 1.0e-12, 10)
112    # algorithm('Newton')
113
114    currentDisp = nodeDisp(3, 1)
115
116 results = open('results.out', 'a+')
117
118 if ok == 0:
119     results.write('PASSED : RCFramePushover.py\n')
120     print("Passed!")
121 else:
122     results.write('FAILED : RCFramePushover.py\n')
123     print("Failed!")
124
125 results.close()
126
127 # Print the state at node 3
128 # print node 3
129
130
131 print("=====")
```

### Three story steel building with rigid beam-column connections and W-section

1. The source code is developed by [Anurag Upadhyay](#) from University of Utah.
2. The source code is shown below, which can be downloaded [here](#).
3. Run the source code in your favorite Python program and should see following plot.



```

1 #####
2 ## 2D steel frame example.
3 ## 3 story steel building with rigid beam-column connections.
4 ## This script uses W-section command inOpensees to create steel..
5 ## .. beam-column fiber sections.
6 ##
7 ## By - Anurag Upadhyay, PhD Student, University of Utah.
8 ## Date - 08/06/2018
9 #####
10 #####
11 print("====")
12 print("Start 2D Steel Frame Example")
13
14 from openseespy.opensees import *
15
16 import numpy as np
17 import matplotlib.pyplot as plt

```

(continues on next page)

(continued from previous page)

```
19 import os
20
21 AnalysisType='Pushover' ; # Pushover Gravity
22
23 ## -----
24 ## Start of model generation
25 ## -----
26 # remove existing model
27 wipe()
28
29 # set modelbuilder
30 model('basic', '-ndm', 2, '-ndf', 3)
31
32 import math
33
34 ##### Units and Constants #####
35 ##### Units and Constants #####
36 ##### Units and Constants #####
37
38 inch = 1;
39 kip = 1;
40 sec = 1;
41
42 # Dependent units
43 sq_in = inch*inch;
44 ksi = kip/sq_in;
45 ft = 12*inch;
46
47 # Constants
48 g = 386.2*inch/(sec*sec);
49 pi = math.acos(-1);
50
51 ##### Dimensions #####
52 ##### Dimensions #####
53 ##### Dimensions #####
54
55 # Dimensions Input
56 H_story=10.0*ft;
57 W_bayX=16.0*ft;
58 W_bayY_ab=5.0*ft+10.0*inch;
59 W_bayY_bc=8.0*ft+4.0*inch;
60 W_bayY_cd=5.0*ft+10.0*inch;
61
62 # Calculated dimensions
63 W_structure=W_bayY_ab+W_bayY_bc+W_bayY_cd;
64
65 #####
66 ### Material
67 #####
68
69 # Steel02 Material
70
71 matTag=1;
72 matConnAx=2;
73 matConnRot=3;
74
75 Fy=60.0*ksi; # Yield stress
```

(continues on next page)

(continued from previous page)

```

76 Es=29000.0*ksi;                      # Modulus of Elasticity of Steel
77 v=0.2;                                # Poisson's ratio
78 Gs=Es/(1+v);                         # Shear modulus
79 b=0.10;                               # Strain hardening ratio
80 params=[18.0, 0.925, 0.15]           # R0, cR1, cR2
81 R0=18.0
82 cR1=0.925
83 cR2=0.15
84 a1=0.05
85 a2=1.00
86 a3=0.05
87 a4=1.0
88 sigInit=0.0
89 alpha=0.05

90
91 uniaxialMaterial('Steel02', matTag, Fy, Es, b, R0, cR1, cR2, a1, a2, a3, a4, sigInit)
92
93 ######
94 # ## Sections
95 #####
96
97 colSecTag1=1;
98 colSecTag2=2;
99 beamSecTag1=3;
100 beamSecTag2=4;
101 beamSecTag3=5;

102
103 # COMMAND: section('WFSection2d', sectTag, matTag, d, tw, bf, tf, Nfw, Nff)
104
105 section('WFSection2d', colSecTag1, matTag, 10.5*inch, 0.26*inch, 5.77*inch, 0.44*inch,
106   ↪ 15, 16)                         # outer Column
107 section('WFSection2d', colSecTag2, matTag, 10.5*inch, 0.26*inch, 5.77*inch, 0.44*inch,
108   ↪ 15, 16)                         # Inner Column
109
110 section('WFSection2d', beamSecTag1, matTag, 8.3*inch, 0.44*inch, 8.11*inch, 0.
111   ↪ 685*inch, 15, 15)                 # outer Beam
112 section('WFSection2d', beamSecTag2, matTag, 8.2*inch, 0.40*inch, 8.01*inch, 0.
113   ↪ 650*inch, 15, 15)                 # Inner Beam
114 section('WFSection2d', beamSecTag3, matTag, 8.0*inch, 0.40*inch, 7.89*inch, 0.
115   ↪ 600*inch, 15, 15)                 # Inner Beam

116
117 # Beam size - W10x26
118 Abeam=7.61*inch*inch;
119 IbeamY=144.* (inch**4);             # Inertia along horizontal axis
120 IbeamZ=14.1* (inch**4);             # inertia along vertical axis
121
122
123 # BRB input data
124 Acore=2.25*inch;
125 Aend=10.0*inch;
126 LR_BRB=0.55;

127 #####
128 # ##### Nodes
129 #####
130
131 # Create All main nodes
132 node(1, 0.0, 0.0)

```

(continues on next page)

(continued from previous page)

```
128 node(2, W_bayX, 0.0)
129 node(3, 2*W_bayX, 0.0)
130
131 node(11, 0.0, H_story)
132 node(12, W_bayX, H_story)
133 node(13, 2*W_bayX, H_story)
134
135 node(21, 0.0, 2*H_story)
136 node(22, W_bayX, 2*H_story)
137 node(23, 2*W_bayX, 2*H_story)
138
139 node(31, 0.0, 3*H_story)
140 node(32, W_bayX, 3*H_story)
141 node(33, 2*W_bayX, 3*H_story)
142
143 # Beam Connection nodes
144
145 node(1101, 0.0, H_story)
146 node(1201, W_bayX, H_story)
147 node(1202, W_bayX, H_story)
148 node(1301, 2*W_bayX, H_story)
149
150 node(2101, 0.0, 2*H_story)
151 node(2201, W_bayX, 2*H_story)
152 node(2202, W_bayX, 2*H_story)
153 node(2301, 2*W_bayX, 2*H_story)
154
155 node(3101, 0.0, 3*H_story)
156 node(3201, W_bayX, 3*H_story)
157 node(3202, W_bayX, 3*H_story)
158 node(3301, 2*W_bayX, 3*H_story)
159
160 # #####
161 # Constraints
162 # #####
163
164 fix(1, 1, 1, 1)
165 fix(2, 1, 1, 1)
166 fix(3, 1, 1, 1)
167
168 # #####
169 # ### Elements
170 # #####
171
172 # ### Assign beam-integration tags
173
174 ColIntTag1=1;
175 ColIntTag2=2;
176 BeamIntTag1=3;
177 BeamIntTag2=4;
178 BeamIntTag3=5;
179
180 beamIntegration('Lobatto', ColIntTag1, colSecTag1, 4)
181 beamIntegration('Lobatto', ColIntTag2, colSecTag2, 4)
182 beamIntegration('Lobatto', BeamIntTag1, beamSecTag1, 4)
183 beamIntegration('Lobatto', BeamIntTag2, beamSecTag2, 4)
184 beamIntegration('Lobatto', BeamIntTag3, beamSecTag3, 4)
```

(continues on next page)

(continued from previous page)

```

185
186 # Assign geometric transformation
187
188 ColTransfTag=1
189 BeamTranfTag=2
190
191 geomTransf('PDelta', ColTransfTag)
192 geomTransf('Linear', BeamTranfTag)
193
194
195 # Assign Elements #####
196
197 # ## Add non-linear column elements
198 element('forceBeamColumn', 1, 1, 11, ColTransfTag, ColIntTag1, '-mass', 0.0)
199 element('forceBeamColumn', 2, 2, 12, ColTransfTag, ColIntTag2, '-mass', 0.0)
200 element('forceBeamColumn', 3, 3, 13, ColTransfTag, ColIntTag1, '-mass', 0.0)
201
202 element('forceBeamColumn', 11, 11, 21, ColTransfTag, ColIntTag1, '-mass', 0.0)
203 element('forceBeamColumn', 12, 12, 22, ColTransfTag, ColIntTag2, '-mass', 0.0)
204 element('forceBeamColumn', 13, 13, 23, ColTransfTag, ColIntTag1, '-mass', 0.0)
205
206 element('forceBeamColumn', 21, 21, 31, ColTransfTag, ColIntTag1, '-mass', 0.0)
207 element('forceBeamColumn', 22, 22, 32, ColTransfTag, ColIntTag2, '-mass', 0.0)
208 element('forceBeamColumn', 23, 23, 33, ColTransfTag, ColIntTag1, '-mass', 0.0)
209
210 #
211
212 # ### Add linear main beam elements, along x-axis
213 #element('elasticBeamColumn', 101, 1101, 1201, Abeam, Es, Gs, Jbeam, IbeamY, IbeamZ, ↴beamTransfTag, '-mass', 0.0)
214
215 element('forceBeamColumn', 101, 1101, 1201, BeamTranfTag, BeamIntTag1, '-mass', 0.0)
216 element('forceBeamColumn', 102, 1202, 1301, BeamTranfTag, BeamIntTag1, '-mass', 0.0)
217
218 element('forceBeamColumn', 201, 2101, 2201, BeamTranfTag, BeamIntTag2, '-mass', 0.0)
219 element('forceBeamColumn', 202, 2202, 2301, BeamTranfTag, BeamIntTag2, '-mass', 0.0)
220
221 element('forceBeamColumn', 301, 3101, 3201, BeamTranfTag, BeamIntTag3, '-mass', 0.0)
222 element('forceBeamColumn', 302, 3202, 3301, BeamTranfTag, BeamIntTag3, '-mass', 0.0)
223
224 # Assign constraints between beam end nodes and column nodes (Rigid beam column, ↴connections)
225 equalDOF(11, 1101, 1,2,3)
226 equalDOF(12, 1201, 1,2,3)
227 equalDOF(12, 1202, 1,2,3)
228 equalDOF(13, 1301, 1,2,3)
229
230 equalDOF(21, 2101, 1,2,3)
231 equalDOF(22, 2201, 1,2,3)
232 equalDOF(22, 2202, 1,2,3)
233 equalDOF(23, 2301, 1,2,3)
234
235 equalDOF(31, 3101, 1,2,3)
236 equalDOF(32, 3201, 1,2,3)
237 equalDOF(32, 3202, 1,2,3)
238 equalDOF(33, 3301, 1,2,3)
239

```

(continues on next page)

(continued from previous page)

```
240
241 ######
242 ## Gravity Load
243 ######
244 # create TimeSeries
245 timeSeries("Linear", 1)
246
247 # create a plain load pattern
248 pattern("Plain", 1, 1)
249
250 # Create the nodal load
251 load(11, 0.0, -5.0*kip, 0.0)
252 load(12, 0.0, -6.0*kip, 0.0)
253 load(13, 0.0, -5.0*kip, 0.0)
254
255 load(21, 0., -5.*kip, 0.0)
256 load(22, 0., -6.*kip, 0.0)
257 load(23, 0., -5.*kip, 0.0)
258
259 load(31, 0., -5.*kip, 0.0)
260 load(32, 0., -6.*kip, 0.0)
261 load(33, 0., -5.*kip, 0.0)
262
263
264 # -----
265 # Start of analysis generation
266 # -----
267
268 NstepsGrav = 10
269
270 system("BandGEN")
271 numberer("Plain")
272 constraints("Plain")
273 integrator("LoadControl", 1.0/NstepsGrav)
274 algorithm("Newton")
275 test('NormUnbalance', 1e-8, 10)
276 analysis("Static")
277
278
279 # perform the analysis
280 data = np.zeros((NstepsGrav+1,2))
281 for j in range(NstepsGrav):
282     analyze(1)
283     data[j+1,0] = nodeDisp(31,2)
284     data[j+1,1] = getLoadFactor(1)*5
285
286 loadConst('-time', 0.0)
287
288 print("Gravity analysis complete")
289
290 wipeAnalysis()
291
292 ######
293 ### PUSHOVER ANALYSIS
294 ######
295
296 if(AnalysisType=="Pushover"):
```

(continues on next page)

(continued from previous page)

```

297
298     print("<<< Running Pushover Analysis >>>")
299
300     # Create load pattern for pushover analysis
301     # create a plain load pattern
302     pattern("Plain", 2, 1)
303
304     load(11, 1.61, 0.0, 0.0)
305     load(21, 3.22, 0.0, 0.0)
306     load(31, 4.83, 0.0, 0.0)
307
308     ControlNode=31
309     ControlDOF=1
310     MaxDisp=0.15*H_story
311     DispIncr=0.1
312     NstepsPush=int (MaxDisp/DispIncr)
313
314     system("ProfileSPD")
315     numberer("Plain")
316     constraints("Plain")
317     integrator("DisplacementControl", ControlNode, ControlDOF, DispIncr)
318     algorithm("Newton")
319     test('NormUnbalance', 1e-8, 10)
320     analysis("Static")
321
322     PushDataDir = r'PushoverOut'
323     if not os.path.exists(PushDataDir):
324         os.makedirs(PushDataDir)
325     recorder('Node', '-file', "PushoverOut/Node2React.out", '-closeOnWrite', '-
326     ↪node', 2, '-dof', 1, 'reaction')
327     recorder('Node', '-file', "PushoverOut/Node31Disp.out", '-closeOnWrite', '-
328     ↪node', 31, '-dof', 1, 'disp')
329     recorder('Element', '-file', "PushoverOut/BeamStress.out", '-closeOnWrite', '-
330     ↪ele', 102, 'section', '4', 'fiber', '1', 'stressStrain')
331
332     # analyze(NstepsPush)
333
334     # Perform pushover analysis
335     dataPush = np.zeros((NstepsPush+1, 5))
336     for j in range(NstepsPush):
337         analyze(1)
338         dataPush[j+1, 0] = nodeDisp(31, 1)
339         reactions()
340         dataPush[j+1, 1] = nodeReaction(1, 1) + nodeReaction(2, 1) +_
341         ↪nodeReaction(3, 1)
342
343         plt.plot(dataPush[:, 0], -dataPush[:, 1])
344         plt.xlim(0, MaxDisp)
345         plt.xticks(np.linspace(0, MaxDisp, 5, endpoint=True))
346         plt.yticks(np.linspace(0, -int(dataPush[NstepsPush, 1]), 10, endpoint=True))
347         plt.grid(linestyle='dotted')
348         plt.xlabel('Top Displacement (inch)')
349         plt.ylabel('Base Shear (kip)')
350         plt.show()
351
352
353     print("Pushover analysis complete")

```

(continues on next page)

(continued from previous page)

350  
351  
352  
353

## 1.12.2 Earthquake Examples

1. *Cantilever 2D EQ ground motion with gravity Analysis*
2. *Reinforced Concrete Frame Earthquake Analysis*
3. *Example name spaced nonlinear SDOF*
4. *RotD Spectra of Ground Motion*
5. *Portal 2D Frame - Dynamic EQ Ground Motion*
6. *2D Column - Dynamic EQ Ground Motion*
7. *Nonlinear Canti Col Uniaxial Inelastic Section- Dyn EQ GM*
8. *Nonlin Canti Col Inelstc Uniaxial Mat in Fiber Sec - Dyn EQ*
9. *Cantilever 2D Column with Units- Dynamic EQ Ground Motion*
10. *Cantilever 2D Column with Units-Static Pushover*
11. *2D Portal Frame with Units- Dynamic EQ Ground Motion*
12. *2D Portal Frame with Units- Multiple Support Dynamic EQ Ground Motion-acctimeseries*
13. *2D Portal Frame with Units- Multiple Support Dynamic EQ Ground Motion-disptimeseries*
14. *2D Portal Frame with Units- Uniform Dynamic EQ -bidirectional-acctimeseries*
15. ExampleFRP

### Cantilever 2D EQ ground motion with gravity Analysis

1. The source code is shown below, which can be downloaded [here](#).
2. The ground motion data file [here](#) must be put in the same folder.
3. Run the source code in your favorite Python program and should see results below

```
=====
Start cantilever 2D EQ ground motion with gravity example
u2 = -0.07441860465116278
Passed!
=====
```

```
1 print("=====")
2 print("Start cantilever 2D EQ ground motion with gravity example")
3
4 from openseespy.opensees import *
5
6
7 # -----
```

(continues on next page)

(continued from previous page)

```

8 # Example 1. cantilever 2D
9 # EQ ground motion with gravity
10 # all units are in kip, inch, second
11 # elasticBeamColumn ELEMENT
12 # Silvia Mazzoni & Frank McKenna, 2006
13 #
14 # ^Y
15 # /
16 # 2      —
17 # /      /
18 # /      /
19 # /      /
20 # (1)    36'
21 # /      /
22 # /      /
23 # /      /
24 # =1=    ---- ----->X
25 #
26
27 # SET UP -----
28 wipe()                                     # clear opensees model
29 model('basic', '-ndm', 2, '-ndf', 3)        # 2 dimensions, 3 dof per node
30 # file mkdir data                          # create data directory
31
32 # define GEOMETRY -----
33 # nodal coordinates:
34 node(1, 0., 0.)                            # node#, X Y
35 node(2, 0., 432.)                          # node#
36
37 # Single point constraints -- Boundary Conditions
38 fix(1, 1, 1, 1)                           # node DX DY RZ
39
40 # nodal masses:
41 mass(2, 5.18, 0., 0.)                     # node#, Mx My Mz, Mass=Weight/g.
42
43 # Define ELEMENTS -----
44 # define geometric transformation: performs a linear geometric transformation of beam
45 # stiffness and resisting force from the basic system to the global-coordinate system
46 geomTransf('Linear', 1)                      # associate a tag to transformation
47
48 # connectivity:
49 element('elasticBeamColumn', 1, 1, 2, 3600.0, 3225.0, 1080000.0, 1)
50
51 # define GRAVITY -----
52 timeSeries('Linear', 1)
53 pattern('Plain', 1, 1, )
54 load(2, 0., -2000., 0.)                    # node#, FX FY MZ --
55 # superstructure-weight
56
57 constraints('Plain')                      # how it handles boundary
58 # conditions
59 numberer('Plain')                         # renumber dof's to minimize band-width
# (optimization), if you want to
60 system('BandGeneral')                     # how to store and solve the system of
# equations in the analysis
61 algorithm('Linear')                       # use Linear algorithm for linear analysis
62 integrator('LoadControl', 0.1)            # determine the next time step
# for an analysis, # apply gravity in 10 steps

```

(continues on next page)

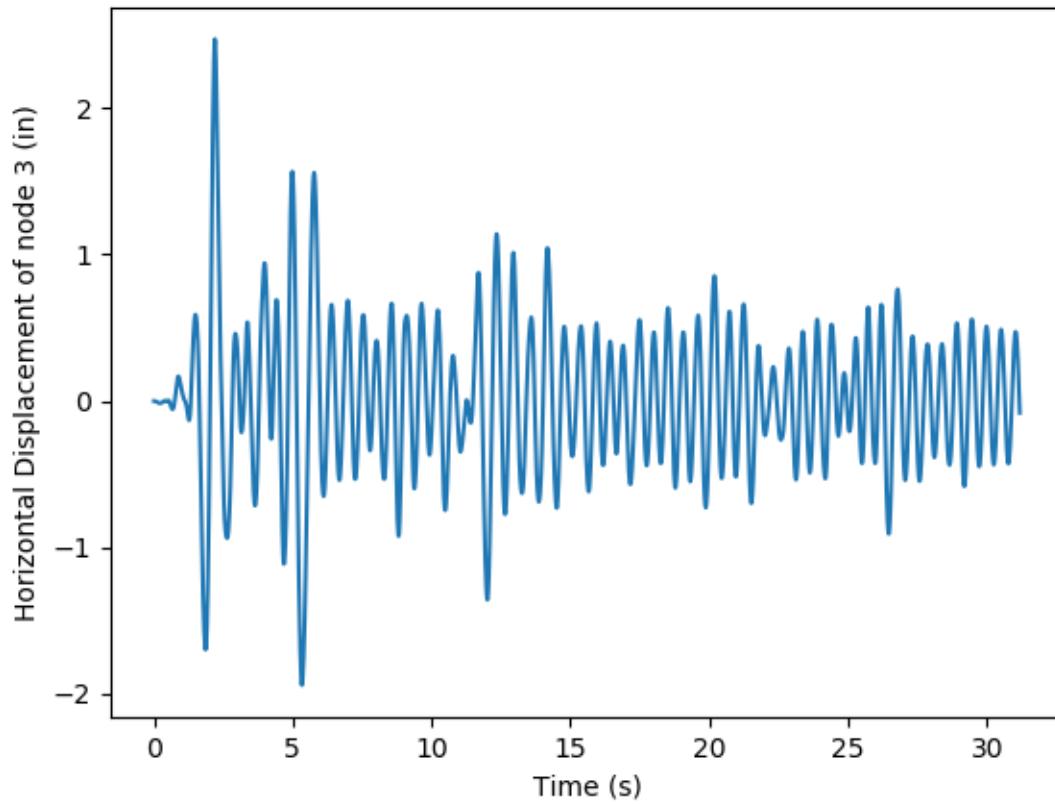
(continued from previous page)

```
60 analysis('Static')                                     # define type of_
  ↵analysis static or transient
61 analyze(10)                                         # perform gravity analysis
62 loadConst('-time', 0.0)                            # hold gravity constant and_
  ↵restart time
63
64 # DYNAMIC ground-motion analysis -----
  ↵
65 # create load pattern
66 G = 386.0
67 timeSeries('Path', 2, '-dt', 0.005, '-filePath', 'A10000.dat', '-factor', G) # define_
  ↵acceleration vector from file (dt=0.005 is associated with the input file gm)
68 pattern('UniformExcitation', 2, 1, '-accel', 2)                         # define_
  ↵where and how (pattern tag, dof) acceleration is applied
69
70 # set damping based on first eigen mode
71 freq = eigen('-fullGenLapack', 1)**0.5
72 dampRatio = 0.02
73 rayleigh(0., 0., 0., 2*dampRatio/freq)
74
75 # create the analysis
76 wipeAnalysis()                                       # clear previously-define analysis_
  ↵parameters
77 constraints('Plain')                                # how it handles boundary conditions
78 numberer('Plain')        # renumber dof's to minimize band-width (optimization), if you_
  ↵want to
79 system('BandGeneral')    # how to store and solve the system of equations in the analysis
80 algorithm('Linear')          # use Linear algorithm for linear analysis
81 integrator('Newmark', 0.5, 0.25)      # determine the next time step for an analysis
82 analysis('Transient')      # define type of analysis: time-dependent
83 analyze(3995, 0.01)           # apply 3995 0.01-sec time steps in analysis
84
85 u2 = nodeDisp(2, 2)
86 print("u2 = ", u2)
87
88
89 if abs(u2+0.07441860465116277579) < 1e-12:
90     print("Passed!")
91 else:
92     print("Failed!")
93
94 wipe()
95
96 print("=====")
```

## Reinforced Concrete Frame Earthquake Analysis

1. The source code is shown below, which can be downloaded [here](#).
2. The file for gravity analysis is also needed [here](#).
3. The ReadRecord is a useful python function for parsing the PEER strong motion data base files and returning the dt, nPts and creating a file containing just data points. The function is kept in a seperate file [here](#) and is imported in the example.
4. The ground motion data file [here](#) must be put in the same folder.

5. Run the source code in your favorite Python program and should see Passed! in the results and a plotting of displacement for node 3



```

1 print("====")
2 print("Start RCFrameEarthquake Example")
3
4 # Units: kips, in, sec
5 #
6 # Written: Minjie
7
8 from openseespy.opensees import *
9
10 import ReadRecord
11 import numpy as np
12 import matplotlib.pyplot as plt
13
14 wipe()
15 # -----
16 # Start of Model Generation & Initial Gravity Analysis
17 # -----
18
19 # Do operations of Example3.1 by sourcing in the tcl file
20 import RCFrameGravity
21 print("Gravity Analysis Completed")
22
23 # Set the gravity loads to be constant & reset the time in the domain

```

(continues on next page)

(continued from previous page)

