OpenSeesPy Documentation

Release 3.2.2.9

Minjie Zhu

Mar 24, 2021

Contents

Important: Version 3.2.2.9 is released!

OpenSeesPy is on *PyPi (Windows, Linux, Mac)*.

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

Note: Questions including modeling issues and the use of OpenSeesPy, please post on OpenSeesPy Forum.

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

For errors in this document, submit on github issues.

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

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

# import OpenSeesPy plotting commands
import openseespy.postprocessing.Get_Rendering as opsplt

# wipe model
ops.wipe()

# create model
ops.model('basic', '-ndm', 2, '-ndf', 3)

# plot model
opsplt.plot_model()
```

To run a test of the pip installation:

```
pytest --pyargs openseespy.test
```

CHAPTER 1

Developer

Minjie Zhu

Research Associate
Civil and Construction Engineering
Oregon State University

1.1 Installation

1. *PyPi (Windows, Linux, Mac)*
2. *Other Instalation*

1.1.1 PyPi (Windows, Linux, Mac)

1. *PyPi (Windows)*
2. *PyPi (Linux)*
3. *PyPi (Mac)*

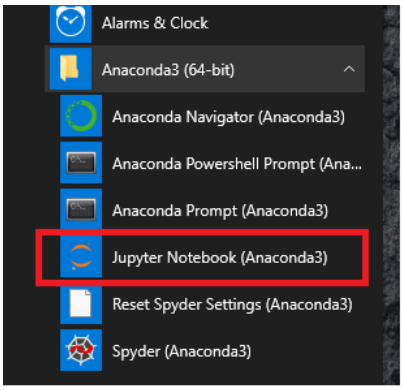
PyPi (Windows)

Install Anaconda

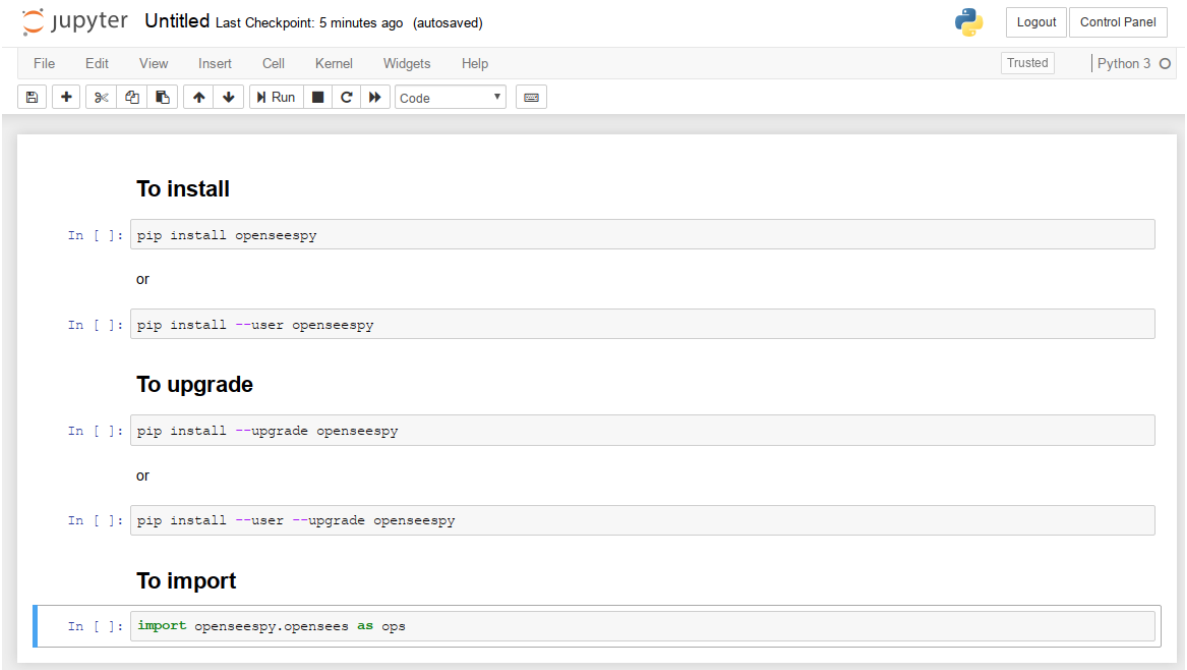
- Install Anaconda

Install In Jupyter Notebook

- Start Jupyter Notebook

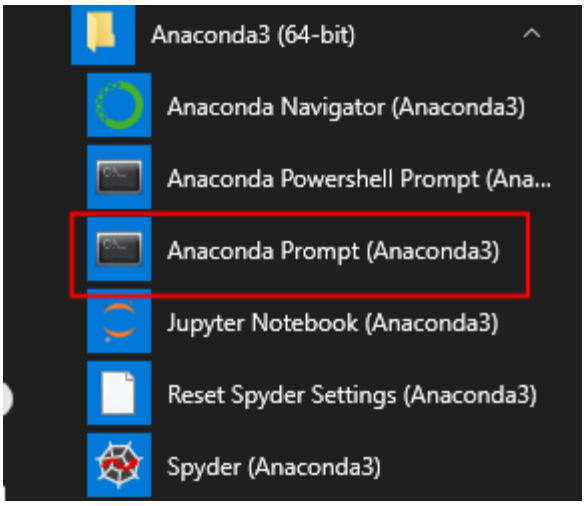


- run install command in the notebook



Install In command line

- Start Anaconda Prompt



- 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
```

- To import

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

PyPi (Linux)

Install Python

- Install Python 3 using your Linux's package manager
  - Ubuntu/Debian: `sudo apt install python3 python3-pip`
  - Centos/Redhat: `sudo yum install python3 python3-pip`

Install in terminal

- To install

```
python3 -m pip install openseespy

python3 -m pip install --user openseespy
```

- To upgrade

```
python3 -m pip install --upgrade openseespy

python3 -m pip install --user --upgrade openseespy
```

- To import

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

PyPi (Mac)

Install Dependencies

- Install MacPorts for your macOs.
- Install gcc8

```
sudo port install gcc8
```

Install Python

- Install Python 3.8 using MacPorts

```
sudo port install python38 py38-pip
```

Install in terminal

- To install

```
python3.8 -m pip install openseespy

python3.8 -m pip install --user openseespy
```

- To upgrade

```
python3.8 -m pip install --upgrade openseespy

python3.8 -m pip install --user --upgrade openseespy
```

- To import

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

1.1.2 Other Instalation

1. *DesignSafe (Web-based)*
2. *Windows Subsystem for Linux (Windows)*

DesignSafe (Web-based)

OpenSeesPy has been offical in DesignSafe.

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

OpenSeesPy in DesignSafe

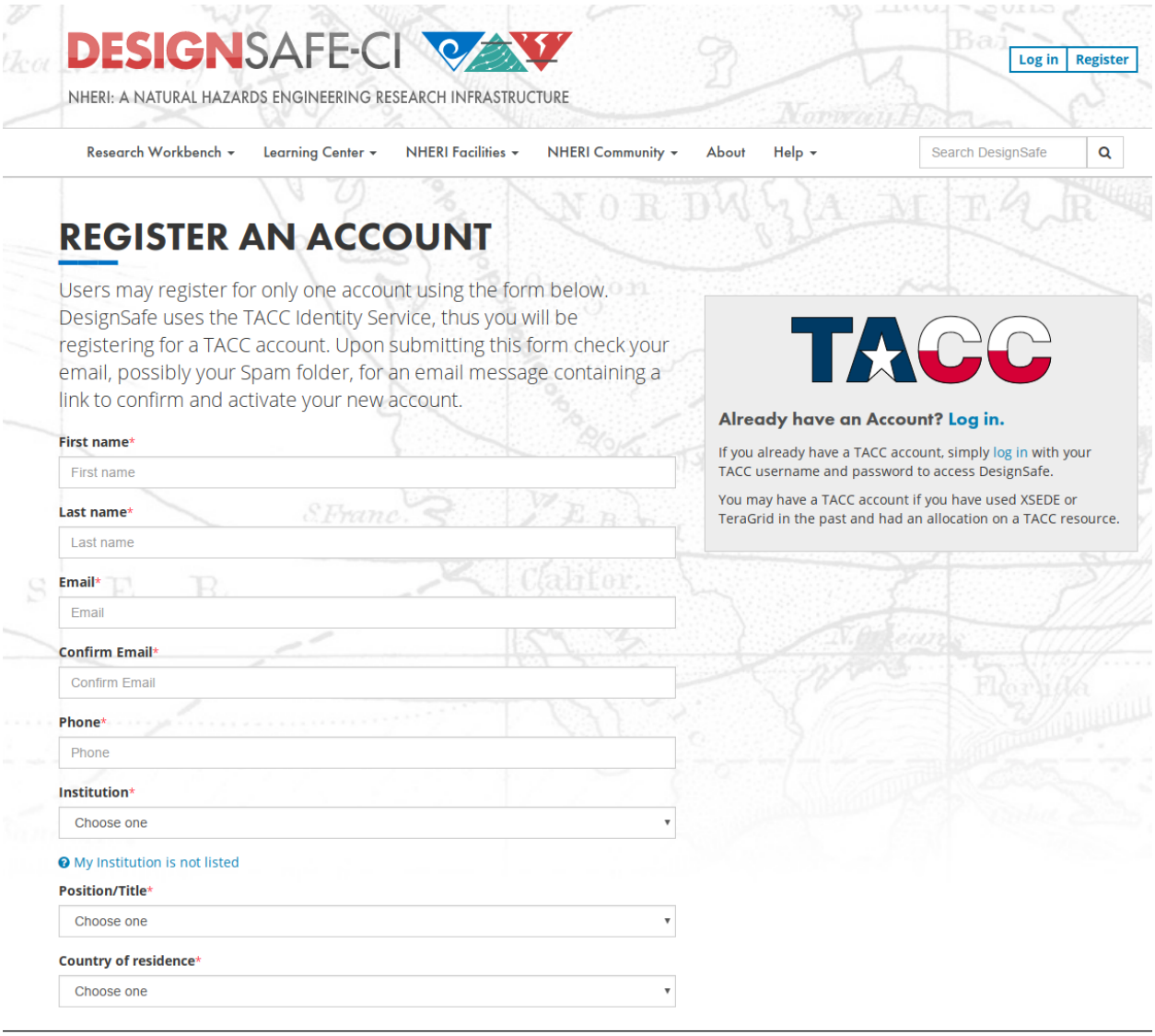
Follow steps below to run OpenSeesPy in DesignSafe.

Tip:

- Go To DesignSafe website and click register

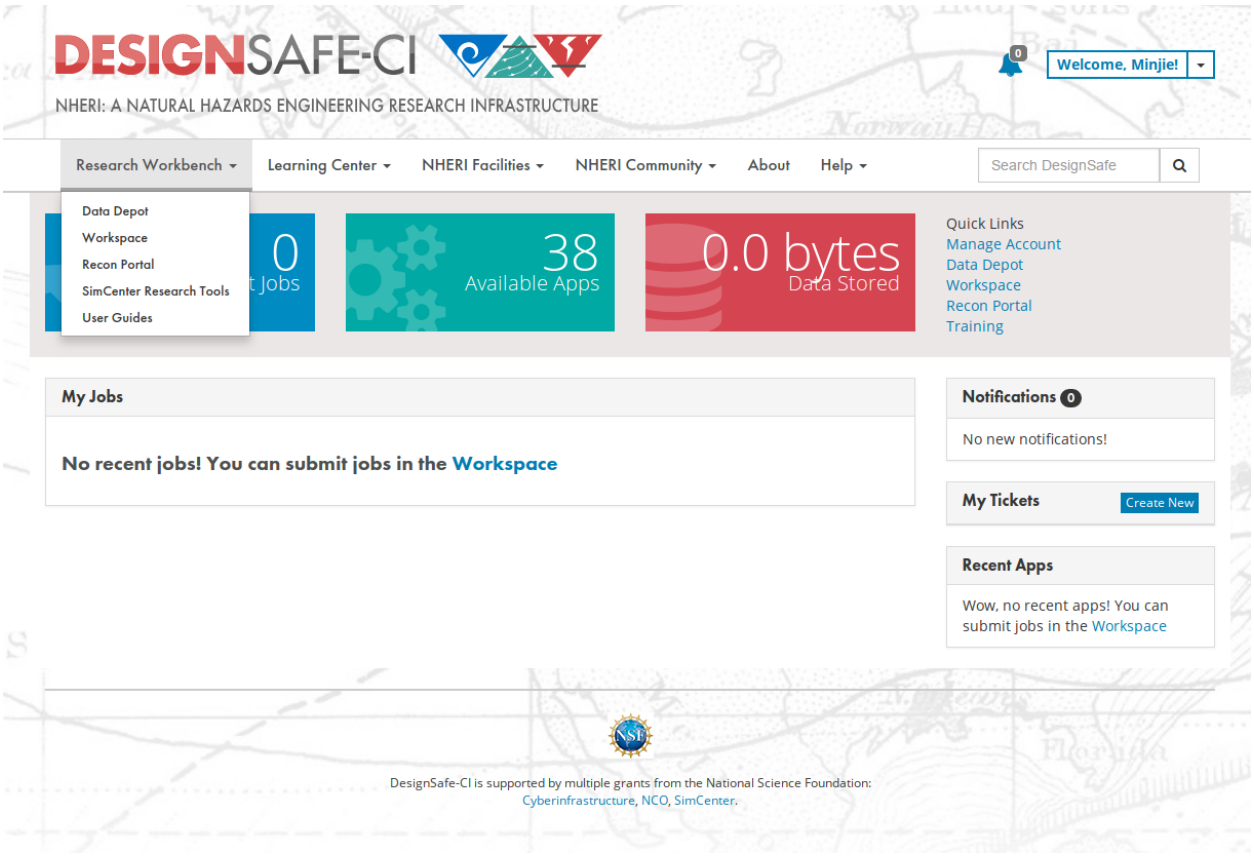
Tip:

- Register an account for free and log in



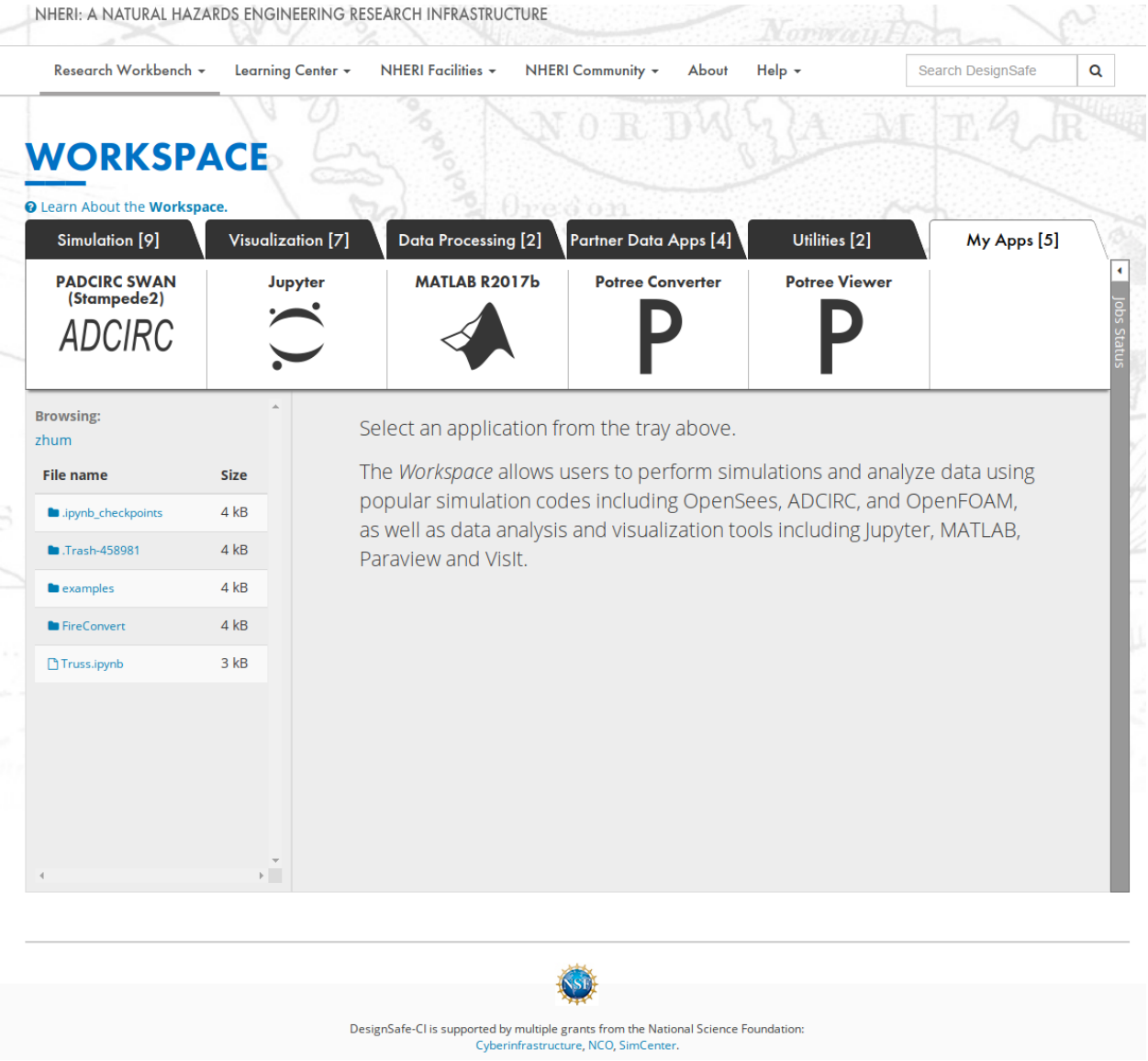
Tip:

- Land on your own portal and go to workspace



Tip:

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



Tip:

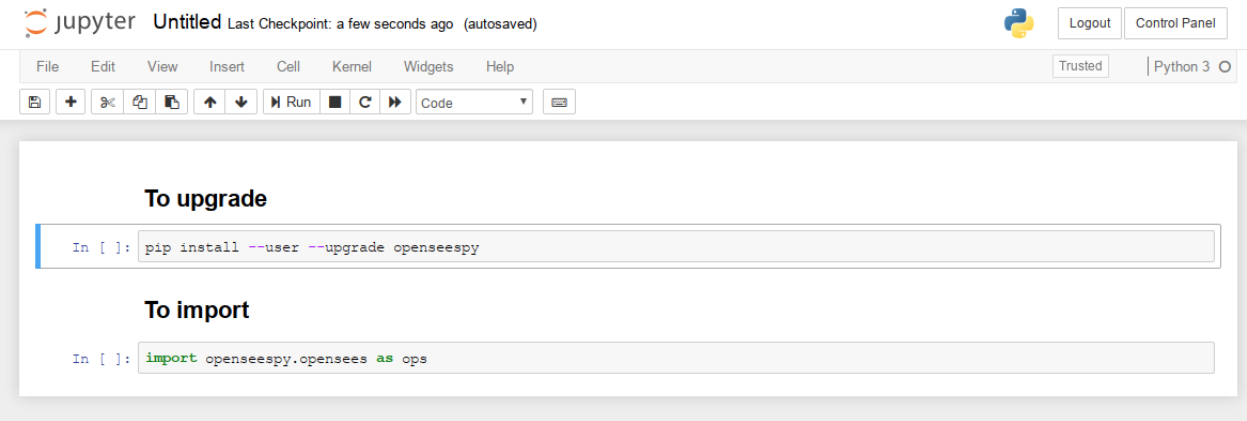
- Now you should be in the `Jupyter` Notebook
- Go to `mydata` and alwasy save your data under this folder

Tip:

- In the `mydata` folder, select `New` and then `Python 3`

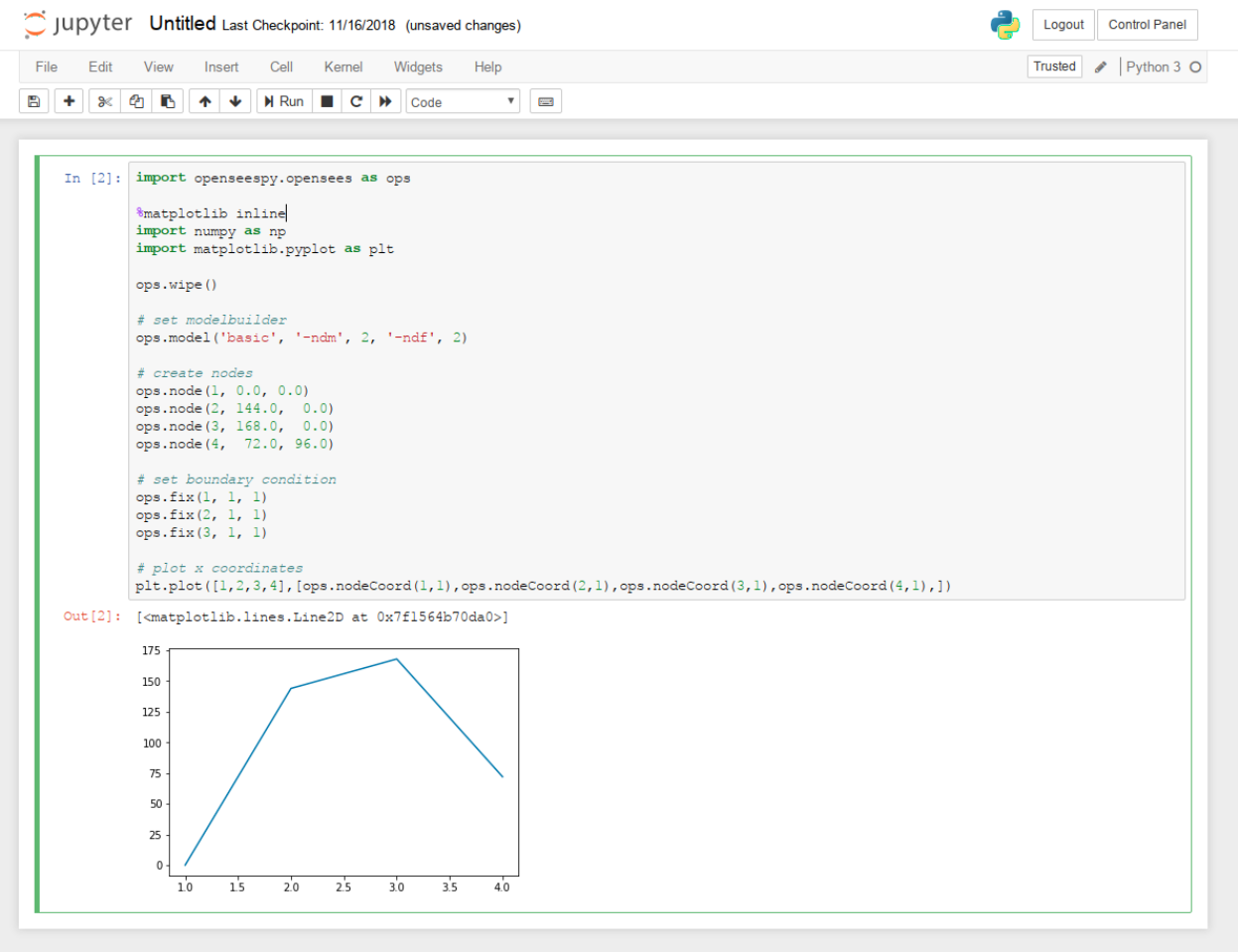
Tip:

- The OpenSeesPy version on DesignSafe is not the latest.
- To update to the latest and import:



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



Paraview in DesignSafe

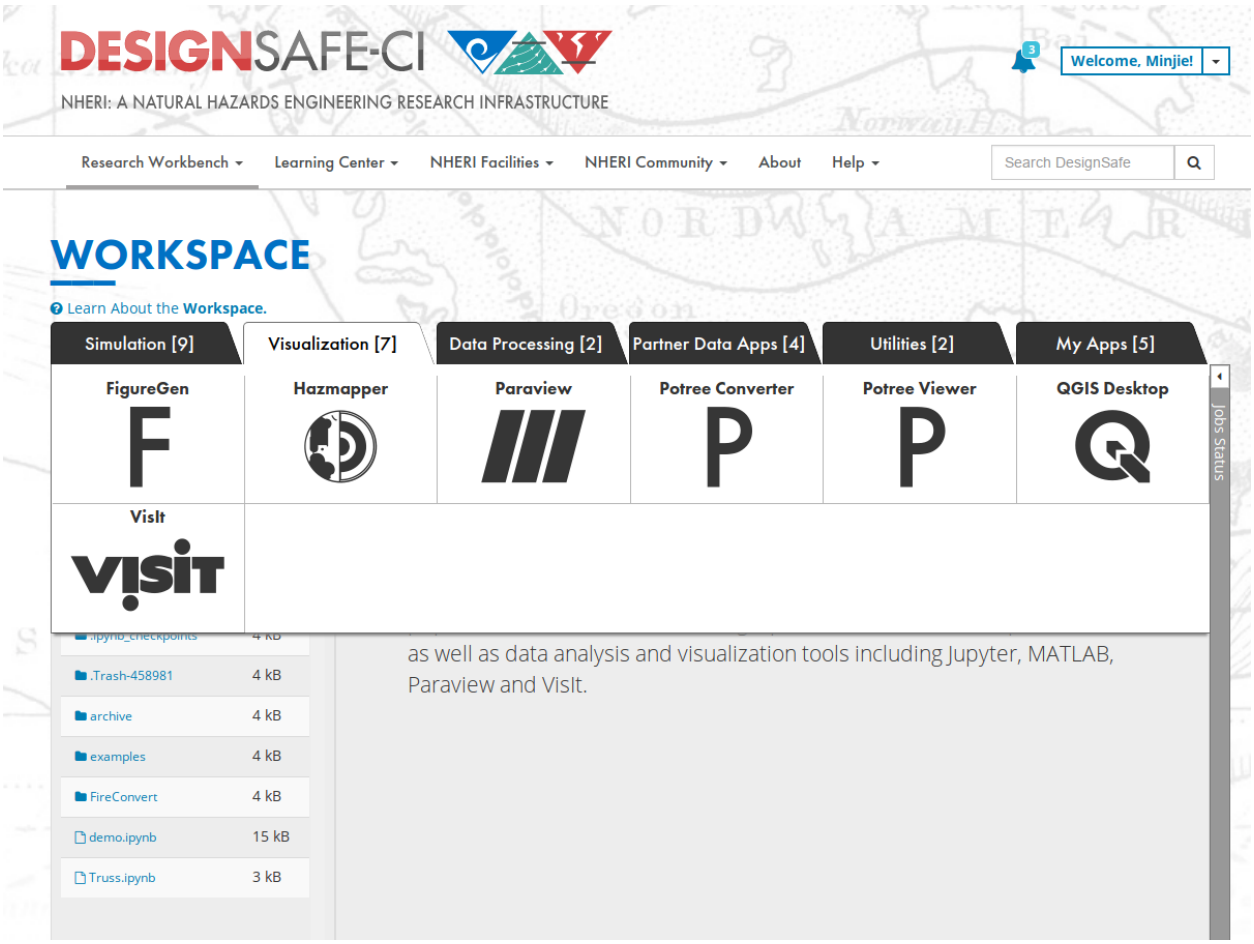
DesignSafe provides paraview for viewing OpenSeesPy results:

Tip:

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

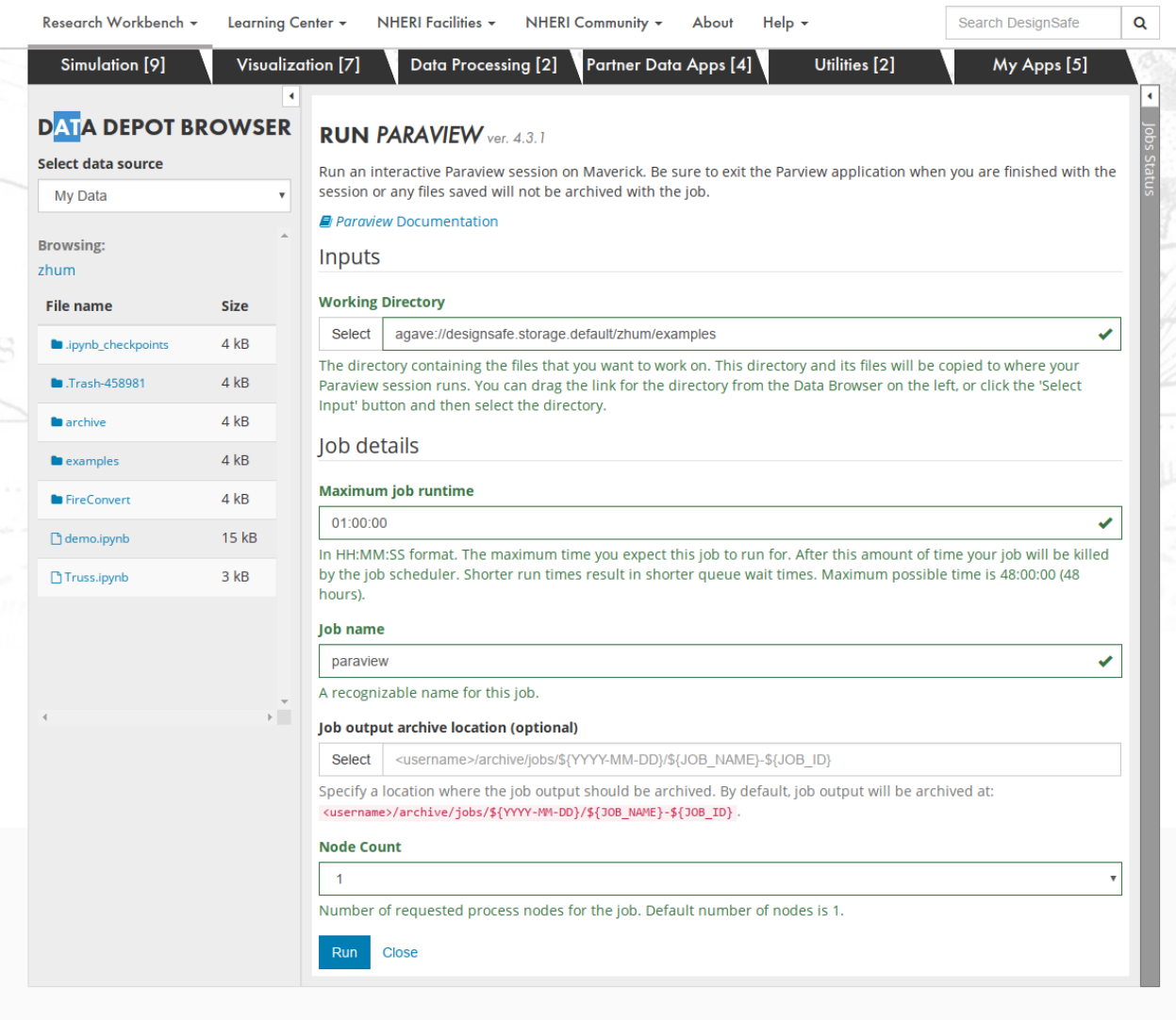
Tip:

- Go to `Workspace` and select `Visualizaion` and `Paraview`.



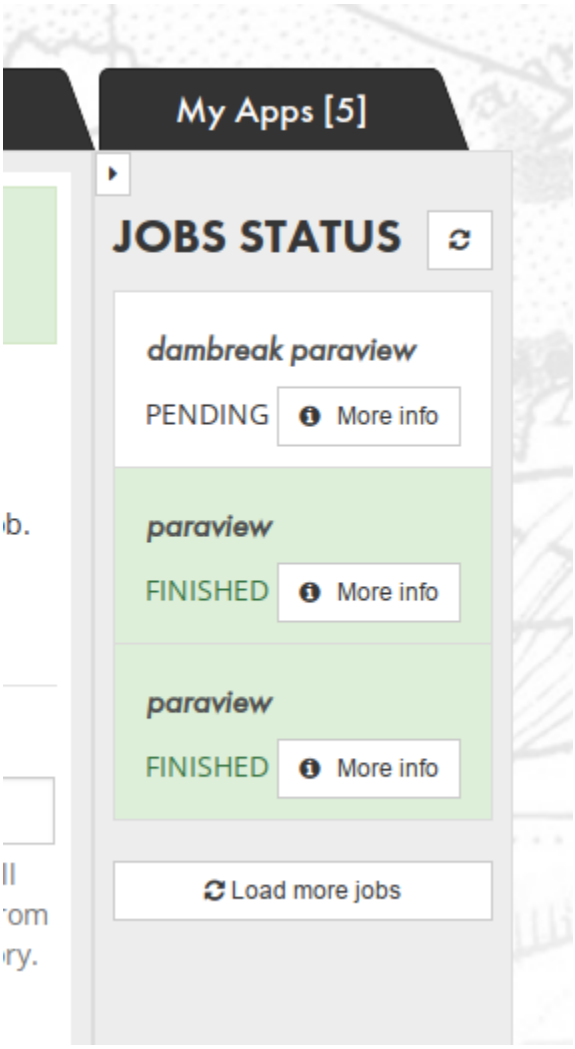
Tip:

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



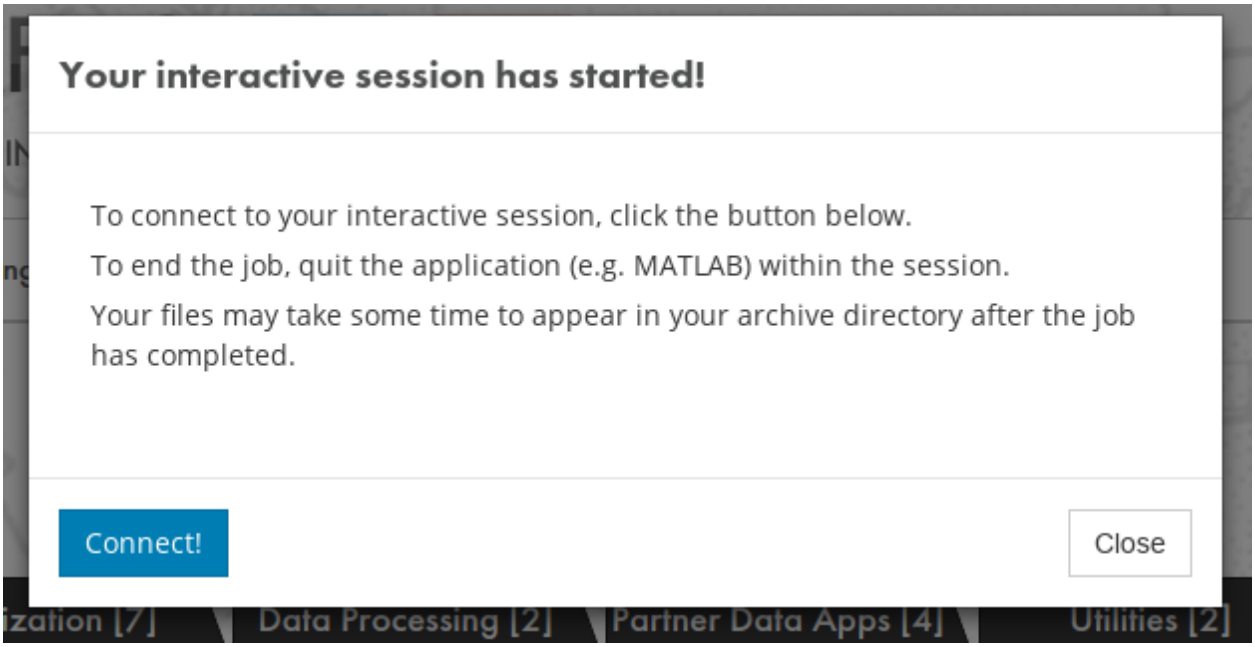
Tip:

- You can see the job status on the right



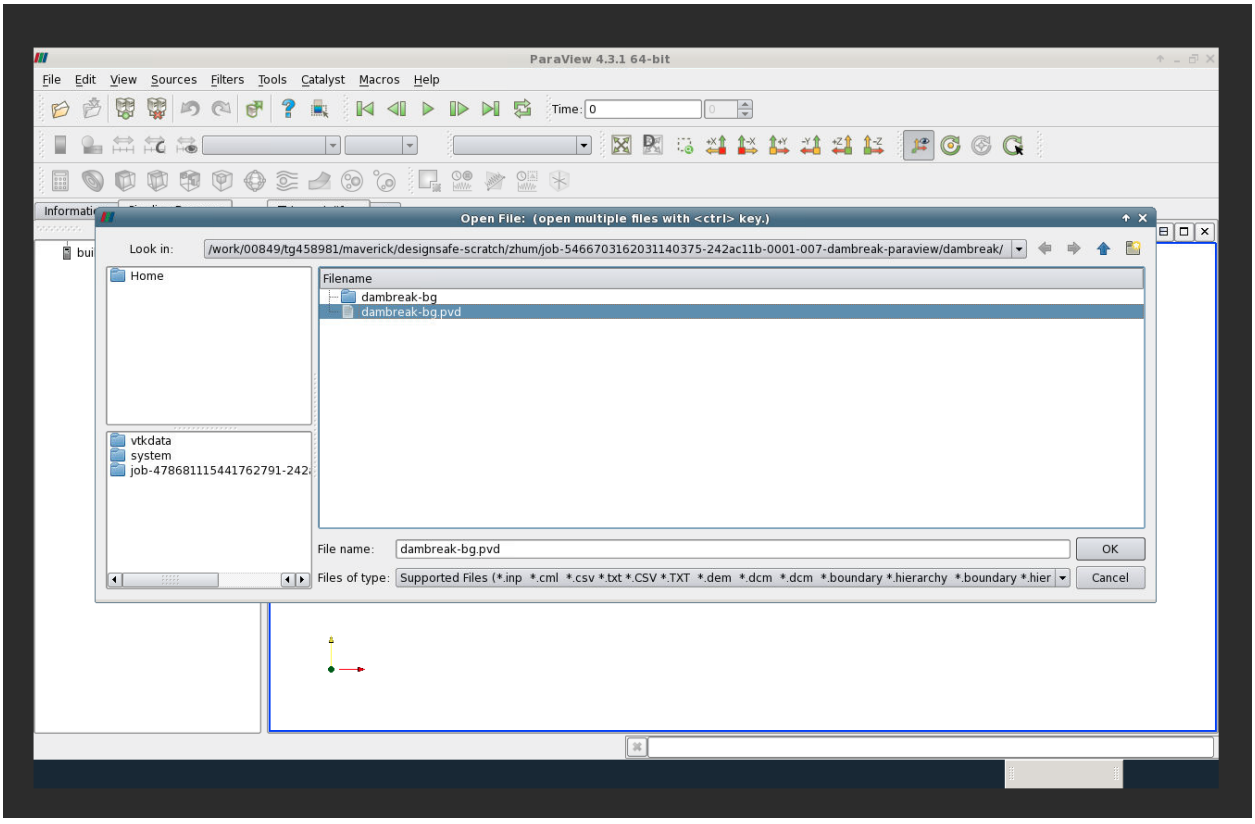
Tip:

- Wait until see this windows and connect to Paraview



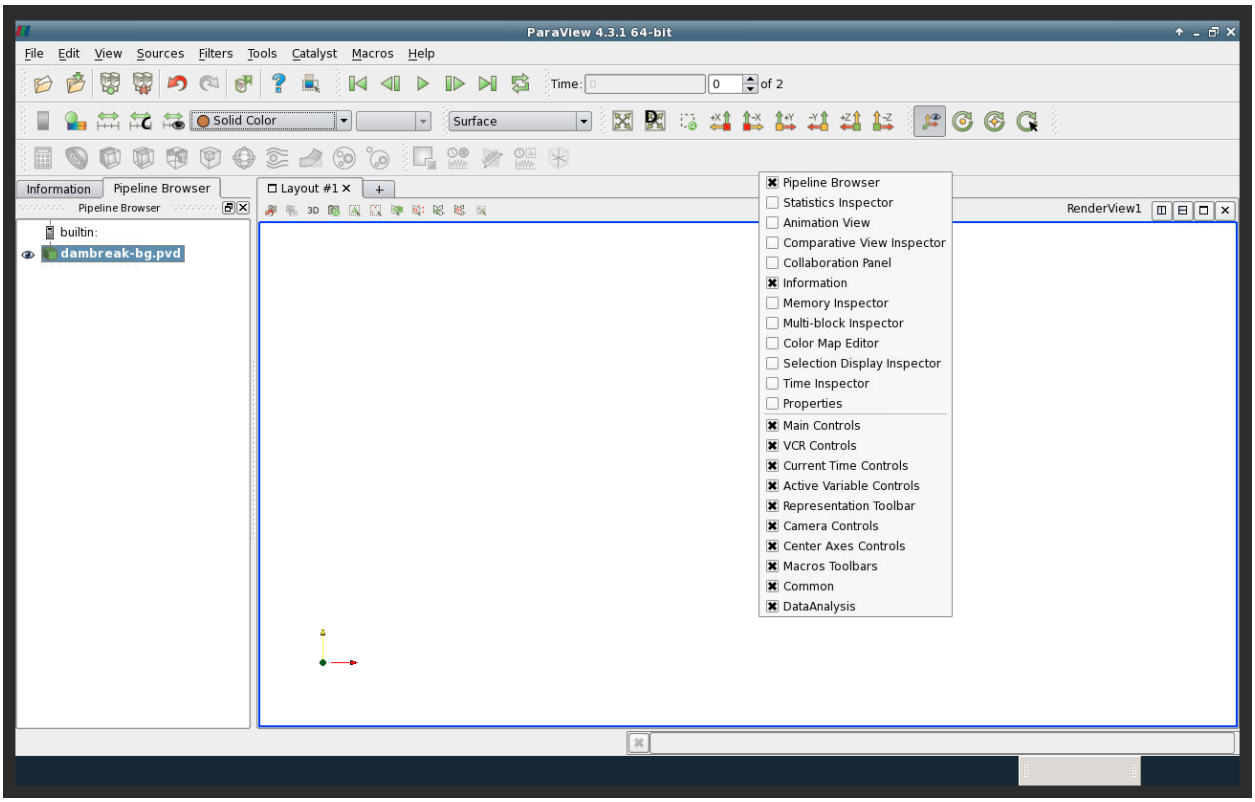
Tip:

- Now open the pvd file in Paraview



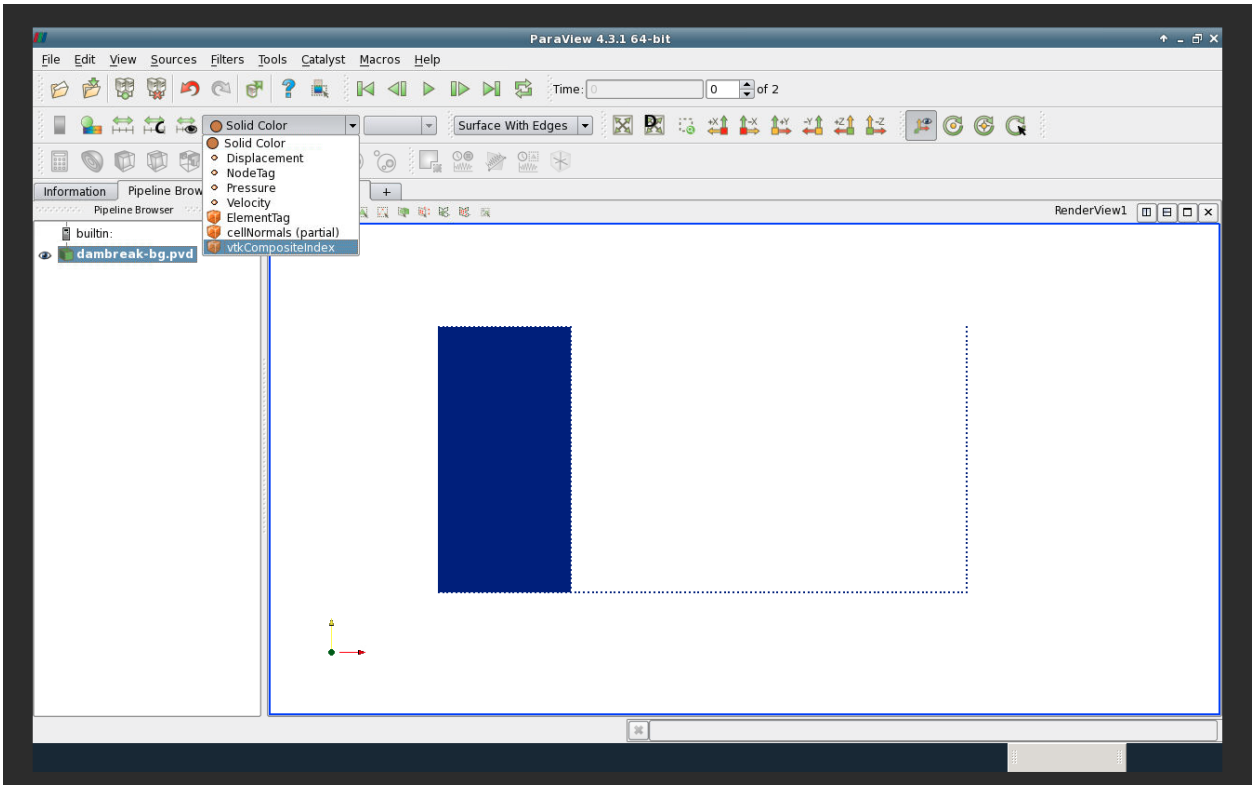
Tip:

- 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`

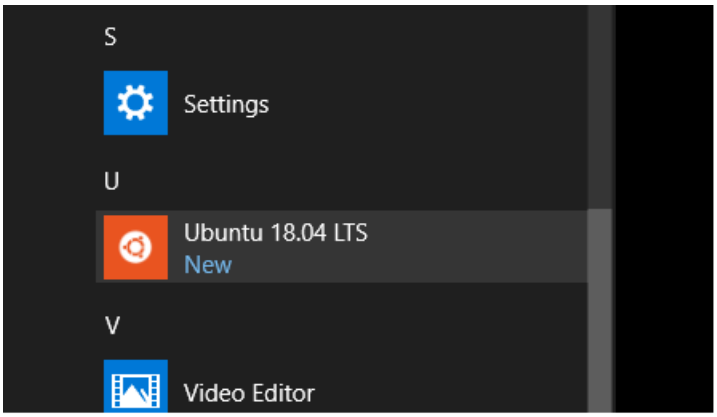


Windows Subsystem for Linux (Windows)

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

Install the Windows Subsystem 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.



Install Anaconda and start Jupyter Notebook

- 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
```

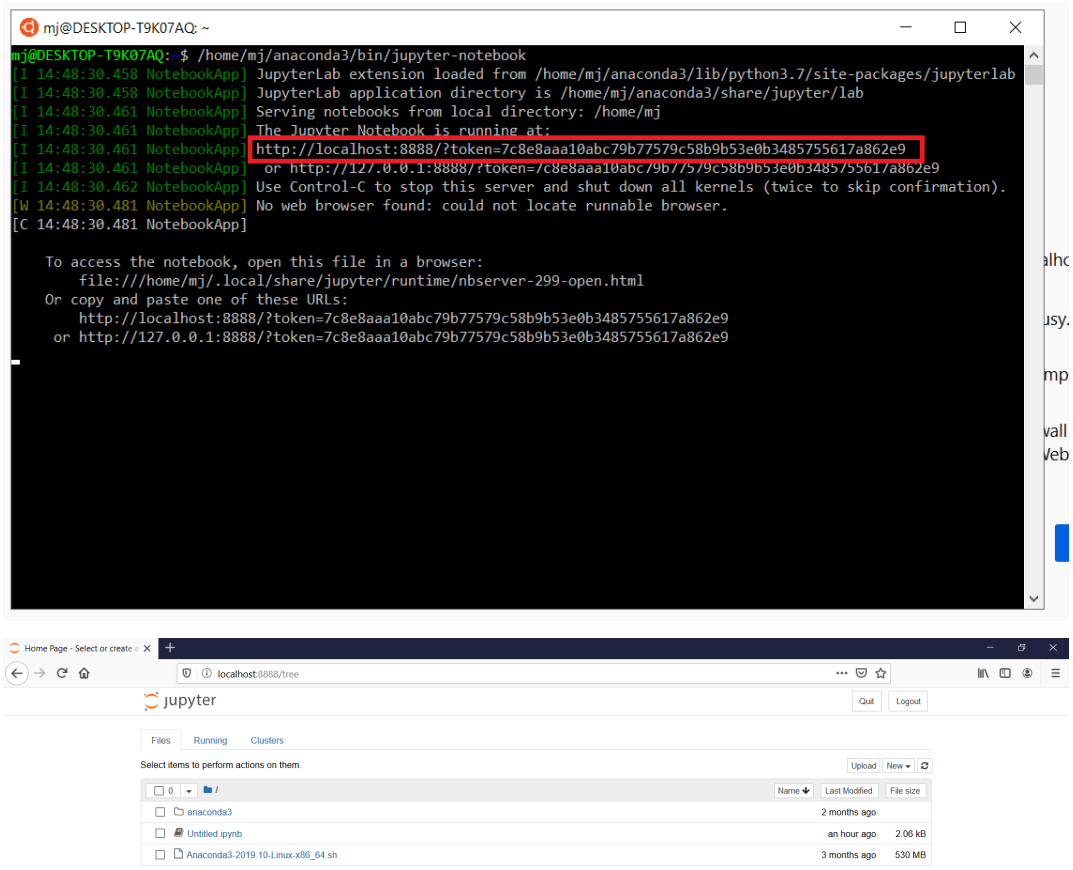
- Install Anconda Linux version with commands

```
~$ 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)
```

- Start Jupyter Notebook

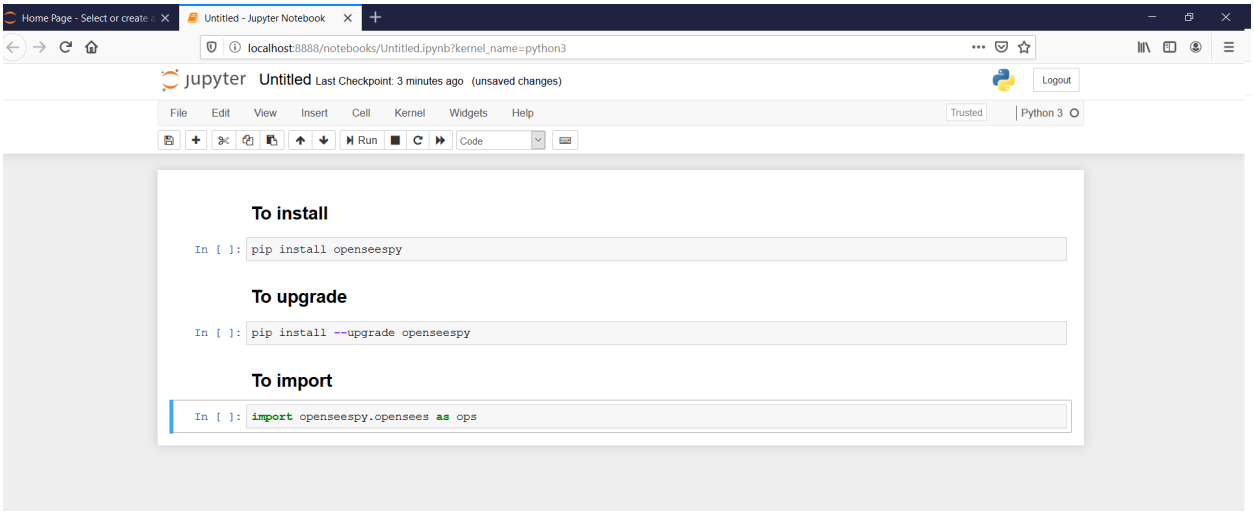
```
~$ /home/username/anaconda3/bin/jupyter-notebook
```

- Copy the address in red box to a web browser



In Jupyter Notebook

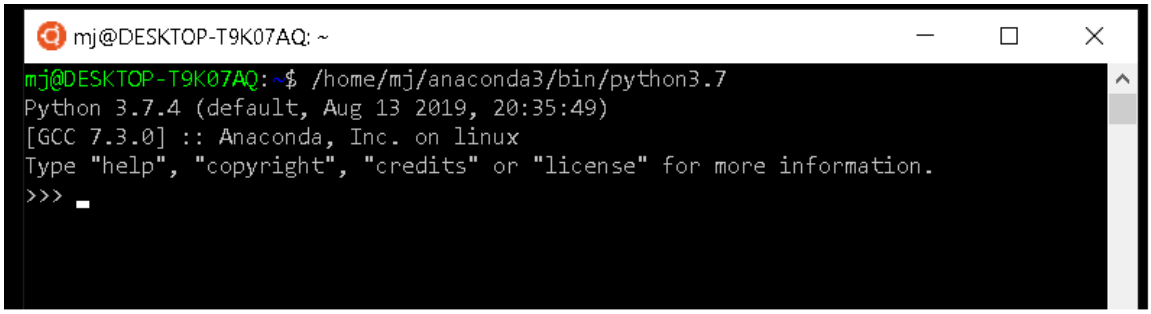
Start a new notebook and then



In the command line (optional)

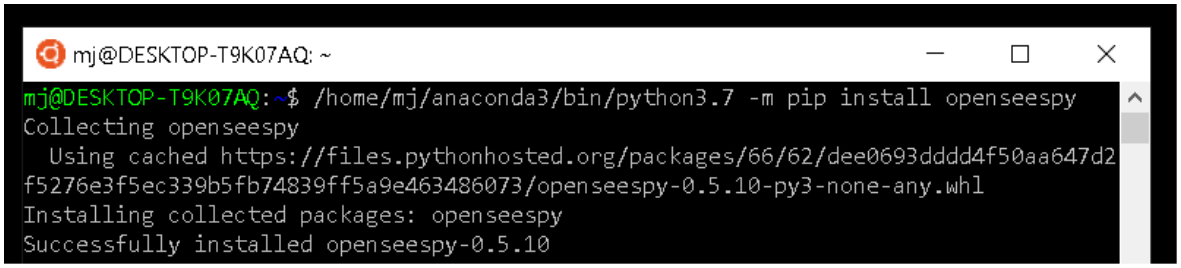
- Run Anaconda with following command, where *username* is your username of your computer. Please use the *username* shown in last step

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



- 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
```