```
24 loadConst('-time', 0.0)
25
26 # -----
27 # End of Model Generation & Initial Gravity Analysis
28 # -----
29
30 # Define nodal mass in terms of axial load on columns
31 g = 386.4
32 m = RCFrameGravity.P/g
33
34 mass(3, m, m, 0.0)
35 mass(4, m, m, 0.0)
36
37 # Set some parameters
38 record = 'elCentro'
39
40 # Perform the conversion from SMD record to OpenSees record
41 dt, nPts = ReadRecord.ReadRecord(record+'.at2', record+'.dat')
42
43 # Set time series to be passed to uniform excitation
44 timeSeries('Path', 2, '-filePath', record+'.dat', '-dt', dt, '-factor', g)
45
46 # Create UniformExcitation load pattern
47 #           tag dir
48 pattern('UniformExcitation', 2, 1, '-accel', 2)
49
50 # set the rayleigh damping factors for nodes & elements
51 rayleigh(0.0, 0.0, 0.0, 0.000625)
52
53 # Delete the old analysis and all it's component objects
54 wipeAnalysis()
55
56 # Create the system of equation, a banded general storage scheme
57 system('BandGeneral')
58
59 # Create the constraint handler, a plain handler as homogeneous boundary
60 constraints('Plain')
61
62 # Create the convergence test, the norm of the residual with a tolerance of
63 # 1e-12 and a max number of iterations of 10
64 test('NormDispIncr', 1.0e-12, 10 )
65
66 # Create the solution algorithm, a Newton-Raphson algorithm
67 algorithm('Newton')
68
69 # Create the DOF numberer, the reverse Cuthill-McKee algorithm
70 numberer('RCM')
71
72 # Create the integration scheme, the Newmark with alpha =0.5 and beta =.25
73 integrator('Newmark', 0.5, 0.25 )
74
75 # Create the analysis object
76 analysis('Transient')
77
78 # Perform an eigenvalue analysis
79 numEigen = 2
80 eigenValues = eigen(numEigen)
```

(continues on next page)

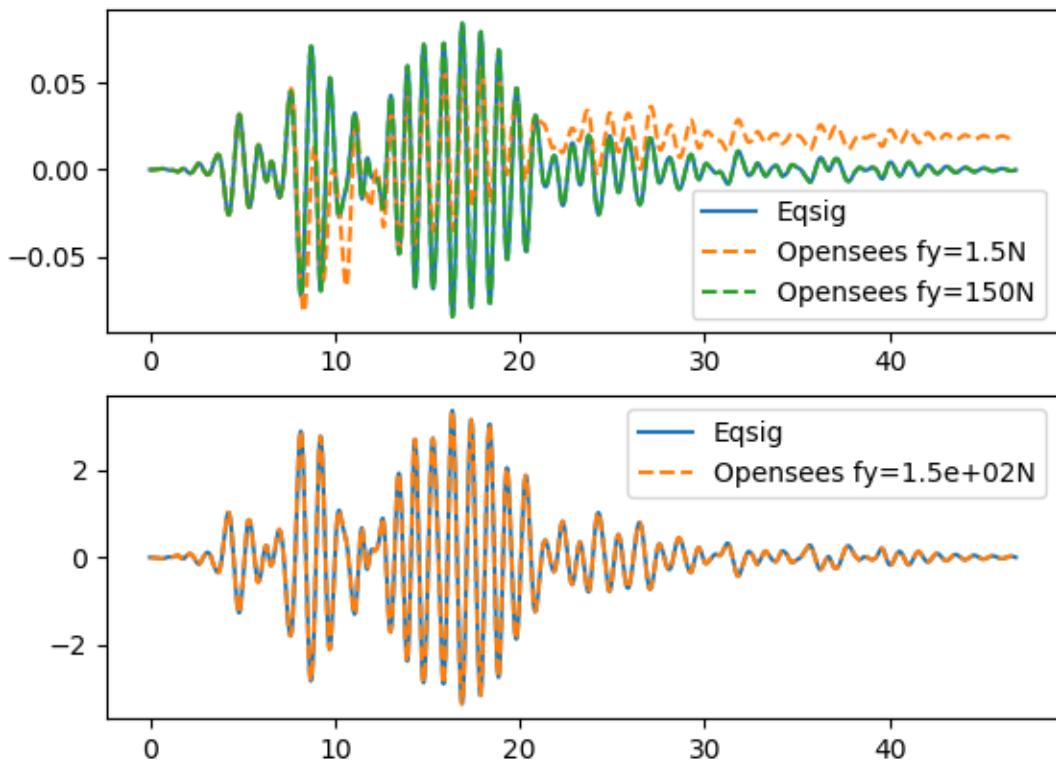
(continued from previous page)

```

81 print("eigen values at start of transient:", eigenValues)
82
83 # set some variables
84 tFinal = nPts*dt
85 tCurrent = getTime()
86 ok = 0
87
88 time = [tCurrent]
89 u3 = [0.0]
90
91 # Perform the transient analysis
92 while ok == 0 and tCurrent < tFinal:
93
94     ok = analyze(1, .01)
95
96     # if the analysis fails try initial tangent iteration
97     if ok != 0:
98         print("regular newton failed .. lets try an initail stiffness for this step")
99         test('NormDispIncr', 1.0e-12, 100, 0)
100        algorithm('ModifiedNewton', '-initial')
101        ok =analyze( 1, .01)
102        if ok == 0:
103            print("that worked .. back to regular newton")
104            test('NormDispIncr', 1.0e-12, 10 )
105            algorithm('Newton')
106
107    tCurrent = getTime()
108
109    time.append(tCurrent)
110    u3.append(nodeDisp(3,1))
111
112
113
114 # Perform an eigenvalue analysis
115 eigenValues = eigen(numEigen)
116 print("eigen values at end of transient:", eigenValues)
117
118 results = open('results.out', 'a+')
119
120 if ok == 0:
121     results.write('PASSED : RCFrameEarthquake.py\n');
122     print("Passed!")
123 else:
124     results.write('FAILED : RCFrameEarthquake.py\n');
125     print("Failed!")
126
127 results.close()
128
129 plt.plot(time, u3)
130 plt.ylabel('Horizontal Displacement of node 3 (in)')
131 plt.xlabel('Time (s)')
132
133 plt.show()
134
135
136
137 print ("=====")
```

### Example name spaced nonlinear SDOF

1. The source code is developed by [Maxim Millen](#) from University of Porto.
2. The source code is shown below, which can be downloaded [here](#).
3. Also download the constants file [here](#), and the ground motion file
4. Make sure the `numpy`, `matplotlib` and `eqsig` packages are installed in your Python distribution.
5. Run the source code in your favorite Python program and should see



```
1 import eqsig
2 from eqsig import duhamels
3 import matplotlib.pyplot as plt
4 import numpy as np
5
6 import openseespy.opensees as op
7 import opensees_constants as opc #opensees_constants.py should be close to main file,
8 #or use sys.path... to its directory
9
10 def get_inelastic_response(mass, k_spring, f_yield, motion, dt, xi=0.05, r_post=0.0):
11     """
12         Run seismic analysis of a nonlinear SDOF
13
14         :param mass: SDOF mass
```

(continues on next page)

(continued from previous page)

```

15     :param k_spring: spring stiffness
16     :param f_yield: yield strength
17     :param motion: list, acceleration values
18     :param dt: float, time step of acceleration values
19     :param xi: damping ratio
20     :param r_post: post-yield stiffness
21     :return:
22     """
23
24     op.wipe()
25     op.model('basic', '-ndm', 2, '-ndf', 3) # 2 dimensions, 3 dof per node
26
27     # Establish nodes
28     bot_node = 1
29     top_node = 2
30     op.node(bot_node, 0., 0.)
31     op.node(top_node, 0., 0.)
32
33     # Fix bottom node
34     op.fix(bot_node, opc.FREE, opc.FIXED, opc.FIXED)
35     op.fix(bot_node, opc.FIXED, opc.FIXED, opc.FIXED)
36     # Set out-of-plane DOFs to be slaved
37     op.equalDOF(1, 2, *[2, 3])
38
39     # nodal mass (weight / g):
40     op.mass(top_node, mass, 0., 0.)
41
42     # Define material
43     bilinear_mat_tag = 1
44     mat_type = "Steel01"
45     mat_props = [f_yield, k_spring, r_post]
46     op.uniaxialMaterial(mat_type, bilinear_mat_tag, *mat_props)
47
48     # Assign zero length element
49     beam_tag = 1
50     op.element('zeroLength', beam_tag, bot_node, top_node, "-mat", bilinear_mat_tag,
51     ↪"-dir", 1, '-doRayleigh', 1)
52
53     # Define the dynamic analysis
54     load_tag_dynamic = 1
55     pattern_tag_dynamic = 1
56
57     values = list(-1 * motion) # should be negative
58     op.timeSeries('Path', load_tag_dynamic, '-dt', dt, '-values', *values)
59     op.pattern('UniformExcitation', pattern_tag_dynamic, opc.X, '-accel', load_tag_
60     ↪dynamic)
61
62     # set damping based on first eigen mode
63     angular_freq = op.eigen('-fullGenLapack', 1) ** 0.5
64     alpha_m = 0.0
65     beta_k = 2 * xi / angular_freq
66     beta_k_comm = 0.0
67     beta_k_init = 0.0
68
69     op.rayleigh(alpha_m, beta_k, beta_k_init, beta_k_comm)

```

(continues on next page)

(continued from previous page)

```
70     op.wipeAnalysis()
71
72     op.algorithm('Newton')
73     op.system('SparseGeneral')
74     op.numberer('RCM')
75     op.constraints('Transformation')
76     op.integrator('Newmark', 0.5, 0.25)
77     op.analysis('Transient')
78
79
80     tol = 1.0e-10
81     iterations = 10
82     op.test('EnergyIncr', tol, iterations, 0, 2)
83     analysis_time = (len(values) - 1) * dt
84     analysis_dt = 0.001
85     outputs = {
86         "time": [],
87         "rel_disp": [],
88         "rel_accel": [],
89         "rel_vel": [],
90         "force": []
91     }
92
93     while op.getTime() < analysis_time:
94         curr_time = op.getTime()
95         op.analyze(1, analysis_dt)
96         outputs["time"].append(curr_time)
97         outputs["rel_disp"].append(op.nodeDisp(top_node, 1))
98         outputs["rel_vel"].append(op.nodeVel(top_node, 1))
99         outputs["rel_accel"].append(op.nodeAccel(top_node, 1))
100        op.reactions()
101        outputs["force"].append(-op.nodeReaction(bot_node, 1)) # Negative since diff_
102    ↵node
103    op.wipe()
104    for item in outputs:
105        outputs[item] = np.array(outputs[item])
106
107    return outputs
108
109 def show_single_comparison():
110     """
111     Create a plot of an elastic analysis, nonlinear analysis and closed form elastic
112
113     :return:
114     """
115
116     record_filename = 'test_motion_dt0p01.txt'
117     motion_step = 0.01
118     rec = np.loadtxt(record_filename)
119     acc_signal = eqsig.AccSignal(rec, motion_step)
120     period = 1.0
121     xi = 0.05
122     mass = 1.0
123     f_yield = 1.5 # Reduce this to make it nonlinear
124     r_post = 0.0
125
```

(continues on next page)

(continued from previous page)

```

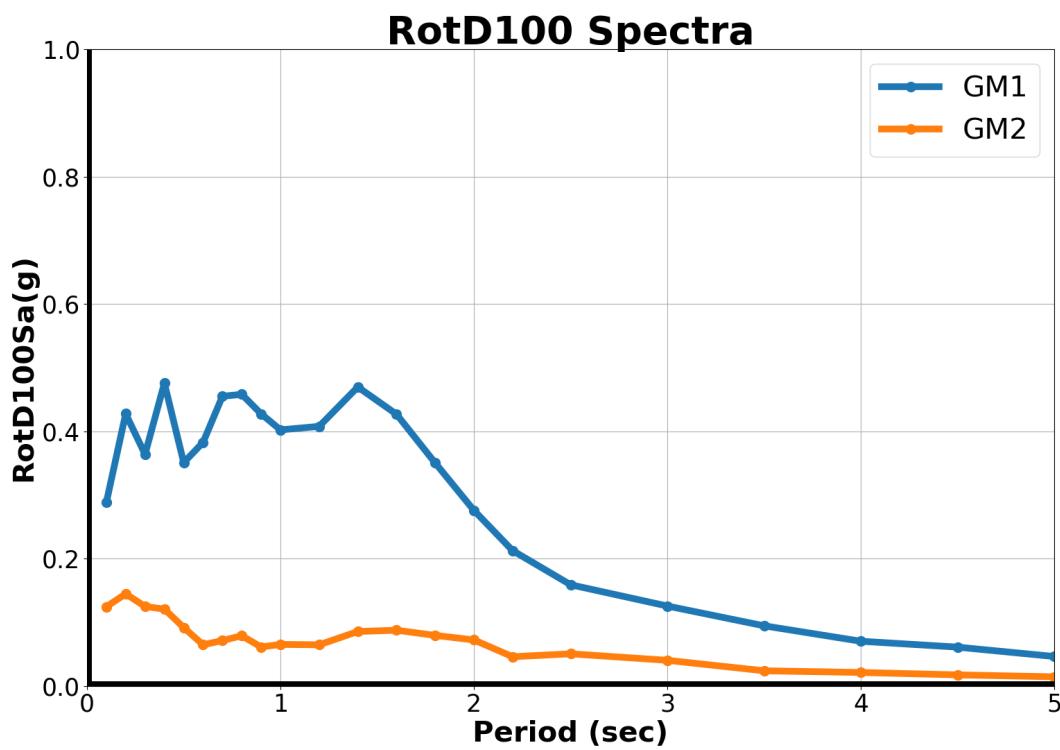
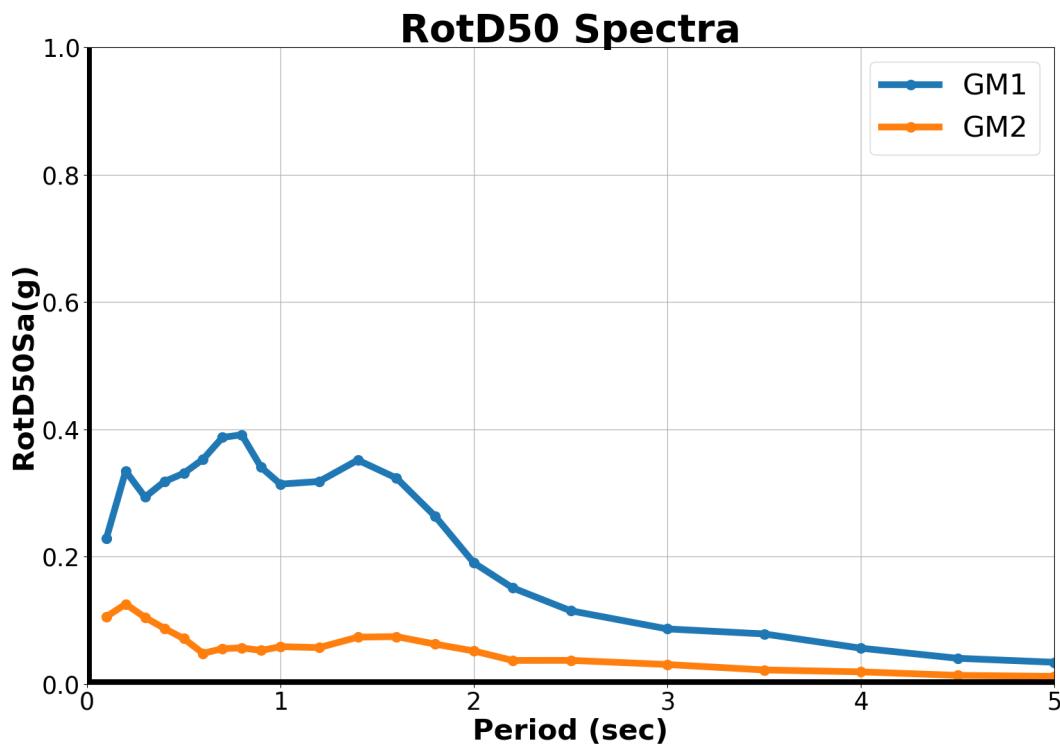
126     periods = np.array([period])
127     resp_u, resp_v, resp_a = duhamels.response_series(motion=rec, dt=motion_step, ↴
128     ↪periods=periods, xi=xi)
129
130     k_spring = 4 * np.pi ** 2 * mass / period ** 2
131     outputs = get_inelastic_response(mass, k_spring, f_yield, rec, motion_step, xi=xi, ↴
132     ↪r_post=r_post)
133     outputs_elastic = get_inelastic_response(mass, k_spring, f_yield * 100, rec, ↴
134     ↪motion_step, xi=xi, r_post=r_post)
135     ux_opensees = outputs["rel_disp"]
136     ux_opensees_elastic = outputs_elastic["rel_disp"]
137
138     bf, sps = plt.subplots(nrows=2)
139     sps[0].plot(acc_signal.time, resp_u[0], label="Eqsig")
140     sps[0].plot(outputs["time"], ux_opensees, label="Opensees fy=%.3gN" % f_yield, ls=↪
141     ↪"--")
142     sps[0].plot(outputs["time"], ux_opensees_elastic, label="Opensees fy=%.3gN" % (f_↪
143     ↪yield * 100), ls="--")
144     sps[1].plot(acc_signal.time, resp_a[0], label="Eqsig") # Elastic solution
145     time = acc_signal.time
146     acc_opensees_elastic = np.interp(time, outputs_elastic["time"], outputs_elastic[↪
147     ↪"rel_accel"])
148     print("diff", sum(acc_opensees_elastic - resp_a[0]))
149     sps[1].plot(time, acc_opensees_elastic, label="Opensees fy=%.2gN" % (f_yield * ↪
150     ↪100), ls="--")
151     sps[0].legend()
152     sps[1].legend()
153     plt.show()

if __name__ == '__main__':
    show_single_comparison()

```

## RotD Spectra of Ground Motion

1. The source code is developed by [Jawad Fayaz](#) from University of California- Irvine.
2. The source code is shown below, which can be downloaded [here](#).
3. Also download the code to read the provided GM file [here](#).
4. The example bi-directional ground motion time histories are given GM11, GM21, GM12, GM22.
5. Run the source code in any Python IDE (e.g Spyder, Jupyter Notebook) and should see



```

"""
author : JAWAD FAYAZ (email: jfayaz@uci.edu) (website: https://jfayaz.github.io)

----- Instructions -----
This code develops the RotD50 Sa and RotD100 Sa Spectra of the Bi-Directional
Ground Motion records as '.AT2' files provided in the current directory

The two directions of the ground motion record must be named as 'GM1i' and 'GM2i',
where 'i' is the ground motion number which goes from 1 to 'n', 'n' being the total
number of ground motions for which the Spectra needs to be generated. The extension
of the files must be '.AT2'

For example: If the Spectra of two ground motion records are required, 4 files with
the following names must be provided in the given 'GM' folder:
    'GM11.AT2' - Ground Motion 1 in direction 1 (direction 1 can be either one of the
    ↪bi-directional GM as we are rotating the ground motions it does not matter)
    'GM21.AT2' - Ground Motion 1 in direction 2 (direction 2 is the other direction
    ↪of the bi-directional GM)
    'GM12.AT2' - Ground Motion 2 in direction 1 (direction 1 can be either one of the
    ↪bi-directional GM as we are rotating the ground motions it does not matter)
    'GM22.AT2' - Ground Motion 2 in direction 2 (direction 2 is the other direction
    ↪of the bi-directional GM)

The Ground Motion file must be a vector file with 4 header lines. The first 3 lines
↪can have
any content, however, the 4th header line must be written exactly as per the
↪following example:
    'NPTS= 15864, DT= 0.0050'
The 'ReadGMFile.py' can be edited accordingly for any other format

You may run this code in python IDE: 'Spyder' or any other similar IDE

Make sure you have the following python libraries installed:
    os
    sys
    pathlib
    fnmatch
    shutil
    IPython
    pandas
    numpy
    matplotlib.pyplot

INPUT:
This codes provides the option to have 3 different regions of developing the Spectra
↪of ground motions with different period intervals (discretizations)
The following inputs within the code are required:
    'Path_to_openpyfiles'--> Path where the library files 'opensees.pyd' and 'LICENSE'.
    ↪rst' of OpenSeesPy are included (for further details go to https://openseespydoc.
    ↪readthedocs.io/en/latest/windows.html)
    'Int_T_Reg_1'           --> Period Interval for the first region of the Spectrum
    'End_T_Reg_1'           --> Last Period of the first region of the Spectrum (where
    ↪to end the first region)
    'Int_T_Reg_2'           --> Period Interval for the second region of the Spectrum
    'End_T_Reg_2'           --> Last Period of the second region of the Spectrum (where
    ↪to end the second region)
    'Int_T_Reg_3'           --> Period Interval for the third region of the Spectrum

```

(continues on next page)

(continued from previous page)

```

'End_T_Reg_3'           --> Last Period of the third region of the Spectrum (where
→to end the third region)
'Plot_Spectra'          --> whether to plot the generated Spectra of the ground
→motions (options: 'Yes', 'No')

OUTPUT:
The output will be provided in a saperate 'GMi_Spectra.txt' file for each ground
→motion record, where 'i' denotes the number of ground motion in the same of
provided 'GM1i.AT2' and 'GM2i.AT2' files. The output files will be generated in a
→saperate folder 'Spectra' which will be created in the current folder
The 'GMi_Spectra.txt' file will consist of space-saperated file with:
'Periods (secs)' 'RotD50 Sa (g)' 'RotD100 Sa (g)'

%%%%%
=====
→%%%%%
%%%%%
→%%%%%
→%%%%%
=====

"""
#####
===== INPUTS ===== #####
# Path where the library files 'opensees.pyd' and 'LICENSE.rst' are included (for
→further details go to https://openseespydoc.readthedocs.io/en/latest/windows.html)
Path_to_openpyfiles = 'C:\Tcl'

# For periods 0 to 'End_T_Reg_1' in an interval of 'Int_T_Reg_1'
Int_T_Reg_1      = 0.1
End_T_Reg_1     = 1

# For periods ['End_T_Reg_1'+'Int_T_Reg_2'] to 'End_T_Reg_2' in an interval of 'Int_T_
→Reg_2'
Int_T_Reg_2      = 0.2
End_T_Reg_2     = 2

# For periods ['End_T_Reg_2'+'Int_T_Reg_3'] to 'End_T_Reg_3' in an interval of 'Int_T_
→Reg_3'
Int_T_Reg_3      = 0.5
End_T_Reg_3     = 5

# Plot Spectra (options: 'Yes' or 'No')
Plot_Spectra     = 'Yes'

#####
===== CODE BEGINS ===== #####
## Importing Libraries
import os, sys, pathlib, fnmatch
import shutil as st
from IPython import get_ipython

from openseespy.opensees import *

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

```

(continues on next page)

(continued from previous page)

```

import warnings
import matplotlib.cbook
warnings.filterwarnings("ignore", category=matplotlib.cbook.mplDeprecation)
wipe()

# Getting Number of Ground Motions from the GM folder
GMdir = os.getcwd()
No_of_GMs = int(len(fnmatch.filter(os.listdir(GMdir), '*.AT2')))/2
print('\nGenerating Spectra for {} provided GMs \n\n'.format(np.round(No_of_GMs, 0)))

# Initializations
DISPLACEMENTS = pd.DataFrame(columns=['uX', 'uY'])
GM_SPECTRA = pd.DataFrame(columns=['Period(s)', 'RotD50Sa(g)', 'RotD100Sa(g)'])
SDOF_RESPONSE = []
GM_RESPONSE = []

# Spectra Generation
for iEQ in range(1, No_of_GMs+1):
    print('Generating Spectra for GM: {} ... \n'.format(np.round(iEQ, 0)))
    Periods = np.concatenate((list(np.arange(Int_T_Reg_1, End_T_Reg_1+Int_T_Reg_1, Int_T_Reg_1)), list(np.arange(End_T_Reg_1+Int_T_Reg_2, End_T_Reg_2+Int_T_Reg_2, Int_T_Reg_2)), list(np.arange(End_T_Reg_2+Int_T_Reg_3, End_T_Reg_3+Int_T_Reg_3, Int_T_Reg_3))), axis=0)
    ii = 0

    for T in Periods:
        ii = ii+1
        GMinter = 0

        # Storing Periods
        GM_SPECTRA.loc[ii-1, 'Period(s)'] = T

        # Setting modelbuilder
        model('basic', '-ndm', 3, '-ndf', 6)

        # Setting SODF Variables
        g = 386.1 # value of g
        L = 1.0 # Length
        d = 2 # Diameter
        r = d/2 # Radius
        A = np.pi*(r**2) # Area
        E = 1.0 # Elastic Modulus
        G = 1.0 # Shear Modulus
        I3 = np.pi*(r**4)/4 # Moment of Inertia (zz)
        J = np.pi*(r**4)/2 # Polar Moment of Inertia
        I2 = np.pi*(r**4)/4 # Moment of Inertia (yy)
        K = 3*E*I3/(L**3) # Stiffness
        M = K*(T**2)/4/(np.pi**2) # Mass
        omega = np.sqrt(K/M) # Natural Frequency
        Tn = 2*np.pi/omega # Natural Period

        # Creating nodes
        node(1, 0.0, 0.0, 0.0)
        node(2, 0.0, 0.0, L)

        # Transformation
        transfTag = 1

```

(continues on next page)

(continued from previous page)

```

geomTransf('Linear',transfTag,0.0,1.0,0.0)

# Setting boundary condition
fix(1, 1, 1, 1, 1, 1, 1)

# Defining materials
uniaxialMaterial("Elastic", 11, E)

# Defining elements
element("elasticBeamColumn",12,1,2,A,E,G,J,I2,I3,1)

# Defining mass
mass(2,M,M,0.0,0.0,0.0,0.0)

# Eigen Value Analysis (Verifying Period)
numEigen = 1
eigenValues = eigen(numEigen)
omega = np.sqrt(eigenValues)
T = 2*np.pi/omega
print('    Calculating Spectral Ordinate for Period = {} secs'.format(np.
˓→round(T,3)))

## Reading GM Files
exec(open("ReadGMFile.py").read()) # read in procedure

Multinitiation
iGMinput = 'GM1'+str(iEQ)+' GM2'+str(iEQ) ;
GMinput = iGMinput.split(' ');
gmXY = {}
for i in range(0,2):
    inFile = GMdir + '\\'+ GMinput[i]+'.AT2';
    dt, NumPts , gmXY = ReadGMfile()

# Storing GM Histories
gmX = gmXY[1]
gmY = gmXY[2]
gmXY_mat = np.column_stack((gmX,gmX, gmY, gmY))

# Bidirectional Uniform Earthquake ground motion (uniform acceleration input_
˓→at all support nodes)
iGMfile = 'GM1'+str(iEQ)+' GM2'+str(iEQ) ;
GMfile = iGMfile.split(' ')
GMdirection = [1,1,2,2];
GMfact = [np.cos(GMinter*np.pi/180),np.sin(-GMinter*np.pi/180),_
˓→np.sin(GMinter*np.pi/180), np.cos(GMinter*np.pi/180)];
IDTag = 2
loop = [1,2,3,4]

for i in loop:
    # Setting time series to be passed to uniform excitation
    timeSeries('Path',IDTag+i, '-dt', dt, '-values', *list(gmXY_mat[:,i-1]),
˓→'-factor', GMfact[i-1]*g)
        # Creating UniformExcitation load pattern
        pattern('UniformExcitation', IDTag+i, GMdirection[i-1], '-accel',
˓→IDTag+i)

# Defining Damping
# Applying Rayleigh Damping from $xDamp

```

(continues on next page)

(continued from previous page)

```

# D=$alphaM*M + $betaKcurr*Kcurrent + $betaKcomm*KlastCommit + $beatKinit*
$Kinitial
    xDamp          = 0.05;
# 5% damping ratio
alphaM          = 0.;                                # M-prop. damping;
D = alphaM*M
    betaKcurr     = 0.;                                # K-proportional damping;
#+beatKcurr*KCurrent
    betaKcomm      = 2.*xDamp/omega;                  # K-
prop. damping parameter; +betaKcomm*KlastCommitt
    betaKinit      = 0.;                                # initial-stiffness proportional
# damping +beatKinit*Kini
rayleigh(alphaM,betaKcurr,betaKinit,betaKcomm); # RAYLEIGH damping

# Creating the analysis
wipeAnalysis()                                     # clear previously-defined
analysis parameters
constraints("Penalty",1e18, 1e18)                  # how to handle boundary conditions
numberer("RCM")                                    # renumber dof's to minimize band-width
#(optimization), if you want to
system('SparseGeneral')                           # how to store and solve the system of
equations in the analysis
algorithm('Linear')                                # use Linear algorithm for linear
analysis
integrator("TRBDF2")                             # determine the next time step for an
analysis
algorithm("NewtonLineSearch")                     # define type of analysis: time-dependent
test('EnergyIncr',1.0e-6, 100, 0)
analysis("Transient")

# Variables (Can alter the speed of analysis)
dtAnalysis   = dt
TmaxAnanlysis = dt*NumPts
tFinal       = int(TmaxAnanlysis/dtAnalysis)
tCurrent     = getTime()
ok           = 0
time         = [tCurrent]

# Initializations of response
u1           = [0.0]
u2           = [0.0]

# Performing the transient analysis (Performance is slow in this loop, can be
altered by changing the parameters)
while ok == 0 and tCurrent < tFinal:
    ok = analyze(1, dtAnalysis)
    # if the analysis fails try initial tangent iteration
    if ok != 0:
        print("Iteration failed .. lets try an initial stiffness for this step
")
        test('NormDispIncr', 1.0e-12, 100, 0)
        algorithm('ModifiedNewton', '-initial')
        ok =analyze( 1, .001)

```

(continues on next page)

(continued from previous page)

```

if ok == 0:
    print("that worked .. back to regular newton")
    test('NormDispIncr', 1.0e-12, 10 )
    algorithm('Newton')

tCurrent = getTime()
time.append(tCurrent)
u1.append(nodeDisp(2,1))
u2.append(nodeDisp(2,2))

# Storing responses
DISPLACEMENTS.loc[ii-1,'uX'] = np.array(u1)
DISPLACEMENTS.loc[ii-1,'uY'] = np.array(u2)
DISP_X_Y = np.column_stack((np.array(u1),np.array(u2)))

# Rotating the Spectra (Projections)
Rot_Matrix = np.zeros((2,2))
Rot_Displ = np.zeros((180,1))
for theta in range (0,180,1):
    Rot_Matrix [0,0] = np.cos(np.deg2rad(theta))
    Rot_Matrix [0,1] = np.sin(np.deg2rad(-theta))
    Rot_Matrix [1,0] = np.sin(np.deg2rad(theta))
    Rot_Matrix [1,1] = np.cos(np.deg2rad(theta))
    Rot_Displ[theta,0] = np.max(np.matmul(DISP_X_Y,Rot_Matrix)[:,0])

# Storing Spectra
Rot_Acc = np.dot(Rot_Displ, (omega**2)/g)
GM_SPECTRA.loc[ii-1,'RotD50Sa(g)'] = np.median(Rot_Acc)
GM_SPECTRA.loc[ii-1,'RotD100Sa(g)']= np.max(Rot_Acc)
wipe()

# Writing Spectra to Files
if not os.path.exists('Spectra'):
    os.makedirs('Spectra')
GM_SPECTRA.to_csv('Spectra//GM'+str(iEQ)+'_Spectra.txt', sep=' ', header=True,
index=False)

# Plotting Spectra
if Plot_Spectra == 'Yes':

    def plot_spectra(PlotTitle,SpectraType,iGM):
        axes = fig.add_subplot(1, 1, 1)
        axes.plot(GM_SPECTRA['Period(s)'] , GM_SPECTRA[SpectraType] , '.-',lw=7,
markerSize=20, label='GM'+str(iGM))
        axes.set_xlabel('Period (sec)',fontSize=30,fontWeight='bold')
        axes.set_ylabel(SpectraType,fontSize=30,fontWeight='bold')
        axes.set_title(PlotTitle,fontSize=40,fontWeight='bold')
        axes.tick_params(labelsize= 25)
        axes.grid(True)
        axes.set_xlim(0, np.ceil(max(GM_SPECTRA['Period(s')])))
        axes.set_ylim(0, np.ceil(max(GM_SPECTRA[SpectraType])))
        axes.axhline(linewidth=10,color='black')
        axes.axvline(linewidth=10,color='black')
        axes.hold(True)
        axes.legend(fontsize =30)

    fig = plt.figure(1,figsize=(18,12))

```

(continues on next page)

(continued from previous page)

```

plot_spectra('RotD50 Spectra', 'RotD50Sa(g)', iEQ)

fig = plt.figure(2, figsize=(18,12))
plot_spectra('RotD100 Spectra', 'RotD100Sa(g)', iEQ)

SDOF_RESPONSE.insert(iEQ-1, DISPLACEMENTS)
GM_RESPONSE.insert(iEQ-1, GM_SPECTRA)

print('\nGenerated Spectra for GM: {} \n'.format(np.round(iEQ, 0)))

```

## Portal 2D Frame - Dynamic EQ Ground Motion

Converted to openseespy by: Pavan Chigullapally  
 University of Auckland  
 Email: pchi893@aucklanduni.ac.nz

1. This is a simple model of an elastic portal frame with EQ ground motion and gravity loading. Here the structure is excited using uniform excitation load pattern
2. All units are in kip, inch, second
3. To run EQ ground-motion analysis, BM68elc.acc needs to be downloaded into the same directory
4. The source code is shown below, which can be downloaded [here](#).
5. The detailed problem description can be found [here](#) (example: 1b)

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Mon Apr 22 17:29:26 2019
4
5 @author: pchi893
6 """
7 # Converted to openseespy by: Pavan Chigullapally
8 # University of Auckland
9 # Email: pchi893@aucklanduni.ac.nz
10 # Example 1b. portal frame in 2D
11 # This is a simple model of an elastic portal frame with EQ ground motion and gravity
12 # loading. Here the structure is excited using uniform excitation load pattern
13 # all units are in kip, inch, second
14 # To run EQ ground-motion analysis (BM68elc.acc needs to be downloaded into the same
15 # directory).
16 # the detailed problem description can be found here: http://opensees.berkeley.edu/
17 # wiki/index.php/Examples_Manual (example: 1b)
18 # -----
19 # elasticBeamColumn ELEMENT
20 # OpenSees (Tcl) code by: Silvia Mazzoni & Frank McKenna, 2006
21 #
22 # ^Y
23 # /
24 # 3_____ (3)_____ 4 _____
25 # / | | | | | | | | | | | |

```

(continues on next page)

(continued from previous page)

```
26 # (1)                                (2)      LCol
27 #   /                                /      /
28 #   /                                /      /
29 #   /                                /      /
30 #   =1=                               =2=      _/_<----->X
31 #   /-----LBeam-----/
32 #
33
34 # SET UP -----
35
36 import openseespy.opensees as op
37 #import the os module
38 import os
39 op.wipe()
40
41 ######
42 ######
43 ######
44 op.model('basic', '-ndm', 2, '-ndf', 3)
45
46 #to create a directory at specified path with name "Data"
47 os.chdir('C:\\\\Opensees Python\\\\OpenseesPy examples')
48
49 #this will create the directory with name 'Data' and will update it when we rerun the_
50 #analysis, otherwise we have to keep deleting the old 'Data' Folder
51 dir = "C:\\\\Opensees Python\\\\OpenseesPy examples\\\\Data-1b"
52 if not os.path.exists(dir):
53     os.makedirs(dir)
54
55 #this will create just 'Data' folder
56 #os.mkdir("Data-1b")
57
58 #detect the current working directory
59 #path1 = os.getcwd()
60 #print(path1)
61
62 h = 432.0
63 w = 504.0
64
65 op.node(1, 0.0, 0.0)
66 op.node(2, h, 0.0)
67 op.node(3, 0.0, w)
68 op.node(4, h, w)
69
70 op.fix(1, 1, 1, 1)
71 op.fix(2, 1, 1, 1)
72 op.fix(3, 0, 0, 0)
73 op.fix(4, 0, 0, 0)
74
75 op.mass(3, 5.18, 0.0, 0.0)
76 op.mass(4, 5.18, 0.0, 0.0)
77
78 op.geomTransf('Linear', 1)
79 A = 3600000000.0
80 E = 4227.0
```

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

```
132 u3 = op.nodeDisp(3, 1)
133 print("u2 = ", u3)
134
135 op.wipe()
```

## 2D Column - Dynamic EQ Ground Motion

```
Converted to openseespy by: Pavan Chigullapally
University of Auckland
Email: pchi893@aucklanduni.ac.nz
```

1. EQ ground motion with gravity- uniform excitation of structure
2. All units are in kip, inch, second
3. Note: In this example, all input values for Example 1a are replaced by variables. The objective of this example is to demonstrate the use of variables in defining
4. The OpenSees input and also to run various tests and algorithms at once to increase the chances of convergence
5. To run EQ ground-motion analysis (BM68elc.acc needs to be downloaded into the same directory)
6. The detailed problem description can be found [here](#) (example:2a)
7. The source code is shown below, which can be downloaded [here](#).

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Mon Apr 22 15:12:06 2019
4
5 @author: pchi893
6 """
7 # Converted to openseespy by: Pavan Chigullapally
8 # University of Auckland
9 # Email: pchi893@aucklanduni.ac.nz
10
11 # EQ ground motion with gravity- uniform excitation of structure
12 # all units are in kip, inch, second
13 ##Note: In this example, all input values for Example 1a are replaced by variables.
14 # The objective of this example is to demonstrate the use of variables in defining
15 # the OpenSees input and also to run various tests and algorithms at once to increase
16 # the chances of convergence
17 # Example 2a. 2D cantilever column, dynamic eq ground motion
18 # To run EQ ground-motion analysis (BM68elc.acc needs to be downloaded into the same
19 # directory)
20 # the detailed problem description can be found here: http://opensees.berkeley.edu/
21 # wiki/index.php/Examples_Manual (example:2a)
22 #
23 #      ^Y
24 #      /
25 #      2      —
26 #      /          /
# -----
```

# OpenSees (Tcl) code by: Silvia Mazzoni &amp; Frank McKenna, 2006

(continues on next page)

(continued from previous page)

```

27 #      /          /
28 # (1)      LCol
29 #      /          /
30 #      /          /
31 #      /          /
32 # =1=      _/_   ----->X
33 #
34
35 # SET UP -----
36 import openseespy.opensees as op
37 #import the os module
38 import os
39 import math
40 op.wipe()
41
42 ######
43 ######
44 ######
45 ######
46 op.model('basic', '-ndm', 2, '-ndf', 3)
47
48 #to create a directory at specified path with name "Data"
49 os.chdir('C:\\\\Opensees Python\\\\OpenseesPy examples')
50
51 #this will create the directory with name 'Data' and will update it when we rerun the_
52 #analysis, otherwise we have to keep deleting the old 'Data' Folder
53 dir = "C:\\\\Opensees Python\\\\OpenseesPy examples\\\\Data-2a"
54 if not os.path.exists(dir):
55     os.makedirs(dir)
56
57 #this will create just 'Data' folder
58 #os.mkdir("Data")
59
60 #detect the current working directory
61 #path1 = os.getcwd()
62 #print(path1)
63
64 LCol = 432.0 # column length
65 Weight = 2000.0 # superstructure weight
66
67 # define section geometry
68 HCol = 60.0 # Column Depth
69 BCol = 60.0 # Column Width
70
71 PCol = Weight # nodal dead-load weight per column
72 g = 386.4
73 Mass = PCol/g
74
75 ACol = HCol*BCol*1000 # cross-sectional area, make stiff
76 IzCol = (BCol*math.pow(HCol,3))/12 # Column moment of inertia
77
78 op.node(1, 0.0, 0.0)
79 op.node(2, 0.0, LCol)
80
81 op.fix(1, 1, 1, 1)

```

(continues on next page)

(continued from previous page)

```
81 op.mass(2, Mass, 1e-9, 0.0)
82
83 ColTransfTag = 1
84 op.geomTransf('Linear', ColTransfTag)
85 #A = 3600000000.0
86 #E = 4227.0
87 #Iz = 1080000.0
88
89 fc = -4.0 # CONCRETE Compressive Strength (+Tension, -Compression)
90 Ec = 57*math.sqrt(-fc*1000) # Concrete Elastic Modulus (the term in sqr root needs to_
91 # be in psi
92
93 op.element('elasticBeamColumn', 1, 1, 2, ACol, Ec, IzCol, ColTransfTag)
94
95 op.recorder('Node', '-file', 'Data-2a/DFree.out', '-time', '-node', 2, '-dof', 1,2,3,
96 # disp')
97 op.recorder('Node', '-file', 'Data-2a/DBase.out', '-time', '-node', 1, '-dof', 1,2,3,
98 # disp')
99 op.recorder('Node', '-file', 'Data-2a/RBase.out', '-time', '-node', 1, '-dof', 1,2,3,
100 # reaction')
101 #op.recorder('Drift', '-file', 'Data-2a/Drift.out', '-time', '-node', 1, '-dof', 1,2,3,
102 # disp')
103 op.recorder('Element', '-file', 'Data-2a/FCol.out', '-time', '-ele', 1, 'globalForce')
104 op.recorder('Element', '-file', 'Data-2a/DCol.out', '-time', '-ele', 1, 'deformations')
105
106 #defining gravity loads
107 op.timeSeries('Linear', 1)
108 op.pattern('Plain', 1, 1)
109 op.load(2, 0.0, -PCol, 0.0)
110
111 Tol = 1e-8 # convergence tolerance for test
112 NstepGravity = 10
113 DGravity = 1/NstepGravity
114 op.integrator('LoadControl', DGravity) # determine the next time step for an analysis
115 op.numberer('Plain') # renumber dof's to minimize band-width (optimization), if you_
116 # want to
117 op.system('BandGeneral') # how to store and solve the system of equations in the_
118 # analysis
119 op.constraints('Plain') # how it handles boundary conditions
120 op.test('NormDispIncr', Tol, 6) # determine if convergence has been achieved at the_
121 # end of an iteration step
122 op.algorithm('Newton') # use Newton's solution algorithm: updates tangent stiffness_
123 # at every iteration
124 op.analysis('Static') # define type of analysis static or transient
125 op.analyze(NstepGravity) # apply gravity
126
127 op.loadConst('-time', 0.0) #maintain constant gravity loads and reset time to zero
128
129 #applying Dynamic Ground motion analysis
130 GMdirection = 1
131 GMfile = 'BM68elc.acc'
132 GMfact = 1.0
133
134
135
136
137 Lambda = op.eigen('-fullGenLapack', 1) # eigenvalue mode 1
138 import math
```

(continues on next page)

(continued from previous page)

```

129 Omega = math.pow(Lambda, 0.5)
130 betaKcomm = 2 * (0.02/Omega)
131
132 xDamp = 0.02                                # 2% damping ratio
133 alphaM = 0.0                                 # M-prop. damping; D = alphaM*M
134 betaKcurr = 0.0                               # K-proportional damping;      +beatKcurr*KCurrent
135 betaKinit = 0.0 # initial-stiffness proportional damping      +beatKinit*Kini
136
137 op.rayleigh(alphaM,betaKcurr, betaKinit, betaKcomm) # RAYLEIGH damping
138
139 # Uniform EXCITATION: acceleration input
140 IDloadTag = 400                                # load tag
141 dt = 0.01                                     # time step for input ground motion
142 GMfatt = 1.0                                  # data in input file is in g Units --_
143     ↪ACCELERATION TH
144 maxNumIter = 10
145 op.timeSeries('Path', 2, '-dt', dt, '-filePath', GMfile, '-factor', GMfact)
146 op.pattern('UniformExcitation', IDloadTag, GMdirection, '-accel', 2)
147
148 op.wipeAnalysis()
149 op.constraints('Transformation')
150 op.numberer('Plain')
151 op.system('BandGeneral')
152 op.test('EnergyIncr', Tol, maxNumIter)
153 op.algorithm('ModifiedNewton')
154
155 NewmarkGamma = 0.5
156 NewmarkBeta = 0.25
157 op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
158 op.analysis('Transient')
159
160 DtAnalysis = 0.01
161 TmaxAnalysis = 10.0
162 Nsteps = int(TmaxAnalysis/ DtAnalysis)
163
164 ok = op.analyze(Nsteps, DtAnalysis)
165 tCurrent = op.getTime()
166
167 # for gravity analysis, load control is fine, 0.1 is the load factor increment (http://opensees.berkeley.edu/wiki/index.php/Load\_Control)
168
169 test = {1:'NormDispIncr', 2: 'RelativeEnergyIncr', 4: 'RelativeNormUnbalance', 5:
170     ↪'RelativeNormDispIncr', 6: 'NormUnbalance'}
171 algorithm = {1:'KrylovNewton', 2: 'SecantNewton', 4: 'RaphsonNewton', 5:
172     ↪'PeriodicNewton', 6: 'BFGS', 7: 'Broyden', 8: 'NewtonLineSearch'}
173
174 for i in test:
175     for j in algorithm:
176
177         if ok != 0:
178             if j < 4:
179                 op.algorithm(algorithm[j], '-initial')
180
181         else:
182             op.algorithm(algorithm[j])

```

(continues on next page)

(continued from previous page)

```

182     op.test(test[i], Tol, 1000)
183     ok = op.analyze(Nsteps, DtAnalysis)
184     print(test[i], algorithm[j], ok)
185     if ok == 0:
186         break
187     else:
188         continue
189
190 u2 = op.nodeDisp(2, 1)
191 print("u2 = ", u2)
192
193 op.wipe()

```

## Nonlinear Canti Col Uniaxial Inelastic Section- Dyn EQ GM

Converted to openseespy by: Pavan Chigullapally  
 University of Auckland  
 Email: pchi893@aucklanduni.ac.nz

1. EQ ground motion with gravity- uniform excitation of structure
2. The nonlinear beam-column element that replaces the elastic element of Example 2a requires the definition of the element cross section, or its behavior. In this example,
3. The Uniaxial Section used to define the nonlinear moment-curvature behavior of the element section is “aggregated” to an elastic response for the axial behavior to define
4. The required characteristics of the column element in the 2D model. In a 3D model, torsional behavior would also have to be aggregated to this section.
5. Note:In this example, both the axial behavior (typically elastic) and the flexural behavior (moment curvature) are defined independently and are then “aggregated” into a section.
6. This is a characteristic of the uniaxial section: there is no coupling of behaviors.
7. To run EQ ground-motion analysis (BM68elc.acc needs to be downloaded into the same directory)
8. The problem description can be found [here](#) (example:2b)
9. The source code is shown below, which can be downloaded [here](#).

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Mon Apr 22 15:12:06 2019
4
5 @author: pchi893
6 """
7 # Converted to openseespy by: Pavan Chigullapally
8 # University of Auckland
9 # Email: pchi893@aucklanduni.ac.nz
10 # Example 2b. 2D cantilever column, dynamic eq ground motion
11 # EQ ground motion with gravity- uniform excitation of structure
12 #he nonlinear beam-column element that replaces the elastic element of Example 2a_
13 #requires the definition of the element cross section, or its behavior. In this_
14 #example,
15 #the Uniaxial Section used to define the nonlinear moment-curvature behavior of the_
16 #element section is "aggregated" to an elastic response for the axial behavior to_
17 #define

```

(continues on next page)

(continued from previous page)

```

14 #the required characteristics of the column element in the 2D model. In a 3D model, torsional behavior would also have to be aggregated to this section.
15 #Note:In this example, both the axial behavior (typically elastic) and the flexural behavior (moment curvature) are defined independently and are then "aggregated" into a section.
16 #This is a characteristic of the uniaxial section: there is no coupling of behaviors.
17
18 #To run EQ ground-motion analysis (BM68elc.acc needs to be downloaded into the same directory)
19 #the problem description can be found here: http://opensees.berkeley.edu/wiki/index.php/Examples_Manual(example:2b)
20 #
21 # -----
22 #          OpenSees (Tcl) code by:      Silvia Mazzoni & Frank McKenna, 2006
23 #
24 #      ^Y
25 #      |
26 #      2      —
27 #      |      /
28 #      |      /
29 #      (1)    LCol
30 #      |      /
31 #      |      /
32 #      |      /
33 #      =1=    _/_<----->X
34 #
35
36 # SET UP -----
37 import openseespy.opensees as op
38 #import the os module
39 import os
40 import math
41 op.wipe()
42 ##########
43 #to create a directory at specified path with name "Data"
44 os.chdir('C:\\\\Opensees Python\\\\OpenseesPy examples')
45
46
47 #this will create the directory with name 'Data' and will update it when we rerun the analysis, otherwise we have to keep deleting the old 'Data' Folder
48 dir = "C:\\\\Opensees Python\\\\OpenseesPy examples\\\\Data-2b"
49 if not os.path.exists(dir):
50     os.makedirs(dir)
51 #this will create just 'Data' folder
52 #os.mkdir("Data")
53 #detect the current working directory
54 #path1 = os.getcwd()
55 #print(path1)
56 #####
57 #####
58 #####
59 op.model('basic', '-ndm', 2, '-ndf', 3)
60 LCol = 432.0 # column length

```

(continues on next page)

(continued from previous page)

```

61 Weight = 2000.0 # superstructure weight
62
63 # define section geometry
64 HCol = 60.0 # Column Depth
65 BCol = 60.0 # Column Width
66
67 PCol =Weight # nodal dead-load weight per column
68 g = 386.4
69 Mass = PCol/g
70
71 ACol = HCol*BCol*1000 # cross-sectional area, make stiff
72 IzCol = (BCol*math.pow(HCol,3))/12 # Column moment of inertia
73
74 op.node(1, 0.0, 0.0)
75 op.node(2, 0.0, LCol)
76
77 op.fix(1, 1, 1, 1)
78
79 op.mass(2, Mass, 1e-9, 0.0)
80
81 #Define Elements and Sections
82 ColMatTagFlex = 2
83 ColMatTagAxial = 3
84 ColSecTag = 1
85 BeamSecTag = 2
86
87 fc = -4.0 # CONCRETE Compressive Strength (+Tension, -Compression)
88 Ec = 57*math.sqrt(-fc*1000) # Concrete Elastic Modulus (the term in sqr root needs to
     ↪be in psi
89
90 #Column Section
91 EICol = Ec*IzCol # EI, for moment-curvature relationship
92 EACol = Ec*ACol # EA, for axial-force-strain relationship
93 MyCol = 130000.0 #yield Moment calculated
94 PhiYCol = 0.65e-4 # yield curvature
95 EIColCrack = MyCol/PhiYCol # cracked section inertia
96 b = 0.01 # strain-hardening ratio (ratio between post-yield tangent and initial
     ↪elastic tangent)
97
98 op.uniaxialMaterial('Steel01', ColMatTagFlex, MyCol, EIColCrack, b) #steel moment
     ↪curvature is used for Mz of the section only, # bilinear behavior for flexure
99 op.uniaxialMaterial('Elastic', ColMatTagAxial, EACol) # this is not used as a
     ↪material, this is an axial-force-strain response
100 op.section('Aggregator', ColSecTag, ColMatTagAxial, 'P', ColMatTagFlex, 'Mz') #
     ↪combine axial and flexural behavior into one section (no P-M interaction here)
101
102 ColTransfTag = 1
103 op.geomTransf('Linear', ColTransfTag)
104 numIntgrPts = 5
105 eleTag = 1
106 op.element('nonlinearBeamColumn', eleTag, 1, 2, numIntgrPts, ColSecTag, ColTransfTag)
107
108 op.recorder('Node', '-file', 'Data-2b/DFree.out', '-time', '-node', 2, '-dof', 1, 2, 3,
     ↪'disp')
109 op.recorder('Node', '-file', 'Data-2b/DBase.out', '-time', '-node', 1, '-dof', 1, 2, 3,
     ↪'disp')
110 op.recorder('Node', '-file', 'Data-2b/RBase.out', '-time', '-node', 1, '-dof', 1, 2, 3,
     ↪'reaction')