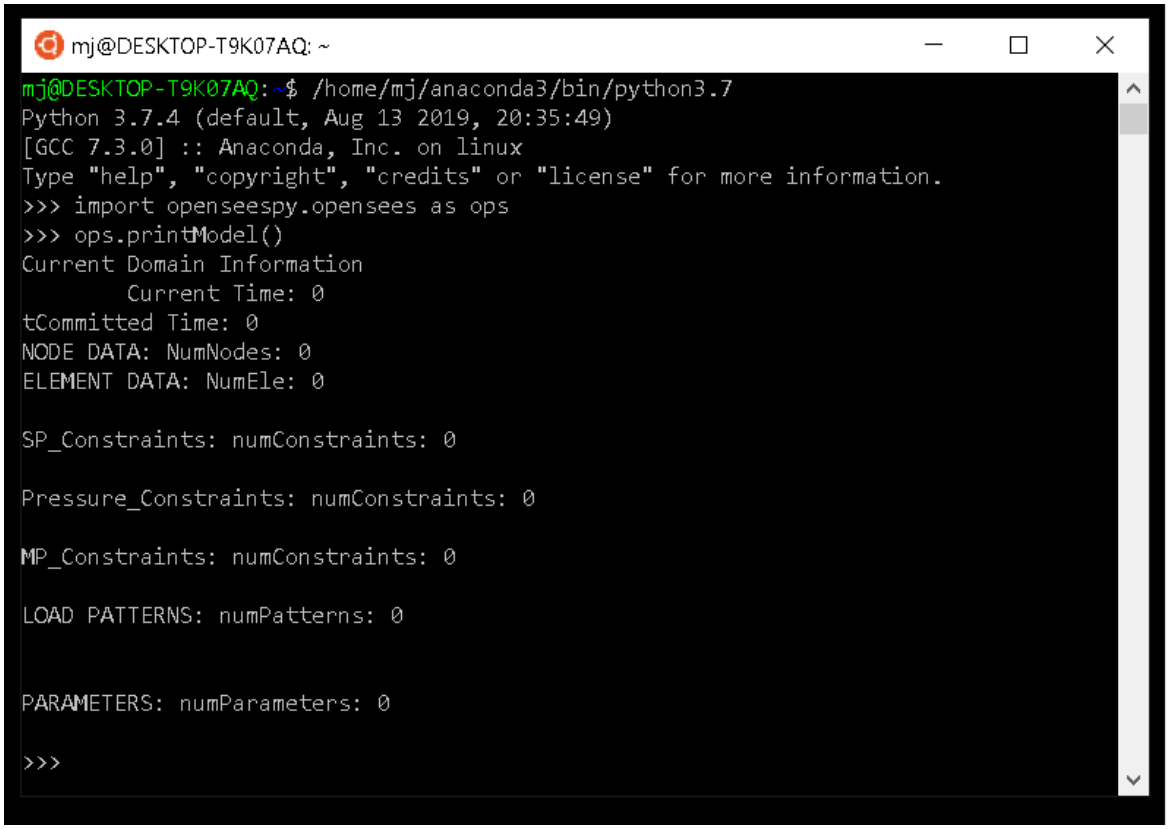
- Run OpenSeesPy

  First run Anaconda with

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

  Then import OpenSeesPy with

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



- run OpenSeesPy scripts

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

1.2 Compilation

1. *Compilation for Windows*
2. *Compilation for MacOS*
3. *Compilation using QT*

1.2.1 Compilation for Windows

1.2.2 Compilation for MacOS

This is a suggestive guide for MacOS.

- Download OpenSees Source code

```
git clone https://github.com/OpenSees/OpenSees.git
```

- Install MacPorts for your macOs.
- Install gcc8

```
sudo port install gcc8
```

- Install Python 3.8 using MacPorts

```
sudo port install python38 python38-devel
```

- Install boost using Macports

```
sudo port install boost
```

- Create Makefile.def
  - Copy one of `MAKES/Makefile.def.MacOS10.x` to the root and rename to `Makefile.def`
  - Change the path and variables for your system
- Compile
  - Run `make python -j` from root
  - The library file will be at `SRC/interpreter/opensees.so`

1.2.3 Compilation using QT

Following is the OpenSees Compilation using QT by

Stevan Gavrilovic **github**

PhD Candidate
University of British Columbia

Original instructions can be found at here.

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.

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.

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!

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

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.

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
```

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/)

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.

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)

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

SUPERLUDIST

SUPERLU but for for for distributed memory parallel machines. Supports manycore heterogeous 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)

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'

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'

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)

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
```

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
```

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 3.2.2.9** (1/28/2021)
  - The pip installation will only install needed libraries based on the Operating System, i.e. Windows, Linux, or Mac. The installation time and size are now one third of before.
  - Bug fixes for Pinching4Material, Concrete07, H5DRM, RCCircularSectionIntegration, ResponseSpectrumAnalysis, PML, FiberSection2d, DriftRecorder,
- **Version 3.2.2.8** (1/8/2021)

  - Linux version is tested with Centos 7, 8, Ubuntu 18.04, 20.04, Fedora, and Debian.
  - Mac version uses MacPorts for installing Python and dependencies.
  - Bug fixes for recorders, FourNodeTetrahedron, ASD_SMA_3K, nodeMass,
  - Add ExpressNewton, RockingBC, CBDI3d, Concrete02IS
  - Update PETSc Solver, ZeroLengthSection, ForceBeamColumn3d, OOHystereticMaterial, SSPbrickUP, HardeningMaterial, BilinearOilDamper,

- **Version 3.2.2.6** (10/15/2020)
  - OpenSeesPy is available now on Mac, just type *import openseespy.opensees as ops* on the MacOS. Python3.8 is required and HomeBrew Python is strongly recommended.
  - SixNodeTri element by Seweryn
  - PostProcessing package ops_vis by Seweryn
- **Version 3.2.2.5** (9/16/2020)
  - Fix a Windows issue for virtual environment
- **Version 3.2.2.4** (9/10/2020)
  - Bug fixes in Truss and ForceBeamColumn2d
  - Adding *ops.__version__* variable
  - Bug fixes in 3D elastic beam
  - Bug fixes in Tri31
  - Adding *ops_vis* module for plotting
  - OpenSees commit b0f6b06
- **Version 3.2.2.3** (8/11/2020)
  - Fix typos in documentation
  - Add *testNorm* and *testIter* commands
  - Add Python3.7 and Python 3.8 support
  - Support latest Anaconda
  - Improvements of ploting commands
  - *ShellDKGT* command
  - Include OpenSees commits upto 380239c on 8/9
- **Version 3.2.2.1** (5/18/2020)
  - add gimmeMCK integrator
- **Version 3.2.2** (5/8/2020)
  - Fix Get_Renderiing tab problem
  - Ship with dependent libraries for more Linux systems
- **Version 3.2.0** (4.17.2020)
  - Add background mesh command
  - Add partition command
  - Add OpenSeesPy test

  - Many bug fixes

- **Version 3.1.5.11** (1.10.2020)
  - Change versioning method. First two digits match the current OpenSees framework version. The last two digits are the versions for OpenSeesPy.
  - For Windows, only support the Python version that corresponds to the current version of Anaconda.
  - Add openseespy.postprocessing.Get_Rendering
  - Add '-init' option to Newmark integrator
  - Some function can return empty or one-element lists
  - Spaces in string input will be automatically removed
  - Bug fixes
- **Version 0.5.4**
  - Support Mac
  - Support Python3.7 on Windows and Linux
- **Version 0.5.3**
  - Fix bug in LimitState UniaxialMaterial
  - 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 compatability
  - Add "updateMaterialStage" function
  - Add "RCCircular" seciton
  - 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` (int)	number of dimensions (1,2,3)
`ndf` (int)	number of dofs (optional)

1.4.2 element commands

element (*eleType*, *eleTag*, **eleNodes*, **eleArgs*)
Create a OpenSees element.

`eleType` (str)	element type
`eleTag` (int)	element tag.
`eleNodes` (list (int))	a list of element nodes, must be preceded with `*`.
`eleArgs` (list)	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.

`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 UniaxialMaterials
`dirs` (list (int))	a list of material directions: • **1,2,3 - translation along local x,y,z axes, respectively;** – 4,5,6 - rotation about local x,y,z axes, respectively
`rFlag` (float)	optional, default = 0 * `rFlag` = 0 NO RAYLEIGH DAMPING (default)
`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` = 1 include rayleigh damping

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 exes.
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*, *secTag*, *<'-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
`secTag` (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 • `rFlag` = 0 NO RAYLEIGH DAMPING (default) • `rFlag` = 1 include rayleigh damping

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
`dirn1 dirn2` (int)	the two directions, 1 through ndof.
`rFlag` (float)	optional, default = 0 • `rFlag` = 0 NO RAYLEIGH DAMPING (default) • `rFlag` = 1 include rayleigh damping

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 constrained and a retained 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 constrained and a retained 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: • 1 Out normal of the master plane pointing to +X direction • 2 Out normal of the master plane pointing to +Y direction • 3 Out normal of the master plane pointing to +Z direction

See also:

Notes

zeroLengthContactNTS2D

element (*'zeroLengthContactNTS2D'*, *eleTag*, *'-sNdNum'*, *sNdNum*, *'-mNdNum'*, *mNdNum*, *'-Nodes'*, **NodesTags*, *kn*, *kt*, *phi*)

eleTag (int)	unique element object tag
sNdNum (int)	Number of Slave Nodes
mNdNum (int)	Number of Master nodes
NodesTags (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 ϕ 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'*, **NodesTags*, *kn*, *kt*, *phi*)

eleTag (int)	unique element object tag
sNdNum (int)	Number of Slave Nodes
mNdNum (int)	Number of Master nodes
sdof, mdof (int)	Slave and Master degree of freedom
NodesTags (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 ϕ 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` (int)	unique element object tag
`eleNodes` (list (int))	a list of a constrained and a retained nodes
`direction` (int)	• `1` if out-normal vector of master plane points to +X direction • `2` if out-normal vector of master plane points to +Y direction • `3` if out-normal vector of master plane points to +Z direction
`initGap` (float)	Initial gap between master plane and slave plane
`frictionRatio` (float)	Friction ratio in two tangential directions (parallel to master and slave planes)
`Kt` (float)	Penalty in two tangential directions
`Kn` (float)	Penalty in normal direction (normal to master and slave planes)
`Kn2` (float)	Penalty in normal direction after yielding based on Hertz impact model
`Delta_y` (float)	Yield deformation based on Hertz impact model
`cohesion` (float)	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 • `cFlag` = 0 lumped mass matrix (default) • `cFlag` = 1 consistent mass matrix
`rFlag` (float)	Rayleigh damping flag, optional, default = 0 • `rFlag` = 0 NO RAYLEIGH DAMPING (default) • `rFlag` = 1 include Rayleigh damping

---

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` (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 • `cFlag` = 0 lumped mass matrix (default) • `cFlag` = 1 consistent mass matrix
`rFlag` (float)	Rayleigh damping flag, optional, default = 0 • `rFlag` = 0 NO RAYLEIGH DAMPING (default) • `rFlag` = 1 include Rayleigh damping

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*, *Area*, *E_mod*, *Iz*, *transfTag*, *<'-mass'*, *mass>*, *<'-cMass'>*, *<'-release'*, *releaseCode>*)
For a two-dimensional problem

element(*'elasticBeamColumn'*, *eleTag*, **eleNodes*, *Area*, *E_mod*, *G_mod*, *Jxx*, *Iy*, *Iz*, *transfTag*, *<'-mass'*, *mass>*, *<'-cMass'>*)
For a three-dimensional problem

`eleTag` (int)	unique element object tag
`eleNodes` (list (int))	a list of two element nodes
`Area` (float)	cross-sectional area of element
`E_mod` (float)	Young's Modulus
`G_mod` (float)	Shear Modulus
`Jxx` (float)	torsional moment of inertia of cross section
`Iz` (float)	second moment of area about the local z-axis
`Iy` (float)	second moment of area about the local y-axis
`transfTag` (int)	identifier for previously-defined coordinate-transformation (CrdTransf) object
`mass` (float)	element mass per unit length (optional, default = 0.0)
`'-cMass'` (str)	to form consistent mass matrix (optional, default = lumped mass matrix)
`'releaseCode'` (int)	moment release (optional, 2d only, 0=no release (default), 1=release at I, 2=release at J, 3=release at I and J)

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*, *Area*, *E_mod*, *Iz*, *K11*, *K33*, *K44*, *transfTag*, *<'-mass'*, *massDens>*, *<'-cMass'>*)

`eleTag` (int)	unique element object tag
`eleNodes` (list (int))	a list of two element nodes
`Area` (float)	cross-sectional area of element
`E_mod` (float)	Young's Modulus
`Iz` (float)	second moment of area about the local z-axis
`K11` (float)	stiffness modifier for translation
`K33` (float)	stiffness modifier for translation
`K44` (float)	stiffness modifier for rotation
`transfTag` (int)	identifier for previously-defined coordinate-transformation (CrdTransf) object
`massDens` (float)	element mass per unit length (optional, default = 0.0)
`'-cMass'` (str)	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_mod*, *G_mod*, *Area*, *Iz*, *Avy*, *transfTag*, *<'-mass'*, *massDens>*, *<'-cMass'>*)
For a two-dimensional problem:

element (*'ElasticTimoshenkoBeam'*, *eleTag*, **eleNodes*, *E_mod*, *G_mod*, *Area*, *Iz*, *Jxx*, *Iy*, *Iz*, *Avy*, *Avz*, *transfTag*, *<'-mass'*, *massDens>*, *<'-cMass'>*)
For a three-dimensional problem:

`eleTag` (int)	unique element object tag
`eleNodes` (list (int))	a list of two element nodes
`E_mod` (float)	Young's Modulus
`G_mod` (float)	Shear Modulus
`Area` (float)	cross-sectional area of element
`Jxx` (float)	torsional moment of inertia of cross section
`Iy` (float)	second moment of area about the local y-axis
`Iz` (float)	second moment of area about the local z-axis
`Avy` (float)	Shear area for the local y-axis
`Avz` (float)	Shear area for the local z-axis
`transfTag` (int)	identifier for previously-defined coordinate-transformation (CrdTransf) object
`massDens` (float)	element mass per unit length (optional, default = 0.0)
`'-cMass'` (str)	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 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 lenghts 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 4LpI and 4LpJ 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 in to 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 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*, **eleNodes*, *transfTag*, *integrationTag*, *'-cMass'*, *'-mass'*, *mass=0.0*)
Create a dispBeamColumn element.

`eleTag` (int)	tag of the element
`eleNodes` (list (int))	list of two node tags
`transfTag` (int)	tag of transformation
`integrationTag` (int)	tag of *`beamIntegration()`*
`'-cMass'`	to form consistent mass matrix (optional, default = lumped mass matrix)
`mass` (float)	element mass density (per unit length), from which a lumped-mass matrix is formed (optional)

forceBeamColumn

element (*'forceBeamColumn'*, *eleTag*, **eleNodes*, *transfTag*, *integrationTag*, *'-iter'*, *maxIter=10*, *tol=1e-12*, *'-mass'*, *mass=0.0*)
Create a ForceBeamColumn element.

`eleTag` (int)	tag of the element
`eleNodes` (list (int))	a list of two element nodes
`transfTag` (int)	tag of transformation
`integrationTag` (int)	tag of *`beamIntegration()`*
`maxIter` (int)	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*, **eleNodes*, *numIntgrPts*, *secTag*, *transfTag*, *'-iter'*, *maxIter=10*, *tol=1e-12*, *'-mass'*, *mass=0.0*, *'-integration'*, *intType*)
Create a nonlinearBeamColumn element. This element is for backward compatability.

`eleTag` (int)	tag of the element
`eleNodes` (list (int))	a list of two element nodes
`numIntgrPts` (int)	number of integration points.
`secTag` (int)	tag of section
`transfTag` (int)	tag of transformation
`maxIter` (int)	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'`) • `'Lobatto'` • `'Legendre'` • `'Radau'` • `'NewtonCotes'` • `'Trapezoidal'`

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*, *secTag*, *transfTag*, *cRot*, *<'-mass'*, *massDens>*)

`eleTag` (int)	unique element object tag
`eleNodes` (list (int))	a list of two element nodes
`numIntgrPts` (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'*, **thick*, *'-width'*, **widths*, *'-rho'*, **rho*, *'-matConcrete'*, **matConcreteTags*, *'-matSteel'*, **matSteelTags*, *'-matShear'*, *matShearTag*)

`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)
`thick` (list (float))	a list of `m` macro-fiber thicknesses
`widths` (list (float))	a list of `m` macro-fiber widths
`rho` (list (float))	a list of m reinforcing ratios corresponding to macro-fibers; for each fiber: $rho_i = A_{s,i}/A_{gross,i} (1 < i < m)$
`matConcreteTags` (list (int))	a list of `m` uniaxialMaterial tags for concrete
`matSteelTags` (list (int))	a list of `m` uniaxialMaterial tags for steel
`matShearTag` (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 (*'SFI_MVLEM'*, *eleTag*, **eleNodes*, *m*, *c*, *'-thick'*, **thick*, *'-width'*, **widths*, *'-mat'*, **mat_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*, *Mat1Tag*, *Mat2Tag*, *Mat3Tag*, *Mat4Tag*, *Mat5Tag*, *Mat6Tag*, *Mat7Tag*, *Mat8Tag*, *Mat9Tag*, *Mat10Tag*, *Mat11Tag*, *Mat12Tag*, *Mat13Tag*, <*eleHeightFac=1.0*, *eleWidthFac=1.0*>)

`eleTag` (int)	unique element object tag
`eleNodes` (list (int))	a list of four element nodes
`Mat1Tag` (int)	uniaxial material tag for left bar-slip spring at node 1
`Mat2Tag` (int)	uniaxial material tag for right bar-slip spring at node 1
`Mat3Tag` (int)	uniaxial material tag for interface-shear spring at node 1
`Mat4Tag` (int)	uniaxial material tag for lower bar-slip spring at node 2
`Mat5Tag` (int)	uniaxial material tag for upper bar-slip spring at node 2
`Mat6Tag` (int)	uniaxial material tag for interface-shear spring at node 2
`Mat7Tag` (int)	uniaxial material tag for left bar-slip spring at node 3
`Mat8Tag` (int)	uniaxial material tag for right bar-slip spring at node 3
`Mat9Tag` (int)	uniaxial material tag for interface-shear spring at node 3
`Mat10Tag` (int)	uniaxial material tag for lower bar-slip spring at node 4
`Mat11Tag` (int)	uniaxial material tag for upper bar-slip spring at node 4
`Mat12Tag` (int)	uniaxial material tag for interface-shear spring at node 4
`Mat13Tag` (int)	uniaxial material tag for shear-panel
`eleHeightFac` (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)
`eleWidthFac` (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 < 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 < 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
`eleNodes` (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)
`Mat1` (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)
`Mat2` (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)
`Mat3` (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)
`Mat4` (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)
`MatC` (int)	uniaxial material tag for rotational spring of the central node that describes shear panel behavior
`LrgDspTag` (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'*, **dir*, *<'-orient'*, **vecx*, **vecyp*>, *<'-pDelta'*, **pDeltaVals*>, *<'-shearDist'*, **shearDist*>, *<'-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 UniaxialMaterial objects
`dirs` (list (int))	a list material directions: • 2D-case: `1` , `2` - translations along local x,y axes; `3` - rotation about local z axis • 3D-case: `1`, `2`, `3` - translations along local x,y,z axes; `4`, `5`, `6` - rotations about local x,y,z axes
`vecx` (list (float))	vector components in global coordinates defining local x-axis (optional)
`vecyp` (list (float))	vector components in global coordinates defining local y-axis (optional)
`pDeltaVals` (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'*, *PMatTag*, *'-Mz'*, *MzMatTag*, *<'-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'*, *PMatTag*, *'-T'*, *TMatTag*, *'-My'*, *MyMatTag*, *'-Mz'*, *MzMatTag*, *<'-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
`PMatTag` (int)	tag associated with previously-defined UniaxialMaterial in axial direction
`TMatTag` (int)	tag associated with previously-defined UniaxialMaterial in torsional direction
`MyMatTag` (int)	tag associated with previously-defined UniaxialMaterial in moment direction around local y-axis
`MzMatTag` (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

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'*, *PMatTag*, *'-Mz'*, *MzMatTag*, *<'-orient'*, **orientVals>*, *<'-shearDist'*, *shearDist>*, *<'-doRayleigh'>*, *<'-mass'*, *mass>*)
For a two-dimensional problem

element (*'ElastomericBearingBoucWen'*, *eleTag*, **eleNodes*, *kInit*, *qd*, *alpha1*, *alpha2*, *mu*, *eta*, *beta*, *gamma*, *'-P'*, *PMatTag*, *'-T'*, *TMatTag*, *'-My'*, *MyMatTag*, *'-Mz'*, *MzMatTag*, *<'-orient'*, **orientVals>*, *<'-shearDist'*, *shearDist>*, *<'-doRayleigh'>*, *<'-mass'*, *mass>*)
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)
`PMatTag` (int)	tag associated with previously-defined UniaxialMaterial in axial direction
`TMatTag` (int)	tag associated with previously-defined UniaxialMaterial in torsional direction
`MyMatTag` (int)	tag associated with previously-defined UniaxialMaterial in moment direction around local y-axis
`MzMatTag` (int)	tag associated with previously-defined UniaxialMaterial in moment direction around local z-axis
`orientVals` (list (int))	vector components in global coordinates defining local x-axis (optional), vector components in global coordinates defining local y-axis (optional)
`shearDist` (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)
`mass` (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'*, *PMatTag*, *'-Mz'*, *MzMatTag*, *<'-orient'*, *x1*, *x2*, *x3*, *y1*, *y2*, *y3>*, *<'-shearDist'*, *sDratio>*, *<'-doRayleigh'>*, *<'-mass'*, *m>*, *<'-maxIter'*, *iter*, *tol>*)
For a two-dimensional problem

element (*'flatSliderBearing'*, *eleTag*, **eleNodes*, *frnMdlTag*, *kInit*, *'-P'*, *PMatTag*, *'-T'*, *TMatTag*, *'-My'*, *MyMatTag*, *'-Mz'*, *MzMatTag*, *<'-orient'*, *<x1*, *x2*, *x3>*, *y1*, *y2*, *y3>*, *<'-shearDist'*, *sDratio>*, *<'-doRayleigh'>*, *<'-mass'*, *m>*, *<'-iter'*, *maxIter*, *tol>*)
For a three-dimensional problem

`eleTag` (int)	unique element object tag
`eleNodes` (list (int))	a list of two element nodes
`frnMdlTag` (float)	tag associated with previously-defined FrictionModel
`kInit` (float)	initial elastic stiffness in local shear direction
`PMatTag` (int)	tag associated with previously-defined UniaxialMaterial in axial direction
`TMatTag` (int)	tag associated with previously-defined UniaxialMaterial in torsional direction
`MyMatTag` (int)	tag associated with previously-defined UniaxialMaterial in moment direction around local y-axis
`MzMatTag` (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 = 0.0)
`'-doRayleigh'` (str)	to include Rayleigh damping from the bearing (optional, default = no Rayleigh damping contribution)
`m` (float)	element mass (optional, default = 0.0)
`iter` (int)	maximum number of iterations to undertake to satisfy element equilibrium (optional, default = 20)
`tol` (float)	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'*, *PMatTag*, *'-Mz'*, *MzMatTag*, *<'-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'*, *PMatTag*, *'-T'*, *TMatTag*, *'-My'*, *MyMatTag*, *'-Mz'*, *MzMatTag*, *<'-orient'*, *<x1*, *x2*, *x3>*, *y1*, *y2*, *y3>*, *<'-shearDist'*, *sDratio>*, *<'-doRayleigh'>*, *<'-mass'*, *m>*, *<'-iter'*, *maxIter*, *tol>*)
For a three-dimensional problem

`eleTag` (int)	unique element object tag
`eleNodes` (list (int))	a list of two element nodes
`frnMdlTag` (float)	tag associated with previously-defined FrictionModel
`Reff` (float)	effective radius of concave sliding surface
`kInit` (float)	initial elastic stiffness in local shear direction
`PMatTag` (int)	tag associated with previously-defined UniaxialMaterial in axial direction
`TMatTag` (int)	tag associated with previously-defined UniaxialMaterial in torsional direction
`MyMatTag` (int)	tag associated with previously-defined UniaxialMaterial in moment direction around local y axis
`MzMatTag` (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 = 0.0)
`'-doRayleigh'` (str)	to include Rayleigh damping from the bearing (optional, default = no Rayleigh damping contribution)
`m` (float)	element mass (optional, default = 0.0)
`maxIter` (int)	maximum number of iterations to undertake to satisfy element equilibrium (optional, default = 20)
`tol` (float)	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*, *Db1*, *Db2*, *Db3*, *Db4*, *d1*, *d2*, *d3*, *d4*, *mu1*, *mu2*, *mu3*, *mu4*, *h1*, *h2*, *h3*, *h4*, *H0*, *colLoad*, *<K>*)

`eleTag` (int)	unique element object tag
`eleNodes` (list (int))	a list of two element nodes
`R1` (float)	Radius of inner bottom sliding surface
`R2` (float)	Radius of inner top sliding surface
`R3` (float)	Radius of outer bottom sliding surface
`R4` (float)	Radius of outer top sliding surface
`Db1` (float)	Diameter of inner bottom sliding surface
`Db2` (float)	Diameter of inner top sliding surface
`Db3` (float)	Diameter of outer bottom sliding surface
`Db4` (float)	Diameter of outer top sliding surface
`d1` (float)	diameter of inner slider
`d2` (float)	diameter of inner slider
`d3` (float)	diameter of outer bottom slider
`d4` (float)	diameter of outer top slider
`mu1` (float)	friction coefficient of inner bottom sliding surface
`mu2` (float)	friction coefficient of inner top sliding surface
`mu3` (float)	friction coefficient of outer bottom sliding surface
`mu4` (float)	friction coefficient of outer top sliding surface
`h1` (float)	height from inner bottom sliding surface to center of bearing
`h2` (float)	height from inner top sliding surface to center of bearing
`h3` (float)	height from outer bottom sliding surface to center of bearing
`h4` (float)	height from inner top sliding surface to center of bearing
`H0` (float)	total height of bearing
`colLoad` (float)	initial axial load on bearing (only used for first time step then load come from model)
`K` (float)	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. Li = R_i - h_i (see Figure 1)
d1 d2 d3 (float)	= displacement limits of pendulums (Figure 1). Displacement limit of the bearing is 2 d1 + d2 + d3 + L1. d3/ L3 - L1. d2/ L2
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'*, *lim>*, *<'-orient'*, *<x1*, *x2*, *x3>*, *yp1*, *yp2*, *yp3>*, *<'-mass'*, *mass>*)

`eleTag` (int)	unique element object tag
`eleNodes` (list (int))	a list of two element nodes
`nSpring` (int)	number of springs
`matTag` (int)	tag associated with previously-defined UniaxialMaterial object
`lim` (float)	minimum deformation to calculate equivalent coefficient (see note 1)
`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
`mass` (float)	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>*, *<'-noPDInput'>*, *<'-noTilt'>*, *<'-adjustPDOutput'*, *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 (defaulut: 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
`'-noPDInput'` (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`), tolorance 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'*, **vecx*, **vecyp>*, *<'-mass'*, *m>*)

`eleTag` (int)	unique element object tag
`eleNodes` (list (int))	a list of two element nodes
`Tp` (int)	compound type = 1 : X0.6R manufactured by Bridgestone corporation.
`DDo` (float)	outer diameter [m]
`DDi` (float)	bore diameter [m]
`Hr` (float)	total thickness of rubber layer [m] Optional Data
`cr cs` (float)	coefficients for shear stress components of tau_r and tau_s
`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.
`m` (float)	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 LeadRubberX 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>*)

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)
qL (float)	density of lead (optional, default = 11200 kg/m3)
cL (float)	specific heat of lead (optional, default = 130 N-m/kg oC)
kS (float)	thermal conductivity of steel (optional, default = 50 W/m oC)
aS (float)	thermal diffusivity of steel (optional, default = 1.41e-05 m2/s)
tag1 (int)	Tag to include cavitation and post-cavitation (optional, default = 0)
tag2 (int)	Tag to include buckling load variation (optional, default = 0)
tag3 (int)	Tag to include horizontal stiffness variation (optional, default = 0)
tag4 (int)	Tag to include vertical stiffness variation (optional, default = 0)
tag5 (int)	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): $Q_d = F_y * (1 - alpha)$

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'*, *PMatTag*, *'-Vy'*, *VyMatTag*, *'-Mz'*, *MzMatTag*, *<'-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'*, *PMatTag*, *'-Vy'*, *VyMatTag*, *'-Vz'*, *VzMatTag*, *'-T'*, *TMatTag*, *'-My'*, *MyMatTag*, *'-Mz'*, *MzMatTag*, *<'-orient'*, *<x1*, *x2*, *x3>*, *y1*, *y2*, *y3>*, *<'-shearDist'*, *sDratio>*, *<'-doRayleigh'>*, *<'-mass'*, *m>*, *<'-iter'*, *maxIter*, *tol>*)

For a three-dimensional problem

`eleTag` (int)	unique element object tag
`eleNodes` (list (int))	a list of two element nodes
`frnMdlTag` (float)	tag associated with previously-defined FrictionModel
`kInit` (float)	initial stiffness of sliding friction component in local shear direction
`'-P'` `PMatTag` (int)	tag associated with previously-defined UniaxialMaterial in axial direction
`'-Vy'` `VyMatTag` (int)	tag associated with previously-defined UniaxialMaterial in shear direction along local y-axis (MER spring behavior not including friction)
`'-Vz'` `VzMatTag` (int)	tag associated with previously-defined UniaxialMaterial in shear direction along local z-axis (MER spring behavior not including friction)
`'-T'` `TMatTag` (int)	tag associated with previously-defined UniaxialMaterial in torsional direction
`'-My'` `MyMatTag` (int)	tag associated with previously-defined UniaxialMaterial in moment direction around local y-axis
`'-Mz'` `MzMatTag` (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 = 0.0)
`'-doRayleigh'` (str)	to include Rayleigh damping from the bearing (optional, default = no Rayleigh damping contribution)
`m` (float)	element mass (optional, default = 0.0)
`maxIter` (int)	maximum number of iterations to undertake to satisfy element equilibrium (optional, default = 20)
`tol` (float)	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*, *ReffectiveFP*, *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. • `1`: N, m, s, C • `2`: kN, m, s, C • `3`: N, mm, s, C • `4`: kN, mm, s, C • `5`: lb, in, s, C • `6`: kip, in, s, C • `7`: lb, ft, s, C • `8`: kip, ft, s, C

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=0.0*, *rho=0.0*, *b1=0.0*, *b2=0.0*>)

`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

ShellNLDKGQ

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

element (*'ShellNLDKGQ'*, *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

ShellNLDKGT

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

element (*'ShellNLDKGT'*, *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*, *secTag*)

`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
`secTag` (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 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 (*'20NodeBrick'*, *eleTag*, **eleNodes*, *matTag*, *bf1*, *bf2*, *bf3*, *massDen*)

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

Note: The valid queries to a 20NodeBrick 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>*)

`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)	unique integer tag associated with previously-defined nDMaterial object
`b1` `b2` `b3` (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
`matTag` (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 (*'quadUP'*, *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 Bc 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 (2.2×10^6 kPa (or 3.191×10^5 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 Bc 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 (2.2×10^6 kPa (or 3.191×10^5 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 Bc 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 (2.2×10^6 kPa (or 3.191×10^5 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 Bc 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 (2.2×10^6 kPa (or 3.191×10^5 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 Bc 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 (2.2×10^6 kPa (or 3.191×10^5 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

Twenty_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 (*'20_8_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 twenty element nodes
`matTag` (int)	Tag of an NDMaterial object (previously defined) of which the element is composed
`bulk` (float)	Combined undrained bulk modulus Bc 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 (2.2×10^6 kPa (or 3.191×10^5 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=0.0*, *b2=0.0>*)

eleTag (int)	unique element object tag
eleNodes (list (int))	a list of four element nodes in counter-clockwise order
matTag (int)	unique integer tag associated with previously-defined nDMaterial object
thick (float)	thickness of the element in out-of-plane direction
fBulk (float)	bulk modulus of the pore fluid
fDen (float)	mass density of the pore fluid
k1 k2 (float)	permeability coefficients in global x- and y-directions, respectively
void (float)	voids ratio
alpha (float)	spatial pressure field stabilization parameter (see discussion below for more information)
b1 b2 (float)	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 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_QuadUP 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 α parameter should be a small number. With a properly defined α 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*, *cNode*, *lNode*, *matTag*, *gTol*, *fTol*)

`eleTag` (int)	unique element object tag
`iNode jNode` (int)	retained nodes (-ndm 2 -ndf 2)
`cNode` (int)	constrained 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
`gTol` (float)	gap tolerance
`fTol` (float)	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*, *cNode*, *lagr_node*, *matTag*, *gTol*, *fTol*)

`eleTag` (int)	unique element object tag
`iNode jNode kNode lNode` (int)	master nodes (-ndm 3 -ndf 3)
`cNode` (int)	constrained node (-ndm 3 -ndf 3)
`lagr_node` (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*, *iNode*, *jNode*, *sNode*, *lNode*, *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*, *cNode*, *lNode*, *radius*, *crdTransf*, *matTag*, *gTol*, *fTol*, *<cFlag>*)

eleTag (int)	unique element object tag
iNode jNode (int)	master nodes (-ndm 3 -ndf 6)
cNode (int)	constrained 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 cFlag = 0 >> contact between bodies is initially assumed (DEFAULT) cFlag = 1 >> no contact between bodies is initially assumed

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*, *cNode*, *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)
cNode (int)	constrained 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` (int)	unique element object tag
`iNode jNode` (int)	end nodes (3 dof per node)
`weight` (float)	undefined
`E` (float)	elastic modulus of the cable material
`A` (float)	cross-sectional area of element
`L0` (float)	unstretched length of the cable
`alpha` (float)	coefficient of thermal expansion
`temperature_change` (float)	temperature change for the element
`rho` (float)	mass per unit length
`errorTol` (float)	allowed tolerance for within-element equilbrium (Newton-Rhapson iterations)
`Nsubsteps` (int)	number of within-element substeps into which equilibrium iterations are subdivided (not number of steps to convergence)
`massType` (int)	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-Rhapson 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*, **eleNodes*, *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
`eleNodes` (list (int))	A list of three or four element nodes, four are required for 3D
`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*, **eleNodes*, *rho*, *mu*, *b1*, *b2*, *<thickness*, *kappa>*)
Create a PFEM compressible element, which is a fluid element for FSI analysis.

eleTag (int)	tag of the element
eleNodes (list (int))	A list of four element nodes, last one is middle node
rho (float)	fluid density
mu (float)	fluid viscosity
b1 (float)	body body acceleration in x direction
b2 (float)	body body acceleration in y direction
thickness (float)	element thickness (optional)
kappa (float)	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*)

eleTag (int)	unique element object tag
eleNodes (list (int))	the four nodes defining the element, input in counterclockwise order (-ndm 3 -ndf 3)
p (float)	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*)

`eleTag` (int)	unique element object tag
`*eleNodes1` (list (int))	four nodes defining structure domain of element boundaries
`*eleNodes2` (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*)

`eleTag` (int)	unique element object tag
`eleNodes` (list (int))	4 end nodes
`matTag` (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.

`nodeTag` (int)	node tag.
`crds` (list (float))	nodal coordinates.
`ndf` (float)	nodal ndf. (optional)
`mass` (list (float))	nodal mass. (optional)
`vel` (list (float))	nodal velocities. (optional)
`accel` (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 constriant.

nodeTag (int)	tag of node to be constrained
constrValues (list (int))	a list of constraint values (0 or 1), must be preceded with *. • 0 free • 1 fixed

For example,

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

fixX command

fixX (*x*, **constrValues*, *'-tol'*, *tol=1e-10*)
Create homogeneous SP constriants.

x (float)	x-coordinate of nodes to be constrained
constrValues (list (int))	a list of constraint values (0 or 1), must be preceded with *. • 0 free • 1 fixed
tol (float)	user-defined tolerance (optional)

fixY command

fixY (*y*, **constrValues*, *'-tol'*, *tol=1e-10*)
Create homogeneous SP constriants.

y (float)	y-coordinate of nodes to be constrained
constrValues (list (int))	a list of constraint values (0 or 1), must be preceded with *. • 0 free • 1 fixed
tol (float)	user-defined tolerance (optional)

fixZ command

fixZ (*z*, **constrValues*, *'-tol'*, *tol=1e-10*)
Create homogeneous SP constriants.

`z` (float)	z-coordinate of nodes to be constrained
`constrValues` (list (int))	a list of constraint values (0 or 1), must be preceded with `*`. • `0` free • `1` fixed
`tol` (float)	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.

`rNodeTag` (int)	integer tag identifying the retained, or master node.
`cNodeTag` (int)	integer tag identifying the constrained, or slave node.
`dofs` (list (int))	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.

`rNodeTag` (int)	integer tag identifying the retained, or master node.
`cNodeTag` (int)	integer tag identifying the constrained, or slave node.
`numDOF` (int)	number of dofs to be constrained
`rcdofs` (list (int))	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. `rcdofs = [rdof1, cdof1, rdof2, cdof2, ...]`

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.

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

rigidLink command

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

Create a multi-point constraint between nodes.

`type` (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.
`rNodeTag` (int)	integer tag identifying the master node
`cNodeTag` (int)	integar 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, λ, in the load pattern with which the TimeSeries object is associated, i.e. $\lambda = F(t)$.

`tsType` (str)	time series type.
`tsTag` (int)	time series tag.
`tsArgs` (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*, *'-tStart'*, *tStart=0.0*)

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.

$\lambda = f(t) = cFactor * (t - tStart)$. (0 if t < tStart)

`tag` (int)	unique tag among TimeSeries objects
`factor` (float)	Linear factor (optional)
`tStart` (float)	start time (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 trigonemtric function of the time in the domain

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

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

`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.

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

$$slope = \frac{cFactor}{period/4}$$

$$k = \frac{t + \phi - tStart}{period} - floor(\frac{t + \phi - tStart}{period})$$

$$\phi = shift - \frac{zeroShift}{slope}$$

`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.

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

`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.

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

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

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. (optinal)
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'*, *fact*)

This commnand 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
`fact` (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>*, *Wx=0.0*, *'-beamPoint'*, *Py*, *<Pz>*, *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 uniformily distributed ref load acting in direction along member length. (optional)
`Wy` (float)	mag of uniformily distributed ref load acting in local y direction of element
`Wz` (float)	mag of uniformily distributed ref load acting in local z direction of element. (required 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. (required only for 3D)
`xL` (float)	location of point load relative to node I, prescribed as fraction of element length
`tempPts` (list (float))	temperature points: `temPts = [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 load 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 element.
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'*, *fact*)
The UniformExcitation pattern allows the user to apply a uniform excitation to a model acting in a certain direction. The command is as follows:

`patternTag` (int)	unique tag among load patterns
`dir` (int)	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
`dispSeriesTag` (int)	tag of the TimeSeries series defining the displacement history. (optional)
`velSeriesTag` (int)	tag of the TimeSeries series defining the velocity history. (optional)
`accelSeriesTag` (int)	tag of the TimeSeries series defining the acceleration history. (optional)
`vel0` (float)	the initial velocity (optional, default=0.0)
`fact` (float)	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. ==========

`patternTag` (int) integer tag identifying pattern ==========

Note:

1. The results for the responses at the nodes are the ABSOLUTE values, and not relative values as in the case of a UniformExciatation.
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

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

`gmTag` (int)	unique tag among ground motions in load pattern
`gmTags` (list (int))	the tags of existing ground motions in pattern to be used for interpolation
`facts` (list (float))	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.

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

1.4.8 mass command

mass (*nodeTag*, **massValues*)
This command is used to set the mass at a node

nodeTag (int)	integer tag identifying node whose mass is set
massValues (list (float))	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*, *betaK*, *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` (int)	number of elements in local x directions of the block.
`numY` (int)	number of elements in local y directions of the block.
`numZ` (int)	number of elements in local z directions of the block.
`startNode` (int)	node from which the mesh generation will start.
`startElem` (int)	element from which the mesh generation will start.
`eleType` (str)	element type (`'stdBrick'`, `'bbarBrick'`, `'Brick20N'`)
`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, <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-Element (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)

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

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: `Nc` 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 $i = 2...N-1$, $w_1 = (x_1 + x_2)/2$, and $w_N = 1 - (x_{N-1} + x_N)/2$.

`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.