```

(continues on next page)

(continued from previous page)

```

111 #op.recorder('Drift', '-file', 'Data-2b/Drift.out', '-time', '-node', 1, '-dof', 1,2,3,
112   ↪ 'disp')
113 op.recorder('Element', '-file', 'Data-2b/FCol.out', '-time', '-ele', 1, 'globalForce')
114 op.recorder('Element', '-file', 'Data-2b/ForceColSec1.out', '-time', '-ele', 1,
115   ↪ 'section', 1, 'force')
116 #op.recorder('Element', '-file', 'Data-2b/DCol.out', '-time', '-ele', 1, 'deformations
117   ↪ ')
118
119 #defining gravity loads
120 op.timeSeries('Linear', 1)
121 op.pattern('Plain', 1, 1)
122 op.load(2, 0.0, -PCol, 0.0)
123
124 Tol = 1e-8 # convergence tolerance for test
125 NstepGravity = 10
126 DGravity = 1/NstepGravity
127 op.integrator('LoadControl', DGravity) # determine the next time step for an analysis
128 op.numberer('Plain') # renumber dof's to minimize band-width (optimization), if you
129   ↪ want to
130 op.system('BandGeneral') # how to store and solve the system of equations in the
131   ↪ analysis
132 op.constraints('Plain') # how it handles boundary conditions
133 op.test('NormDispIncr', Tol, 6) # determine if convergence has been achieved at the
134   ↪ end of an iteration step
135 op.algorithm('Newton') # use Newton's solution algorithm: updates tangent stiffness
136   ↪ at every iteration
137 op.analysis('Static') # define type of analysis static or transient
138 op.analyze(NstepGravity) # apply gravity
139
140
141 Lambda = op.eigen('-fullGenLapack', 1) # eigenvalue mode 1
142 import math
143 Omega = math.pow(Lambda, 0.5)
144 betaKcomm = 2 * (0.02/Omega)
145
146 xDamp = 0.02 # 2% damping ratio
147 alphaM = 0.0 # M-prop. damping; D = alphaM*M
148 betaKcurr = 0.0 # K-proportional damping; +beatKcurr*KCurrent
149 betaKinit = 0.0 # initial-stiffness proportional damping +beatKinit*Kini
150
151 op.rayleigh(alphaM,betaKcurr, betaKinit, betaKcomm) # RAYLEIGH damping
152
153
154 # Uniform EXCITATION: acceleration input
155 IDloadTag = 400 # load tag
156 dt = 0.01 # time step for input ground motion
157 GMfatt = 1.0 # data in input file is in g Units --
158   ↪ ACCELERATION TH
159 maxNumIter = 10
160 op.timeSeries('Path', 2, '-dt', dt, '-filePath', GMfile, '-factor', GMfact)

```

(continues on next page)

(continued from previous page)

```
160 op.pattern('UniformExcitation', IDloadTag, GMdirection, '-accel', 2)
161
162 op.wipeAnalysis()
163 op.constraints('Transformation')
164 op.numberer('Plain')
165 op.system('BandGeneral')
166 op.test('EnergyIncr', Tol, maxNumIter)
167 op.algorithm('ModifiedNewton')
168
169 NewmarkGamma = 0.5
170 NewmarkBeta = 0.25
171 op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
172 op.analysis('Transient')
173
174 DtAnalysis = 0.01
175 TmaxAnalysis = 10.0
176
177 Nsteps = int(TmaxAnalysis / DtAnalysis)
178
179 ok = op.analyze(Nsteps, DtAnalysis)
180
181 tCurrent = op.getTime()
182
183 # for gravity analysis, load control is fine, 0.1 is the load factor increment (http://opensees.berkeley.edu/wiki/index.php/Load\_Control)
184
185 test = {1:'NormDispIncr', 2: 'RelativeEnergyIncr', 4: 'RelativeNormUnbalance', 5:
186     ↪'RelativeNormDispIncr', 6: 'NormUnbalance'}
187 algorithm = {1:'KrylovNewton', 2: 'SecantNewton', 4: 'RaphsonNewton', 5:
188     ↪'PeriodicNewton', 6: 'BFGS', 7: 'Broyden', 8: 'NewtonLineSearch'}
189
190 for i in test:
191     for j in algorithm:
192
193         if ok != 0:
194             if j < 4:
195                 op.algorithm(algorithm[j], '-initial')
196
197             else:
198                 op.algorithm(algorithm[j])
199
200                 op.test(test[i], Tol, 1000)
201                 ok = op.analyze(Nsteps, DtAnalysis)
202                 print(test[i], algorithm[j], ok)
203                 if ok == 0:
204                     break
205                 else:
206                     continue
207
208 u2 = op.nodeDisp(2, 1)
209 print("u2 = ", u2)
210
211 op.wipe()
```

## Nonlin Canti Col Inelstc Uniaxial Mat in Fiber Sec - Dyn EQ

Converted to openseespy by: Pavan Chigullapally  
 University of Auckland  
 Email: pchi893@aucklanduni.ac.nz

1. EQ ground motion with gravity- uniform excitation of structure
2. In this example, the Uniaxial Section of Example 2b is replaced by a fiber section. Inelastic uniaxial materials are used in this example,
3. Which are assigned to each fiber, or patch of fibers, in the section.
4. In this example the axial and flexural behavior are coupled, a characteristic of the fiber section.
5. The nonlinear/inelastic behavior of a fiber section is defined by the stress-strain response of the uniaxial materials used to define it.
6. To run EQ ground-motion analysis (BM68elc.acc needs to be downloaded into the same directory)
7. The problem description can be found [here](#) (example:2c)
8. The source code is shown below, which can be downloaded [here](#).

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Mon Apr 22 15:12:06 2019
4
5 @author: pchi893
6 """
7 # Converted to openseespy by: Pavan Chigullapally
8 # University of Auckland
9 # Email: pchi893@aucklanduni.ac.nz
10 # Example 2c. 2D cantilever column, dynamic eq ground motion
11 # EQ ground motion with gravity- uniform excitation of structure
12 #In this example, the Uniaxial Section of Example 2b is replaced by a fiber section. ↴
13 #Inelastic uniaxial materials are used in this example,
14 #which are assigned to each fiber, or patch of fibers, in the section.
15 #In this example the axial and flexural behavior are coupled, a characteristic of the ↴
16 #fiber section.
17 #The nonlinear/inelastic behavior of a fiber section is defined by the stress-strain ↴
18 #response of the uniaxial materials used to define it.
19 #
20 # -----
21 # -----
22 # OpenSees (Tcl) code by: Silvia Mazzoni & Frank McKenna, 2006
23 #
24 # ^Y
25 # /
26 # 2 —
27 # / /
28 # / /
29 # (1) LCol
30 # / /

```

(continues on next page)

(continued from previous page)

```

31 #      /          /
32 #      /          /
33 #  =1=      _/_  ----->X
34 #
35
36 # SET UP -----
37 import openseespy.opensees as op
38 #import the os module
39 import os
40 import math
41 op.wipe()
42 ######
43 #to create a directory at specified path with name "Data"
44 os.chdir('C:\\\\Opensees Python\\\\OpenseesPy examples')
45
46 #this will create the directory with name 'Data' and will update it when we rerun the_
47 #analysis, otherwise we have to keep deleting the old 'Data' Folder
48 dir = "C:\\\\Opensees Python\\\\OpenseesPy examples\\\\Data-2c"
49 if not os.path.exists(dir):
50     os.makedirs(dir)
51 #this will create just 'Data' folder
52 #os.mkdir("Data")
53 #detect the current working directory
54 path1 = os.getcwd()
55 #print(path1)
56
57 ######
58 op.model('basic', '-ndm', 2, '-ndf', 3)
59 LCol = 432.0 # column length
60 Weight = 2000.0 # superstructure weight
61
62 # define section geometry
63 HCol = 60.0 # Column Depth
64 BCol = 60.0 # Column Width
65
66 PCol =Weight # nodal dead-load weight per column
67 g = 386.4
68 Mass = PCol/g
69
70 ACol = HCol*BCol*1000 # cross-sectional area, make stiff
71 IzCol = (BCol*math.pow(HCol,3))/12 # Column moment of inertia
72
73 op.node(1, 0.0, 0.0)
74 op.node(2, 0.0, LCol)
75
76 op.fix(1, 1, 1, 1)
77
78 op.mass(2, Mass, 1e-9, 0.0)
79
80 ColSecTag = 1           # assign a tag number to the column section
81 coverCol = 5.0          # Column cover to reinforcing steel NA.
82 numBarsCol = 16          # number of longitudinal-reinforcement bars in column. (symmetric_
#top & bot)

```

(continues on next page)

(continued from previous page)

```

83 barAreaCol = 2.25 # area of longitudinal-reinforcement bars
84
85 # MATERIAL parameters
86 IDconcU = 1 # material ID tag -- unconfined cover concrete
87 # (here used for complete section)
88 IDreinf = 2 # material ID tag -- reinforcement
89
90 # nominal concrete compressive strength
91 fc = -4.0 # CONCRETE Compressive Strength (+Tension, -
92 # Compression)
93 Ec = 57*math.sqrt(-fc*1000) # Concrete Elastic Modulus (the term in sqr root needs to
94 # be in psi
95
96 # unconfined concrete
97 fc1U = fc # UNCONFINED concrete (todeschini parabolic model),
98 # maximum stress
99 eps1U = -0.003 # strain at maximum strength of unconfined
100 # concrete
101 fc2U = 0.2*fc1U # ultimate stress
102 eps2U = -0.01 # strain at ultimate stress
103 Lambda = 0.1 # ratio between unloading slope at $eps2
104 # and initial slope $Ec
105
106 # tensile-strength properties
107 ftU = -0.14* fc1U # tensile strength +tension
108 Ets = ftU/0.002 # tension softening stiffness
109
110 Fy = 66.8 # STEEL yield stress
111 Es = 29000.0 # modulus of steel
112 Bs = 0.01 # strain-hardening ratio
113 R0 = 18.0 # control the transition from elastic to
114 # plastic branches
115 cR1 = 0.925 # control the transition from elastic to
116 # plastic branches
117 cR2 = 0.15 # control the transition from elastic to
118 # plastic branches
119
120 op.uniaxialMaterial('Concrete02', IDconcU, fc1U, eps1U, fc2U, eps2U, Lambda, ftU,
121 # build cover concrete (unconfined)
122 op.uniaxialMaterial('Steel02', IDreinf, Fy, Es, Bs, R0,cR1,cR2) # build reinforcement
123 # material
124
125 # FIBER SECTION properties -----
126 # -----
127 # symmetric section
128 # Y
129 # ^
130 # /
131 #
132 # -----
133 # / o o o / / -- cover
134 # / / / / / /
135 # z <-- / + / H
136 # / / / / /
137 # / o o o / / -- cover
138 # -----
139 # / ----- B -----/

```

(continues on next page)

(continued from previous page)

```

128 #
129 # RC section:
130
131 coverY = HCol/2.0          # The distance from the section z-axis to the edge of the
132   ↵cover concrete -- outer edge of cover concrete
132 coverZ = BCol/2.0          # The distance from the section y-axis to the edge of the
133   ↵cover concrete -- outer edge of cover concrete
133 coreY = coverY-coverCol
134 coreZ = coverZ-coverCol
135 nfY = 16      # number of fibers for concrete in y-direction
136 nfZ = 4           # number of fibers for concrete in z-direction
137
138 op.section('Fiber', ColSecTag)
139 op.patch('quad', IDconcU, nfZ, nfY, -coverY, coverZ, -coverY, -coverZ, coverY, -coverZ,
140   ↵-coverY, coverZ) # Define the concrete patch
141 op.layer('straight', IDreinf, numBarsCol, barAreaCol, -coreY, coreZ, -coreY, -coreZ)
141 op.layer('straight', IDreinf, numBarsCol, barAreaCol, coreY, coreZ, coreY, -coreZ)
142 ColTransfTag = 1
143 op.geomTransf('Linear', ColTransfTag)
144 numIntgrPts = 5
145 eleTag = 1
146
147 #import InelasticFiberSection
148
149 op.element('nonlinearBeamColumn', eleTag, 1, 2, numIntgrPts, ColSecTag, ColTransfTag)
150
151 op.recorder('Node', '-file', 'Data-2c/DFree.out', '-time', '-node', 2, '-dof', 1, 2, 3,
152   ↵'disp')
152 op.recorder('Node', '-file', 'Data-2c/DBase.out', '-time', '-node', 1, '-dof', 1, 2, 3,
153   ↵'disp')
153 op.recorder('Node', '-file', 'Data-2c/RBase.out', '-time', '-node', 1, '-dof', 1, 2, 3,
154   ↵'reaction')
154 #op.recorder('Drift', '-file', 'Data-2c/Drift.out', '-time', '-node', 1, '-dof', 1, 2, 3,
155   ↵'disp')
155 op.recorder('Element', '-file', 'Data-2c/FCol.out', '-time', '-ele', 1, 'globalForce')
156 op.recorder('Element', '-file', 'Data-2c/ForceColSec1.out', '-time', '-ele', 1,
157   ↵'section', 1, 'force')
157 #op.recorder('Element', '-file', 'Data-2c/DCol.out', '-time', '-ele', 1, 'deformations
158   ↵')

159 #defining gravity loads
160 op.timeSeries('Linear', 1)
161 op.pattern('Plain', 1, 1)
162 op.load(2, 0.0, -PCol, 0.0)
163
164 Tol = 1e-8 # convergence tolerance for test
165 NstepGravity = 10
166 DGravity = 1/NstepGravity
167 op.integrator('LoadControl', DGravity) # determine the next time step for an analysis
168 op.numberer('Plain') # renumber dof's to minimize band-width (optimization), if you
169   ↵want to
169 op.system('BandGeneral') # how to store and solve the system of equations in the
170   ↵analysis
170 op.constraints('Plain') # how it handles boundary conditions
171 op.test('NormDispIncr', Tol, 6) # determine if convergence has been achieved at the
172   ↵end of an iteration step
172 op.algorithm('Newton') # use Newton's solution algorithm: updates tangent stiffness
173   ↵at every iteration

```

(continues on next page)

(continued from previous page)

```

173 op.analysis('Static') # define type of analysis static or transient
174 op.analyze(NstepGravity) # apply gravity
175
176 op.loadConst('-time', 0.0) #maintain constant gravity loads and reset time to zero
177
178 #applying Dynamic Ground motion analysis
179 GMdirection = 1
180 GMfile = 'BM68elc.acc'
181 GMfact = 1.0
182
183
184 Lambda = op.eigen('-fullGenLapack', 1) # eigenvalue mode 1
185 import math
186 Omega = math.pow(Lambda, 0.5)
187 betaKcomm = 2 * (0.02/Omega)
188
189 xDamp = 0.02 # 2% damping ratio
190 alphaM = 0.0 # M-prop. damping; D = alphaM*M
191 betaKcurr = 0.0 # K-proportional damping; +beatKcurr*KCurrent
192 betaKinit = 0.0 # initial-stiffness proportional damping +beatKinit*Kini
193
194 op.rayleigh(alphaM,betaKcurr, betaKinit, betaKcomm) # RAYLEIGH damping
195
196 # Uniform EXCITATION: acceleration input
197 IDloadTag = 400 # load tag
198 dt = 0.01 # time step for input ground motion
199 GMfatt = 1.0 # data in input file is in g Units --_
200 ↪ACCELERATION TH
201 maxNumIter = 10
202 op.timeSeries('Path', 2, '-dt', dt, '-filePath', GMfile, '-factor', GMfact)
203 op.pattern('UniformExcitation', IDloadTag, GMdirection, '-accel', 2)
204
205 op.wipeAnalysis()
206 op.constraints('Transformation')
207 op.numberer('Plain')
208 op.system('BandGeneral')
209 op.test('EnergyIncr', Tol, maxNumIter)
210 op.algorithm('ModifiedNewton')
211
212 NewmarkGamma = 0.5
213 NewmarkBeta = 0.25
214 op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
215 op.analysis('Transient')
216
217 DtAnalysis = 0.01
218 TmaxAnalysis = 10.0
219
220 Nsteps = int(TmaxAnalysis/ DtAnalysis)
221
222 ok = op.analyze(Nsteps, DtAnalysis)
223
224 tCurrent = op.getTime()
225
226 # for gravity analysis, load control is fine, 0.1 is the load factor increment (http://opensees.berkeley.edu/wiki/index.php/Load\_Control)
227

```

(continues on next page)

(continued from previous page)

```
228 test = {1:'NormDispIncr', 2: 'RelativeEnergyIncr', 4: 'RelativeNormUnbalance', 5:  
229     ↪'RelativeNormDispIncr', 6: 'NormUnbalance'}  
230 algorithm = {1:'KrylovNewton', 2: 'SecantNewton', 4: 'RaphsonNewton', 5:  
231     ↪'PeriodicNewton', 6: 'BFGS', 7: 'Broyden', 8: 'NewtonLineSearch'}  
232  
233 for i in test:  
234     for j in algorithm:  
235  
236         if ok != 0:  
237             if j < 4:  
238                 op.algorithm(algorithm[j], '-initial')  
239  
240             else:  
241                 op.algorithm(algorithm[j])  
242  
243             op.test(test[i], Tol, 1000)  
244             ok = op.analyze(Nsteps, DtAnalysis)  
245             print(test[i], algorithm[j], ok)  
246             if ok == 0:  
247                 break  
248             else:  
249                 continue  
250  
251 u2 = op.nodeDisp(2, 1)  
252 print("u2 = ", u2)  
253  
254 op.wipe()
```

## Cantilever 2D Column with Units- Dynamic EQ Ground Motion

```
Converted to openseespy by: Pavan Chigullapally  
University of Auckland  
Email: pchi893@aucklanduni.ac.nz
```

1. To run Uniaxial Inelastic Material, Fiber Section, Nonlinear Mode, Uniform Earthquake Excitation
2. First import the `InelasticFiberSection.py` (upto gravity loading is already in this script) and run the current script
3. To run EQ ground-motion analysis `BM68elc.acc` needs to be downloaded into the same directory
4. Same acceleration input at all nodes restrained in specified direction (uniform acceleration input at all support nodes)
5. The problem description can be found [here](#) (example:3)
6. The source code is shown below, which can be downloaded [here](#).

```
1 # -*- coding: utf-8 -*-  
2 """  
3 Created on Mon Apr 22 15:12:06 2019  
4  
5 @author: pchi893  
6 """  
7 # Converted to openseespy by: Pavan Chigullapally  
8 #                                         University of Auckland
```

(continues on next page)

(continued from previous page)

```

9      # Email: pchi893@aucklanduni.ac.nz
10     # Example 3. 2D Cantilever -- EQ ground motion
11     #To run Uniaxial Inelastic Material, Fiber Section, Nonlinear Mode, Uniform_
12     #→Earthquake Excitation:First import the InelasticFiberSection.py(upto gravity_
13     #→loading is already in this script)
14     #and run the current script
15     #To run EQ ground-motion analysis (BM68elc.acc needs to be downloaded into the same_
16     #→directory)
17     # Same acceleration input at all nodes restrained in specified direction (uniform_
18     #→acceleration input at all support nodes)
19     #the detailed problem description can be found here: http://opensees.berkeley.edu/
20     #→wiki/index.php/Examples_Manual (example: 3)
21     #
22     # -----
23     #-----#
24     # OpenSees (Tcl) code by: Silvia Mazzoni & Frank McKenna, 2006
25     ######
26     import openseespy.opensees as op
27     #import the os module
28     #import os
29     import math
30     op.wipe()
31     ######
32     import InelasticFiberSection
33     #applying Dynamic Ground motion analysis
34     Tol = 1e-8
35     GMdirection = 1
36     GMfile = 'BM68elc.acc'
37     GMfact = 1.0
38     Lambda = op.eigen('-fullGenLapack', 1) # eigenvalue mode 1
39     Omega = math.pow(Lambda, 0.5)
40     betaKcomm = 2 * (0.02/Omega)
41
42     xDamp = 0.02                      # 2% damping ratio
43     alphaM = 0.0                         # M-prop. damping; D = alphaM*M
44     betaKcurr = 0.0                      # K-proportional damping;      +beatKcurr*KCurrent
45     betaKinit = 0.0 # initial-stiffness proportional damping      +beatKinit*Kini
46
47     op.rayleigh(alphaM,betaKcurr, betaKinit, betaKcomm) # RAYLEIGH damping
48
49     # Uniform EXCITATION: acceleration input
50     IDloadTag = 400                      # load tag
51     dt = 0.01                            # time step for input ground motion
52     GMfatt = 1.0                          # data in input file is in g Units --
53     #→ACCELERATION TH
54     maxNumIter = 10
55     op.timeSeries('Path', 2, '-dt', dt, '-filePath', GMfile, '-factor', GMfact)
56     op.pattern('UniformExcitation', IDloadTag, GMdirection, '-accel', 2)
57
58     op.wipeAnalysis()
59     op.constraints('Transformation')
60     op.numberer('Plain')
61     op.system('BandGeneral')
62     op.test('EnergyIncr', Tol, maxNumIter)
63     op.algorithm('ModifiedNewton')

```

(continues on next page)

(continued from previous page)

```
57 NewmarkGamma = 0.5
58 NewmarkBeta = 0.25
59 op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
60 op.analysis('Transient')
61
62 DtAnalysis = 0.01 # time-step Dt for lateral analysis
63 TmaxAnalysis = 10.0 # maximum duration of ground-motion analysis
64
65 Nsteps = int(TmaxAnalysis/ DtAnalysis)
66
67 ok = op.analyze(Nsteps, DtAnalysis)
68
69 tCurrent = op.getTime()
70
71 # for gravity analysis, load control is fine, 0.1 is the load factor increment (http://opensees.berkeley.edu/wiki/index.php/Load\_Control)
72
73 test = {1:'NormDispIncr', 2: 'RelativeEnergyIncr', 4: 'RelativeNormUnbalance', 5:
74     'RelativeNormDispIncr', 6: 'NormUnbalance'}
75 algorithm = {1:'KrylovNewton', 2: 'SecantNewton' , 4: 'RaphsonNewton', 5:
76     'PeriodicNewton', 6: 'BFGS', 7: 'Broyden', 8: 'NewtonLineSearch'}
77
78 for i in test:
79     for j in algorithm:
80
81         if ok != 0:
82             if j < 4:
83                 op.algorithm(algorithm[j], '-initial')
84
85             else:
86                 op.algorithm(algorithm[j])
87
88             op.test(test[i], Tol, 1000)
89             ok = op.analyze(Nsteps, DtAnalysis)
90             print(test[i], algorithm[j], ok)
91             if ok == 0:
92                 break
93             else:
94                 continue
95
96 u2 = op.nodeDisp(2, 1)
97 print("u2 = ", u2)
98
99 op.wipe()
```

## Cantilever 2D Column with Units-Static Pushover

```
Converted to openseespy by: Pavan Chigullapally
University of Auckland
Email: pchi893@aucklanduni.ac.nz
```

1. To run Uniaxial Inelastic Material, Fiber Section, Nonlinear Mode, Static Pushover Analysis
2. First import the `InelasticFiberSection.py` (upto gravity loading is already in this script) and run the current script

3. To run EQ ground-motion analysis BM68elc.acc needs to be downloaded into the same directory)
4. Same acceleration input at all nodes restrained in specified direction (uniform acceleration input at all support nodes)
5. The problem description can be found [here](#) (example:3)
6. The source code is shown below, which can be downloaded [here](#).

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Mon Apr 22 15:12:06 2019
4
5 @author: pchi893
6 """
7 # Converted to openseespy by: Pavan Chigullapally
8 # University of Auckland
9 # Email: pchi893@aucklanduni.ac.nz
10 # Example 3. 2D Cantilever -- Static Pushover
11 #To run Uniaxial Inelastic Material, Fiber Section, Nonlinear Mode, Static Pushover
#Analysis: First import the InelasticFiberSection.py(upto gravity loading is already
#in this script)
12 #and run the current script
13 #the detailed problem description can be found here: http://opensees.berkeley.edu/
#wiki/index.php/Examples_Manual (example: 3)
14 # -----
15 # OpenSees (Tcl) code by: Silvia Mazzoni & Frank McKenna, 2006
16 # characteristics of pushover analysis
17 ######
######
18 import openseespy.opensees as op
19 #import the os module
20 #import os
21 import math
22 op.wipe()
23
24 from InelasticFiberSection import *
25 Dmax = 0.05*LCol
26 Dincr = 0.001*LCol
27 Hload = Weight
28 maxNumIter = 6
29 tol = 1e-8
30
31 op.timeSeries('Linear', 2)
32 op.pattern('Plain', 200, 2)
33 op.load(2, Hload, 0.0,0.0)
34
35 op.wipeAnalysis()
36 op.constraints('Plain')
37 op.numberer('Plain')
38 op.system('BandGeneral')
39 op.test('EnergyIncr', Tol, maxNumIter)
40 op.algorithm('Newton')
41
42 op.integrator('DisplacementControl', IDctrlNode, IDctrlDOF, Dincr)
43 op.analysis('Static')
44

```

(continues on next page)

(continued from previous page)

```
46 Nsteps = int(Dmax/ Dincr)
47
48 ok = op.analyze(Nsteps)
49 print(ok)
50
51 # for gravity analysis, load control is fine, 0.1 is the load factor increment (http://opensees.berkeley.edu/wiki/index.php/Load\_Control)
52
53 test = {1:'NormDispIncr', 2: 'RelativeEnergyIncr', 4: 'RelativeNormUnbalance', 5:
54     ↪'RelativeNormDispIncr', 6: 'NormUnbalance'}
55 algorithm = {1:'KrylovNewton', 2: 'SecantNewton', 4: 'RaphsonNewton', 5:
56     ↪'PeriodicNewton', 6: 'BFGS', 7: 'Broyden', 8: 'NewtonLineSearch'}
57
58 for i in test:
59     for j in algorithm:
60
61         if ok != 0:
62             if j < 4:
63                 op.algorithm(algorithm[j], '-initial')
64
65             else:
66                 op.algorithm(algorithm[j])
67
68             op.test(test[i], Tol, 1000)
69             ok = op.analyze(Nsteps)
70             print(test[i], algorithm[j], ok)
71             if ok == 0:
72                 break
73             else:
74                 continue
75
76 u2 = op.nodeDisp(2, 1)
77 print("u2 = ", u2)
78
79 op.wipe()
```

## 2D Portal Frame with Units- Dynamic EQ Ground Motion

Converted to openseespy by: Pavan Chigullapally  
University of Auckland  
Email: pchi893@aucklanduni.ac.nz

1. To run Uniaxial Inelastic Material, Fiber Section, Nonlinear Mode, Uniform Earthquake Excitation
2. First import the `InelasticFiberSectionPortal2Dframe.py`
3. To run EQ ground-motion analysis (`ReadRecord.py`, `H-E12140.AT2` needs to be downloaded into the same directory)
4. Same acceleration input at all nodes restrained in specified direction (uniform acceleration input at all support nodes)
5. The problem description can be found [here](#) (example:4)
6. The source code is shown below, which can be downloaded [here](#).