```
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*, *secETag*, *npL*, **secsLTags*, **locsL*, **wtsL*, *npR*, **secsRTags*, **locsR*, **wtsR*)

Create a UserHinge beamIntegration object.

tag (int)	tag of the beam integration
secETag (int)	A previous-defined section objects for non-hinge area.
npL (int)	number of integration points along the left hinge.
secsLTags (list (int))	A list of previous-defined section objects for left hinge area.
locsL (list (float))	A list of locations of integration points for left hinge area.
wtsL (list (float))	A list of weights of integration points for left hinge area.
npR (int)	number of integration points along the right hinge.
secsRTags (list (int))	A list of previous-defined section objects for right hinge area.
locsR (list (float))	A list of locations of integration points for right hinge area.
wtsR (list (float))	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.

tag (int)	tag of the beam integration.
secI (int)	A previous-defined section object for hinge at I.
lpI (float)	The plastic hinge length at I.
secJ (int)	A previous-defined section object for hinge at J.
lpJ (float)	The plastic hinge length at J.
secE (int)	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',tag,secI,lpI,secJ,lpJ,secE)
```

HingeRadau

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

Create a HingeRadau 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 palstic hinge length.

Arguments and examples see *HingeMidpoint*.

HingeRadauTwo

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

Create a HingeRadauTwo 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 (equals 1/4 of the plastic hinge length).

Arguments and examples see *HingeMidpoint*.

HingeEndpoint

beamIntegration (*'HingeEndpoint'*, *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.

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

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 (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)
params (list (float))	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 (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)

sigInit (float)	Initial Stress Value (optional, default: 0.0) the strain is calculated from epsP=sigInit/E
	`if (sigInit!= 0.0) {`
	`  double epsInit = sigInit/E;`
	`  eps = trialStrain+epsInit;`
	`} else {`
	`  eps = trialStrain;`
	`}`

See also:

Steel02

Steel4

uniaxialMaterial (*'Steel4'*, *matTag*, *Fy*, *E0*, *'-asym'*, *'-kin'*, *b_k*, **params*, *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.

matTag (int)	integer tag identifying material
Fy (float)	yield strength
E0 (float)	initial elastic tangent
'-kin' (str)	apply kinematic hardening
b_k (float)	hardening ratio (E_k/E_0)
params (list (float))	control the exponential transition from linear elastic to hardening asymptote params=[R_0, r_1,r_2]. Recommended values: R_0 = 20, r_1 = 0.90, r_2 = 0.15
'-iso' (str)	apply isotropic hardening
b_i (float)	initial hardening ratio (E_i/E_0)
b_l (float)	saturated hardening ratio (E_is/E_0)
rho_i (float)	specifies the position of the intersection point between initial and saturated hardening asymptotes
R_i (float)	control the exponential transition from initial to saturated asymptote
l_yp (float)	length of the yield plateau in eps_y0 = f_y / E_0 units
'-ult' (str)	apply an ultimate strength limit
f_u (float)	ultimate strength
R_u (float)	control the exponential transition from kinematic hardening to perfectly plastic asymptote
'-asym' (str)	assume non-symmetric behavior
'-init' (str)	apply initial stress
sig_init (float)	initial stress value
'-mem' (str)	configure the load history memory
cycNum (float)	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 cycNum = 50 Load history memory can be turned off by setting cycNum = 0.

See also:

Steel4

Hysteretic

uniaxialMaterial (*'Hysteretic'*, *matTag*, **p1*, **p2*, **p3=p2*, **n1*, **n2*, **n3=n2*, *pinchX*, *pinchY*, *damage1*, *damage2*, *beta=0.0*)

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*, *eps_sh*, *eps_ult*, *'-GABuck'*, *lsr*, *beta*, *r*, *gamma*, *'-DMBuck'*, *lsr*, *alpha=1.0*, *'-CMFatigue'*, *Cf*, *alpha*, *Cd*, *'-IsoHard'*, *a1=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.

matTag (int)	integer tag identifying material
fy (float)	Yield stress in tension
fu (float)	Ultimate stress in tension
Es (float)	Initial elastic tangent
Esh (float)	Tangent at initial strain hardening
eps_sh (float)	Strain corresponding to initial strain hardening
eps_ult (float)	Strain at peak stress
'-GABuck' (str)	Buckling Model Based on Gomes and Appleton (1997)
lsr (float)	Slenderness Ratio
beta (float)	Amplification factor for the buckled stress strain curve.
r (float)	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
gamma (float)	Buckling constant
'-DMBuck' (str)	Buckling model based on Dhakal and Maekawa (2002)
lsr (float)	Slenderness Ratio
alpha (float)	Adjustment Constant usually between 0.75 and 1.0 Default: alpha=1.0, this parameter is optional.
'-CMFatigue' (str)	Coffin-Manson Fatigue and Strength Reduction
Cf (float)	Coffin-Manson constant C
alpha (float)	Coffin-Manson constant a
Cd (float)	Cyclic strength reduction constant
'-IsoHard' (str)	Isotropic Hardening / Diminishing Yield Plateau
a1 (float)	Hardening constant (default = 4.3)
limit (float)	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)
'-MPCurveParams' (str)	Menegotto and Pinto Curve Parameters
R1 (float)	(default = 0.333)
R2 (float)	(default = 18)
R3 (float)	(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
fy (float)	Yield strength
E0 (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*, *fyp*, *fyn*, *E0*, *bp*, *bn*, **params*, *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)
`params` (list (float))	parameters to control the transition from elastic to plastic branches. `params=[R0,cR1, cR2]`. Recommended values: `R0=20`, `cR1=0.925`, `cR2=0.15` or `cR2=0.0015`
`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. *FRPConfinedConcrete02*
11. *ConcreteCM*
12. *TDConcrete*
13. *TDConcreteEXP*
14. *TDConcreteMC10*
15. *TDConcreteMC10NL*

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` (int)	integer tag identifying material
`fpc` (float)	concrete compressive strength at 28 days (compression is negative)
`epsc0` (float)	concrete strain at maximum strength
`fpcu` (float)	concrete crushing strength
`epsU` (float)	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).

matTag (int)	integer tag identifying material
fpc (float)	concrete compressive strength at 28 days (compression is negative)
epsc0 (float)	concrete strain at maximum strength
fpcu (float)	concrete crushing strength
epsU (float)	concrete strain at crushing strength
lambda (float)	ratio between unloading slope at $epscu and initial slope
ft (float)	tensile strength
Ets (float)	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*, *epsc*, *epscu*, *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.

matTag (int)	integer tag identifying material
fc (float)	floating point values defining concrete compressive strength at 28 days (compression is negative)
epsc (float)	floating point values defining concrete strain at maximum strength
epscu (float)	floating point values defining concrete strain at crushing strength
Ec (float)	floating point values defining initial stiffness
fct (float)	floating point value defining the maximum tensile strength of concrete (optional)
et (float)	floating point value defining ultimate tensile strain of concrete (optional)
beta (float)	loating 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 $Ec = 57000 * sqrt(|fcc|)$ (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)	α_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)	α_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*, *epsc*, *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)
epsc (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*, *epsc0*, *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)
epsc0 (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/epsc0)

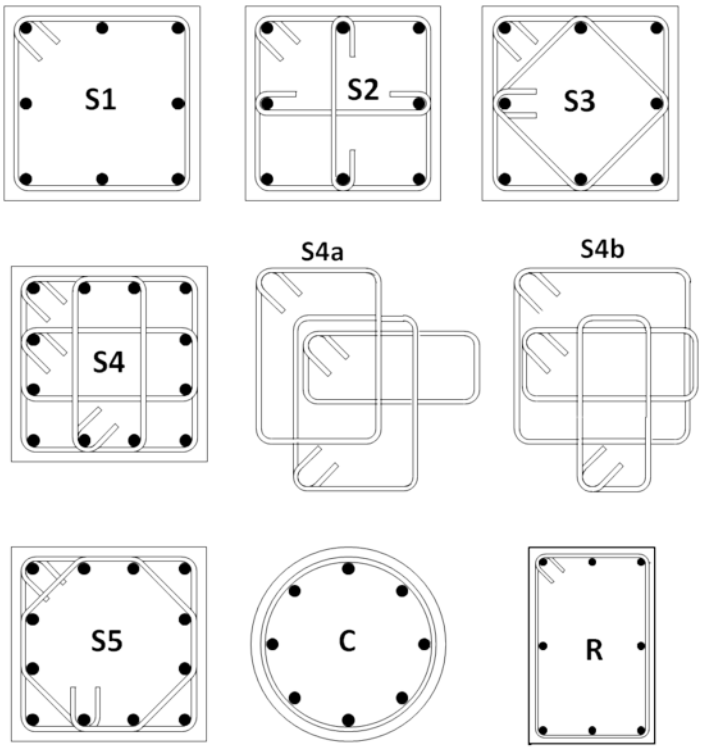
See also:

Notes

ConfinedConcrete01

uniaxialMaterial (*'ConfinedConcrete01'*, *matTag*, *secType*, *fpc*, *Ec*, *epscu_type*, *epscu_val*, *nu*, *L1*, *L2*, *L3*, *phis*, *S*, *fyh*, *Es0*, *haRatio*, *mu*, *phiLon*, *'-internal'*, **internalArgs*, *'-wrap'*, **wrapArgs*, *'-gravel'*, *'-silica'*, *'-tol'*, *tol*, *'-maxNumIter'*, *maxNumIter*, *'-epscuLimit'*, *epscuLimit*, *'-stRatio'*, *stRatio*)

matTag (int)	integer tag identifying material
secType (str)	tag for the transverse reinforcement configuration. see image below. • 'S1' square section with S1 type of transverse reinforcement with or without external FRP wrapping • 'S2' square section with S2 type of transverse reinforcement with or without external FRP wrapping • 'S3' square section with S3 type of transverse reinforcement with or without external FRP wrapping • 'S4a' square section with S4a type of transverse reinforcement with or without external FRP wrapping • 'S4b' square section with S4b type of transverse reinforcement with or without external FRP wrapping • 'S5' square section with S5 type of transverse reinforcement with or without external FRP wrapping • 'C' circular section with or without external FRP wrapping • 'R' rectangular section with or without external FRP wrapping.
fpc (float)	unconfined cylindrical strength of concrete specimen.
Ec (float)	initial elastic modulus of unconfined concrete.
epscu_type (str)	Method to define confined concrete ultimate strain - -epscu then value is confined concrete ultimate strain, - -gamma then value is the ratio of the strength corresponding to ultimate strain to the peak strength of the confined concrete stress-strain curve. If gamma cannot be achieved in the range [0, epscuLimit] then epscuLimit (optional, default: 0.05) will be assumed as ultimate strain.
epscu_val (float)	Value for the definition of the concrete ultimate strain
nu (str) or (list)	Definition for Poisson's Ratio. - ['-nu', <value of Poisson's ratio>] - '-varub' 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 -'-varnoub' 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.
L1 (float)	length/diameter of square/circular core section measured respect to the hoop center line.
L2 (float)	additional dimensions when multiple hoops are being used.
L3 (float)	additional dimensions when multiple hoops are being used.
phis (float)	hoop diameter. If section arrangement has multiple hoops it refers to the external hoop.
S (float)	hoop spacing.
fyh (float)	yielding strength of the hoop steel.
Es0 (float)	elastic modulus of the hoop steel.
haRatio (float)	hardening ratio of the hoop steel.



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.

`matTag` (int)	integer tag identifying material
`fc` (float)	concrete compressive strength
`epsc` (float)	concrete strain at corresponding to compressive strength
`ft` (float)	concrete tensile strength
`epst` (float)	concrete strain at corresponding to tensile strength
`Ec` (float)	concrete initial Elastic modulus
`alphac` (float)	compressive descending parameter
`alphat` (float)	tensile descending parameter
`cesp` (float)	plastic parameter, recommended values: 0.2~0.3
`etap` (float)	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*, *nu0*, *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.
nu0 (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

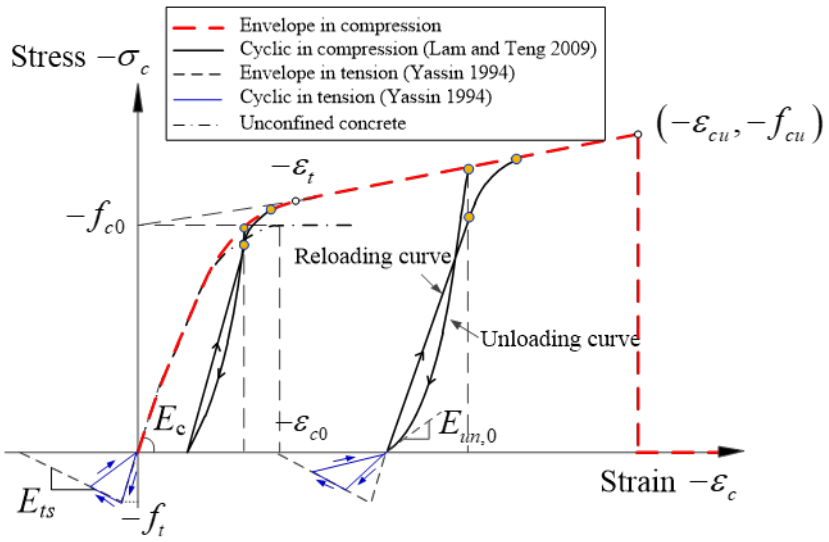
FRPConfinedConcrete02

DEVELOPED AND IMPLEMENTED BY:

Jin-Yu LU, Southeast University, Nanjing, China
Guan LIN (guanlin@polyu.edu.hk), Hong Kong Polytechnic University, Hong Kong, China.

uniaxialMaterial (*'FRPConfinedConcrete02'*, *matTag*, *fc0*, *Ec*, *ec0*, *<'-JacketC'*, *tfrp*, *Efrp*, *erup*, *R>*, *<'-Ultimate'*, *fcu*, *ecu>*, *ft*, *Ets*, *Unit*)

Figure 1 Hysteretic Stress-Strain Relation



This command is used to construct a uniaxial hysteretic stress-strain model for fiber-reinforced polymer (FRP)-confined concrete. The envelope compressive stress-strain response is described by a parabolic first portion and a linear second portion with smooth connection between them (Figure 1). The hysteretic rules of compression are based on Lam and Teng's (2009) model. The cyclic linear tension model of Yassin (1994) for unconfined concrete (as adopted in Concrete02) is used with slight modifications to describe the tensile behavior of FRP-confined concrete (Teng et al. 2015).

`matTag` (int)	integer tag identifying material
`fc0` (float)	compressive strength of unconfined concrete (compression is negative)
`Ec` (float)	elastic modulus of unconfined concrete (=4730(-$fc0(MPa)))
`ec0` (float)	axial strain corresponding to unconfined concrete strength (0.002)
`-JacketC` (str)	input parameters of the FRP jacket in a circular section
`tfrp` (float)	thickness of an FRP jacket
`Efrp` (float)	tensile elastic modulus of an FRP jacket
`erup` (float)	hoop rupture strain of an FRP jacket
`R` (float)	radius of circular column section
`-Ultimate` (str)	input ultimate stress/strain directly
`fcu` (float)	ultimate stress of FRP-confined concrete ($fcu $fc0)
`ecu` (float)	ultimate strain of FRP-confined concrete
`ft` (float)	tensile strength of unconfined concrete (=0.632(-$fc0(MPa)))
`Ets` (float)	stiffness of tensile softening (0.05 Ec)
`Unit` (float)	unit indicator, Unit = 1 for SI Metric Units; Unit = 0 for US Customary Units

Note:

1. Compressive concrete parameters should be input as negative values.
2. The users are required to input either the FRP jacket properties in an FRP-confined circular column (<-JacketC>)

or directly input the ultimate point (ϵcu, fcu) (<-Ultimate>). If <-JacketC> is used, the ultimate stress and strain are automatically calculated based on Teng et al.'s (2009) model which is a refined version of Lam and Teng's (2003) stress-strain model for FRP-confined concrete in circular columns. If <-Ultimate> is used, the ultimate stress and strain can be calculated by the users in advance based on other stress-strain models of FRP-confined concrete and thus can be used for other cross section shapes (e.g., square, rectangular, or elliptical). If none of them is specified, a stress-strain curve (parabola + horizontal linear curve) for unconfined concrete will be defined (Figure 1). Both <-JacketC> and <-Ultimate> adopt the envelope compressive stress-strain curve with a parabolic first portion and a linear second portion.

3. Unit indicator: $Unit = 1 for SI Metric Units (e.g., N, mm, MPa); $Unit = 0 for US Customary Units (e.g., kip, in, sec, ksi).

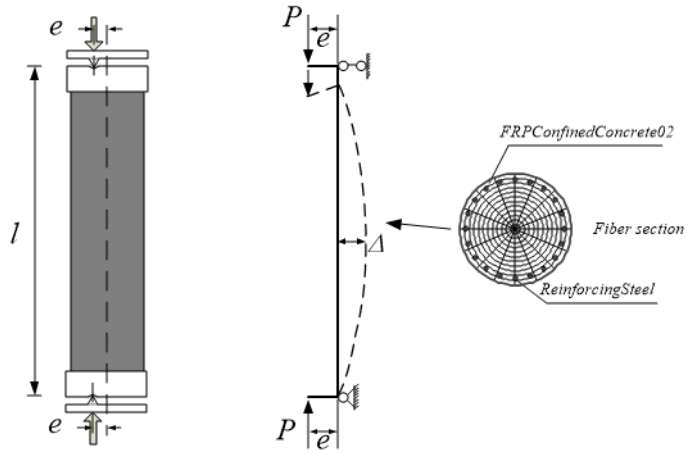
Calibration:

1. The implemented new material has been calibrated using a simple-supported Force-Based Beam-Column element subjected to axial load only (http://opensees.berkeley.edu/wiki/index.php/Calibration_of_Maxwell_Material). The output stress-strain responses were compared with the desired curves defined by the input parameters.

Examples:

1. Example 1: Pin-ended FRP-confined reinforced concrete (RC) columns

Figure 2 Simulation of pin-ended FRP-confined RC column



(a) Column under eccentric loading (b) Idealized column model

1. The first example is a pin-ended FRP-confined circular RC column subjected to eccentric compression (load eccentricity = 20 mm) at both ends tested by Bisby and Ranger (2010) (Figure 2). Due to the symmetry in geometry and loading, only half of the column needs to be modelled. In this case, three forceBeamColumn elements each with 5 integration points were used for the half column. The FRPConfinedConcrete02 model was used to describe the stress-strain behavior of FRP-confined concrete. Either <-JacketC> or <-Ultimate> can be used. If the former is used, the properties of the FRP jacket need to be input; if the latter is used, the ultimate stress and strain need to be calculated by the users and input directly. The eccentric loading is applied with a combined axial load and bending moment at each end node. An increasing vertical displacement is applied to the top node of the column model. The analysis terminated until the ultimate axial strain of FRP-confined concrete was reached by the extreme compression concrete fiber at the mid-height (equivalent to FRP rupture). SI Metric Unit (e.g., N, mm, MPa) is used in the script of this example ($Unit = 1).

2. Figure 3 shows the comparison of axial load-lateral displacement curve between the test results and the theoretical results. Figure 4 shows the variation of column slenderness ratio (l/D) on

the axial load-lateral displacement response of the column. Please refer to Lin (2016) for more details about the modeling.

Figure 3 Experimental results vs theoretical results

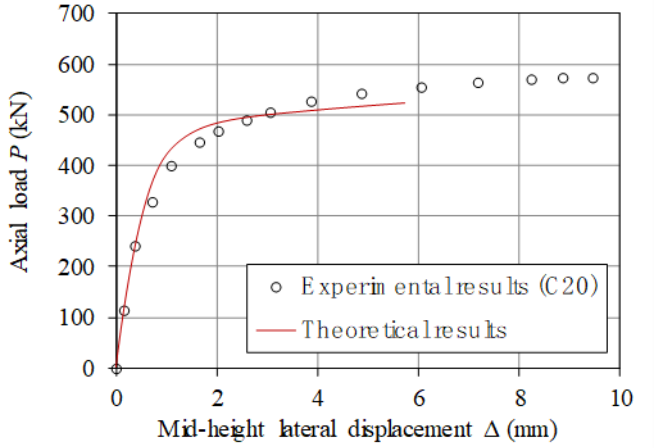
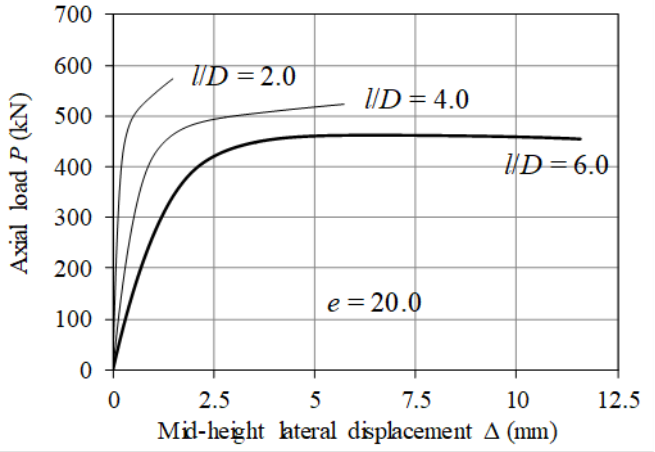
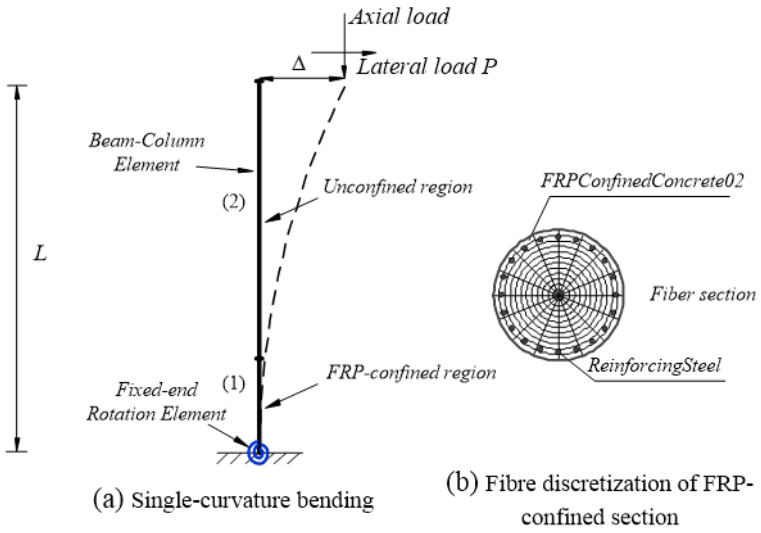


Figure 4 Parametric study (effect of column slenderness ratio)



2. Example 2: Cantilever column subjected to constant axial compression and cyclic lateral loading

Figure 5 Simulation of columns under cyclic latera loading



1. The second example is a cantilever FRP-confined circular RC column subjected to constant axial compression and cyclic lateral loading (Column C5 tested by Saadatmanesh et al. 1997). The US Customary Units (e.g., kip, in, sec, ksi) were used in this example. The twenty-five (25)-in.-height region (potential plastic hinge region) above the footing of the column was wrapped with an FRP jacket; the remaining portion of the column with a height of 71 in. was conventional RC section without FRP jacketing. The column was modelled using two forceBeamColumn elements to cater for the variation of section characteristic along the column height. A zero length section element at the column-footing interface was used to simulate fixed-end rotations due to the strain penetration of longitudinal steel bars (Figure 5) (Lin et al. 2012). The bond-slip model of Zhao and Sritharan (2007) (Bond_SP01) was used to depict the bar stress-slip response. In addition, another zero length section element was used at the column-footing interface to consider the possible rotations of the footing (Teng et al. 2015). The rotation stiffness of the zero length section element was adjusted to achieve close matching between the test response and the predicted response during the initial stage of loading. This zero length section element was found to have little effect on the ultimate displacement of the column (Teng et al. 2015). Moreover, the inclination of axial load in the column test needs to be accounted for when comparing predicted results with test results (Teng et al. 2015). Figure 6 shows the comparison of lateral load-lateral displacement curve between the test results and the theoretical results.

References:

1. Bisby, L. and Ranger, M. (2010). "Axial-flexural interaction in circular FRP-confined reinforced concrete columns", Construction and Building Materials, Vol. 24, No. 9, pp. 1672-1681.

2. Lam, L. and Teng, J.G. (2003). "Design-oriented stress-strain model for FRP-confined concrete", Construction and Building Materials, Vol. 17, No. 6, pp. 471-489.

3. Lam, L. and Teng, J.G. (2009). "Stress-strain model for FRP-confined concrete under cyclic axial compression", Engineering Structures, Vol. 31, No. 2, pp. 308-321.

4. Lin, G. (2016). Seismic Performance of FRP-confined RC Columns: Stress-Strain Models and Numerical Simulation, Ph.D. thesis, Department of Civil and Environmental Engineering, The Hong Kong Polytechnic University, Hong Kong, China.

5. Lin, G. and Teng, J.G. (2015). "Numerical simulation of cyclic/seismic lateral response of square RC columns confined with fibre-reinforced polymer jackets", Proceedings, Second International Conference on Performance-based and Life-cycle Structural Engineering (PLSE 2015), pp. 481-489 (http://plse2015.org/cms/USB/pdf/full-paper_7408.pdf).

6. Lin, G., Teng, J.G. and Lam, L. (2012). "Numerical simulation of FRP-jacketed RC columns under cyclic

loading: modeling of the strain penetration effect", First International Conference on Performance-based and Life-cycle Structural Engineering (PLSE2012), December 5-7, Hong Kong, China.

7. Saadatmanesh, H., Ehsani, M. and Jin, L. (1997). "Seismic retrofitting of rectangular bridge columns with composite straps", Earthquake Spectra, Vol. 13, No. 2, pp. 281-304.
8. Teng, J.G., Lam, L., Lin, G., Lu, J.Y. and Xiao, Q.G. (2015). "Numerical Simulation of FRP-Jacketed RC Columns Subjected to Cyclic and Seismic Loading", Journal of Composites for Construction, ASCE, Vol. 20, No. 1, pp. 04015021.
9. Yassin, M.H.M. (1994). Nonlinear Analysis of Prestressed Concrete Structures under Monotonic and Cyclic Loads, Ph.D. thesis, University of California at Berkeley, California, USA.
10. Zhao, J. and Sritharan, S. (2007). "Modeling of strain penetration effects in fiber-based analysis of reinforced concrete structuresconcrete structures", ACI Structural Journal, Vol. 104, No. 2, pp. 133-141.

ConcreteCM

uniaxialMaterial (*'ConcreteCM'*, *matTag*, *fpcc*, *epcc*, *Ec*, *rc*, *xcrn*, *ft*, *et*, *rt*, *xcrp*, *'-GapClose'*, *GapClose=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 (ϵ'_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 (ϵ^-_{cr}, where the envelope curve starts following a straight line)
`ft` (float)	Tensile strength (f_t)
`et` (float)	Strain at tensile strength (ϵ_t)
`rt` (float)	Shape parameter in Tsai's equation defined for tension (r_t)
`xcrp` (float)	Non-dimensional critical strain on tension envelope (ϵ^+_{cr}, where the envelope curve starts following a straight line - large value [e.g., 10000] recommended when tension stiffening is considered)
`GapClose` (float)	GapClose = 0, less gradual gap closure (default); GapClose = 1, more gradual gap closure

See also:

Notes

TDConcrete

uniaxialMaterial(*'TDConcrete'*, *matTag*, *fc*, *fct*, *Ec*, *beta*, *tD*, *epsshu*, *psish*, *Tcr*, *phiu*, *psicr1*, *psicr2*, *tcast*)

This command is used to construct a uniaxial time-dependent concrete material object with linear behavior in compression, nonlinear behavior in tension (REF: Tamai et al., 1988) and creep and shrinkage according to ACI 209R-92.

`matTag` (int)	integer tag identifying material
`fc` (float)	concrete compressive strength (compression is negative)
`fct` (float)	concrete tensile strength (tension is positive)
`Ec` (float)	concrete modulus of elasticity
`beta` (float)	tension softening parameter (tension softening exponent)
`tD` (float)	analysis time at initiation of drying (in days)
`epsshu` (float)	ultimate shrinkage strain as per ACI 209R-92 (shrinkage is negative)
`psish` (float)	fitting parameter of the shrinkage time evolution function as per ACI 209R-92
`Tcr` (float)	creep model age (in days)
`phiu` (float)	ultimate creep coefficient as per ACI 209R-92
`psicr1` (float)	fitting parameter of the creep time evolution function as per ACI 209R-92
`psicr2` (float)	fitting parameter of the creep time evolution function as per ACI 209R-92
`tcast` (float)	analysis time corresponding to concrete casting (in days; minimum value 2.0)

Note:

1. Compressive concrete parameters should be input as negative values (if input as positive, they will be converted to negative internally).
2. Shrinkage concrete parameters should be input as negative values (if input as positive, they will be converted to negative internally).

See also:

Detailed descriptions of the model and its implementation can be found in the following: (1) Knaack, A.M., Kurama, Y.C. 2018. Modeling Time-Dependent Deformations: Application for Reinforced Concrete Beams with Recycled Concrete Aggregates. ACI Structural J. 115, 175-190. doi:10.14359/51701153 (2) Knaack, A.M., 2013. Sustainable concrete structures using recycled concrete aggregate: short-term and long-term behavior considering material variability. PhD Dissertation, Civil and Environmental Engineering and Earth Sciences, University of Notre Dame, Notre Dame, Indiana, USA, 680 pp A manual describing the use of the model and sample files can be found at: https://data.mendeley.com/datasets/z4gxnhchky/1

TDConcreteEXP

uniaxialMaterial(*'TDConcreteEXP'*, *matTag*, *fc*, *fct*, *Ec*, *beta*, *tD*, *epsshu*, *psish*, *Tcr*, *epscru*, *sigCr*, *psicr1*, *psicr2*, *tcast*)

This command is used to construct a uniaxial time-dependent concrete material object with linear behavior in compression, nonlinear behavior in tension (REF: Tamai et al., 1988) and creep and shrinkage according to ACI 209R-92.

matTag (int)	integer tag identifying material
fc (float)	concrete compressive strength (compression is negative)
fct (float)	concrete tensile strength (tension is positive)
Ec (float)	concrete modulus of elasticity
beta (float)	tension softening parameter (tension softening exponent)
tD (float)	analysis time at initiation of drying (in days)
epsshu (float)	ultimate shrinkage strain as per ACI 209R-92 (shrinkage is negative)
psish (float)	fitting parameter of the shrinkage time evolution function as per ACI 209R-92
Tcr (float)	creep model age (in days)
epscru (float)	ultimate creep strain (e.g., taken from experimental measurements)
sigCr (float)	concrete compressive stress (input as negative) associated with $epscru (e.g., experimentally applied)
psicr1 (float)	fitting parameter of the creep time evolution function as per ACI 209R-92
psicr2 (float)	fitting parameter of the creep time evolution function as per ACI 209R-92
tcast (float)	analysis time corresponding to concrete casting (in days; minimum value 2.0)

Note:

1. Compressive concrete parameters should be input as negative values (if input as positive, they will be converted to negative internally).
2. Shrinkage concrete parameters should be input as negative values (if input as positive, they will be converted to negative internally).

See also:

Detailed descriptions of the model and its implementation can be found in the following: (1) Knaack, A.M., Kurama, Y.C. 2018. Modeling Time-Dependent Deformations: Application for Reinforced Concrete Beams with Recycled Concrete Aggregates. ACI Structural J. 115, 175-190. doi:10.14359/51701153 (2) Knaack, A.M., 2013. Sustainable concrete structures using recycled concrete aggregate: short-term and long-term behavior considering material variability. PhD Dissertation, Civil and Environmental Engineering and Earth Sciences, University of Notre Dame, Notre Dame, Indiana, USA, 680 pp A manual describing the use of the model and sample files can be found at: https://data.mendeley.com/datasets/z4gxnhchky/1

TDConcreteMC10

uniaxialMaterial (*'TDConcreteMC10'*, *matTag*, *fc*, *fct*, *Ec*, *Ecm*, *beta*, *tD*, *epsba*, *epsbb*, *epsda*, *epsdb*, *phiba*, *phibb*, *phida*, *phidb*, *tcast*, *cem*)

This command is used to construct a uniaxial time-dependent concrete material object with linear behavior in compression, nonlinear behavior in tension (REF: Tamai et al., 1988) and creep and shrinkage according to fib Model Code 2010.

matTag (int)	integer tag identifying material
fc (float)	concrete compressive strength (compression is negative)
fct (float)	concrete tensile strength (tension is positive)
Ec (float)	concrete modulus of elasticity at loading age
Ecm (float)	concrete modulus of elasticity at 28 days
beta (float)	tension softening parameter (tension softening exponent)
tD (float)	analysis time at initiation of drying (in days)
epsba (float)	ultimate basic shrinkage strain (input as negative) as per fib Model Code 2010
epsbb (float)	fitting parameter of the basic shrinkage time evolution function as per fib Model Code 2010
epsda (float)	product of ultimate drying shrinkage strain and relative humidity function as per fib Model Code 2010
epsdb (float)	fitting parameter of the basic shrinkage time evolution function as per fib Model Code 2010
phiba (float)	parameter for the effect of compressive strength on basic creep as per fib Model Code 2010
phibb (float)	fitting parameter of the basic creep time evolution function as per fib Model Code 2010
phida (float)	product of the effect of compressive strength and relative humidity on drying creep as per fib Model Code 2010
phidb (float)	fitting parameter of the drying creep time evolution function as per fib Model Code 2010
tcast (float)	analysis time corresponding to concrete casting (in days; minimum value 2.0)
cem (float)	coefficient dependent on the type of cement as per fib Model Code 2010

Note:

1. Compressive concrete parameters should be input as negative values (if input as positive, they will be converted to negative internally).
2. Shrinkage concrete parameters should be input as negative values (if input as positive, they will be converted to negative internally).

See also:

Detailed descriptions of the model and its implementation can be found in the following: (1) Knaack, A.M., Kurama, Y.C. 2018. Modeling Time-Dependent Deformations: Application for Reinforced Concrete Beams with Recycled Concrete Aggregates. ACI Structural J. 115, 175-190. doi:10.14359/51701153 (2) Knaack, A.M., 2013. Sustainable concrete structures using recycled concrete aggregate: short-term and long-term behavior considering material variability. PhD Dissertation, Civil and Environmental Engineering and Earth Sciences, University of Notre Dame, Notre Dame, Indiana, USA, 680 pp A manual describing the use of the model and sample files can be found at: https://data.mendeley.com/datasets/z4gxnhchky/1

TDConcreteMC10NL

uniaxialMaterial (*'TDConcreteMC10NL'*, *matTag*, *fc*, *fcu*, *epscu*, *fct*, *Ec*, *Ecm*, *beta*, *tD*, *epsba*, *epsbb*, *epsda*, *epsdb*, *phiba*, *phibb*, *phida*, *phidb*, *tcast*, *cem*)

This command is used to construct a uniaxial time-dependent concrete material object with non-linear behavior

in compression (REF: Concrete02), nonlinear behavior in tension (REF: Tamai et al., 1988) and creep and shrinkage according to fib Model Code 2010.

`matTag` (int)	integer tag identifying material
`fc` (float)	concrete compressive strength (compression is negative)
`fcu` (float)	concrete crushing strength (compression is negative)
`epscu` (float)	concrete strain at crushing strength (input as negative)
`fct` (float)	concrete tensile strength (tension is positive)
`Ec` (float)	concrete modulus of elasticity at loading age
`Ecm` (float)	concrete modulus of elasticity at 28 days
`beta` (float)	tension softening parameter (tension softening exponent)
`tD` (float)	analysis time at initiation of drying (in days)
`epsba` (float)	ultimate basic shrinkage strain (input as negative) as per fib Model Code 2010
`epsbb` (float)	fitting parameter of the basic shrinkage time evolution function as per fib Model Code 2010
`epsda` (float)	product of ultimate drying shrinkage strain and relative humidity function as per fib Model Code 2010
`epsdb` (float)	fitting parameter of the basic shrinkage time evolution function as per fib Model Code 2010
`phiba` (float)	parameter for the effect of compressive strength on basic creep as per fib Model Code 2010
`phibb` (float)	fitting parameter of the basic creep time evolution function as per fib Model Code 2010
`phida` (float)	product of the effect of compressive strength and relative humidity on drying creep as per fib Model Code 2010
`phidb` (float)	fitting parameter of the drying creep time evolution function as per fib Model Code 2010
`tcast` (float)	analysis time corresponding to concrete casting (in days; minimum value 2.0)
`cem` (float)	coefficient dependent on the type of cement as per fib Model Code 2010

Note:

1. Compressive concrete parameters should be input as negative values (if input as positive, they will be converted to negative internally).
2. Shrinkage concrete parameters should be input as negative values (if input as positive, they will be converted to negative internally).

See also:

Detailed descriptions of the model and its implementation can be found in the following: (1) Knaack, A.M., Kurama, Y.C. 2018. Modeling Time-Dependent Deformations: Application for Reinforced Concrete Beams with Recycled Concrete Aggregates. ACI Structural J. 115, 175-190. doi:10.14359/51701153 (2) Knaack, A.M., 2013. Sustainable concrete structures using recycled concrete aggregate: short-term and long-term behavior considering material variability. PhD Dissertation, Civil and Environmental Engineering and Earth Sciences, University of Notre Dame, Notre Dame, Indiana, USA, 680 pp A manual describing the use of the model and sample files can be found at: https://data.mendeley.com/datasets/z4gxnhchky/1

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*, **MatTags*, *'-factors'*, **factorArgs*)

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
MatTags (list (int))	identification tags of materials making up the material model
factorArgs (list (float))	factors to create a linear combination of the specified materials. Factors can be negative to subtract one material from an other. (optional, default = 1.0)

See also:

Notes

Series Material

uniaxialMaterial (*'Series'*, *matTag*, **matTags*)

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
matTags (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*, *soilType*, *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 Vijayvergiya'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 Vijayvergiya'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'*, *timeSeriesTag*)

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 FourNodeQuad elements, or PressureDependMultiYield or PressureDependMultiYield02 materials in FourNodeQuadUP or NineFourQuadUP elements. There are two possible forms:

`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
`pRes` (float)	sets the minimum (or residual) peak resistance that the material retains as the adjacent solid soil elements liquefy
`ele1` `ele2` (float)	are the eleTag (element numbers) for the two solid elements from which PyLiq1 will obtain mean effective stresses and excess pore pressures
`timeSeries` (float)	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'*, *timeSeriesTag*)

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.

matTag (int)	integer tag identifying material
tzType (int)	tzType = 1 Backbone of t-z curve approximates Reese and O'Neill (1987). tzType = 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).
ele1 ele2 (float)	are the eleTag (element numbers) for the two solid elements from which PyLiq1 will obtain mean effective stresses and excess pore pressures
timeSeriesTag (float)	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 (ModIMKPeakOriented 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` (int)	integer tag identifying material
`E` (float)	tangent stiffness
`sigmaY` (float)	yield stress or force
`H_iso` (float)	isotropic hardening Modulus
`H_kin` (float)	kinematic hardening Modulus
`eta` (float)	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.

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

$P_p = nb_o h^2 f_y/4L$

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

$K_p = nb_o E h^3 f_y/6L^3$

See also:

Notes

ViscousDamper Material

uniaxialMaterial (*'ViscousDamper'*, *matTag*, *K_el*, *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` (int)	integer tag identifying material
`K_el` (float)	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` (float)	Damping coefficient
`alpha` (float)	Velocity exponent
`LGap` (float)	Gap length to simulate the gap length due to the pin tolerance
`NM` (int)	Employed adaptive numerical algorithm (default value NM = 1; * `1` = Dormand-Prince54, * `2` = 6th order Adams-Bashforth-Moulton, * `3` = modified Rosenbrock Triple)
`RelTol` (float)	Tolerance for absolute relative error control of the adaptive iterative algorithm (default value 10^-6)
`AbsTol` (float)	Tolerance for absolute error control of adaptive iterative algorithm (default value 10^-10)
`MaxHalf` (int)	Maximum number of sub-step iterations within an integration step (default value 15)

See also:

Notes

BilinearOilDamper Material

uniaxialMaterial(*'BilinearOilDamper'*, *matTag*, *K_el*, *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 (int)	integer tag identifying material
K_el (float)	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 (float)	Damping coefficient
Fr (float)	Damper relief load (default=1.0, Damper property)
p (float)	Post-relief viscous damping coefficient ratio (default=1.0, linear oil damper)
LGap (float)	Gap length to simulate the gap length due to the pin tolerance
NM (int)	Employed adaptive numerical algorithm (default value NM = 1; • 1 = Dormand-Prince54, • 2 = 6th order Adams-Bashforth-Moulton, • 3 = modified Rosenbrock Triple)
RelTol (float)	Tolerance for absolute relative error control of the adaptive iterative algorithm (default value 10^-6)
AbsTol (float)	Tolerance for absolute error control of adaptive iterative algorithm (default value 10^-10)
MaxHalf (int)	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=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=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=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_p_Plus (float)	pre-capping rotation for positive loading direction (often noted as plastic rotation capacity)
theta_p_Neg (float)	pre-capping rotation for negative loading direction (often noted as plastic rotation capacity) (positive value)
theta_pc_Plus (float)	post-capping rotation for positive loading direction
theta_pc_Neg (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_u_Plus (float)	ultimate rotation capacity for positive loading direction
theta_u_Neg (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=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=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=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=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_Plus (float)	pre-capping rotation for positive loading direction (often noted as plastic rotation capacity)
theta_p_Neg (float)	pre-capping rotation for negative loading direction (often noted as plastic rotation capacity) (must be defined as a positive value)
theta_pc_Plus (float)	post-capping rotation for positive loading direction
theta_pc_Neg (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_Plus (float)	ultimate rotation capacity for positive loading direction
theta_u_Neg (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=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=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=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=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_Plus (float)	pre-capping rotation for positive loading direction (often noted as plastic rotation capacity)
theta_p_Neg (float)	pre-capping rotation for negative loading direction (often noted as plastic rotation capacity) (must be defined as a positive value)
theta_pc_Plus (float)	post-capping rotation for positive loading direction
theta_pc_Neg (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_Plus (float)	ultimate rotation capacity for positive loading direction
theta_u_Neg (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*, *alpha*, *beta*)

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

`matTag` (int)	integer tag identifying material
`F0` (float)	Intercept strength of the shear wall spring element for the asymtotic line to the envelope curve F0 > FI > 0
`FI` (float)	Intercept strength of the spring element for the pinching branch of the hysteretic curve. (FI > 0).
`DU` (float)	Spring element displacement at ultimate load. (DU > 0).
`S0` (float)	Initial stiffness of the shear wall spring element (S0 > 0).
`R1` (float)	Stiffness ratio of the asymptotic line to the spring element envelope curve. The slope of this line is R1 S0. (0 < R1 < 1.0).
`R2` (float)	Stiffness ratio of the descending branch of the spring element envelope curve. The slope of this line is R2 S0. (R2 < 0).
`R3` (float)	Stiffness ratio of the unloading branch off the spring element envelope curve. The slope of this line is R3 S0. (R3 1).
`R4` (float)	Stiffness ratio of the pinching branch for the spring element. The slope of this line is R4 S0. (R4 > 0).
`alpha` (float)	Stiffness degradation parameter for the shear wall spring element. (ALPHA > 0).
`beta` (float)	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 (int)	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*, *otherTag*, *'-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
`otherTag` (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:
   - `Kmax` = 20300 kN/m of abutment width
   - `Kcur` = `Kmax`
   - `Rf` = 0.7
   - `Fult` = -326 kN per meter of abutment width
   - `gap` = -2.54 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.

`matTag` (int)	integer tag identifying material
`s1p` `e1p` (float)	stress and strain (or force & deformation) at first point of the envelope in the positive direction
`s2p` `e2p` (float)	stress and strain (or force & deformation) at second point of the envelope in the positive direction
`s3p` `e3p` (float)	stress and strain (or force & deformation) at third point of the envelope in the positive direction
`s1n` `e1n` (float)	stress and strain (or force & deformation) at first point of the envelope in the negative direction
`s2n` `e2n` (float)	stress and strain (or force & deformation) at second point of the envelope in the negative direction
`s3n` `e3n` (float)	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(m-1)
`damage2` (float)	damage due to energy: D2(Ei/Eult)
`beta` (float)	power used to determine the degraded unloading stiffness based on ductility, m-b (optional, default=0.0)
`curveTag` (int)	an integer tag for the Limit Curve defining the limit surface
`curveType` (int)	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 stains: 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 stains: 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'*, **strain*, *'-stress'*, **stress*)
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` (int)	integer tag identifying material
`eta` (float)	damping tangent (optional, default=0.0)
`strain` (list (float))	list of strain points along stress-strain curve
`stress` (list (float))	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` (int)	integer tag identifying material
`pts` (list (float))	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` (int)	integer tag identifying material
`otherTag` (int)	tag of the other material
`initStrain` (float)	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 strain that corresponds to the initial stress will be calculated from the other material.

matTag (int)	integer tag identifying material
otherTag (float)	tag of the other material
initStress (float)	initial stress

See also:

Notes

PathIndependent Material

uniaxialMaterial (*'PathIndependent'*, *matTag*, *OtherTag*)

This command is to create a PathIndependent material

matTag (int)	integer tag identifying material
OtherTag (int)	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.

matTag (int)	integer tag identifying material
ePf1 ePf2 ePf3 ePf4 (float)	floating point values defining force points on the positive response envelope
ePd1 ePd2 ePd3 ePd4 (float)	floating point values defining deformation points on the positive response envelope
eNf1 eNf2 eNf3 eNf4 (float)	floating point values defining force points on the negative response envelope
eNd1 eNd2 eNd3 eNd4 (float)	floating point values defining deformation points on the negative response envelope
rDispP (float)	floating point value defining the ratio of the deformation at which reloading occurs to the maximum historic deformation demand
fFoceP (float)	floating point value defining the ratio of the force at which reloading begins to force corresponding to the maximum historic deformation demand
uForceP (float)	floating point value defining the ratio of strength developed upon unloading from negative load to the maximum strength developed under monotonic loading
rDispN (float)	floating point value defining the ratio of the deformation at which reloading occurs to the minimum historic deformation demand
fFoceN (float)	floating point value defining the ratio of the force at which reloading begins to force corresponding to the minimum historic deformation demand
uForceN (float)	floating point value defining the ratio of strength developed upon unloading from negative load to the minimum strength developed under monotonic loading
gK1 gK2 gK3 gK4 gKLim (float)	floating point values controlling cyclic degradation model for unloading stiffness degradation
gD1 gD2 gD3 gD4 gDLim (float)	floating point values controlling cyclic degradation model for reloading stiffness degradation
gF1 gF2 gF3 gF4 gFLim (float)	floating point values controlling cyclic degradation model for strength degradation
gE (float)	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.
dmgType (str)	string to indicate type of damage (option: 'cycle', 'energy')

See also:

Notes

Engineered Cementitious Composites Material

uniaxialMaterial (*'ECC01'*, *matTag*, *sigt0*, *epst0*, *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
epst2 (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=0*, *epsBear=0*, *rBear=k1*)

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 epsSlip = 0, there will be no slippage)
epsBear (float)	Bearing Strain/Deformation (if epsBear = 0, there will be no bearing)
rBear (float)	Ratio of Bearing Stiffness to Initial Stiffness k1

See also:

Notes

Viscous Material

uniaxialMaterial (*'Viscous'*, *matTag*, *C*, *alpha*)

This command is used to construct a uniaxial viscous material object. stress =C(strain-rate)^alpha

matTag (int)	integer tag identifying material
C (float)	damping coeficient
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 stiffenss (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 grater 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)).

matTag (int)	integer tag identifying material
alpha (float)	ratio of post-yield stiffness to the initial elastic stiffenss (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 grater 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 BoucWen Material)
Ao (float)	parameter that controls tangent stiffness
q zetas p Shi deltaShi lambda (float)	parameters that control pinching
tol (float)	tolerance
maxIter (float)	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).

matTag (int)	integer tag identifying material
tp (str)	rubber type (see note 1)
ar (float)	area of rubber [unit: m^2] (see note 2)
hr (float)	total thickness of rubber [unit: m] (see note 2)
cg ch cu (float)	correction coefficients for equivalent shear modulus (cg), equivalent viscous daming ratio (ch), ratio of shear force at zero displacement (cu).
rs rf (float)	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^2] and [m], respectively.

3) `rs` and `rf` areavailable 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` = $\frac{1}{\sum_{i=0}^{n-1}\sin(\pi*i/n)}$, 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.

`matTag` (int)	integer tag identifying material
`type` (int)	rubber type (see note 1)
`ar` (float)	area of rubber [unit: m^2]
`hr` (float)	total thickness of rubber [unit: m]
`gr` (float)	shear modulus of rubber [unit: N/m^2]
`ap` (float)	area of lead plug [unit: m^2]
`tp` (float)	yield stress of lead plug [unit: N/m^2]
`alph` (float)	shear modulus of lead plug [unit: N/m^2]
`beta` (float)	ratio of initial stiffness to yielding stiffness
`temp` (float)	temperature [unit: °C]
`rk rq` (float)	reduction rate for yielding stiffness (`rk`) and force at zero displacement (`rq`)
`rs rf` (float)	reduction rate for stiffness (`rs`) and force (`rf`) (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^2], [N/m^2].
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` = $\frac{1}{\sum_{i=0}^{n-1}\sin(\pi*i/n)}$, 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.

matTag (int)	integer tag identifying material
sce (float)	compressive modulus
fty fcy (float)	yield stress under tension (fty) and compression (fcy) (see note 1)
bte bty bcy (float)	reduction rate for tensile elastic range (bte), tensile yielding (bty) and compressive yielding (bcy) (see note 1)
fcr (float)	target point stress (see note 1)

Note:

1. Input parameters are required to satisfy followings.

   fcy < 0.0 < fty

   0.0 <= bty < bte <= 1.0

   0.0 <= bcy <= 1.0

   fcy <= fcr <= 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.

matTag (int)	integer tag identifying material
sce (float)	compressive modulus
fty fcy (float)	yield stress under tension (fty) and compression (fcy) (see note 1)
bte bty bth bcy (float)	reduction rate for tensile elastic range (bte), tensile yielding (bty), tensile hardening (bth) and compressive yielding (bcy) (see note 1)
fcr (float)	target point stress (see note 1)
ath (float)	hardening strain ratio to yield strain

Note:

1. Input parameters are required to satisfy followings.

   fcy < 0.0 < fty

   0.0 <= bty < bth < bte <= 1.0

   0.0 <= bcy <= 1.0

   fcy <= fcr <= 0.0

   1.0 <= 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*, *crvTag*, *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 (int)	integer tag identifying material
nodeT (int)	integer node tag to define the first node at the extreme end of the associated flexural frame member (L3 or D5 in Figure)
nodeB (int)	integer node tag to define the last node at the extreme end of the associated flexural frame member (L2 or D2 in Figure)
driftAxis (int)	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) driftAxis = 1 - Drift along the x-axis driftAxis = 2 - Drift along the y-axis driftAxis = 3 - Drift along the z-axis
Kelas (float)	floating point value to define the initial material elastic stiffness (Kelastic); Kelas > 0
crvTyp (int)	integer flag to indicate the type of limit curve associated with this material. crvTyp = 0 - No limit curve crvTyp = 1 - axial limit curve crvTyp = 2 - RotationShearCurve
crvTag (int)	integer tag for the unique limit curve object associated with this material
YpinchLDN (float)	floating point unloading force pinching factor for loading in the negative direction. **Note: This value must be between zero and unity**
YpinchRDN (float)	floating point reloading force pinching factor for loading in the negative direction. **Note: This value must be between negative one and unity**
XpinchRDN (float)	floating point reloading displacement pinching factor for loading in the negative direction. **Note: This value must be between negative one and unity**
YpinchLDP (float)	floating point unloading force pinching factor for loading in the positive direction. **Note: This value must be between zero and unity**
YpinchRDP (float)	floating point reloading force pinching factor for loading in the positive direction. **Note: This value must be between negative one and unity**
XpinchRDP (float)	floating point reloading displacement pinching factor for loading in the positive direction. **Note: This value must be between negative one and unity**
dmgStrength (float)	floating point force limit for elastic stiffness damage (typically defined as the lowest of shear strength or shear at flexrual yielding). This value is used to compute the maximum deformation at flexural yield (δmax 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
dmgDisp (float)	floating point for ultimate drift at failure (δmax Eq. 1) and is used for strength and stiffness damage. 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 δmax. 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)	floating point elastic stiffness damage factors α1,α2,α3,α4 shown in Eq. 1
dmgE3 dmgE4 (float)	floating point elastic stiffness damage factors α1,α2,α3,α4 shown in Eq. 1
dmgELim (float)	floating point elastic stiffness damage limit Dlim shown in Eq. 1; **Note: This value must be between zero and unity**
dmgR1 dmgR2 (float)	floating point reloading stiffness damage factors α1,α2,α3,α4 shown in Eq. 1
dmgR3 dmgR4 (float)	floating point reloading stiffness damage factors α1,α2,α3,α4 shown in Eq. 1
dmgRLim (float)	floating point reloading stiffness damage limit Dlim shown in Eq. 1; **Note: This value must be between zero and unity**
dmgRCyc (float)	floating point cyclic reloading stiffness damage index; **Note: This value must be between zero and unity**
dmgS1 dmgS2	floating point backbone strength damage factors α1,α2,α3,α4 shown in Eq. 1

uniaxialMaterial (*'PinchingLimitStateMaterial'*, *matTag*, *dnodeT*, *nodeB*, *driftAxis*, *Kelas*, *crvTyp*, *crvTag*, *eleTag*, *b*, *d*, *h*, *a*, *st*, *As*, *Acc*, *ld*, *db*, *rhot*, *fc*, *fy*, *fyt*)

MODE 2: Calibrated Model for Shear-Critical Concrete Columns

matTag (int)	integer tag identifying material
nodeT (int)	integer node tag to define the first node at the extreme end of the associated flexural frame member (L3 or D5 in Figure)
nodeB (int)	integer node tag to define the last node at the extreme end of the associated flexural frame member (L2 or D2 in Figure)
driftAxis (int)	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) `driftAxis` = 1 - Drift along the x-axis `driftAxis` = 2 - Drift along the y-axis `driftAxis` = 3 - Drift along the z-axis
Kelas (float)	floating point value to define the shear stiffness (Kelastic) of the shear spring prior to shear failure `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 (cantelever) 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)
crvTag (int)	integer tag for the unique limit curve object associated with this material
eleTag (int)	integer element tag to define the associated beam-column element used to extract axial load
b (float)	floating point column width (inches)
d (float)	floating point column depth (inches)
h (float)	floating point column height (inches)
a (float)	floating point shear span length (inches)
st (float)	floating point transverse reinforcement spacing (inches) along column height
As (float)	floating point total area (inches squared) of longitudinal steel bars in section
Acc (float)	floating point gross confined concrete area (inches squared) bounded by the transverse reinforcement in column section
ld (float)	floating point development length (inches) of longitudinal bars using ACI 318-11 Eq. 12-1 and Eq. 12-2
db (float)	floating point diameter (inches) of longitudinal bars in column section
rhot (float)	floating point transverse reinforcement ratio (Ast/st.db)
f'c (float)	floating point concrete compressive strength (ksi)
fy (float)	floating point longitudinal steel yield strength (ksi)
fyt (float)	floating point transverse steel yield strength (ksi)

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.

`matTag` (int)	integer tag identifying material
`height` (float)	SWP's height (mm)
`width` (float)	SWP's width (mm)
`fut` (float)	Tensile strength of framing members (MPa)
`tf` (float)	Framing thickness (mm)
`Ife` (float)	Moment of inertia of the double end-stud (mm4)
`Ifi` (float)	Moment of inertia of the intermediate stud (mm4)
`ts` (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)
`nc` (float)	Total number of screws located on the SWP perimeter
`type` (int)	Integer identifier used to define wood sheathing type (DFP=1, OSB=2, CSP=3)
`openingArea` (float)	Total area of openings (mm2)
`openingLength` (float)	Cumulative length of openings (mm)

See also:

Notes

CFSSSWP Steel-Sheathed Cold-formed Steel Shear Wall Panel

uniaxialMaterial (*'CFSSSWP'*, *matTag*, *height*, *width*, *fuf*, *fyf*, *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)
`fyf` (float)	Yield strength of framing members (MPa)
`tf` (float)	Framing thickness (mm)
`Af` (float)	Framing cross section area (mm2)
`fus` (float)	Tensile strength of steel sheet sheathing (MPa)
`fys` (float)	Yield strength of steel sheet sheathing (MPa)
`ts` (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 (mm2)
`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)
```