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Mon Apr 22 15:12:06 2019
4
5 @author: pchi893
6 """
7 # Converted to openseespy by: Pavan Chigullapally
8 # University of Auckland
9 # Email: pchi893@aucklanduni.ac.nz
10 # Example4. 2D Portal Frame-- Dynamic EQ input analysis
11
12 #To run Uniaxial Inelastic Material, Fiber Section, Nonlinear Mode, Uniform_
13 #Earthquake Excitation:First import the InelasticFiberSectionPortal2Dframe.py
14 #(upto gravity loading is already in this script) and run the current script
15 #To run EQ ground-motion analysis (ReadRecord.py, H-E12140.AT2 needs to be downloaded_
16 #into the same directory)
17 #Same acceleration input at all nodes restrained in specified direction (uniform_
18 #acceleration input at all support nodes)
19 #the problem description can be found here:
20 #http://opensees.berkeley.edu/wiki/index.php/Examples_Manual and http://opensees.
21 #berkeley.edu/wiki/index.php/OpenSees_Example_4._Portal_Frame(example: 4)
22 #
23 # -----
24 # -----
25 # OpenSees (Tcl) code by: Silvia Mazzoni & Frank McKenna, 2006
26 ######
27 import openseespy.opensees as op
28 #import the os module
29 #import os
30 import math
31 op.wipe()
32 #####
33 from InelasticFiberSectionPortal2Dframe import *
34 #applying Dynamic Ground motion analysis
35 Tol = 1e-8
36 maxNumIter = 10
37 GMdirection = 1
38 GMfact = 1.5
39 GMfatt = g*GMfact
40 DtAnalysis = 0.01*sec # time-step Dt for lateral analysis
41 TmaxAnalysis = 10.0*sec # maximum duration of ground-motion analysis
42
43 Lambda = op.eigen('-fullGenLapack', 1) # eigenvalue mode 1
44 Omega = math.pow(Lambda, 0.5)
45 betaKcomm = 2 * (0.02/Omega)
46
47 xDamp = 0.02 # 2% damping ratio
48 alphaM = 0.0 # M-prop. damping; D = alphaM*M
49 betaKcurr = 0.0 # K-proportional damping; +beatKcurr*Kcurrent
50 betaKinit = 0.0 # initial-stiffness proportional damping +beatKinit*Kini
51
52 op.rayleigh(alphaM,betaKcurr, betaKinit, betaKcomm) # RAYLEIGH damping
53
54 # Set some parameters

```

(continues on next page)

(continued from previous page)

```
51 record = 'H-E12140'
52
53 import ReadRecord
54 # Perform the conversion from SMD record to OpenSees record
55 nPts = ReadRecord.ReadRecord(record+'.at2', record+'.dat')
56 #print(dt, nPts)
57
58 # Uniform EXCITATION: acceleration input
59 IDloadTag = 400 # load tag
60 op.timeSeries('Path', 2, '-dt', dt, '-filePath', record+'.dat', '-factor', GMfatt)
61 op.pattern('UniformExcitation', IDloadTag, GMdirection, '-accel', 2)
62
63 op.wipeAnalysis()
64 op.constraints('Transformation')
65 op.numberer('RCM')
66 op.system('BandGeneral')
67 #op.test('EnergyIncr', Tol, maxNumIter)
68 #op.algorithm('ModifiedNewton')
69 #NewmarkGamma = 0.5
70 #NewmarkBeta = 0.25
71 #op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
72 #op.analysis('Transient')
73
74
75 #Nsteps = int(TmaxAnalysis/ DtAnalysis)
76 #
77 #ok = op.analyze(Nsteps, DtAnalysis)
78
79 tCurrent = op.getTime()
80
81 # for gravity analysis, load control is fine, 0.1 is the load factor increment (http://opensees.berkeley.edu/wiki/index.php/Load\_Control)
82
83 test = {1:'NormDispIncr', 2: 'RelativeEnergyIncr', 3:'EnergyIncr', 4:
84     ↪'RelativeNormUnbalance',5: 'RelativeNormDispIncr', 6: 'NormUnbalance'}
85 algorithm = {1:'KrylovNewton', 2: 'SecantNewton', 3:'ModifiedNewton', 4:
86     ↪'RaphsonNewton',5: 'PeriodicNewton', 6: 'BFGS', 7: 'Broyden', 8: 'NewtonLineSearch'}
87
88 tFinal = nPts*dt
89
90 #tFinal = 10.0*sec
91 time = [tCurrent]
92 u3 = [0.0]
93 u4 = [0.0]
94 ok = 0
95 while tCurrent < tFinal:
96     # ok = op.analyze(1, .01)
97     for i in test:
98         for j in algorithm:
99             if j < 4:
100                 op.algorithm(algorithm[j], '-initial')
101
102             else:
103                 op.algorithm(algorithm[j])
104             while ok == 0 and tCurrent < tFinal:
105
106                 op.test(test[i], Tol, maxNumIter)
```

(continues on next page)

(continued from previous page)

```

105     NewmarkGamma = 0.5
106     NewmarkBeta = 0.25
107     op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
108     op.analysis('Transient')
109     ok = op.analyze(1, .01)

110
111     if ok == 0 :
112         tCurrent = op.getTime()
113         time.append(tCurrent)
114         u3.append(op.nodeDisp(3,1))
115         u4.append(op.nodeDisp(4,1))
116         print(test[i], algorithm[j], 'tCurrent=', tCurrent)

117
118
119 import matplotlib.pyplot as plt
120 plt.figure(figsize=(8,8))
121 plt.plot(time, u3)
122 plt.ylabel('Horizontal Displacement of node 3 (in)')
123 plt.xlabel('Time (s)')
124 plt.savefig('Horizontal Disp at Node 3 vs time.jpeg', dpi = 500)
125 plt.show()

126
127 plt.figure(figsize=(8,8))
128 plt.plot(time, u4)
129 plt.ylabel('Horizontal Displacement of node 4 (in)')
130 plt.xlabel('Time (s)')
131 plt.savefig('Horizontal Disp at Node 4 vs time.jpeg', dpi = 500)
132 plt.show()

133
134
135 op.wipe()

```

## 2D Portal Frame with Units- Multiple Support Dynamic EQ Ground Motion-acctimeseries

Converted to openseespy by: Pavan Chigullapally  
 University of Auckland  
 Email: pchi893@aucklanduni.ac.nz

1. To run Uniaxial Inelastic Material, Fiber Section, Nonlinear Mode, MultipleSupport Earthquake ground motion
2. First import the `InelasticFiberSectionPortal2Dframe.py`
3. Upto gravity loading is already in this script and run the current script
4. To run EQ ground-motion analysis (`ReadRecord.py`, `H-E12140.AT2` needs to be downloaded into the same directory)
5. MultipleSupport Earthquake ground motion (different acceleration input at specified support nodes) – two nodes here
6. The problem description can be found [here](#) (example:4)
7. The source code is shown below, which can be downloaded [here](#).

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Mon Apr 22 15:12:06 2019

```

(continues on next page)

(continued from previous page)

```

4
5 @author: pchi893
6 """
7 # Converted to openseespy by: Pavan Chigullapally
8 # University of Auckland
9 # Email: pchi893@aucklanduni.ac.nz
10 # Example4. 2D Portal Frame-- Dynamic EQ input analysis-- multiple-support_
11 # excitation using acceleration timeseries
12
13 #To run Uniaxial Inelastic Material, Fiber Section, Nonlinear Mode, MultipleSupport_
14 #Earthquake ground motion:First import the InelasticFiberSectionPortal2Dframe.py
15 #upto gravity loading is already in this script) and run the current script
16 #To run EQ ground-motion analysis (ReadRecord.py, H-E12140.AT2 needs to be downloaded_
17 #into the same directory)
18 # MultipleSupport Earthquake ground motion (different acceleration input at specified_
19 #support nodes) -- two nodes here
20 #the problem description can be found here:
21 #http://opensees.berkeley.edu/wiki/index.php/Examples_Manual and http://opensees.
22 #berkeley.edu/wiki/index.php/OpenSees_Example_4._Portal_Frame(example: 4)
23 #
24 -----
25 #
26 # OpenSees (Tcl) code by: Silvia Mazzoni & Frank McKenna, 2006
27 ######
28 import openseespy.opensees as op
29 #import the os module
30 #import os
31 import math
32 op.wipe()
33 #####
34 from InelasticFiberSectionPortal2Dframe import *
35 # execute this file after you have built the model, and after you apply gravity
36 #
37
38 # MultipleSupport Earthquake ground motion (different displacement input at spec'd_
39 #support nodes) -- two nodes here
40
41 #applying Dynamic Ground motion analysis
42 iSupportNode = [1, 2]
43 iGMfact = [1.5, 1.5]
44 iGMdirection = [1, 1]
45 iGMfile = ['H-E12140', 'H-E12140']
46 DtAnalysis = 0.01*sec # time-step Dt for lateral analysis
47 TmaxAnalysis = 10.0*sec # maximum duration of ground-motion analysis
48 Tol = 1e-8
49
50 # define DAMPING-----
51 #
52 # apply Rayleigh DAMPING from $xDamp
53 # D=$alphaM*M + $betaKcurr*Kcurrent + $betaKcomm*KlastCommit + $beatKinit*$Kinitial
54 Lambda = op.eigen('-fullGenLapack', 1) # eigenvalue mode 1
55 Omega = math.pow(Lambda, 0.5)
56 betaKcomm = 2 * (0.02/Omega)
57
58 xDamp = 0.02 # 2% damping ratio

```

(continues on next page)

(continued from previous page)

```

51 alphaM = 0.0                                # M-prop. damping; D = alphaM*M
52 betaKcurr = 0.0                            # K-proportional damping;      +beatKcurr*KCurrent
53 betaKinit = 0.0 # initial-stiffness proportional damping      +beatKinit*Kini
54
55 op.rayleigh(alphaM,betaKcurr, betaKinit, betaKcomm) # RAYLEIGH damping
56 #-----
57 #----- perform Dynamic Ground-Motion Analysis
58 # the following commands are unique to the Multiple-Support Earthquake excitation
59 # Set some parameters
60 IDloadTag = 400 # load tag
61 IDgmSeries = 500 # for multipleSupport Excitation
62
63 # read a PEER strong motion database file, extracts dt from the header and converts_
64 #the file
65 # to the format OpenSees expects for Uniform/multiple-support ground motions
66 record = ['H-E12140', 'H-E12140']
67 #dt = []
68 #nPts = []
69
70 import ReadRecord
71 # Perform the conversion from SMD record to OpenSees record
72 #dt, nPts = ReadRecord.ReadRecord(record+'.at2', record+'.dat')
73 #print(dt, nPts)
74 count = 2
75 #use displacement series, create time series('Path'), then create multi-support_
76 #excitation patter (gmtag, 'Plain'), then create imposed ground motion
77 #using groundmotion('nodetag', gmtag), run this in a loop for each support or node_
78 #where the earthquake load is going to be applied.
79 op.pattern('MultipleSupport', IDloadTag)
80 for i in range(len(iSupportNode)):
81     record_single = record[i]
82     GMfatt = (iGMfact[i])*g
83     dt, nPts = ReadRecord.ReadRecord(record_single+'.AT2', record_single+'.dat')
84     op.timeSeries('Path', count, '-dt', dt, '-filePath', record_single+'.dat', '-_
85     #factor', GMfatt)
86     op.groundMotion(IDgmSeries+count, 'Plain', '-accel', count)
87     op.imposedMotion(iSupportNode[i], iGMDirection[i], IDgmSeries+count)
88     count = count + 1
89
90 maxNumIter = 10
91 op.wipeAnalysis()
92 op.constraints('Transformation')
93 op.numberer('RCM')
94 op.system('BandGeneral')
95 #op.test('EnergyIncr', Tol, maxNumIter)
96 #op.algorithm('ModifiedNewton')
97 #NewmarkGamma = 0.5
98 #NewmarkBeta = 0.25
99 #op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
100 #op.analysis('Transient')
101 ##
102 ##
103 #Nsteps = int(TmaxAnalysis/ DtAnalysis)
104 ##
105 #ok = op.analyze(Nsteps, DtAnalysis)

```

(continues on next page)

(continued from previous page)

```

103 tCurrent = op.getTime()
104
105 # for gravity analysis, load control is fine, 0.1 is the load factor increment (http://opensees.berkeley.edu/wiki/index.php/Load\_Control)
106
107 test = {1:'NormDispIncr', 2: 'RelativeEnergyIncr', 3:'EnergyIncr', 4:
108     ↪'RelativeNormUnbalance',5: 'RelativeNormDispIncr', 6: 'NormUnbalance'}
109 algorithm = {1:'KrylovNewton', 2: 'SecantNewton' , 3:'ModifiedNewton' , 4:
110     ↪'RaphsonNewton',5: 'PeriodicNewton', 6: 'BFGS', 7: 'Broyden', 8: 'NewtonLineSearch'}
111
112 #tFinal = TmaxAnalysis
113 tFinal = nPts*dt
114 time = [tCurrent]
115 u3 = [0.0]
116 u4 = [0.0]
117 ok = 0
118
119 while tCurrent < tFinal:
120     #     ok = op.analyze(1, .01)
121     for i in test:
122         for j in algorithm:
123             if j < 4:
124                 op.algorithm(algorithm[j], '-initial')
125             else:
126                 op.algorithm(algorithm[j])
127     while ok == 0 and tCurrent < tFinal:
128
129         op.test(test[i], Tol, maxNumIter)
130         NewmarkGamma = 0.5
131         NewmarkBeta = 0.25
132         op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
133         op.analysis('Transient')
134         ok = op.analyze(1, .01)
135
136         if ok == 0 :
137             tCurrent = op.getTime()
138             time.append(tCurrent)
139             u3.append(op.nodeDisp(3,1))
140             u4.append(op.nodeDisp(4,1))
141             print(test[i], algorithm[j], 'tCurrent=', tCurrent)
142
143 import matplotlib.pyplot as plt
144 plt.figure(figsize=(8,8))
145 plt.plot(time, u3)
146 plt.ylabel('Horizontal Displacement of node 3 (in)')
147 plt.xlabel('Time (s)')
148 plt.savefig('Horizontal Disp at Node 3 vs time-multiple support excitation-acctime.
149 ↪jpeg', dpi = 500)
150 plt.show()
151
152 plt.figure(figsize=(8,8))
153 plt.plot(time, u4)
154 plt.ylabel('Horizontal Displacement of node 4 (in)')
155 plt.xlabel('Time (s)')
156 plt.savefig('Horizontal Disp at Node 4 vs time-multiple support excitation-acctime.
157 ↪jpeg', dpi = 500)

```

(continues on next page)

(continued from previous page)

```

155 plt.show()
156
157
158 op.wipe()
```

## 2D Portal Frame with Units- Multiple Support Dynamic EQ Ground Motion-disptimeseries

Converted to openseespy by: Pavan Chigullapally  
 University of Auckland  
 Email: pchi893@aucklanduni.ac.nz

1. To run Uniaxial Inelastic Material, Fiber Section, Nonlinear Mode, MultipleSupport Earthquake ground motion
2. First import the `InelasticFiberSectionPortal2Dframe.py`
3. To run EQ ground-motion analysis (`ReadRecord.py`, `H-E12140.DT2` needs to be downloaded into the same directory)
4. MultipleSupport Earthquake ground motion (different displacement input at specified support nodes) – two nodes here
5. The problem description can be found [here](#) (example:4)
6. The source code is shown below, which can be downloaded [here](#).

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Mon Apr 22 15:12:06 2019
4
5 @author: pchi893
6 """
7 # Converted to openseespy by: Pavan Chigullapally
8 # University of Auckland
9 # Email: pchi893@aucklanduni.ac.nz
10 # Example4. 2D Portal Frame-- Dynamic EQ input analysis-- multiple-support_
11 # excitation using displacement timeseries
12 # To run Uniaxial Inelastic Material, Fiber Section, Nonlinear Mode, MultipleSupport_
13 # Earthquake ground motion:First import the InelasticFiberSectionPortal2Dframe.py
14 # (upto gravity loading is already in this script) and run the current script
15 # To run EQ ground-motion analysis (ReadRecord.py, H-E12140.DT2 needs to be downloaded_
16 # into the same directory)
17 # MultipleSupport Earthquake ground motion (different displacement input at specified_
18 # support nodes) -- two nodes here
19 # the problem description can be found here:
20 # http://opensees.berkeley.edu/wiki/index.php/Examples_Manual and http://opensees.
21 # berkeley.edu/wiki/index.php/OpenSees_Example_4._Portal_Frame(example: 4)
22 #
23 # -----
24 # OpenSees (Tcl) code by: Silvia Mazzoni & Frank McKenna, 2006
25 #####import openseespy.opensees as op
26 #import the os module
27 #import os
28 import math
```

(continues on next page)

(continued from previous page)

```

25 op.wipe()
26 ######
27 #-----#
28 from InelasticFiberSectionPortal2Dframe import *
29 # execute this file after you have built the model, and after you apply gravity
30 #
31 #
32 # MultipleSupport Earthquake ground motion (different displacement input at spec'd
33 # support nodes) -- two nodes here
34 #
35 #applying Dynamic Ground motion analysis
36 iSupportNode = [1, 2]
37 iGMfact = [1.5, 1.25]
38 iGMdirection = [1, 1]
39 iGMfile = ['H-E12140', 'H-E12140']
40 DtAnalysis = 0.01*sec # time-step Dt for lateral analysis
41 TmaxAnalysis = 10.0*sec # maximum duration of ground-motion analysis
42 Tol = 1e-8
43 #
44 # define DAMPING-----
45 #-----#
46 # apply Rayleigh DAMPING from $xDamp
47 # D=$alphaM*M + $betaKcurr*Kcurrent + $betaKcomm*KlastCommit + $beatKinit*$Kinitial
48 Lambda = op.eigen('-fullGenLapack', 1) # eigenvalue mode 1
49 Omega = math.pow(Lambda, 0.5)
50 betaKcomm = 2 * (0.02/Omega)
51 #
52 xDamp = 0.02 # 2% damping ratio
53 alphaM = 0.0 # M-prop. damping; D = alphaM*M
54 betaKcurr = 0.0 # K-proportional damping; +beatKcurr*KCurrent
55 betaKinit = 0.0 # initial-stiffness proportional damping +beatKinit*Kini
56 #
57 op.rayleigh(alphaM,betaKcurr, betaKinit, betaKcomm) # RAYLEIGH damping
58 #
59 # -----
60 # ----- perform Dynamic Ground-Motion Analysis
61 # the following commands are unique to the Multiple-Support Earthquake excitation
62 # Set some parameters
63 IDloadTag = 400 # load tag
64 IDgmSeries = 500 # for multipleSupport Excitation
65 #
66 # read a PEER strong motion database file, extracts dt from the header and converts
67 # the file
68 # to the format OpenSees expects for Uniform/multiple-support ground motions
69 record = ['H-E12140', 'H-E12140']
70 #dt = []
71 #nPts = []
72 #
73 import ReadRecord
74 # Perform the conversion from SMD record to OpenSees record
75 #dt, nPts = ReadRecord.ReadRecord(record+'.at2', record+'.dat')
76 #print(dt, nPts)
77 count = 2
78 #use displacement series, create time series('Path'), then create multi-support
79 #excitation patter (gmtag, 'Plain'), then create imposed ground motion
80 #using groundmotion('nodetag', gmtag), run this in a loop for each support or node
81 #where the earthquake load is going to be applied.

```

(continues on next page)

(continued from previous page)

```

76 op.pattern('MultipleSupport', IDloadTag)
77 for i in range(len(iSupportNode)):
78     record_single = record[i]
79     GMfatt = (iGMfact[i])*cm
80     nPts = ReadRecord.ReadRecord(record_single+'.DT2', record_single+'.dat')
81     op.timeSeries('Path', count, '-dt', dt, '-filePath', record_single+'.dat', '-
82     ↳factor', GMfatt)
83     op.groundMotion(IDgmSeries+count, 'Plain', '-disp', count)
84     op.imposedMotion(iSupportNode[i], iGMDirection[i], IDgmSeries+count)
85     count = count + 1
86
87 maxNumIter = 10
88 op.wipeAnalysis()
89 op.constraints('Transformation')
90 op.numberer('RCM')
91 op.system('BandGeneral')
92 #op.test('EnergyIncr', Tol, maxNumIter)
93 #op.algorithm('ModifiedNewton')
94 #NewmarkGamma = 0.5
95 #NewmarkBeta = 0.25
96 #op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
97 #op.analysis('Transient')
98 #
99 #
100 #Nsteps = int(TmaxAnalysis/ DtAnalysis)
101 #
102 #ok = op.analyze(Nsteps, DtAnalysis)
103 tCurrent = op.getTime()
104
105 # for gravity analysis, load control is fine, 0.1 is the load factor increment (http://opensees.berkeley.edu/wiki/index.php/Load\_Control)
106
107 test = {1:'NormDispIncr', 2: 'RelativeEnergyIncr', 3:'EnergyIncr', 4:
108     ↳'RelativeNormUnbalance', 5: 'RelativeNormDispIncr', 6: 'NormUnbalance'}
109 algorithm = {1:'KrylovNewton', 2: 'SecantNewton', 3:'ModifiedNewton', 4:
110     ↳'RaphsonNewton', 5: 'PeriodicNewton', 6: 'BFGS', 7: 'Broyden', 8: 'NewtonLineSearch'}
111
112 #tFinal = TmaxAnalysis
113 tFinal = nPts*dt
114 time = [tCurrent]
115 u3 = [0.0]
116 u4 = [0.0]
117 ok = 0
118
119 while tCurrent < tFinal:
120     #    ok = op.analyze(1, .01)
121     for i in test:
122         for j in algorithm:
123             if j < 4:
124                 op.algorithm(algorithm[j], '-initial')
125
126             else:
127                 op.algorithm(algorithm[j])
128             while ok == 0 and tCurrent < tFinal:
129
130                 op.test(test[i], Tol, maxNumIter)

```

(continues on next page)

(continued from previous page)

```

129     NewmarkGamma = 0.5
130     NewmarkBeta = 0.25
131     op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
132     op.analysis('Transient')
133     ok = op.analyze(1, .01)
134
135     if ok == 0 :
136         tCurrent = op.getTime()
137         time.append(tCurrent)
138         u3.append(op.nodeDisp(3,1))
139         u4.append(op.nodeDisp(4,1))
140         print(test[i], algorithm[j], 'tCurrent=', tCurrent)
141
142 import matplotlib.pyplot as plt
143 plt.figure(figsize=(8,8))
144 plt.plot(time, u3)
145 plt.ylabel('Horizontal Displacement of node 3 (in)')
146 plt.xlabel('Time (s)')
147 plt.savefig('Horizontal Disp at Node 3 vs time-multiple support excitation-disptime.
148             →jpeg', dpi = 500)
149 plt.show()
150
151 plt.figure(figsize=(8,8))
152 plt.plot(time, u4)
153 plt.ylabel('Horizontal Displacement of node 4 (in)')
154 plt.xlabel('Time (s)')
155 plt.savefig('Horizontal Disp at Node 4 vs time-multiple support excitation-disptime.
156             →jpeg', dpi = 500)
157 plt.show()
158
159 op.wipe()

```

## 2D Portal Frame with Units- Uniform Dynamic EQ -bidirectional-acctimeseries

Converted to openseespy by: Pavan Chigullapally  
 University of Auckland  
 Email: pchi893@aucklanduni.ac.nz

1. To run Uniaxial Inelastic Material, Fiber Section, Nonlinear Mode, Bidirectional-uniform earthquake ground motion
2. First import the `InelasticFiberSectionPortal2Dframe.py`
3. To run EQ ground-motion analysis (`ReadRecord.py`, `H-E12140.AT2`, `H-E01140.AT2` needs to be downloaded into the same directory)
4. Bidirectional-uniform support excitation using acceleration timeseries (different accelerations are input at all support nodes in two directions) – two support nodes here
5. The problem description can be found [here](#) (example:4)
6. The source code is shown below, which can be downloaded [here](#).