Standard Models

The following contain information about available `matType`:

1. *ElasticIsotropic*
2. *ElasticOrthotropic*
3. *J2Plasticity*
4. *DruckerPrager*
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*, *nu*, *rho=0.0*)
This command is used to construct an ElasticIsotropic material object.

`matTag` (int)	integer tag identifying material
`E` (float)	elastic modulus
`nu` (float)	Poisson's ratio
`rho` (float)	mass density (optional)

The material formulations for the ElasticIsotropic object are:

- `'ThreeDimensional'`
- `'PlaneStrain'`
- `'Plane Stress'`
- `'AxiSymmetric'`
- `'PlateFiber'`

ElasticOrthotropic

nDMaterial (*'ElasticOrthotropic'*, *matTag*, *Ex*, *Ey*, *Ez*, *nu_xy*, *nu_yz*, *nu_zx*, *Gxy*, *Gyz*, *Gzx*, *rho=0.0*)
This command is used to construct an ElasticOrthotropic material object.

`matTag` (int)	integer tag identifying material
`Ex` (float)	elastic modulus in x direction
`Ey` (float)	elastic modulus in y direction
`Ez` (float)	elastic modulus in z direction
`nu_xy` (float)	Poisson's ratios in x and y plane
`nu_yz` (float)	Poisson's ratios in y and z plane
`nu_zx` (float)	Poisson's ratios in z and x plane
`Gxy` (float)	shear modulii in x and y plane
`Gyz` (float)	shear modulii in y and z plane
`Gzx` (float)	shear modulii in z and x plane
`rho` (float)	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

$$\sigma = K * trace(\epsilon_e) + (2 * G) * dev(\epsilon_e)$$

Yield Function

$$\phi(\sigma, q) = ||dev(\sigma)|| - \sqrt{(\frac{2}{3} * q(x_i))}$$

Saturation Isotropic Hardening with linear term

$$q(x_i) = \sigma_0 + (\sigma_\infty - \sigma_0) * exp(-delta * \xi) + H * \xi$$

Flow Rules

$$\dot{\epsilon_p} = \gamma * \frac{\partial \phi}{\partial \sigma}$$
$$\dot{\xi} = -\gamma * \frac{\partial \phi}{\partial q}$$

Linear Viscosity

$$\gamma = \frac{\phi}{\eta}(if\phi > 0)$$

Backward Euler Integration Routine Yield condition enforced at time n+1

set η = 0 for rate independent case

DruckerPrager

nDMaterial (*'DruckerPrager'*, *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, $0 \leq rhoBar \leq rho$.
`Kinf` (float)	nonlinear isotropic strain hardening parameter, $Kinf \geq 0$.
`Ko` (float)	nonlinear isotropic strain hardening parameter, $Ko \geq 0$.
`delta1` (float)	nonlinear isotropic strain hardening parameter, $delta1 \geq 0$.
`delta2` (float)	tension softening parameter, $delta2 \geq 0$.
`H` (float)	linear hardening parameter, $H \geq 0$.
`theta` (float)	controls relative proportions of isotropic and kinematic hardening, $0 \leq theta \leq 1$.
`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.

matTag (int)	integer tag identifying material
fcc (float)	concrete compressive strength, negative real value (positive input is changed in sign automatically)
fct (float)	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)
E (float)	optional Young modulus, $57000 * sqrt(abs(fcc))$ if $abs(fcc) > 2000$ because fcc is assumed in psi (see ACI 318)
ni (float)	optional Poisson coefficient, 0.15 (from comparison with tests by Kupfer Hilsdorf Rusch 1969)
Gt (float)	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)
Gc (float)	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)
rho_bar (float)	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)
H (float)	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)
theta (float)	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)
tangent (float)	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)

The material formulations for the Damage2p object are:

- `'ThreeDimensional'`
- `'PlaneStrain'`
- `'Plane Stress'`
- `'AxiSymmetric'`
- `'PlateFiber'`

See also here

PlaneStress

nDMaterial (*'PlaneStress'*, *matTag*, *mat3DTag*)

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.

matTag (int)	integer tag identifying material
mat3DTag (int)	tag of perviously defined 3d ndMaterial material

The material formulations for the PlaneStress object are:

- `'Plane Stress'`

PlaneStrain

nDMaterial (*'PlaneStrain'*, *matTag*, *mat3DTag*)
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
`mat3DTag` (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*, *sXTag*, *sYTag*, *concTag*, *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
`sXTag` (int)	Tag of uniaxialMaterial simulating horizontal (x) reinforcement
`sYTag` (int)	Tag of uniaxialMaterial simulating vertical (y) reinforcement
`concTag` (int)	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*, *D_r*, *G_o*, *h_po*, *Den*, *P_atm*, *h_o*, *e_max*, *e_min*, *n_b*, *n_d*, *A_do*, *z_max*, *c_z*, *c_e*, *phi_cv*, *nu*, *g_degr*, *c_dr*, *c_kaf*, *Q_bolt*, *R_bolt*, *m_par*, *F_sed*, *p_sed*)

This command is used to construct a 2-dimensional PM4Sand material.

matTag (int)	integer tag identifying material
D_r (float)	Relative density, in fraction
G_o (float)	Shear modulus constant
h_po (float)	Contraction rate parameter
Den (float)	Mass density of the material
P_atm (float)	Optional, Atmospheric pressure
h_o (float)	Optional, Variable that adjusts the ratio of plastic modulus to elastic modulus
e_max (float)	Optional, Maximum and minimum void ratios
e_min (float)	Optional, Maximum and minimum void ratios
n_b (float)	Optional, Bounding surface parameter, $n_b \geq 0$
n_d (float)	Optional, Dilatancy surface parameter $n_d \geq 0$
A_do (float)	Optional, Dilatancy parameter, will be computed at the time of initialization if input value is negative
z_max (float)	Optional, Fabric-dilatancy tensor parameter
c_z (float)	Optional, Fabric-dilatancy tensor parameter
c_e (float)	Optional, Variable that adjusts the rate of strain accumulation in cyclic loading
phi_cv (float)	Optional, Critical state effective friction angle
nu (float)	Optional, Poisson's ratio
g_degr (float)	Optional, Variable that adjusts degradation of elastic modulus with accumulation of fabric
c_dr (float)	Optional, Variable that controls the rotated dilatancy surface
c_kaf (float)	Optional, Variable that controls the effect that sustained static shear stresses have on plastic modulus
Q_bolt (float)	Optional, Critical state line parameter
R_bolt (float)	Optional, Critical state line parameter
m_par (float)	Optional, Yield surface constant (radius of yield surface in stress ratio space)
F_sed (float)	Optional, Variable that controls the minimum value the reduction factor of the elastic moduli can get during reconsolidation
p_sed (float)	Optional, Mean effective stress up to which reconsolidation strains are enhanced

The material formulations for the PM4Sand object are:

- 'PlaneStrain'

See als 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 (*'stressDensity'*, *matTag*, *mDen*, *eNot*, *A*, *n*, *nu*, *a1*, *b1*, *a2*, *b2*, *a3*, *b3*, *fd*, *muNot*, *muCyc*, *sc*, *M*, *patm*, **ssls*, *hsl*, *p1*)

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).

`matTag` (int)	integer tag identifying material
`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 ($etaMax = a1 + b1 * Is$)
`b1` (float)	peak stress ratio coefficient ($etaMax = a1 + b1 * Is$)
`a2` (float)	max shear modulus coefficient ($Gn_max = a2 + b2 * Is$)
`b2` (float)	max shear modulus coefficient ($Gn_max = a2 + b2 * Is$)
`a3` (float)	min shear modulus coefficient ($Gn_min = a3 + b3 * Is$)
`b3` (float)	min shear modulus coefficient ($Gn_min = a3 + b3 * Is$)
`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)
`ssls` (list (float))	void ratio of quasi steady state (QSS-line) at pressures [pmin, 10kPa, 30kPa, 50kPa, 100kPa, 200kPa, 400kPa] (default = [0.877, 0.877, 0.873, 0.870, 0.860, 0.850, 0.833])
`hsl` (float)	void ratio of upper reference state (UR-line) for all pressures (default = 0.895)
`p1` (float)	pressure corresponding to ssl1 (default = 1.0 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.

matTag (int)	integer tag identifying material
K (float)	bulk module of the acoustic medium
rho (float)	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 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.

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
Mfc (float)	Stress ratio at failure in triaxial compression
dre1 (float)	Coefficient for reversible dilatancy generation
Mdc (float)	Stress ratio at which the reversible dilatancy sign changes
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
ein (float)	Initial void ratio
rho (float)	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 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*
4. *PlasticDamageConcretePlaneStress*

PlaneStressUserMaterial

nDMaterial (*'PlaneStressUserMaterial'*, *matTag*, *nstatevs*, *nprops*, *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.

`nstatevs` (int)	number of state/history variables (usually 40)
`nprops` (int)	number of material properties (usually 7)
`matTag` (int)	integer tag identifying material
`fc` (float)	concrete compressive strength at 28 days (positive)
`ft` (float)	concrete tensile strength (positive)
`fcu` (float)	concrete crushing strength (negative)
`epsc0` (float)	concrete strain at maximum strength (negative)
`epscu` (float)	concrete strain at crushing strength (negative)
`epstu` (float)	ultimate tensile strain (positive)
`stc` (float)	shear retention factor

PlateFromPlaneStress

nDMaterial (*'PlateFromPlaneStress'*, *matTag*, *pre_def_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.

`matTag` (int)	new integer tag identifying material deriving from pre-defined PlaneStress material
`pre_def_matTag` (int)	integer tag identifying PlaneStress material
`OutofPlaneModulus` (float)	shear modulus for out of plane stresses

PlateRebar

nDMaterial (*'PlateRebar'*, *matTag*, *pre_def_matTag*, *sita*)

This command is used to create the multi-dimensional reinforcement material.

`matTag` (int)	new integer tag identifying material deriving from pre-defined uniaxial material
`pre_def_matTag` (int)	integer tag identifying uniaxial material
`sita` (float)	define the angle of reinforcement layer, 90 (longitudinal), 0 (tranverse)

PlasticDamageConcretePlaneStress

nDMaterial (*'PlasticDamageConcretePlaneStress'*, *matTag*, *E*, *nu*, *ft*, *fc*, <*beta*, *Ap*, *An*, *Bn*>)
No documentation is available yet. If you have the manual, please let me know.

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 Wilely & 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 Wilely & Sons, Ltd, West Sussex, England.

Wrapper material for Initial State Analysis

1. *InitialStateAnalysisWrapper*
2. *Initial Stress Material*
3. *Initial Strain Material*

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.

`matTag` (int)	integer tag identifying material
`nDMatTag` (int)	the tag of the associated nDMaterial object
`nDim` (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.

Initial Stress Material

nDMaterial (*'InitStressNDMaterial'*, *matTag*, *otherTag*, *initStress*, *nDim*)

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 strain that corresponds to the initial stress will be calculated from the other material.

`matTag` (int)	integer tag identifying material
`otherTag` (float)	tag of the other material
`initStress` (float)	initial stress
`nDim` (int)	Number of dimensions (e.g. if plane strain nDim=2)

See also:

Notes

Initial Strain Material

nDMaterial (*'InitStrainNDMaterial'*, *matTag*, *otherTag*, *initStrain*, *nDim*)

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` (int)	integer tag identifying material
`otherTag` (int)	tag of the other material
`initStrain` (float)	initial strain
`nDim` (float)	Number of dimensions

See also:

Notes

UC San Diego soil models

1. *PressureIndependMultiYield*
2. *PressureDependMultiYield*
3. *PressureDependMultiYield02*
4. *PressureDependMultiYield03*

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)	(G_r) Reference low-strain shear modulus, specified at a reference mean effective confining pressure refPress of p'r (see below).
refBulkModul (float)	(B_r) 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.
peakShearStra (float)	(γ_{max}) 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)	(ϕ) Friction angle at peak shear strength in degrees, optional (default is 0.0).
refPress (float)	(p'_r) Reference mean effective confining pressure at which G_r, B_r, and γ_{max} are defined, optional (default is 100. kPa).
pressDependCoe (float)	(d) A positive constant defining variations of G and B as a function of instantaneous effective confinement p' (default is 0.0) $G = G_r(\frac{p'}{p'_r})^d$ $B = B_r(\frac{p'}{p'_r})^d$ 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 (Gs) 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, noYieldSurf=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.

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)	(G_r) Reference low-strain shear modulus, specified at a reference mean effective confining pressure refPress of p'r (see below).
refBulkModul (float)	(B_r) Reference bulk modulus, specified at a reference mean effective confining pressure refPress of p'r (see below).
frictionAng (float)	(ϕ) Friction angle at peak shear strength in degrees, optional (default is 0.0).
peakShearStra (float)	(γ_{max}) Maximum 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).
refPress (float)	(p'_r) Reference mean effective confining pressure at which G_r, B_r, and γ_{max} are defined, optional (default is 100. kPa).
pressDependCoe (float)	(d) A positive constant defining variations of G and B as a function of instantaneous effective confinement p' (default is 0.0) $G = G_r(\frac{p'}{p'_r})^d$ $B = B_r(\frac{p'}{p'_r})^d$ If $\phi = 0$, d is reset to 0.0.
PTAng (float)	(ϕ_{PT}) Phase transformation angle, in degrees.
contrac (float)	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.
dilat (list (float))	Non-negative constants defining the rate of shear-induced volume increase (dilation). Larger values correspond to stronger dilation rate. `dilat = [dilat1, dilat2]`.
liquefac (list (float))	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 γ_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 (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))	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 (float)	Initial void ratio, optional (default is 0.6).
params (list (float))	`params=[cs1, cs2, cs3, pa]` defining a straight critical-state line ec in e-p' space. If cs3=0, ec = cs1-cs2 log(p'/pa) else (Li and Wang, JGGE, 124(12)), ec = cs1-cs2(p'/pa)cs3 where pa is atmospheric pressure for normalization (typically 101 kPa in SI units or 14.65 psi in English units). All four constants are optional
c (float)	Numerical constant (default value = 0.3 kPa)

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_σ 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
`nd` (float)	Number of dimensions, 2 for plane-strain, and 3 for 3D analysis.
`rho` (float)	Saturated soil mass density.
`refShearModul` (float)	(G_r) Reference low-strain shear modulus, specified at a reference mean effective confining pressure refPress of p'r (see below).
`refBulkModul` (float)	(B_r) Reference bulk modulus, specified at a reference mean effective confining pressure refPress of p'r (see below).
`frictionAng` (float)	(phi) Friction angle at peak shear strength in degrees, optional (default is 0.0).
`peakShearStra` (float)	(γ_{max}) 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).
`refPress` (float)	(p'_r) Reference mean effective confining pressure at which G_r, B_r, and γ_{max} are defined, optional (default is 100. kPa).
`pressDependCoe` (float)	(d) A positive constant defining variations of G and B as a function of instantaneous effective confinement p' (default is 0.0) $G = G_r(\frac{p'}{p'_r})^d$ $B = B_r(\frac{p'}{p'_r})^d$ If $\phi = 0$, d is reset to 0.0.
`PTAng` (float)	(ϕ_{PT}) Phase transformation angle, in degrees.
`contrac[2]` (float)	A non-negative constant reflecting K_σ effect.
`dilat[2]` (float)	A non-negative constant reflecting K_σ 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

PressureDependMultiYield03

nDMaterial (*'PressureDependMultiYield03'*, *matTag*, *nd*, *rho*, *refShearModul*, *refBulkModul*, *frictionAng*, *peakShearStra*, *refPress*, *pressDependCoe*, *PTAng*, *ca*, *cb*, *cc*, *cd*, *ce*, *da*, *db*, *dc*, *noYieldSurf=20.0*, **yieldSurf=[]*, *liquefac1=1*, *liquefac2=0.*, *pa=101*, *s0=1.73*)

The reference for PressureDependMultiYield03 material: Khosravifar, A., Elgamal, A., Lu, J., and Li, J. [2018]. "A 3D model for earthquake-induced liquefaction triggering and post-liquefaction response." Soil Dynamics and Earthquake Engineering, 110, 43-52)

PressureDependMultiYield03 is modified from PressureDependMultiYield02 material to comply with the established guidelines on the dependence of liquefaction triggering to the number of loading cycles, effective overburden stress (Kσ), and static shear stress (Kα).

The explanations of parameters

See notes

UC San Diego Saturated Undrained soil

1. *FluidSolidPorousMaterial*

FluidSolidPorousMaterial

nDMaterial (*'FluidSolidPorous'*, *matTag*, *nd*, *soilMatTag*, *combinedBulkModul*, *pa=101.0*)

FluidSolidPorous material 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).
`combinedBulkModul` (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 (2.2x106 kPa (or 3.191x105 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.

`secType` (str)	section type
`secTag` (int)	section tag.
`secArgs` (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_mod*, *A*, *Iz*, *G_mod=None*, *alphaY=None*)

section (*'Elastic'*, *secTag*, *E_mod*, *A*, *Iz*, *Iy*, *G_mod*, *Jxx*, *alphaY=None*, *alphaZ=None*)

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_mod` (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_mod` (float)	Shear Modulus (optional for 2D analysis, required for 3D analysis)
`Jxx` (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*)
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

section (*'Fiber'*, *secTag*, *'-torsion'*, *torsionMatTag*)
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
`torsionMatTag` (int)	uniaxialMaterial tag assigned to the section for torsional response (can be nonlinear)

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.

Commands to generate all fibers:

1. *Fiber Command*
2. *Patch Command*
3. *Layer 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 circlular arc
`ang` (list (float))	starting and ending angle (optional)

Fiber Thermal Section

section (*'FiberThermal'*, *secTag*, *'-GJ'*, *GJ=0.0*)
This command create 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.

Commands to generate all fibers:

1. *Fiber Command*
2. *Patch Command*
3. *Layer Command*

NDFiber Section

section (*'NDFiber'*, *secTag*)

This commnand 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, Mz, Vy]`. Stress components 11, 12, and 13 lead to all six stress resultants in 3D `[P, Mz, Vy, My, Vz, 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*, *coreMatTag*, *coverMatTag*, *steelMatTag*, *d*, *b*, *cover_depth*, *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
`coreMatTag` (int)	tag of uniaxialMaterial assigned to each fiber in the core region
`coverMatTag` (int)	tag of uniaxialMaterial assigned to each fiber in the cover region
`steelMatTag` (int)	tag of uniaxialMaterial assigned to each reinforcing bar
`d` (float)	section depth
`b` (float)	section width
`cover_depth` (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*, *coreMatTag*, *coverMatTag*, *steelMatTag*, *d*, *cover_depth*, *As*, *NringsCore*, *NringsCover*, *Newedges*, *Nsteel*, *'-GJ'*, *GJ*)

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
`coreMatTag` (int)	tag of uniaxialMaterial assigned to each fiber in the core region
`coverMatTag` (int)	tag of uniaxialMaterial assigned to each fiber in the cover region
`steelMatTag` (int)	tag of uniaxialMaterial assigned to each reinforcing bar
`d` (float)	section radius
`cover_depth` (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
`GJ` (float)	GJ stiffness

Note: For more general reinforced concrete section definitions, use the Fiber Section command.

Parallel Section

section (*'Parallel'*, *secTag*, **SecTags*)
Connect sections in parallel.

secTag (int)	unique section tag
SecTags (list (int))	tags of 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)	list of tags and dofs of previously-defined UniaxialMaterial objects, `mats = [matTag1,dof1, matTag2,dof2,...]` the force-deformation quantity to be modeled by this section object. One of the following section dof may be used: • `'P'` Axial force-deformation • `'Mz'` Moment-curvature about section local z-axis • `'Vy'` Shear force-deformation along section local y-axis • `'My'` Moment-curvature about section local y-axis • `'Vz'` Shear force-deformation along section local z-axis • `'T'` Torsion Force-Deformation
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` (int)	unique section tag
`matTag` (int)	tag of uniaxial material
`quantity` (str)	the force-deformation quantity to be modeled by this section object. One of the following section dof may be used: • `'P'` Axial force-deformation • `'Mz'` Moment-curvature about section local z-axis • `'Vy'` Shear force-deformation along section local y-axis • `'My'` Moment-curvature about section local y-axis • `'Vz'` Shear force-deformation along section local z-axis • `'T'` Torsion Force-Deformation

Elastic Membrane Plate Section

`section` (*'ElasticMembranePlateSection'*, *secTag*, *E_mod*, *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` (int)	unique section tag
`E_mod` (float)	Young's Modulus
`nu` (float)	Poisson's Ratio
`h` (float)	depth of section
`rho` (float)	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` (int)	unique section tag
`matTag` (int)	nDMaterial tag to be assigned to each fiber
`h` (float)	plate thickness

Bidirectional Section

`section` (*'Bidirectional'*, *secTag*, *E_mod*, *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.

secTag (int)	unique section tag
E_mod (float)	elastic modulus
Fy (float)	yield force
Hiso (float)	isotropic hardening modulus
Hkin (float)	kinematic hardening modulus
code1 (str)	section force code for direction 1 (optional)
code2 (str)	section force code for direction 2 (optional) One of the following section code may be used: • 'P' Axial force-deformation • 'Mz' Moment-curvature about section local z-axis • 'Vy' Shear force-deformation along section local y-axis • 'My' Moment-curvature about section local y-axis • 'Vz' Shear force-deformation along section local z-axis • 'T' Torsion Force-Deformation

Isolator2spring Section

section (*'Isolator2spring'*, *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.

secTag (int)	unique section tag
tol (float)	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.
k1 (float)	initial stiffness for lateral force-deformation
Fyo (float)	nominal yield strength for lateral force-deformation
k2o (float)	nominal postyield stiffness for lateral force-deformation
kvo (float)	nominal stiffness in the vertical direction
hb (float)	total height of elastomeric bearing
PE (float)	Euler Buckling load for the bearing
Po (float)	axial load at which nominal yield strength is achieved (optional)

LayeredShell

section (*'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` (int)	unique tag among sections
`nLayers` (int)	total numbers of layers
`mats` (list)	a list of material tags and thickness, `[[mat1,thk1], ..., [mat2,thk2]]`

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` (str)	frictionModel type
`frnTag` (int)	frictionModel tag.
`frnArgs` (list)	a list of frictionModel arguments, must be preceded with `*`.

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` (int)	unique friction model tag
`mu` (float)	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).

`frnTag` (int)	unique friction model tag
`muSlow` (float)	coefficient of friction at low velocity
`muFast` (float)	coefficient of friction at high velocity
`transRate` (float)	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.

`frnTag` (int)	unique friction model tag
`aSlow` (float)	constant for coefficient of friction at low velocity
`nSlow` (float)	exponent for coefficient of friction at low velocity
`aFast` (float)	constant for coefficient of friction at high velocity
`nFast` (float)	exponent for coefficient of friction at high velocity
`alpha0` (float)	constant rate parameter coefficient
`alpha1` (float)	linear rate parameter coefficient
`alpha2` (float)	quadratic rate parameter coefficient
`maxMuFact` (float)	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 μFast, 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'*, **velPoints*, *'-frn'*, **frnPoints*)

This command is used to construct a VelDepMultiLinear friction model object. The friction-velocity relationship is given by a multi-linear curve that is define 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
`velPoints` (list (float))	list of velocity points along friction-velocity curve
`frnPoints` (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.

transfTag (int)	integer tag identifying transformation
vecxz (list (float))	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. 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 (list (float))	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).
dJ (list (float))	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).

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.

transfTag (int)	integer tag identifying transformation
vecxz (list (float))	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. 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 (list (float))	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).
dJ (list (float))	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).

Note: P LARGE Delta effects do not include P small delta effects.

Corotational Transformation

geomTransf (*'Corotational'*, *transfTag*, *'-jntOffset'*, **dI*, **dJ*)

geomTransf (*'Corotational'*, *transfTag*, **vecxz*)

This command is used to construct the Corotational Coordinate Transformation (CorotCrdTransf) object. Corotational transformation can be used in large displacement-small strain problems.

transTag (int)	integer tag identifying transformation
vecxz (list (float))	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. 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 (list (float))	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).
dJ (list (float))	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).

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.

`constraintType` (str)	constraints type
`constraintArgs` (list)	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-pont 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:

`alphaS` (float)	α_S factor on single points.
`alphaM` (float)	α_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)	α_S factor on single points.
`alphaM` (float)	α_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.

`systemType` (str)	system type
`systemArgs` (list)	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 actal 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) • 0: AMD • 1: set by user • 2: AMF • 3: SCOTCH • 4: PORD • 5: Metis • 6: AMD with QADM • 7: automatic

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.

`testType` (str)	test type
`testArgs` (list)	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=maxIncr*)
Create a NormUnbalance test, which uses the 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): • 0 print nothing. • 1 print information on norms each time `test()` is invoked. • 2 print information on norms and number of iterations at end of successful test. • 4 at each step it will print the norms and also the ΔU and $R(U)$ vectors. • 5 if it fails to converge at end of `numIter` it will print an error message **but return a successfull test**.
`nType` (int)	Type of norm, (0 = max-norm, 1 = 1-norm, 2 = 2-norm). (optional)
`maxIncr` (int)	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.

`tol` (float)	Tolerance criteria used to check for convergence.
`iter` (int)	Max number of iterations to check
`pFlag` (int)	Print flag (optional): • 0 print nothing. • 1 print information on norms each time `test()` is invoked. • 2 print information on norms and number of iterations at end of successful test. • 4 at each step it will print the norms and also the ΔU and $R(U)$ vectors. • 5 if it fails to converge at end of `numIter` it will print an error message **but return a successfull test**.
`nType` (int)	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.

`tol` (float)	Tolerance criteria used to check for convergence.
`iter` (int)	Max number of iterations to check
`pFlag` (int)	Print flag (optional): • 0 print nothing. • 1 print information on norms each time `test()` is invoked. • 2 print information on norms and number of iterations at end of successful test. • 4 at each step it will print the norms and also the ΔU and $R(U)$ vectors. • 5 if it fails to converge at end of `numIter` it will print an error message **but return a successfull test**.
`nType` (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.

`tol` (float)	Tolerance criteria used to check for convergence.
`iter` (int)	Max number of iterations to check
`pFlag` (int)	Print flag (optional): • 0 print nothing. • 1 print information on norms each time `test()` is invoked. • 2 print information on norms and number of iterations at end of successful test. • 4 at each step it will print the norms and also the ΔU and $R(U)$ vectors. • 5 if it fails to converge at end of `numIter` it will print an error message **but return a successfull test**.
`nType` (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): • 0 print nothing. • 1 print information on norms each time `test()` is invoked. • 2 print information on norms and number of iterations at end of successful test. • 4 at each step it will print the norms and also the ΔU and $R(U)$ vectors. • 5 if it fails to converge at end of `numIter` it will print an error message **but return a successfull test**.
`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): • 0 print nothing. • 1 print information on norms each time `test()` is invoked. • 2 print information on norms and number of iterations at end of successful test. • 4 at each step it will print the norms and also the ΔU and $R(U)$ vectors. • 5 if it fails to converge at end of `numIter` it will print an error message **but return a successfull test**.
`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): • 0 print nothing. • 1 print information on norms each time `test()` is invoked. • 2 print information on norms and number of iterations at end of successful test. • 4 at each step it will print the norms and also the ΔU and $R(U)$ vectors. • 5 if it fails to converge at end of `numIter` it will print an error message **but return a successfull test**.
`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): • 0 print nothing. • 1 print information on norms each time `test()` is invoked. • 2 print information on norms and number of iterations at end of successful test. • 4 at each step it will print the norms and also the ΔU and $R(U)$ vectors. • 5 if it fails to converge at end of `numIter` it will print an error message **but return a successfull test**.
`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.

`tolIncr` (float)	Tolerance for left hand solution increments
`tolIncr` (float)	Tolerance for right hand residual
`iter` (int)	Max number of iterations to check
`pFlag` (int)	Print flag (optional): • 0 print nothing. • 1 print information on norms each time `test()` is invoked. • 2 print information on norms and number of iterations at end of successful test. • 4 at each step it will print the norms and also the ΔU and $R(U)$ vectors. • 5 if it fails to converge at end of `numIter` it will print an error message **but return a successfull test**.
`nType` (int)	Type of norm, (0 = max-norm, 1 = 1-norm, 2 = 2-norm). (optional)
`maxincr` (int)	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.

`tolIncr` (float)	Tolerance for left hand solution increments
`tolIncr` (float)	Tolerance for right hand residual
`iter` (int)	Max number of iterations to check
`pFlag` (int)	Print flag (optional): • 0 print nothing. • 1 print information on norms each time `test()` is invoked. • 2 print information on norms and number of iterations at end of successful test. • 4 at each step it will print the norms and also the ΔU and $R(U)$ vectors. • 5 if it fails to converge at end of `numIter` it will print an error message **but return a successfull test**.
`nType` (int)	Type of norm, (0 = max-norm, 1 = 1-norm, 2 = 2-norm). (optional)
`maxincr` (int)	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.

`Bisection` (bool)	Flag to use Bisection line search. (optional)
`Secant` (bool)	Flag to use Secant line search. (optional)
`RegulaFalsi` (bool)	Flag to use RegulaFalsi line search. (optional)
`InitialInterpolated` (bool)	Flag to use InitialInterpolated line search.(optional)
`tol` (float)	Tolerance for search. (optional)
`maxIter` (float)	Max num of iterations to try. (optional)
`minEta` (float)	Min η value. (optional)
`maxEta` (float)	Max η 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.

`secant` (bool)	Flag to indicate to use secant stiffness. (optional)
`initial` (bool)	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.

`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)

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.

`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)

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

`intType` (str)	integrator type
`intArgs` (list)	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.

`incr` (float)	Load factor increment λ.
`numIter` (int)	Number of iterations the user would like to occur in the solution algorithm. (optional)
`minIncr` (float)	Min stepsize the user will allow λ_{min}. (optional)
`maxIncr` (float)	Max stepsize the user will allow λ_{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'*, *nodeTag*, *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.

`nodeTag` (int)	tag of node whose response controls solution
`dof` (int)	Degree of freedom at the node, 1 through ndf.
`incr` (float)	First displacement increment ΔU_{dof}.
`numIter` (int)	Number of iterations the user would like to occur in the solution algorithm. (optional)
`minIncr` (float)	Min stepsize the user will allow ΔU_{min}. (optional)
`maxIncr` (float)	Max stepsize the user will allow ΔU_{max}. (optional)

Parallel DisplacementControl

integrator (*'ParallelDisplacementControl'*, *nodeTag*, *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.

`nodeTag` (int)	tag of node whose response controls solution
`dof` (int)	Degree of freedom at the node, 1 through ndf.
`incr` (float)	First displacement increment ΔU_{dof}.
`numIter` (int)	Number of iterations the user would like to occur in the solution algorithm. (optional)
`minIncr` (float)	Min stepsize the user will allow ΔU_{min}. (optional)
`maxIncr` (float)	Max stepsize the user will allow ΔU_{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.