```

1 # -*- coding: utf-8 -*-
2 """

```

(continues on next page)

(continued from previous page)

```

3 Created on Mon Apr 22 15:12:06 2019
4
5 @author: pchi893
6 """
7 # Converted to openseespy by: Pavan Chigullapally
8 # University of Auckland
9 # Email: pchi893@aucklanduni.ac.nz
10
11 # Example4. 2D Portal Frame-- Dynamic EQ input analysis-- Bidirectional-uniform_
12 # support excitation using acceleration timeseries
13
14 #To run Uniaxial Inelastic Material, Fiber Section, Nonlinear Mode, Bidirectional-
15 #uniform earthquake ground motion:First import the_
16 #InelasticFiberSectionPortal2Dframe.py
17 #(upto gravity loading is already in this script) and run the current script
18 #To run EQ ground-motion analysis (ReadRecord.py, H-E12140.AT2 and H-E01140.AT2 needs_
19 #to be downloaded into the same directory)
20 # Bidirectional-uniform support excitation using acceleration timeseries (different_
21 #accelerations are input at all support nodes in two directions) -- two support_
22 #nodes here
23 #the problem description can be found here:
24 #http://opensees.berkeley.edu/wiki/index.php/Examples_Manual and http://opensees.
25 #berkeley.edu/wiki/index.php/OpenSees_Example_4._Portal_Frame(example: 4)
26 #
27 # -----
28 # OpenSees (Tcl) code by: Silvia Mazzoni & Frank McKenna, 2006
29 # -----
30 #####
31 #####
32 #####
33 #####
34 #####
35 #####
36 #####
37 #####
38 #####
39 #####
40 #####
41 #####
42 #####
43 #####
44 #####
45 #####
46 #####

```

**import openseespy.opensees as op**

**#import the os module**

**#import os**

**import math**

**op.wipe()**

**from InelasticFiberSectionPortal2Dframe import \***

**# execute this file after you have built the model, and after you apply gravity**

**#**

**# MultipleSupport Earthquake ground motion (different displacement input at spec'd**

**#support nodes) -- two nodes here**

**#applying Dynamic Ground motion analysis**

**#iSupportNode = [1, 2]**

**iGMfact = [1.5, 0.25]**

**iGMDirection = [1, 2]**

**iGMfile = ['H-E01140', 'H-E12140']**

**DtAnalysis = 0.01\*sec # time-step Dt for lateral analysis**

**TmaxAnalysis = 10.0\*sec # maximum duration of ground-motion analysis**

**Tol = 1e-8**

**# define DAMPING-----**

**# apply Rayleigh DAMPING from \$xDamp**

(continues on next page)

(continued from previous page)

```

47 # $D = $alphaM*M + $betaKcurr*Kcurrent + $betaKcomm*KlastCommit + $beatKinit*$Kinitial
48 Lambda = op.eigen('-fullGenLapack', 1) # eigenvalue mode 1
49 Omega = math.pow(Lambda, 0.5)
50 betaKcomm = 2 * (0.02/Omega)
51
52 xDamp = 0.02 # 2% damping ratio
53 alphaM = 0.0 # M-prop. damping; D = alphaM*M
54 betaKcurr = 0.0 # K-proportional damping; +beatKcurr*KCurrent
55 betaKinit = 0.0 # initial-stiffness proportional damping +beatKinit*Kini
56
57 op.rayleigh(alphaM,betaKcurr, betaKinit, betaKcomm) # RAYLEIGH damping
58 #-----
59 # ----- perform Dynamic Ground-Motion Analysis
60 # the following commands are unique to the Multiple-Support Earthquake excitation
61 # Set some parameters
62 IDloadTag = 400 # load tag
63
64
65 # read a PEER strong motion database file, extracts dt from the header and converts
# the file
66 # to the format OpenSees expects for Uniform/multiple-support ground motions
67 record = ['H-E01140', 'H-E12140']
68 #dt = []
69 #nPts = []
70
71 import ReadRecord
72
73 #this is similar to uniform excitation in single direction
74 count = 2
75 for i in range(len(iGMdirection)):
76     IDloadTag = IDloadTag+count
77     record_single = record[i]
78     GMfatt = (iGMfact[i])*g
79     dt, nPts = ReadRecord.ReadRecord(record_single+'.AT2', record_single+'.dat')
80     op.timeSeries('Path', count, '-dt', dt, '-filePath', record_single+'.dat', '-
#factor', GMfatt)
81     op.pattern('UniformExcitation', IDloadTag, iGMdirection[i], '-accel', 2)
82     count = count + 1
83
84 maxNumIter = 10
85 op.wipeAnalysis()
86 op.constraints('Transformation')
87 op.numberer('RCM')
88 op.system('BandGeneral')
89 #op.test('EnergyIncr', Tol, maxNumIter)
90 #op.algorithm('ModifiedNewton')
91 #NewmarkGamma = 0.5
92 #NewmarkBeta = 0.25
93 #op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
94 #op.analysis('Transient')
95
96
97 #Nsteps = int(TmaxAnalysis/ DtAnalysis)
98 #
99 #ok = op.analyze(Nsteps, DtAnalysis)
100

```

(continues on next page)

(continued from previous page)

```

101 tCurrent = op.getTime()
102
103 # for gravity analysis, load control is fine, 0.1 is the load factor increment (http://opensees.berkeley.edu/wiki/index.php/Load\_Control)
104
105 test = {1:'NormDispIncr', 2: 'RelativeEnergyIncr', 3:'EnergyIncr', 4:
106     ↪'RelativeNormUnbalance',5: 'RelativeNormDispIncr', 6: 'NormUnbalance'}
107 algorithm = {1:'KrylovNewton', 2: 'SecantNewton' , 3:'ModifiedNewton' , 4:
108     ↪'RaphsonNewton',5: 'PeriodicNewton', 6: 'BFGS', 7: 'Broyden', 8: 'NewtonLineSearch'}
109
110 #tFinal = TmaxAnalysis
111 tFinal = nPts*dt
112 time = [tCurrent]
113 u3 = [0.0]
114 u4 = [0.0]
115 ok = 0
116
117 while tCurrent < tFinal:
118     #     ok = op.analyze(1, .01)
119     for i in test:
120         for j in algorithm:
121             if j < 4:
122                 op.algorithm(algorithm[j], '-initial')
123             else:
124                 op.algorithm(algorithm[j])
125     while ok == 0 and tCurrent < tFinal:
126
127         op.test(test[i], Tol, maxNumIter)
128         NewmarkGamma = 0.5
129         NewmarkBeta = 0.25
130         op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
131         op.analysis('Transient')
132         ok = op.analyze(1, .01)
133
134         if ok == 0 :
135             tCurrent = op.getTime()
136             time.append(tCurrent)
137             u3.append(op.nodeDisp(3,1))
138             u4.append(op.nodeDisp(4,1))
139             print(test[i], algorithm[j], 'tCurrent=', tCurrent)
140
141 import matplotlib.pyplot as plt
142 plt.figure(figsize=(8,8))
143 plt.plot(time, u3)
144 plt.ylabel('Horizontal Displacement of node 3 (in)')
145 plt.xlabel('Time (s)')
146 plt.savefig('Horizontal Disp at Node 3 vs time-uniform excitation-acctime.jpeg', dpi_
147     ↪= 500)
148 plt.show()
149
150 plt.figure(figsize=(8,8))
151 plt.plot(time, u4)
152 plt.ylabel('Horizontal Displacement of node 4 (in)')
153 plt.xlabel('Time (s)')
154 plt.savefig('Horizontal Disp at Node 4 vs time-uniform excitation-acctime.jpeg', dpi_
155     ↪= 500)

```

(continues on next page)

(continued from previous page)

```
153 plt.show()  
154 #  
155  
156 op.wipe()
```

### 1.12.3 Tsunami Examples

1. *Dambreak Analysis*
2. *Dambreak with Elastic Obstacle Analysis*

#### Dambreak Analysis

1. The source code is shown below, which can be downloaded [here](#).
2. The folder `dambreak/` must exist before running the script.
3. Run the source code in your favorite Python program.
4. The `ParaView` is needed to view the results. To view the displaced shape of fluid, use the “Warp By Vector” filter with scale factor = 1.0.

```
1 from openseespy.opensees import *  
2  
3 import numpy as np  
4 import matplotlib.pyplot as plt  
5  
6 # -----  
7 # Start of model generation  
8 # -----  
9  
10 # remove existing model  
11 wipe()  
12  
13 # set modelbuilder  
14 model('basic', '-ndm', 2, '-ndf', 2)  
15  
16 # geometric  
17 L = 0.146  
18 H = L*2  
19 H2 = 0.3  
20 h = 0.005  
21 alpha = 1.2  
22 tw = 3*h  
23  
24 # material  
25 rho = 1000.0  
26 mu = 0.0001  
27 b1 = 0.0  
28 b2 = -9.81  
29 thk = 0.012  
30 kappa = -1.0  
31  
32 # time steps  
33 dtmax = 1e-3
```

(continues on next page)

(continued from previous page)

```

34 dtmin = 1e-6
35 totaltime = 1.0
36
37 # filename
38 filename = 'dambreak'
39
40 # recorder
41 recorder('PVD', filename, 'disp', 'vel', 'pressure')
42
43 # nodes
44 node(1, 0.0, 0.0)
45 node(2, L, 0.0)
46 node(3, L, H)
47 node(4, 0.0, H)
48 node(5, 0.0, H2)
49 node(6, 4*L, 0.0)
50 node(7, 4*L, H2)
51 node(8, -tw, H2)
52 node(9, -tw, -tw)
53 node(10, 4*L+tw, -tw)
54 node(11, 4*L+tw, H2)
55
56 # fluid mesh
57 fluid = 4
58 ndf = 2
59 id = -1
60 mesh('line', 1, 9, 4,5,8,9,10,11,7,6,2, id, ndf, h)
61 mesh('line', 2, 3, 2,1,4, id, ndf, h)
62 mesh('line', 3, 3, 2,3,4, id, ndf, h)
63
64 eleArgs = ['PFEMElementBubble', rho, mu, b1, b2, thk, kappa]
65 mesh('tri', fluid, 2, 2,3, id, ndf, h, *eleArgs)
66
67 # wall mesh
68 wall = 5
69 id = 1
70 mesh('tri', wall, 2, 1,2, id, ndf, h)
71
72 for nd in getNodeTags('-mesh', wall):
73     fix(nd, 1,1)
74
75 # save the original modal
76 record()
77
78 # create constraint object
79 constraints('Plain')
80
81 # create numberer object
82 numberer('Plain')
83
84 # create convergence test object
85 test('PFEM', 1e-5, 1e-5, 1e-5, 1e-5, 1e-15, 1e-15, 100, 3, 1, 2)
86
87 # create algorithm object
88 algorithm('Newton')
89
90 # create integrator object

```

(continues on next page)

(continued from previous page)

```
91 integrator('PFEM')
92
93 # create SOE object
94 system('PFEM')
95
96 # create analysis object
97 analysis('PFEM', dtmax, dtmin, b2)
98
99 # analysis
100 while getTime() < totaltime:
101
102     # analysis
103     if analyze() < 0:
104         break
105
106     remesh(alpha)
107
108
109
```

## Dambreak with Elastic Obstacle Analysis

1. The source code is shown below, which can be downloaded [here](#).
2. The folder `obstacle/` must exist before running the script.
3. Run the source code in your favorite Python program.
4. The `ParaView` is needed to view the results. To view the displaced shape of fluid, use the “Warp By Vector” filter with scale factor = 1.0.

```
1 from openseespy.opensees import *
2
3
4 import numpy as np
5 import matplotlib.pyplot as plt
6
7 # -----
8 # Start of model generation
9 # -----
10
11 # remove existing model
12 wipe()
13
14 # set modelbuilder
15 model('basic', '-ndm', 2, '-ndf', 3)
16
17 # geometric
18 L = 0.146
19 H = 2*L
20 H2 = 0.3
21 b = 0.012
22 h = 0.005
23 alpha = 1.2
24 Hb = 20.0*b/3.0
25 tw = 3*h
```

(continues on next page)

(continued from previous page)

```

26
27 # material
28 rho = 1000.0
29 mu = 0.0001
30 b1 = 0.0
31 b2 = -9.81
32 thk = 0.012
33 kappa = -1.0
34 #kappa = 2.15e9
35
36 rhos = 2500.0
37 A = thk*thk
38 E = 1e6
39 Iz = thk*thk*thk*thk/12.0
40 bmass = A*Hb*rhos
41
42 # analysis
43 dtmax = 1e-3
44 dtmin = 1e-6
45 totaltime = 1.0
46
47 filename = 'obstacle'
48
49
50 # recorder
51 recorder('PVD', filename, 'disp', 'vel', 'pressure')
52
53 # nodes
54 node(1, 0.0, 0.0)
55 node(2, L, 0.0)
56 node(3, L, H, '-ndf', 2)
57 node(4, 0.0, H)
58 node(5, 0.0, H2)
59 node(6, 4*L, 0.0)
60 node(7, 4*L, H2)
61 node(8, -tw, H2)
62 node(9, -tw, -tw)
63 node(10, 4*L+tw, -tw)
64 node(11, 4*L+tw, H2)
65 node(12, 2*L, 0.0)
66 node(13, 2*L, Hb)
67
68 # transformation
69 transfTag = 1
70 geomTransf('Corotational', transfTag)
71
72 # section
73 secTag = 1
74 section('Elastic', secTag, E, A, Iz)
75
76 # beam integration
77 inteTag = 1
78 numpts = 2
79 beamIntegration('Legendre', inteTag, secTag, numpts)
80
81 # beam mesh
82 beam = 6

```

(continues on next page)

(continued from previous page)

```
83 id = 1
84 ndf = 3
85 mesh('line', beam, 2, 12, 13, id, ndf, h, 'dispBeamColumn', transfTag, intetTag)
86
87 # fluid mesh
88 fluid = 4
89 ndf = 2
90 id = -1
91 mesh('line', 1, 10, 4, 5, 8, 9, 10, 11, 7, 6, 12, 2, id, ndf, h)
92 mesh('line', 2, 3, 2, 1, 4, id, ndf, h)
93 mesh('line', 3, 3, 2, 3, 4, id, ndf, h)
94
95 eleArgs = ['PFEMElementBubble', rho, mu, b1, b2, thk, kappa]
96 mesh('tri', fluid, 2, 2, 3, id, ndf, h, *eleArgs)
97
98 # wall mesh
99 wall = 5
100 id = 1
101 mesh('tri', wall, 2, 1, 2, id, ndf, h)
102
103 for nd in getNodeTags('-mesh', wall):
104     fix(nd, 1, 1, 1)
105
106 # save the original modal
107 record()
108
109 # create constraint object
110 constraints('Plain')
111
112 # create numberer object
113 numberer('Plain')
114
115 # create convergence test object
116 test('PFEM', 1e-5, 1e-5, 1e-5, 1e-5, 1e-15, 1e-15, 100, 3, 1, 2)
117
118 # create algorithm object
119 algorithm('Newton')
120
121 # create integrator object
122 integrator('PFEM')
123
124 # create SOE object
125 system('PFEM')
126
127 # create analysis object
128 analysis('PFEM', dtmax, dtmin, b2)
129
130 # analysis
131 while getTime() < totaltime:
132
133     # analysis
134     if analyze() < 0:
135         break
136
137     remesh(alpha)
```

## 1.12.4 GeoTechnical Examples

### 1. Laterally-Loaded Pile Foundation

#### Laterally-Loaded Pile Foundation

1. The original model can be found [here](#).
2. The Python code is converted by **Pavan Chigullapally** from **University of Auckland, Auckland** ([pchi893@aucklanduni.ac.nz](mailto:pchi893@aucklanduni.ac.nz)), and shown below, which can be downloaded [here](#).

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Thu Jan 10 18:24:47 2019
4
5 @author: pchi893
6 """
7
8
9
10 #####
11 # Procedure to compute ultimate lateral resistance, p_u, #
12 # and displacement at 50% of lateral capacity, y50, for #
13 # p-y springs representing cohesionless soil. #
14 # Converted to openseespy by: Pavan Chigullapally #
15 # University of Auckland #
16 #
17 # Created by: Hyung-suk Shin #
18 # University of Washington #
19 # Modified by: Chris McGann #
20 # Pedro Arduino #
21 # Peter Mackenzie-Helnwein #
22 # University of Washington #
23 #
24 #####
25 #
26 # references
27 # American Petroleum Institute (API) (1987). Recommended Practice for Planning,_
# Designing and
28 # Constructing Fixed Offshore Platforms. API Recommended Practice 2A(RP-2A),_
# Washington D.C,
29 # 17th edition.
30 #
31 # Brinch Hansen, J. (1961). "The ultimate resistance of rigid piles against_
# transversal forces."
32 # Bulletin No. 12, Geoteknisk Institute, Copenhagen, 59.
33 #
34 # Boulanger, R. W., Kutter, B. L., Brandenberg, S. J., Singh, P., and Chang, D.#
# (2003). Pile
35 # Foundations in liquefied and laterally spreading ground during earthquakes:#
# Centrifuge experiments
36 # and analyses. Center for Geotechnical Modeling, University of California at Davis,
# Davis, CA.
37 # Rep. UCD/CGM-03/01.
38 #
39 #
40 
```

(continues on next page)

(continued from previous page)

```
41 # Reese, L.C. and Van Impe, W.F. (2001), Single Piles and Pile Groups Under Lateral  
42 # Loading.  
43 # A.A. Balkema, Rotterdam, Netherlands.  
44  
45 import math  
46  
47 def get_pyParam ( pyDepth, gamma, phiDegree, b, pEleLength, puSwitch, kSwitch,  
48 # gwtSwitch ):  
49  
50 #-----  
51 # define ultimate lateral resistance, pult  
52 #-----  
53  
54 # pult is defined per API recommendations (Reese and Van Impe, 2001 or API, 1987)  
55 # for puSwitch = 1  
56 # OR per the method of Brinch Hansen (1961) for puSwitch = 2  
57  
58 pi = 3.14159265358979  
59 phi = phiDegree * (pi/180)  
60 zbRatio = pyDepth / b  
61  
62 #-----API recommended method-----  
63  
64 if puSwitch == 1:  
65  
66     # obtain loading-type coefficient A for given depth-to-diameter ratio zb  
67     # ---> values are obtained from a figure and are therefore approximate  
68     zb = []  
69     dataNum = 41  
70     for i in range(dataNum):  
71         b1 = i * 0.125  
72         zb.append(b1)  
73     As = [2.8460, 2.7105, 2.6242, 2.5257, 2.4271, 2.3409, 2.2546, 2.1437, 2.0575,  
74     # 1.9589, 1.8973, 1.8111, 1.7372, 1.6632, 1.5893, 1.5277, 1.4415, 1.3799, 1.3368, 1.  
75     # 2.690, 1.2074, 1.1581,  
76     # 1.1211, 1.0780, 1.0349, 1.0164, 0.9979, 0.9733, 0.9610, 0.9487, 0.9363, 0.  
77     # 9117, 0.8994, 0.8994, 0.8871, 0.8871, 0.8809, 0.8809, 0.8809, 0.8809, 0.8809]  
78  
79     # linear interpolation to define A for intermediate values of depth:diameter  
80     # ratio  
81     for i in range(dataNum):  
82         if zbRatio >= 5.0:  
83             A = 0.88  
84         elif zb[i] <= zbRatio and zbRatio <= zb[i+1]:  
85             A = (As[i+1] - As[i])/(zb[i+1] - zb[i]) * (zbRatio-zb[i]) + As[i]  
86  
87     # define common terms  
88     alpha = phi / 2  
89     beta = pi / 4 + phi / 2  
90     K0 = 0.4  
91  
92     tan_1 = math.tan(pi / 4 - phi / 2)  
93     Ka = math.pow(tan_1 , 2)  
94  
95     # terms for Equation (3.44), Reese and Van Impe (2001)  
96     tan_2 = math.tan(phi)  
97     tan_3 = math.tan(beta - phi)
```

(continues on next page)

(continued from previous page)

```

91     sin_1 = math.sin(beta)
92     cos_1 = math.cos(alpha)
93     c1 = K0 * tan_2 * sin_1 / (tan_3*cos_1)
94
95     tan_4 = math.tan(beta)
96     tan_5 = math.tan(alpha)
97     c2 = (tan_4/tan_3)*tan_4 * tan_5
98
99     c3 = K0 * tan_4 * (tan_2 * sin_1 - tan_5)
100
101    c4 = tan_4 / tan_3 - Ka
102
103    # terms for Equation (3.45), Reese and Van Impe (2001)
104    pow_1 = math.pow(tan_4,8)
105    pow_2 = math.pow(tan_4,4)
106    c5 = Ka * (pow_1-1)
107    c6 = K0 * tan_2 * pow_2
108
109    # Equation (3.44), Reese and Van Impe (2001)
110    pst = gamma * pyDepth * (pyDepth * (c1 + c2 + c3) + b * c4)
111
112    # Equation (3.45), Reese and Van Impe (2001)
113    psd = b * gamma * pyDepth * (c5 + c6)
114
115    # pult is the lesser of pst and psd. At surface, an arbitrary value is defined
116    if pst <=psd:
117        if pyDepth == 0:
118            pu = 0.01
119
120        else:
121            pu = A * pst
122
123    else:
124        pu = A * psd
125
126    # PySimple1 material formulated with pult as a force, not force/length, ↵
127    # multiply by trib. length
128    pult = pu * pEleLength
129
130    #-----Brinch Hansen method-----
131    elif puSwitch == 2:
132        # pressure at ground surface
133        cos_2 = math.cos(phi)
134
135        tan_6 = math.tan(pi/4+phi/2)
136
137        sin_2 = math.sin(phi)
138        sin_3 = math.sin(pi/4 + phi/2)
139
140        exp_1 = math.exp((pi/2+phi)*tan_2)
141        exp_2 = math.exp(-(pi/2-phi) * tan_2)
142
143        Kqo = exp_1 * cos_2 * tan_6 - exp_2 * cos_2 * tan_1
144        Kco = (1/tan_2) * (exp_1 * cos_2 * tan_6 - 1)
145
146        # pressure at great depth
147        exp_3 = math.exp(pi * tan_2)

```

(continues on next page)

(continued from previous page)

```

147     pow_3 = math.pow(tan_2, 4)
148     pow_4 = math.pow(tan_6, 2)
149     dcinf = 1.58 + 4.09 * (pow_3)
150     Nc = (1/tan_2)*(exp_3)*(pow_4 - 1)
151     Ko = 1 - sin_2
152     Kcinf = Nc * dcinf
153     Kqinf = Kcinf * Ko * tan_2
154
155     # pressure at an arbitrary depth
156     aq = (Kqo/(Kqinf - Kqo))*(Ko*sin_2/sin_3)
157     KqD = (Kqo + Kqinf * aq * zbRatio)/(1 + aq * zbRatio)
158
159     # ultimate lateral resistance
160     if pyDepth == 0:
161         pu = 0.01
162     else:
163         pu = gamma * pyDepth * KqD * b
164
165     # PySimple1 material formulated with pult as a force, not force/length, ↵
166     # multiply by trib. length
167     pult = pu * pEleLength
168
169     #-----#
170     # define displacement at 50% lateral capacity, y50
171     #-----#
172
173     # values of y50 depend of the coefficient of subgrade reaction, k, which can be ↵
174     # defined in several ways.
175     # for gwtSwitch = 1, k reflects soil above the groundwater table
176     # for gwtSwitch = 2, k reflects soil below the groundwater table
177     # a linear variation of k with depth is defined for kSwitch = 1 after API (1987)
178     # a parabolic variation of k with depth is defined for kSwitch = 2 after ↵
179     # Boulanger et al. (2003)
180
181     # API (1987) recommended subgrade modulus for given friction angle, values ↵
182     # obtained from figure (approximate)
183
184     ph = [28.8, 29.5, 30.0, 31.0, 32.0, 33.0, 34.0, 35.0, 36.0, 37.0, 38.0, 39.0, 40. ↵
185     0]
186
187     # subgrade modulus above the water table
188     if gwtSwitch == 1:
189         k = [10, 23, 45, 61, 80, 100, 120, 140, 160, 182, 215, 250, 275]
190
191     else:
192         k = [10, 20, 33, 42, 50, 60, 70, 85, 95, 107, 122, 141, 155]
193
194     dataNum = 13
195     for i in range(dataNum):
196         if ph[i] <= phiDegree and phiDegree <= ph[i+1]:
197             khat = (k[i+1]-k[i])/(ph[i+1]-ph[i])*(phiDegree - ph[i]) + k[i]
198
199     # change units from (lb/in^3) to (kN/m^3)
200     k_SIunits = khat * 271.45
201
202     # define parabolic distribution of k with depth if desired (i.e. lin_par switch ↵
203     == 2)

```

(continues on next page)

(continued from previous page)

```

198     sigV = pyDepth * gamma
199
200     if sigV == 0:
201         sigV = 0.01
202
203     if kSwitch == 2:
204         # Equation (5-16), Boulanger et al. (2003)
205         cSigma = math.pow(50 / sigV, 0.5)
206         # Equation (5-15), Boulanger et al. (2003)
207         k_SIunits = cSigma * k_SIunits
208
209     # define y50 based on pult and subgrade modulus k
210
211     # based on API (1987) recommendations, p-y curves are described using tanh_
212     ↪functions.
213     # tcl does not have the atanh function, so must define this specifically
214
215     # i.e. atanh(x) = 1/2*ln((1+x)/(1-x)), |x| < 1
216
217     # when half of full resistance has been mobilized, p(y50)/pult = 0.5
218     x = 0.5
219     log_1 = math.log((1+x)/(1-x))
220     atanh_value = 0.5 * log_1
221
222     # need to be careful at ground surface (don't want to divide by zero)
223     if pyDepth == 0.0:
224         pyDepth = 0.01
225
226     y50 = 0.5 * (pu/ A) / (k_SIunits * pyDepth) * atanh_value
227     # return pult and y50 parameters
228     outResult = []
229     outResult.append(pult)
230     outResult.append(y50)
231
232     return outResult
233
234 ######
235 ######
236 ######
237 ######
238 ######
239 ######
240 ######
241 ######
242 ######
243 ######
244 ######
245 ######
246 ######
247 ######
248 ######
249 ######
250 ######
251 # references
252 # Meyerhof G.G. (1976). "Bearing capacity and settlement of pile foundations."

```

(continues on next page)

(continued from previous page)

```

252 # J. Geotech. Eng., ASCE, 102(3), 195-228.
253 #
254 # Vijayvergya, V.N. (1977). "Load-movement characteristics of piles."
255 # Proc., Ports 77 Conf., ASCE, New York.
256 #
257 # Kulhawy, F.H. ad Mayne, P.W. (1990). Manual on Estimating Soil Properties for
258 # Foundation Design. Electrical Power Research Institute. EPRI EL-6800,
259 # Project 1493-6 Final Report.
260
261 def get_qzParam (phiDegree, b, sigV, G):
262
263     # define required constants; pi, atmospheric pressure (kPa), pa, and coeff. of
264     # lat earth pressure, Ko
265     pi = 3.14159265358979
266     pa = 101
267     sin_4 = math.sin(phiDegree * (pi/180))
268     Ko = 1 - sin_4
269
270     # ultimate tip pressure can be computed by qult = Nq*sigV after Meyerhof (1976)
271     # where Nq is a bearing capacity factor, phi is friction angle, and sigV is eff.
272     # overburden
273     # stress at the pile tip.
274     phi = phiDegree * (pi/180)
275
276     # rigidity index
277     tan_7 = math.tan(phi)
278     Ir = G/(sigV * tan_7)
279     # bearing capacity factor
280     tan_8 = math.tan(pi/4+phi/2)
281     sin_5 = math.sin(phi)
282     pow_4 = math.pow(tan_8,2)
283     pow_5 = math.pow(Ir,(4*sin_5)/(3*(1+sin_5)))
284     exp_4 = math.exp(pi/2-phi)
285
286     Nq = (1+2*Ko)*(1/(3-sin_5))*exp_4*(pow_4)*(pow_5)
287     # tip resistance
288     qu = Nq * sigV
289     # QzSimple1 material formulated with qult as force, not stress, multiply by area of
290     # pile tip
291     pow_6 = math.pow(b, 2)
292     qult = qu * pi*pow_6/4
293
294     # the q-z curve of Vijayvergya (1977) has the form, q(z) = qult*(z/zc)^(1/3)
295     # where zc is critical tip deflection given as ranging from 3-9% of the
296     # pile diameter at the tip.
297
298     # assume zc is 5% of pile diameter
299     zc = 0.05 * b
300
301     # based on Vijayvergya (1977) curve, z50 = 0.125*zc
302     z50 = 0.125 * zc
303
304     # return values of qult and z50 for use in q-z material
305     outResult = []
306     outResult.append(qult)
307     outResult.append(z50)

```

(continues on next page)

(continued from previous page)

```

306     return outResult
307
308 ##########
309 #####
310 ##########
311 #####
312 #####
313 # Procedure to compute ultimate resistance, tult, and #
314 # displacement at 50% mobilization of tult, z50, for #
315 # use in t-z curves for cohesionless soil. #
316 # Converted to openseespy by: Pavan Chigullapally #
317 # University of Auckland #
318 # Created by: Chris McGann #
319 # University of Washington #
320 #
321 ##########
322
323 def get_tzParam ( phi, b, sigV, pEleLength):
324
325 # references
326 # Mosher, R.L. (1984). "Load transfer criteria for numerical analysis of
327 # axial loaded piles in sand." U.S. Army Engineering and Waterways
328 # Experimental Station, Automatic Data Processing Center, Vicksburg, Miss.
329 #
330 # Kulhawy, F.H. (1991). "Drilled shaft foundations." Foundation engineering
331 # handbook, 2nd Ed., Chap 14, H.-Y. Fang ed., Van Nostrand Reinhold, New York
332
333 pi = 3.14159265358979
334
335 # Compute tult based on tult = Ko*sigV*pi*dia*tan(delta), where
336 # Ko is coeff. of lateral earth pressure at rest,
337 # taken as Ko = 0.4
338 # delta is interface friction between soil and pile,
339 # taken as delta = 0.8*phi to be representative of a
340 # smooth precast concrete pile after Kulhawy (1991)
341
342 delta = 0.8 * phi * pi/180
343
344 # if z = 0 (ground surface) need to specify a small non-zero value of sigV
345
346 if sigV == 0.0:
347     sigV = 0.01
348
349 tan_9 = math.tan(delta)
350 tu = 0.4 * sigV * pi * b * tan_9
351
352 # TzSimple1 material formulated with tult as force, not stress, multiply by_
353 # tributary length of pile
354 tult = tu * pEleLength
355
356 # Mosher (1984) provides recommended initial tangents based on friction angle
357 # values are in units of psf/in
358 kf = [6000, 10000, 10000, 14000, 14000, 18000]
359 fric = [28, 31, 32, 34, 35, 38]

```

(continues on next page)

(continued from previous page)

```

360     dataNum = len(fric)
361
362
363         # determine kf for input value of phi, linear interpolation for intermediate_
364         ↪values
364     if phi < fric[0]:
365         k = kf[0]
366     elif phi > fric[5]:
367         k = kf[5]
368     else:
369         for i in range(dataNum):
370             if fric[i] <= phi and phi <= fric[i+1]:
371                 k = ((kf[i+1] - kf[i])/(fric[i+1] - fric[i])) * (phi - fric[i]) +_
372         ↪kf[i]
373
374     # need to convert kf to units of kN/m^3
375     kSIunits = k * 1.885
376
377     # based on a t-z curve of the shape recommended by Mosher (1984), z50 = tult/kf
378     z50 = tult / kSIunits
379
380     # return values of tult and z50 for use in t-z material
381     outResult = []
382     outResult.append(tult)
383     outResult.append(z50)
384
385     return outResult
386
387
388 ######
389 ######
390 ######
391 ######
392 ######
393 #
394 # Static pushover of a single pile, modeled as a beam on #
395 # a nonlinear Winkler foundation. Lateral soil response #
396 # is described by p-y springs. Vertical soil response #
397 # described by t-z and q-z springs. #
398 # Converted to openseespy by: Pavan Chigullapally #
399 # University of Auckland #
400 # Created by: Chris McGann #
401 # HyungSuk Shin #
402 # Pedro Arduino #
403 # Peter Mackenzie-Helnwein #
404 # --University of Washington-- #
405 #
406 # ---> Basic units are kN and meters #
407 #
408 ######
409
410
411 from openseespy.opensees import *
412

```

(continues on next page)

(continued from previous page)

```

413 op.wipe()
414
415 ######
416 # all the units are in SI units N and mm
417 ######
418
419 # pile geometry and mesh
420
421 #-----#
422 # pile geometry and mesh
423 #-----#
424
425 # length of pile head (above ground surface) (m)
426 L1 = 1.0
427 # length of embedded pile (below ground surface) (m)
428 L2 = 20.0
429 # pile diameter
430 diameter = 1.0
431
432 # number of pile elements
433 nElePile = 84
434 # pile element length
435 eleSize = (L1+L2)/nElePile
436
437 # number of total pile nodes
438 nNodePile = 1 + nElePile
439
440 #-----#
441 # create spring nodes
442 #-----#
443 # spring nodes created with 3 dim, 3 dof
444 op.model('basic', '-ndm', 3, '-ndf', 3)
445
446 # counter to determine number of embedded nodes
447 count = 0
448
449 # create spring nodes
450
451 #1 to 85 are spring nodes
452
453 pile_nodes = dict()
454
455 for i in range(nNodePile):
456     zCoord = eleSize * i
457     if zCoord <= L2:
458         op.node(i+1, 0.0, 0.0, zCoord)
459         op.node(i+101, 0.0, 0.0, zCoord)
460         pile_nodes[i+1] = (0.0, 0.0, zCoord)
461         pile_nodes[i+101] = (0.0, 0.0, zCoord)
462         count = count + 1
463
464 print("Finished creating all spring nodes...")
465
466 # number of embedded nodes
467 nNodeEmbed = count

```

(continues on next page)

(continued from previous page)

```

468
469 # spring node fixities
470 for i in range(nNodeEmbed):
471     op.fix(i+1, 1, 1, 1)
472     op.fix(i+101, 0, 1, 1)
473
474 print("Finished creating all spring node fixities...")
475
476 #-----
477 # soil properties
478 #-----
479
480 # soil unit weight (kN/m^3)
481 gamma = 17.0
482 # soil internal friction angle (degrees)
483 phi = 36.0
484 # soil shear modulus at pile tip (kPa)
485 Gsoil = 150000.0
486
487 # select pult definition method for p-y curves
488 # API (default) --> 1
489 # Brinch Hansen --> 2
490 puSwitch = 1
491
492 # variation in coefficent of subgrade reaction with depth for p-y curves
493 # API linear variation (default) --> 1
494 # modified API parabolic variation --> 2
495 kSwitch = 1
496
497 # effect of ground water on subgrade reaction modulus for p-y curves
498 # above gwt --> 1
499 # below gwt --> 2
500 gwtSwitch = 1
501
502 #-----
503 # create spring material objects
504 #-----
505
506 # p-y spring material
507
508 for i in range(1, nNodeEmbed+1):
509     # depth of current py node
510     pyDepth = L2 - eleSize * (i-1)
511     # procedure to define pult and y50
512     pyParam = get_pyParam(pyDepth, gamma, phi, diameter, eleSize, puSwitch, kSwitch,
513     ↴gwtSwitch)
514     pult = pyParam [0]
515     y50 = pyParam [1]
516     op.uniaxialMaterial('PySimple1', i, 2, pult, y50, 0.0)
517
518 # t-z spring material
519 for i in range(2, nNodeEmbed+1):
520     # depth of current tz node
521     pyDepth = eleSize * (i-1)
522     # vertical effective stress at current depth
523     sigV = gamma * pyDepth

```

(continues on next page)

(continued from previous page)

```

524 # procedure to define tult and z50
525 tzParam = get_tzParam(phi, diameter, sigV, eleSize)
526 tult = tzParam [0]
527 z50 = tzParam [1]
528 op.uniaxialMaterial('TzSimple1', i+100, 2, tult, z50, 0.0)
529
530
531 # q-z spring material
532
533     # vertical effective stress at pile tip, no water table (depth is embedded pile_
534     ↵length)
535 sigVq = gamma * L2
536     # procedure to define qult and z50
537 qzParam = get_qzParam (phi, diameter, sigVq, Gsoil)
538 qult = qzParam [0]
539 z50q = qzParam [1]
540
541 #op.uniaxialMaterial('QzSimple1', 101, 2, qult, z50q) #, 0.0, 0.0
542 op.uniaxialMaterial('TzSimple1', 101, 2, qult, z50q, 0.0)
543
544 print("Finished creating all p-y, t-z, and z-z spring material objects...")
545
546 #-----
547 #   create zero-length elements for springs
548 #-----
549
550 # element at the pile tip (has q-z spring)
551 op.element('zeroLength', 1001, 1, 101, '-mat', 1, 101, '-dir', 1, 3)
552
553 # remaining elements
554 for i in range(2, nNodeEmbed+1):
555     op.element('zeroLength', 1000+i, i, 100+i, '-mat', i, 100+i, '-dir', 1, 3)
556
557 print("Finished creating all zero-Length elements for springs...")
558
559 #-----
560 #   create pile nodes
561 #-----
562
563 # pile nodes created with 3 dimensions, 6 degrees of freedom
564 op.model('basic', '-ndm', 3, '-ndf', 6)
565
566 # create pile nodes
567 for i in range(1, nNodePile+1):
568     zCoord = eleSize * i
569     op.node(i+200, 0.0, 0.0, zCoord)
570
571 print("Finished creating all pile nodes...")
572
573 # create coordinate-transformation object
574 op.geomTransf('Linear', 1, 0.0, -1.0, 0.0)
575
576
577 # create fixity at pile head (location of loading)
578 op.fix(200+nNodePile, 0, 1, 0, 1, 0, 1)
579

```

(continues on next page)

(continued from previous page)

```
580
581 # create fixities for remaining pile nodes
582 for i in range(201, 200+nNodePile):
583     op.fix(i, 0, 1, 0, 1, 0, 1)
584
585 print("Finished creating all pile node fixities...")
586
587 #-----
588 # define equal dof between pile and spring nodes
589 #-----
590
591 for i in range(1, nNodeEmbed+1):
592     op.equalDOF(200+i, 100+i, 1, 3)
593
594 print("Finished creating all equal degrees of freedom...")
595
596 #-----
597 # pile section
598 #-----
599 ##########
600 #>#####
601 ##########
602 #>#####
603 #-----
604 # create elastic pile section
605 #-----
606
607 secTag = 1
608 E = 25000000.0
609 A = 0.785
610 Iz = 0.049
611 Iy = 0.049
612 G = 9615385.0
613 J = 0.098
614
615 matTag = 3000
616 op.section('Elastic', 1, E, A, Iz, Iy, G, J)
617
618 # elastic torsional material for combined 3D section
619 op.uniaxialMaterial('Elastic', 3000, 1e10)
620
621 # create combined 3D section
622 secTag3D = 3
623 op.section('Aggregator', secTag3D, 3000, 'T', '-section', 1)
624
625
626 ##########
627 #>#####
628 ##########
629 #>#####
630 # elastic pile section
631 #import elasticPileSection
```

(continues on next page)

(continued from previous page)

```

633 #-----
634 #  create pile elements
635 #-----
636 op.beamIntegration('Legendre', 1, secTag3D, 3) # we are using gauss-Legendre
#integration as it is the default integration scheme used in opensees tcl (check
#dispBeamColumn)
637
638 for i in range(201, 201+nElePile):
639     op.element('dispBeamColumn', i, i, i+1, 1, 1)
640
641 print("Finished creating all pile elements...")
642
643 #-----
644 #  create recorders
645 #-----
646
647 # record information at specified increments
timeStep = 0.5
649
650 # record displacements at pile nodes
op.recorder('Node', '-file', 'pileDisp.out', '-time', '-dT', timeStep, '-nodeRange',
#201, 200 + nNodePile, '-dof', 1,2,3, 'disp')
652
653 # record reaction force in the p-y springs
op.recorder('Node', '-file', 'reaction.out', '-time', '-dT', timeStep, '-nodeRange', 1,
# nNodePile, '-dof', 1, 'reaction')
655
656 # record element forces in pile elements
op.recorder('Element', '-file', 'pileForce.out', '-time', '-dT', timeStep, '-eleRange',
# 201, 200+nElePile, 'globalForce')
658
659 print("Finished creating all recorders...")
660
661 #-----
662 #  create the loading
663 #-----
664
665 op.setTime(10.0)
666
667 # apply point load at the uppermost pile node in the x-direction
values = [0.0, 0.0, 1.0, 1.0]
time = [0.0, 10.0, 20.0, 10000.0]
670
671 nodeTag = 200+nNodePile
672 loadValues = [3500.0, 0.0, 0.0, 0.0, 0.0, 0.0]
673 op.timeSeries('Path', 1, '-values', *values, '-time', *time, '-factor', 1.0)
674
675 op.pattern('Plain', 10, 1)
676 op.load(nodeTag, *loadValues)
677
678 print("Finished creating loading object...")
679
680 #-----
681 #  create the analysis
682 #-----
683 op.integrator('LoadControl', 0.05)
684 op.numberer('RCM')

```

(continues on next page)

(continued from previous page)

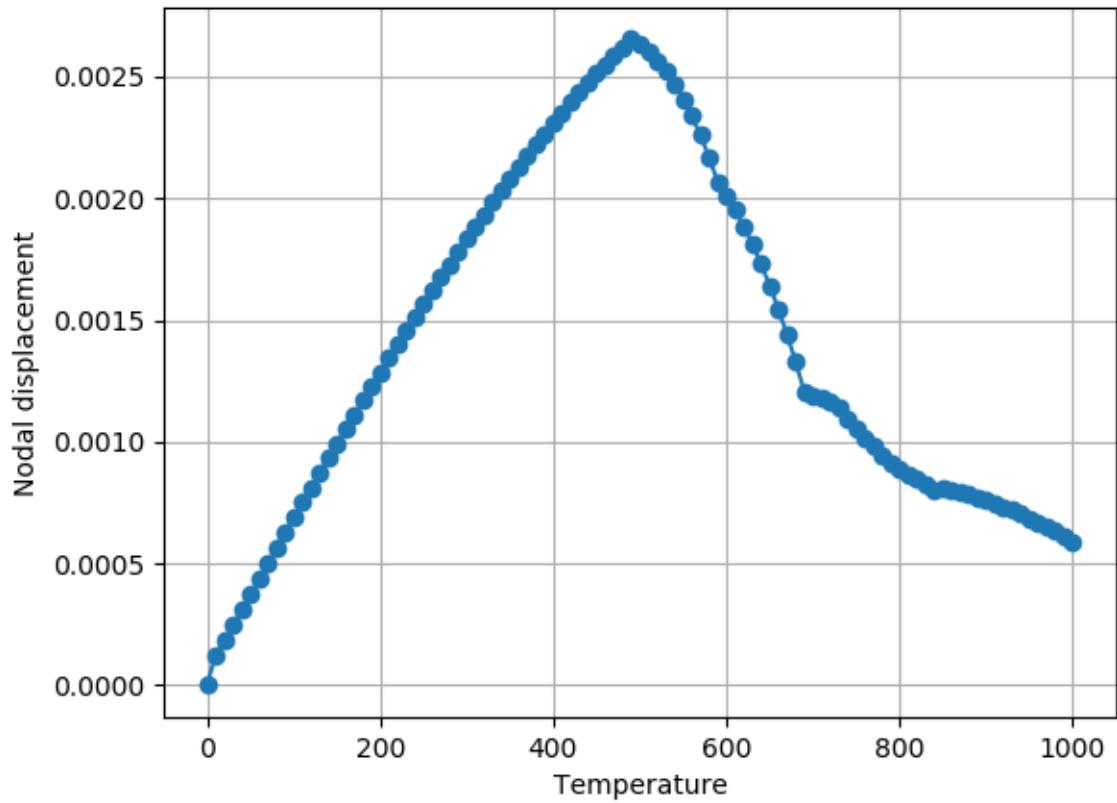
```
685 op.system('SparseGeneral')
686 op.constraints('Transformation')
687 op.test('NormDispIncr', 1e-5, 20, 1)
688 op.algorithm('Newton')
689 op.analysis('Static')
690
691 print("Starting Load Application...")
692 op.analyze(201)
693
694 print("Load Application finished...")
# print("Loading Analysis execution time: [expr $endT-$startT] seconds.")
695
696 #op.wipe
697
698 op.reactions()
699 Nodereactions = dict()
700 Nodedisplacements = dict()
701 for i in range(201,nodeTag+1):
702     Nodereactions[i] = op.nodeReaction(i)
703     Nodedisplacements[i] = op.nodeDisp(i)
704 print('Node Reactions are: ', Nodereactions)
705 print('Node Displacements are: ', Nodedisplacements)
706
707
708
709
710
711
```

## 1.12.5 Thermal Examples

### 1. *Restrained beam under thermal expansion*

#### Restrained beam under thermal expansion

1. The original model can be found [here](#).
2. The Pypton source code is shown below, which can be downloaded [here](#).
3. Make sure the `numpy` and `matplotlib` packages are installed in your Python distribution.
4. Run the source code in your favorite Python program and should see



```

1  from openseespy.opensees import *
2
3  import numpy as np
4  import matplotlib.pyplot as plt
5
6  # define model
7  model('basic', '-ndm', 2, '-ndf', 3)
8
9  #define node
10 node(1, 0.0, 0.0)
11 node(2, 2.0, 0.0)
12 node(3, 1.0, 0.0)
13
14 #define boundary condition
15 fix(1, 1, 1, 1)
16 fix(2, 1, 1, 1)
17 fix(3, 0, 1, 1)
18
19 #define an elastic material with Tag=1 and E=2e11.
20 matTag = 1
21 uniaxialMaterial('Steel01Thermal', 1, 2e11, 2e11, 0.01)
22
23 #define fibred section Two fibres: fiber $yLoc $zLoc $A $matTag
24 secTag = 1
25 section('FiberThermal', secTag)

```

(continues on next page)

(continued from previous page)

```
26 fiber(-0.025, 0.0, 0.005, matTag)
27 fiber(0.025, 0.0, 0.005, matTag)
28
29 #define coordinate transformation
30 #three transformation types can be chosen: Linear, PDelta, Corotational)
31 transfTag = 1
32 geomTransf('Linear', transfTag)
33
34 # beam integration
35 np = 3
36 biTag = 1
37 beamIntegration('Lobatto',biTag, secTag, np)
38
39 #define beam element
40 element('dispBeamColumnThermal', 1, 1, 3, transfTag, biTag)
41 element('dispBeamColumnThermal', 2, 3, 2, transfTag, biTag)
42
43 # define time series
44 tsTag = 1
45 timeSeries('Linear',tsTag)
46
47 # define load pattern
48 patternTag = 1
49 maxtemp = 1000.0
50 pattern('Plain', patternTag, tsTag)
51 eleLoad('-ele', 1, '-type', '-beamThermal', 1000.0, -0.05, 1000.0, 0.05)
52 #eleLoad -ele 2 -type -beamThermal 0 -0.05 0 0.05
53
54 # define analysis
55 incrtemp = 0.01
56 system('BandGeneral')
57 constraints('Plain')
58 numberer('Plain')
59 test('NormDispIncr', 1.0e-3, 100, 1)
60 algorithm('Newton')
61 integrator('LoadControl', incrtemp)
62 analysis('Static')
63
64 # analysis
65 nstep = 100
66 temp = [0.0]
67 disp = [0.0]
68 for i in range(nstep):
69     if analyze(1) < 0:
70         break
71
72     temp.append(getLoadFactor(patternTag)*maxtemp)
73     disp.append(nodeDisp(3,1))
74
75
76 plt.plot(temp,disp,'-o')
77 plt.xlabel('Temperature')
78 plt.ylabel('Nodal displacement')
79 plt.grid()
80 plt.show()
```

## 1.12.6 Parallel Examples

1. [Hello World Example 1](#)
2. [Hello World Example 2](#)
3. [Parallel Truss Example](#)
4. [Parallel Tri31 Example](#)
5. [Parallel Parametric Study Example](#)

### Hello World Example 1

1. The source code is shown below, which can be downloaded [here](#).
2. Run the source code with 4 processors

```
mpiexec -np 4 python hello.py
```

the outputs look like

```
Hello World Process: 1
Hello World Process: 2
Hello World Process: 0
Total number of processes: 4
Hello World Process: 3
Process 1 Terminating
Process 2 Terminating
Process 0 Terminating
Process 3 Terminating
```

The script is shown below

```
1 import openseespy.opensees as ops
2
3 pid = ops.getPID()
4 np = ops.getNP()
5
6 print('Hello World Process:', pid)
7 if pid == 0:
8     print('Total number of processes:', np)
9
```

### Hello World Example 2

1. The source code is shown below, which can be downloaded [here](#).
2. Run the source code with 4 processors

```
mpiexec -np 4 python hello2.py
```

the outputs look like

```
Random:
Hello from 2
Hello from 1
```

(continues on next page)

(continued from previous page)

```
Hello from 3

Ordered:
Hello from 1
Hello from 2
Hello from 3

Broadcasting:
Hello from 0
Hello from 0
Hello from 0
Process 3 Terminating
Process 2 Terminating
Process 1 Terminating
Process 0 Terminating
```

The script is shown below

```
1 import openseespy.opensees as ops
2
3 pid = ops.getPID()
4 np = ops.getNP()
5
6 # datatype = 'float'
7 # datatype = 'int'
8 datatype = 'str'
9
10 if pid == 0:
11     print('Random: ')
12
13     for i in range(1, np):
14         data = ops.recv('-pid', 'ANY')
15         print(data)
16 else:
17     if datatype == 'str':
18         ops.send('-pid', 0, 'Hello from {}'.format(pid))
19     elif datatype == 'float':
20         ops.send('-pid', 0, float(pid))
21     elif datatype == 'int':
22         ops.send('-pid', 0, int(pid))
23
24 ops.barrier()
25
26 if pid == 0:
27     print('\nOrdered: ')
28
29     for i in range(1, np):
30         data = ops.recv('-pid', i)
31         print(data)
32 else:
33     if datatype == 'str':
34         ops.send('-pid', 0, 'Hello from {}'.format(pid))
35     elif datatype == 'float':
36         ops.send('-pid', 0, float(pid))
37     elif datatype == 'int':
38         ops.send('-pid', 0, int(pid))
39
```

(continues on next page)

(continued from previous page)

```

40 ops.barrier()
41 if pid == 0:
42     print('\nBroadcasting: ')
43     if datatype == 'str':
44         ops.Bcast('Hello from {}'.format(pid))
45     elif datatype == 'float':
46         ops.Bcast(float(pid))
47     elif datatype == 'int':
48         ops.Bcast(int(pid))
49 else:
50     data = ops.Bcast()
51     print(data)

```

## Parallel Truss Example

1. The source code is shown below, which can be downloaded [here](#).
2. Run the source code with 2 processors

```
mpiexec -np 2 python paralleltruss.py
```

the outputs look like

```

Node 4: [[72.0, 96.0], [0.5300927771322836, -0.17789363846931772]]
Node 4: [[72.0, 96.0], [0.5300927771322836, -0.17789363846931772]]
Node 4: [[72.0, 96.0], [1.530092777132284, -0.19400676316761836]]
Node 4: [[72.0, 96.0], [1.530092777132284, -0.19400676316761836]]
opensees.msg: TIME(sec) Real: 0.208238

opensees.msg: TIME(sec) Real: 0.209045

Process 0 Terminating
Process 1 Terminating

```

The script is shown below

```

1 import openseespy.opensees as ops
2
3 pid = ops.getPID()
4 np = ops.getNP()
5 ops.start()
6 if np != 2:
7     exit()
8
9 ops.model('basic', '-ndm', 2, '-ndf', 2)
10 ops.uniaxialMaterial('Elastic', 1, 3000.0)
11
12 if pid == 0:
13     ops.node(1, 0.0, 0.0)
14     ops.node(4, 72.0, 96.0)
15
16     ops.fix(1, 1, 1)
17
18     ops.element('Truss', 1, 1, 4, 10.0, 1)
19     ops.timeSeries('Linear', 1)

```

(continues on next page)

(continued from previous page)

```
20     ops.pattern('Plain', 1, 1)
21     ops.load(4, 100.0, -50.0)
22
23 else:
24     ops.node(2, 144.0, 0.0)
25     ops.node(3, 168.0, 0.0)
26     ops.node(4, 72.0, 96.0)
27
28     ops.fix(2, 1, 1)
29     ops.fix(3, 1, 1)
30
31     ops.element('Truss', 2, 2, 4, 5.0, 1)
32     ops.element('Truss', 3, 3, 4, 5.0, 1)
33
34 ops.constraints('Transformation')
35 ops.numbererer('ParallelPlain')
36 ops.system('Mumps')
37 ops.test('NormDispIncr', 1e-6, 6, 2)
38 ops.algorithm('Newton')
39 ops.integrator('LoadControl', 0.1)
40 ops.analysis('Static')
41
42 ops.analyze(10)
43
44 print('Node 4: ', [ops.nodeCoord(4), ops.nodeDisp(4)])
45
46 ops.loadConst('-time', 0.0)
47
48 if pid == 0:
49     ops.pattern('Plain', 2, 1)
50     ops.load(4, 1.0, 0.0)
51
52 ops.domainChange()
53 ops.integrator('ParallelDisplacementControl', 4, 1, 0.1)
54 ops.analyze(10)
55
56 print('Node 4: ', [ops.nodeCoord(4), ops.nodeDisp(4)])
57 ops.stop()
```

## Parallel Tri31 Example

1. The source code is shown below, which can be downloaded [here](#).
2. Run the source code with 4 processors

```
mpiexec -np 4 python paralleltri31.py
```

the outputs look like

```
opensees.msg: TIME(sec) Real: 0.177647
opensees.msg: TIME(sec) Real: 0.187682
opensees.msg: TIME(sec) Real: 0.193193
```

(continues on next page)

(continued from previous page)

```

opensees.msg: TIME(sec) Real: 0.19473
opensees.msg: TIME(sec) Real: 14.4652
Node 4 [-0.16838893553441528, -2.88399389660282]
opensees.msg: TIME(sec) Real: 14.4618
opensees.msg: TIME(sec) Real: 14.4619
opensees.msg: TIME(sec) Real: 14.4948
Process 0 Terminating
Process 1 Terminating
Process 2 Terminating
Process 3 Terminating

```

The script is shown below

```

1 import openseespy.opensees as ops
2
3 pid = ops.getPID()
4 np = ops.getNP()
5 ops.start()
6
7 ops.model('basic', '-ndm', 2, '-ndf', 2)
8
9 L = 48.0
10 H = 4.0
11
12 Lp = L / np
13 ndf = 2
14 meshsize = 0.05
15
16 ops.node(pid, Lp * pid, 0.0)
17 ops.node(pid + 1, Lp * (pid + 1), 0.0)
18 ops.node(np + pid + 2, Lp * (pid + 1), H)
19 ops.node(np + pid + 1, Lp * pid, H)
20
21 sid = 1
22 ops.setStartNodeTag(2 * np + 2 + pid * int(H / meshsize + 10))
23 ops.mesh('line', 3, 2, pid, np + pid + 1, sid, ndf, meshsize)
24 ops.setStartNodeTag(2 * np + 2 + (pid + 1) * int(H / meshsize + 10))
25 ops.mesh('line', 4, 2, pid + 1, np + pid + 2, sid, ndf, meshsize)
26
27 ops.setStartNodeTag(int(2 * L / meshsize + (np + 1) * H / meshsize * 2) +
28                     pid * int(H * L / meshsize ** 2 * 2))
29 ops.mesh('line', 1, 2, pid, pid + 1, sid, ndf, meshsize)
30 ops.mesh('line', 2, 2, np + pid + 1, np + pid + 2, sid, ndf, meshsize)
31
32 ops.nDMaterial('ElasticIsotropic', 1, 3000.0, 0.3)
33
34 eleArgs = ['tri31', 1.0, 'PlaneStress', 1]
35
36 ops.mesh('quad', 5, 4, 1, 4, 2, 3, sid, ndf, meshsize, *eleArgs)
37
38

```

(continues on next page)

(continued from previous page)

```
39 if pid == 0:
40     ops.fix(pid, 1, 1)
41     ops.fix(np+pid+1, 1, 1)
42 if pid == np-1:
43     ops.timeSeries('Linear', 1)
44     ops.pattern('Plain', 1, 1)
45     ops.load(np + pid + 2, 0.0, -1.0)
46
47
48 ops.constraints('Transformation')
49 ops.numberer('ParallelPlain')
50 ops.system('Mumps')
51 ops.test('NormDispIncr', 1e-6, 6)
52 ops.algorithm('Newton')
53 ops.integrator('LoadControl', 1.0)
54 ops.analysis('Static')
55
56 ops.stop()
57 ops.start()
58 ops.analyze(1)
59
60 if pid == np-1:
61     print('Node', pid+1, ops.nodeDisp(pid+1))
62
63
64 ops.stop()
```

## Parallel Parametric Study Example

1. The source code is shown below, which can be downloaded [here](#).
2. Run the source code with 2 processors

```
mpiexec -np 2 python paralleltruss2.py
```

the outputs look like

```
Processor 0
Node 4 (E = 3000.0 ) Disp : [0.5300927771322836, -0.17789363846931766]
Processor 1

Node 4 (E = 6000.0 ) Disp : [0.2650463885661418, -0.08894681923465883]

Process 1 Terminating
Process 0 Terminating
```

The script is shown below

```
1 import openseespy.opensees as ops
2
3 pid = ops.getPID()
4 np = ops.getNP()
5 ops.start()
6 if np != 2:
```

(continues on next page)

(continued from previous page)

```

7     exit()
8
9 ops.model('basic', '-ndm', 2, '-ndf', 2)
10
11 if pid == 0:
12     E = 3000.0
13 else:
14     E = 6000.0
15
16 ops.uniaxialMaterial('Elastic', 1, E)
17
18 ops.node(1, 0.0, 0.0)
19 ops.node(2, 144.0, 0.0)
20 ops.node(3, 168.0, 0.0)
21 ops.node(4, 72.0, 96.0)
22
23 ops.fix(1, 1, 1)
24 ops.fix(2, 1, 1)
25 ops.fix(3, 1, 1)
26
27 ops.element('Truss', 1, 1, 4, 10.0, 1)
28 ops.timeSeries('Linear', 1)
29 ops.pattern('Plain', 1, 1)
30 ops.load(4, 100.0, -50.0)
31
32 ops.element('Truss', 2, 2, 4, 5.0, 1)
33 ops.element('Truss', 3, 3, 4, 5.0, 1)
34
35 ops.constraints('Transformation')
36 ops.numbererer('ParallelPlain')
37 ops.system('Umfpack')
38 ops.test('NormDispIncr', 1e-6, 6)
39 ops.algorithm('Newton')
40 ops.integrator('LoadControl', 0.1)
41 ops.analysis('Static')
42
43 ops.analyze(10)
44
45
46 if pid == 0:
47     print('Processor 0')
48     print('Node 4 (E =', E, ') Disp :', ops.nodeDisp(4))
49
50 ops.barrier()
51
52 if pid == 1:
53     print('Processor 1')
54     print('Node 4 (E =', E, ') Disp :', ops.nodeDisp(4))
55
56
57 ops.stop()

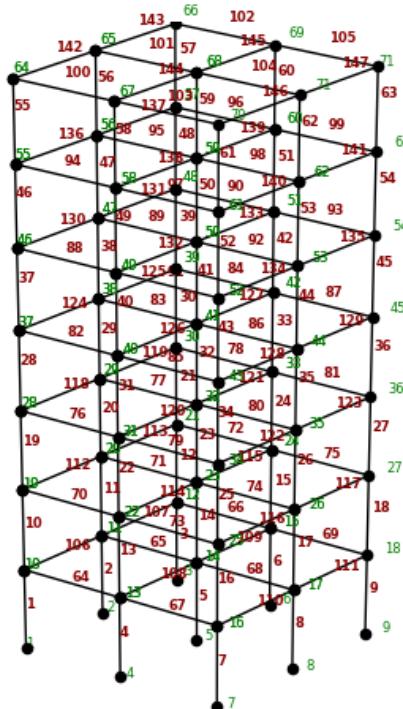
```

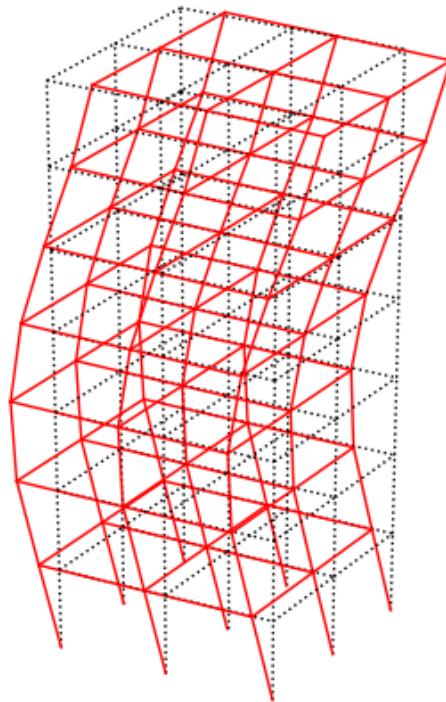
## 1.12.7 Plotting Examples

### 1. A Procedure to Render 2D or 3D OpenSees Model and Mode Shapes

## A Procedure to Render 2D or 3D OpenSees Model and Mode Shapes

1. The source code is developed by [Anurag Upadhyay](#) from University of Utah.
2. The source code can be downloaded [here](#).
3. Below is an example showing how to visualize an OpenSeesPy model.
4. Import by writing in the model file, “from openseespy.postprocessing.Get\_Rendering import \* “. (see line 11 in below example)
5. Plot the model by writing “plot\_model()” after defining all the nodes and elements. (see line 115 in below example)
6. Plot mode shapes by writing “plot\_modeshape(mode\_number)” after performing the eigen analysis. (see line 114 in below example)
7. Update openseespy to the latest version to get this function.





```

1 ######
2 ## 3D frame example to show how to render opensees model and
3 ## plot mode shapes
4 ##
5 ## By - Anurag Upadhyay, PhD Candidate, University of Utah.
6 ## Date - 12/01/2019
7 #####
8 #####
9
10 from openseespy.postprocessing.Get_Rendering import *
11 from openseespy.opensees import *
12
13 import numpy as np
14
15 from math import asin, sqrt
16
17 # set some properties
18 wipe()
19
20 model('Basic', '-ndm', 3, '-ndf', 6)
21
22 # properties
23 # units kip, ft
24
25 numBayX = 2

```

(continues on next page)

(continued from previous page)

```
26 numBayY = 2
27 numFloor = 7
28
29 bayWidthX = 120.0
30 bayWidthY = 120.0
31 storyHeights = [162.0, 162.0, 156.0, 156.0, 156.0, 156.0, 156.0, 156.0, 156.0, 156.0, 156.0, 156.0]
32
33 E = 29500.0
34 massX = 0.49
35 M = 0.
36 coordTransf = "Linear" # Linear, PDelta, Corotational
37 massType = "-lMass" # -lMass, -cMass
38
39 nodeTag = 1
40
41 # add the nodes
42 # - floor at a time
43 zLoc = 0.
44 for k in range(0, numFloor + 1):
45     xLoc = 0.
46     for i in range(0, numBayX + 1):
47         yLoc = 0.
48         for j in range(0, numBayY + 1):
49             node(nodeTag, xLoc, yLoc, zLoc)
50             mass(nodeTag, massX, massX, 0.01, 1.0e-10, 1.0e-10, 1.0e-10)
51             if k == 0:
52                 fix(nodeTag, 1, 1, 1, 1, 1, 1)
53
54             yLoc += bayWidthY
55             nodeTag += 1
56
57             xLoc += bayWidthX
58
59         if k < numFloor:
60             storyHeight = storyHeights[k]
61
62         zLoc += storyHeight
63
64 # add column element
65 geomTransf(coordTransf, 1, 1, 0, 0)
66 geomTransf(coordTransf, 2, 0, 0, 1)
67
68 eleTag = 1
69 nodeTag1 = 1
70
71 for k in range(0, numFloor):
72     for i in range(0, numBayX+1):
73         for j in range(0, numBayY+1):
74             nodeTag2 = nodeTag1 + (numBayX+1)*(numBayY+1)
75             iNode = nodeCoord(nodeTag1)
76             jNode = nodeCoord(nodeTag2)
77             element('elasticBeamColumn', eleTag, nodeTag1, nodeTag2, 50., E, 1000., 1000., 2150., 2150., 1, '-mass', M, massType)
78             eleTag += 1
79             nodeTag1 += 1
```

(continues on next page)

(continued from previous page)

```

81
82 nodeTag1 = 1+ (numBayX+1)* (numBayY+1)
83 #add beam elements
84 for j in range(1, numFloor + 1):
85     for i in range(0, numBayX):
86         for k in range(0, numBayY+1):
87             nodeTag2 = nodeTag1 + (numBayY+1)
88             iNode = nodeCoord(nodeTag1)
89             jNode = nodeCoord(nodeTag2)
90             element('elasticBeamColumn', eleTag, nodeTag1, nodeTag2, 50., ↴
91             ↪E, 1000., 1000., 2150., 2150., 2, '-mass', M, massType)
92             eleTag += 1
93             nodeTag1 += 1
94
95             nodeTag1 += (numBayY+1)
96
97 nodeTag1 = 1+ (numBayX+1)* (numBayY+1)
98 #add beam elements
99 for j in range(1, numFloor + 1):
100    for i in range(0, numBayY+1):
101        for k in range(0, numBayX):
102            nodeTag2 = nodeTag1 + 1
103            iNode = nodeCoord(nodeTag1)
104            jNode = nodeCoord(nodeTag2)
105            element('elasticBeamColumn', eleTag, nodeTag1, nodeTag2, 50., ↴
106             ↪E, 1000., 1000., 2150., 2150., 2, '-mass', M, massType)
107            eleTag += 1
108            nodeTag1 += 1
109
110 # calculate eigenvalues & print results
111 numEigen = 7
112 eigenValues = eigen(numEigen)
113 PI = 2 * asin(1.0)
114
115 #
116 plot_modeshape(5)
117 # plot_model()

```



---

## Index

---

### A

`addToParameter()` (*built-in function*), 233  
`algorithm()` (*built-in function*), 197  
`analysis()` (*built-in function*), 205  
`analyze()` (*built-in function*), 206

### B

`barrier()` (*built-in function*), 238  
`basicDeformation()` (*built-in function*), 208  
`basicForce()` (*built-in function*), 208  
`basicStiffness()` (*built-in function*), 208  
`Bcast()` (*built-in function*), 239  
`beamIntegration()` (*built-in function*), 100  
`block2D()` (*built-in function*), 98  
`block3D()` (*built-in function*), 99

### C

`computeGradients()` (*built-in function*), 234  
`constraints()` (*built-in function*), 187  
`convertBinaryToText()` (*built-in function*), 222  
`convertTextToBinary()` (*built-in function*), 222

### D

`database()` (*built-in function*), 222  
`domainChange()` (*built-in function*), 239

### E

`eigen()` (*built-in function*), 206  
`eleDynamicalForce()` (*built-in function*), 208  
`eleForce()` (*built-in function*), 208  
`eleLoad()` (*built-in function*), 94  
`element()` (*built-in function*), 28  
`eleNodes()` (*built-in function*), 209  
`eleResponse()` (*built-in function*), 209  
`equalDOF()` (*built-in function*), 89  
`equalDOF_Mixed()` (*built-in function*), 89

### F

`fiber()` (*built-in function*), 174

`fix()` (*built-in function*), 88  
`fixX()` (*built-in function*), 88  
`fixY()` (*built-in function*), 88  
`fixZ()` (*built-in function*), 88  
`frictionModel()` (*built-in function*), 182

### G

`geomTransf()` (*built-in function*), 184  
`getEleTags()` (*built-in function*), 209  
`getLoadFactor()` (*built-in function*), 209  
`getNodeTags()` (*built-in function*), 209  
`getNP()` (*built-in function*), 238  
`getNumThread()` (*built-in function*), 227  
`getParamTags()` (*built-in function*), 234  
`getParamValue()` (*built-in function*), 234  
`getPID()` (*built-in function*), 238  
`getTime()` (*built-in function*), 209  
`groundMotion()` (*built-in function*), 96

### I

`imposedMotion()` (*built-in function*), 97  
`InitialStateAnalysis()` (*built-in function*), 223  
`integrator()` (*built-in function*), 205

### L

`layer()` (*built-in function*), 176  
`load()` (*built-in function*), 94  
`loadConst()` (*built-in function*), 223  
`logFile()` (*built-in function*), 221

### M

`mass()` (*built-in function*), 97  
`mesh()` (*built-in function*), 228  
`modalDamping()` (*built-in function*), 223  
`model()` (*built-in function*), 28

### N

`nDMaterial()` (*built-in function*), 155  
`node()` (*built-in function*), 87

nodeAccel () (*built-in function*), 210  
nodeBounds () (*built-in function*), 210  
nodeCoord () (*built-in function*), 210  
nodeDisp () (*built-in function*), 210  
nodeDOFs () (*built-in function*), 211  
nodeEigenvector () (*built-in function*), 210  
nodeMass () (*built-in function*), 211  
nodePressure () (*built-in function*), 211  
nodeReaction () (*built-in function*), 211  
nodeResponse () (*built-in function*), 211  
nodeUnbalance () (*built-in function*), 212  
nodeVel () (*built-in function*), 212  
numberer () (*built-in function*), 188  
numFact () (*built-in function*), 212  
numIter () (*built-in function*), 212

## P

parameter () (*built-in function*), 233  
patch () (*built-in function*), 175  
pattern () (*built-in function*), 93  
plot\_model () (*built-in function*), 227  
plot\_modeshape () (*built-in function*), 227  
printA () (*built-in function*), 212  
printB () (*built-in function*), 213  
printGID () (*built-in function*), 213  
printModel () (*built-in function*), 213

## R

randomVariable () (*built-in function*), 236  
rayleigh () (*built-in function*), 98  
reactions () (*built-in function*), 223  
record () (*built-in function*), 214  
recorder () (*built-in function*), 214  
recv () (*built-in function*), 239  
region () (*built-in function*), 98  
remesh () (*built-in function*), 231  
remove () (*built-in function*), 223  
reset () (*built-in function*), 224  
restore () (*built-in function*), 224  
rigidDiaphragm () (*built-in function*), 90  
rigidLink () (*built-in function*), 90

## S

save () (*built-in function*), 224  
section () (*built-in function*), 173  
sectionDeformation () (*built-in function*), 219  
sectionFlexibility () (*built-in function*), 220  
sectionForce () (*built-in function*), 219  
sectionLocation () (*built-in function*), 220  
sectionStiffness () (*built-in function*), 219  
sectionWeight () (*built-in function*), 220  
send () (*built-in function*), 238  
sensitivityAlgorithm () (*built-in function*), 235  
sensLambda () (*built-in function*), 235

sensNodeAccel () (*built-in function*), 235  
sensNodeDisp () (*built-in function*), 235  
sensNodePressure () (*built-in function*), 236  
sensNodeVel () (*built-in function*), 235  
sensSectionForce () (*built-in function*), 236  
setElementRayleighDampingFactors ()  
    (*built-in function*), 226  
setNodeAccel () (*built-in function*), 225  
setNodeCoord () (*built-in function*), 225  
setNodeDisp () (*built-in function*), 225  
setNodeVel () (*built-in function*), 225  
setNumThread () (*built-in function*), 227  
setParameter () (*built-in function*), 234  
setPrecision () (*built-in function*), 226  
setStartNodeTag () (*built-in function*), 239  
setTime () (*built-in function*), 225  
sp () (*built-in function*), 95  
start () (*built-in function*), 226  
stop () (*built-in function*), 226  
stripXML () (*built-in function*), 226  
system () (*built-in function*), 190  
systemSize () (*built-in function*), 220

## T

test () (*built-in function*), 192  
testIter () (*built-in function*), 220  
testNorm () (*built-in function*), 220  
timeSeries () (*built-in function*), 90

## U

uniaxialMaterial () (*built-in function*), 106  
updateElementDomain () (*built-in function*), 226  
updateParameter () (*built-in function*), 234

## V

version () (*built-in function*), 221

## W

wipe () (*built-in function*), 227  
wipeAnalysis () (*built-in function*), 227