`dlambda1` (float)	First load increment (pseudo-time step) at the first iteration in the next invocation of the analysis command.
`Jd` (int)	Factor relating first load increment at subsequent time steps. (optional)
`minLambda` (float)	Min load increment. (optional)
`maxLambda` (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.

`s` (float)	The arcLength.
`alpha` (float)	α a scaling factor on the reference loads.

Transient integrator objects

1. *Central Difference*
2. *Newmark Method*
3. *Hilber-Hughes-Taylor Method*

4. *Generalized Alpha Method*
5. *TRBDF2*
6. *Explicit Difference*
7. *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, $\frac{\Delta t}{T_n} < \frac{1}{\pi}$

Newmark Method

integrator (*'Newmark'*, *gamma*, *beta*, *'-form'*, *form*)
Create a Newmark integrator.

`gamma` (float)	γ factor.
`beta` (float)	β factor.
`form` (str)	Flag to indicate which variable to be used as primary variable (optional) • `'D'` – displacement (default) • `'V'` – velocity • `'A'` – acceleration

1. If the accelerations are chosen as the unknowns and β 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 $\beta >= \frac{\gamma}{2} >= \frac{1}{4}$

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.

`alpha` (float)	α factor.
`gamma` (float)	γ factor. (optional)
`beta` (float)	β factor. (optional)

1. Like Mewmark 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. α = 1.0 corresponds to the Newmark method.
3. α should be between 0.67 and 1.0. The smaller the α the greater the numerical damping.
4. γ and β are optional. The default values ensure the method is second order accurate and unconditionally stable when α is $\frac{2}{3} <= \alpha <= 1.0$. 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. Δt^2. Depending on choices of input parameters, the method can be unconditionally stable.

`alphaM` (float)	α_M factor.
`alphaF` (float)	α_F factor.
`gamma` (float)	γ factor. (optional)
`beta` (float)	β 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. α_M = 1.0, α_F = 1.0 produces the Newmark Method.
3. α_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 $\alpha_M >= \alpha_F >= \frac{1}{2}, \beta >= \frac{1}{4} + \frac{1}{2}(\gamma_M - \gamma_F)$
6. γ and β 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 *PFEM analysis*.

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.

numEigenvalues (int)	number of eigenvalues required
solver (str)	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

`numIncr` (int)	Number of analysis steps to perform. (required except for *PFEM analysis*)
`dt` (float)	Time-step increment. (required for Transient analysis and VariableTransient analysis.')
`dtMin` (float)	Minimum time steps. (required for VariableTransient analysis)
`dtMax` (float)	Maximum time steps (required for VariableTransient analysis)
`Jd` (float)	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 λ 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(*nodeTag*, *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 (int)	node tag.
dof (int)	specific dof of the response
responseID (int)	the id of responses: • Disp = 1 • Vel = 2 • Accel = 3 • IncrDisp = 4 • IncrDeltaDisp = 5 • Reaction = 6 • Unbalance = 7 • RayleighForces = 8

1.6.22 nodeVel command

nodeVel (*nodeTag*, *dof=-1*)
Returns the current velocity 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.

1.6.23 nodeUnbalance command

nodeUnbalance (*nodeTag*, *dof=-1*)
Returns the unbalanced force 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.

1.6.24 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. The printA command can only be issued after an analyze command.

filename (str)	name of file to which output is sent, by default, print to the screen. (optional)
'-ret' (str)	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.

`filename` (str)	name of file to which output is sent, by default, print to the screen. (optional)
`'-ret'` (str)	return the B vector as a list. (optional)

1.6.28 printGID command

`printGID` (*filename*, *'-append'*, *'-eleRange'*, *startEle*, *endEle*)
Print in GID format.

`filename` (str)	output file name.
`'-append'` (str)	append to existing file. (optional)
`startEle` (int)	start element tag. (optional)
`endEle` (int)	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.

`filename` (str)	name of file to which output is sent, by default, print to the screen. (optional)
`'-JSON'` (str)	print to a JSON file. (optional)
`'-node'` (str)	print node information. (optional)
`flag` (int)	integer flag to be sent to the print() method, depending on the node and element type (optional)
`nodes` (list (int))	a list of nodes tags to be printed, default is to print all, (optional)
`eles` (list (int))	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 commmand.
- -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.

`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.
`inetAddr` (str)	ip address, "xx.xx.xx.xx", of remote machine to which data is sent. (optional)
`port` (int)	port on remote machine awaiting tcp. (optional)
`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 ommitted, (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.
`resType` (list (str))	a string indicating response required. Response types are given in table below • `'disp'` displacement • `'vel'` velocity • `'accel'` acceleration • `'incrDisp'` incremental displacement • `'reaction'` nodal reaction • `'eigen i'` eigenvector for mode i • `'rayleighForces'` damping forces

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 quantaties.

`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 ommitted, (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.
`resType` (list (str))	a string indicating response required. Response types are given in table below • `'disp'` displacement • `'vel'` velocity • `'accel'` acceleration • `'incrDisp'` incremental displacement • `'reaction'` nodal reaction • `'eigen i'` eigenvector for mode i

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.

`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 ommitted, (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)
`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)	a region tag; to specify all nodes in the previously defined region. (optional)
`args` (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 seperate 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 ommitted, (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)
`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)	a 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 `filename.pvd` and `filename/` 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) • `'disp'` • `'vel'` • `'accel'` • `'incrDisp'` • `'reaction'` • `'pressure'` • `'unbalancedLoad'` • `'mass'` • `'eigen'`

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.

`filename` (str)	the name for `filename.pvd` and `filename/` 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) • `'disp'` • `'vel'` • `'accel'` • `'incrDisp'` • `'reaction'` • `'pressure'` • `'unbalancedLoad'` • `'mass'` • `'eigen'`

Collapse Recorder command

recorder (*'Collapse'*, *'-node'*, *nodeTag*, *'-file_infill'*, *fileNameinf*, *'-checknodes'*, *nTagbotn*, *nTagmidn*, *nTagtopn*, *'-global_gravaxis'*, *globgrav*, *'-secondary'*, *'-eles'*, **eleTags*, *'-eleRage'*, *start*, *end*, *'-region'*, *regionTag*, *'-time'*, *'-dT'*, *dT*, *'-file'*, *fileName*, *'-mass'*, **massValues*, *'-g'*, *gAcc*, *gDir*, *gPat*, *'-section'*, **secTags*, *'-crit'*, *critType*, *critValue*)

A progressive collapse algorithm is developed by Talaat, M and Mosalam, K.M. in [1-3] and is implemented in OpenSeesPy interpreter. The different applications of said algorithm are exemplified in references [4-7]. This algorithm is developed using element removal which relies on the dynamic equilibrium and the subsequent transient change in the system kinematics. The theoretical background of the routine is detailed in the references mentioned herein.

`nodeTag` (int)	node tag
`fileNameinf` (str)	is the file used to input the displacement interaction curve. Two columns of data are input in this file where only positive values are input. First column is the OOP displacement in ascending order and second column is the corresponding IP displacement. Full interaction should be defined. In other words, first value of OOP displacement and last value of IP displacement should be zero.
`fileName` (str)	is the file name for element removal log. Only one log file is constructed for all collapse recorder commands (i.e. for all removals). The first file name input to a collapse recorder command is used and any subsequent file names are ignored.
`globgrav` (float)	is the global axis of the model in the direction of gravity. 1, 2 and 3 should be input for X, Y and Z axes, respectively.
`critType` (list (str))	criterial type • `'INFILLWALL'` no value required • `'minStrain'` value required • `'maxStrain'` value required • `'axialDI'` value required • `'flexureDI'` value required • `'axialLS'` value required • `'shearLS'` value required

The progressive collapse algorithm is thus implemented within OpenSeesPy for an automatic removal of elements which have "numerically" collapse during an ongoing dynamic simulation. Main elements of the progressive collapse routine are illustrated in Figures 1 and 2. The implementation is supported in Python as a relatively new OpenSees module. Following each converged step of the dynamic analysis, the algorithm is called to check each element respectively for possible violation given a user-defined removal criteria. The routine calls for the activation of the element removal sequence before accessing the main analysis module on the subsequent analysis step. Activation of the element removal algorithm includes updating nodal masses, checking if the removal of the collapsed element results in leaving behind dangling nodes or floating elements, which must be removed as well and removing all associated element and nodal forces, imposed displacements, and constraints.

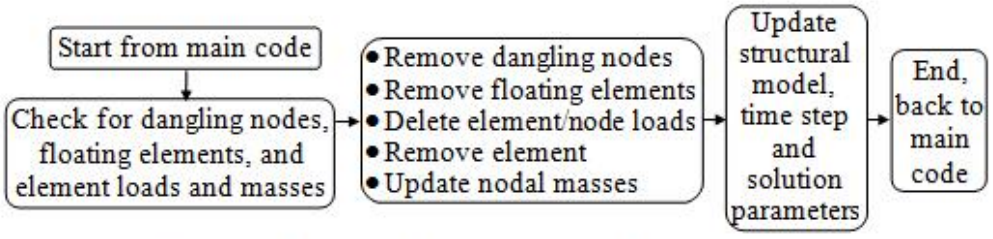


Figure 4. Considered element removal algorithm

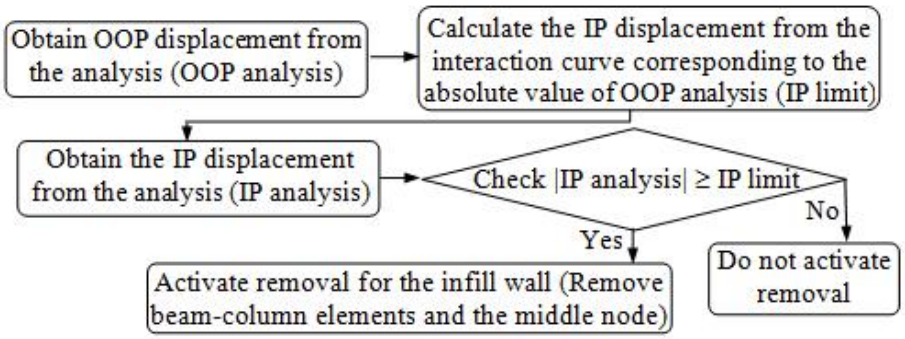


Figure 7. Algorithm of infill wall removal

Furthermore, the aforementioned infill wall element and its removal criteria are defined for force- and displacement-based distributed plasticity fiber elements and lumped plasticity beam–column elements with fiber-discretized plastic hinges. Current version of OpenSeesPy considers only the removal of the infill wall model described in (https://opensees.berkeley.edu/wiki/index.php/Infill_Wall_Model_and_Element_Removal#New_Command_in_ OpenSees_Interpreter). Implementation of the removal of the elements representing the aforementioned infill wall analytical model in the progressive collapse algorithm is performed through defining a removal criterion for the beam-column elements of this model. This criterion is based on the interaction between the in-plane (IP) and out-of-plane (OOP) displacements. IP displacement is the relative horizontal displacement between the top and bottom nodes of the diagonal element. OOP displacement is that of the middle node (where the OOP mass is attached) with respectto the chord which connects the top and bottom nodes. The user is free to choose any interaction relationship between IP and OOP displacements. In the example highlighted above, the interaction between in-plane and out-of-plane is taken into consideration with regards to the displacement interaction between the two mechanisms, where the IP and OOP displacement capacities are obtained using the FEMA 356 formulation for collapse prevention level. During the nonlinear time history simulation, when the mentioned combination of displacements from the analysis exceeds the interaction curve, the two beam-column elements and the middle node, representing the unreinforced masonry infill wall, are removed.

For the example illustrated in the next Figure, the existing Python command and its arguments in the OpenSeesPy interpreter with respect to the infill wall removal is described such that:

```
recorder('Collapse', '-ele', ele1, '-time', '-crit', 'INFILLWALL', '-file', filename,
↪'-file_infill', filenameinf, '-global_gravaxis, globgrav, '-checknodes', nodebot,␣
↪nodemid, nodetop)

recorder('Collapse', '-ele', ele2, '-time', '-crit', 'INFILLWALL', '-file', filename,
↪'-file_infill', filenameinf, '-global_gravaxis, globgrav, '-checknodes', nodebot,␣
↪nodemid, nodetop)

recorder('Collapse', '-ele', ele1, ele2, '-node', nodemid)
```

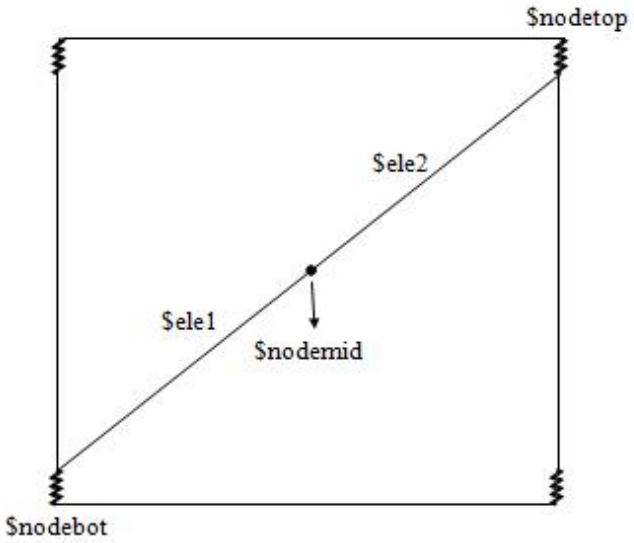


Figure 8: Representation of nodes and elements used in collapse recorder

Note: it might seem that node inputs are unnecessary. However, when there are shear springs in the model, nodetop and nodebot should be the nodes of the springs which connect to the beams, since the shear spring deformation contributes to the IP displacement of the infill wall. These nodes are not the nodes of the diagonal element. Therefore, it is necessary to input these nodes.

References

1. Talaat, M. and Mosalam, K.M. (2008), "Computational Modeling of Progressive Collapse in Reinforced Concrete Frame Structures", Pacific Earthquake Engineering Research Center, PEER 2007/10. 2. Talaat, M. and Mosalam, K.M. (2009), "Modeling Progressive Collapse in Reinforced Concrete Buildings Using Direct Element Removal", Earthquake Engineering and Structural Dynamics, 38(5): 609-634. 3. Talaat, M. and Mosalam, K.M. (2009), Chapter20: How to Simulate Column Collapse and Removal in As- built and Retrofitted Building Structures?, in Seismic Risk Assessment and Retrofitting - with special emphasis on existing low-rise structures, Ilki, A, Karadogan, F, Pala, S & Yuksel, E (Eds), ISBN 978-90-481-2680-4, Springer. 4. Talaat, M. and Mosalam, K.M. (2006), "Progressive Collapse Modeling of Reinforced Concrete Framed Structures Containing Masonry Infill Walls", Proceedings of the 2nd NEES/E-Defense Workshop on Collapse Simulation of Reinforced Concrete Building Structures, Kobe, Japan. 5. Talaat, M. and Mosalam, K.M. (2007), "Towards Modeling Progressive Collapse in Reinforced Concrete Buildings", Proceedings of SEI-ASCE 2007 Structures Congress, Long Beach, California, USA. 6. Mosalam, K.M., Talaat, M., and Park, S. (2008), "Modeling Progressive Collapse in Reinforced Concrete Framed Structures", Proceedings of the 14th World Conference on Earthquake Engineering, Beijing, China, October 12-17, Paper S15-018. 7. Mosalam, K.M., Park, S., Günay, M.S. (2009), "Evaluation of an Element Removal Algorithm for Reinforced Concrete Structures Using Shake Table Experiments," Proceedings of the 2nd International Conference on Computational Methods in structural Dynamics and Earthquake Engineering (COMPDYN 2009), Island of Rhodes, Greece, June 22-24.

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.

eleTag (int)	element tag.
secNum (int)	section number, i.e. the Gauss integratio number
dof (int)	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.

eleTag (int)	element tag.
secNum (int)	section number, i.e. the Gauss integratio number
dof (int)	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.

`filename` (str)	name of the log file
`'-append'` (str)	append to the file
`'-noEcho'` (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*

1.7.1 convertBinaryToText command

convertBinaryToText (*inputfile*, *outputfile*)
Convert binary file to text file

`inputfile` (str)	input file name.
`outputfile` (str)	output file name.

1.7.2 convertTextToBinary command

convertTextToBinary (*inputfile*, *outputfile*)
Convert text file to binary file

`inputfile` (str)	input file name.
`outputfile` (str)	output file name.

1.7.3 database command

database (*type*, *dbName*)
Create a database.

type (str)	database type: • 'File' - outputs database into a file • 'MySQL' - creates a SQL database • 'BerkeleyDB' - creates a BerkeleyDB database
dbName (str)	database name.

1.7.4 InitialStateAnalysis command

InitialStateAnalysis (*flag*)
Set the initial state analysis to 'on' or 'off'

flag (str)	'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 (float)	Time domain is to be set to (optional)

Note: Load Patterns added afer 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 (float)	damping factor.

1.7.7 reactions command

reactions (*'-dynamic'*, *'-rayleigh'*)
Calculate the reactions. Call this command before the *nodeReaction()*.

'-dynamic' (str)	Include dynamic effects.
'-rayleigh' (str)	Include rayleigh damping.

1.7.8 remove command

remove (*type*, *tag*)
This commmand is used to remove components from the model.

type (str)	type of the object, 'ele', 'loadPattern', 'parameter', 'node', 'timeSeries', 'sp', 'mp'.
tag (int)	tag of the object

remove (*'recorders'*)
Remove all recorder objects.

remove (*'sp'*, *nodeTag*, *dofTag*, *patternTag*)
Remove a sp object based on node

nodeTag (int)	node tag
dof (int)	dof the sp constrains
patternTag (int)	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 (int)	a tag identify the commit

1.7.11 save command

save (*commitTag*)
Save current state to database, which should be created through *database()*.

commitTag (int)	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` (float)	mass
`zeta` (float)	damping ratio
`k` (float)	stiffness
`Fy` (float)	yielding strength
`alpha` (float)	strain-hardening ratio
`dtF` (float)	time step for input data
`filename` (str)	input data file, one force per line
`dt` (float)	time step for analysis
`uresidual` (float)	residual displacement at the end of previous analysis (optional, default=0)
`umaxprev` (float)	previous displacement (optional, default=0)

The command returns a list of five response quantities.

`umax` (float)	maximum displacement during analysis
`u` (float)	displacement at end of analysis
`up` (float)	permanent residual displacement at end of analysis
`amax` (float)	maximum acceleration during analysis
`tamax` (float)	time when maximum accleration occurred

1.7.13 setTime command

`setTime` (*pseudoTime*)
This command is used to set the time in the Domain.

`pseudoTime` (float)	Time domain to be set

1.7.14 setNodeCoord command

`setNodeCoord` (*nodeTag*, *dim*, *value*)
set the nodal coodinate at the specified dimension.

`nodeTag` (int)	node tag.
`dim` (int)	the dimension of the coordinate to be set.
`value` (float)	coordinate value

1.7.15 setNodeDisp command

`setNodeDisp` (*nodeTag*, *dof*, *value*, *'-commit'*)
set the nodal displacement at the specified DOF.

`nodeTag` (int)	node tag.
`dof` (int)	the DOF of the displacement to be set.
`value` (float)	displacement value
`'-commit'` (str)	commit nodal state. (optional)

1.7.16 setNodeVel command

setNodeVel (*nodeTag*, *dof*, *value*, *'-commit'*)
set the nodal velocity at the specified DOF.

`nodeTag` (int)	node tag.
`dof` (int)	the DOF of the velocity to be set.
`value` (float)	velocity value
`'-commit'` (str)	commit nodal state. (optional)

1.7.17 setNodeAccel command

setNodeAccel (*nodeTag*, *dof*, *value*, *'-commit'*)
set the nodal acceleration at the specified DOF.

`nodeTag` (int)	node tag.
`dof` (int)	the DOF of the acceleration to be set.
`value` (float)	acceleration value
`'-commit'` (str)	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(*inputml*, *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

setNumThreads(*num*)
set the number of threads to be used in the multi-threaded environment.

num (int)	number of threades

1.7.28 getNumthread command

getNumThreads()
return the total number of threads available

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.

`tag` (int)	mesh tag.
`numnodes` (int)	number of nodes for defining consective lines.
`ndtags` (list (int))	the node tags
`id` (int)	mesh id. Meshes with same id are considered as same structure of fluid identity. • `id` = 0 : not in FSI • `id` > 0 : structure • `id` < 0 : fluid
`ndf` (int)	ndf for nodes to be created.
`meshsize` (float)	mesh size.
`eleType` (str)	the type of the element, (optional) • *Elastic Beam Column Element* • forceBeamColumn-Element • dispBeamColumn-Element if no type is given, only nodes are created
`eleArgs` (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, `eleArgs = ['elasticBeamColumn', A, E, Iz, transfTag]`

triangular mesh

mesh (*'tri'*, *tag*, *numlines*, **ltags*, *id*, *ndf*, *meshsize*, *eleType=''*, **eleArgs=[]*)
Create a triangular mesh object.

`tag` (int)	mesh tag.
`numlines` (int)	number of lines (*line mesh*) for defining a polygon.
`ltags` (list (int))	the *line mesh* tags
`id` (int)	mesh id. Meshes with same id are considered as same structure of fluid identity. • `id` = 0 : not in FSI • `id` > 0 : structure • `id` < 0 : fluid
`ndf` (int)	ndf for nodes to be created.
`meshsize` (float)	mesh size.
`eleType` (str)	the element type, (optional) • *PFEMElementBubble* • *PFEMElementCompressible* • *Tri31 Element* • *Elastic Beam Column Element* • *forceBeamColumn* • *dispBeamColumn* if no type is given, only nodes are created. if beam elements are given, beams are created instead of triangular elements.
`eleArgs` (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, `eleArgs = ['PFEMElementBubble', rho, mu, b1, b2, thickness, kappa]`

quad mesh

mesh (*'quad'*, *tag*, *numlines*, **ltags*, *id*, *ndf*, *meshsize*, *eleType=''*, **eleArgs=[]*)
Create a quad mesh object. The number of lines must be 4. These lines are continuous to form a loop.

`tag` (int)	mesh tag.
`numlines` (int)	number of lines (*line mesh*) for defining a polygon.
`ltags` (list (int))	the *line mesh* tags
`id` (int)	mesh id. Meshes with same id are considered as same structure of fluid identity. • `id` = 0 : not in FSI • `id` > 0 : structure • `id` < 0 : fluid
`ndf` (int)	ndf for nodes to be created.
`meshsize` (float)	mesh size.
`eleType` (str)	the element type, (optional) • *PFEMElementBubble* • *PFEMElementCompressible* • *Tri31 Element* • *Elastic Beam Column Element* • *forceBeamColumn* • *dispBeamColumn* • *Shell Element* if no type is given, only nodes are created. If beam elements are given, beams are created instead of quad elements. If triangular elements are given, they are created by dividing one quad to two triangles.
`eleArgs` (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, `eleArgs = ['PFEMElementBubble', rho, mu, b1, b2, thickness, kappa]`

tetrahedron mesh

mesh (*'tet'*, *tag*, *nummesh*, **mtags*, *id*, *ndf*, *meshsize*, *eleType=''*, **eleArgs=[]*)

Create a 3D tetrahedron mesh object.

`tag` (int)	mesh tag.
`nummesh` (int)	number of 2D mesh for defining a 3D body.
`mtags` (list (int))	the mesh tags
`id` (int)	mesh id. Meshes with same id are considered as same structure of fluid identity. • `id` = 0 : not in FSI • `id` > 0 : structure • `id` < 0 : fluid
`ndf` (int)	ndf for nodes to be created.
`meshsize` (float)	mesh size.
`eleType` (str)	the element type, (optional) • *FourNodeTetrahedron* if no type is given, only nodes are created.
`eleArgs` (list)	a list of element arguments. The arguments are same as in the element commands, but without element tag, and node tags. (optional)

particle mesh

mesh (*'part'*, *tag*, *type*, **pArgs*, *eleType=''*, **eleArgs=[]*, *'-vel'*, **vel0*, *'-pressure'*, *p0*)
Create or return a group of particles which will be used for background mesh.

`tag` (int)	mesh tag.
`type` (str)	type of the mesh
`pArgs` (list (float))	coordinates of points defining the mesh region `nx`, `ny`, `nz` are number of particles in x, y, and z directions • `'quad'` : [x1, y1, x2, y2, x3, y3, x4, y4, nx, ny] Coordinates of four corners in counter-clock wise order. • **`'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 • `'tri'` : [x1, y1, x2, y2, x3, y3, nx, ny] Coordinates of three corners in counter-clock wise order • `'line'` : [x1, y1, x2, y2, nx] Coordinates of two ends in counter-clock wise order • **`'pointlist'`** [[x1, y1, <z1>, vx1, vy1, <vz1>,] ax1, ay1, <az1>, p1, x2, y2, <z2>, vx2, vy2, <vz2>, ax2, ay2, <az2>, p2, ..] input particles' data in a list, in the order of coordinates of last time step, current coordinates, velocity, acceleration, and pressure. • `'pointlist'` without list return a list of current particles' data in this mesh [tag1, x1, y1, <z1>, vx1, vy1, <vz1>, ax1, ay1, <az1>, p1, tag2, x2, y2, <z2>, vx2, vy2, <vz2>, ax2, ay2, <az2>, p2, ..] The format is similar to the input list, but with an additional tag for each particle.
`eleType` (str)	the element type, (optional) • *PFEMElementBubble* • *PFEMElementCompressible* • *Tri31 Element* if no type is given, only nodes are created
`eleArgs` (list)	a list of element arguments. (optional, see *line mesh* and *triangular mesh*)
`vel0` (list (float))	a list of initial velocities. (optional)
`p0` (float)	initial pressure. (optional)

background mesh

mesh (*'bg'*, *basicsize*, **lower*, **upper*, *'-tol'*, *tol*, *'-meshtol'*, *meshtol*, *'-wave'*, *wavefilename*, *numl*, **locations*, *'-numsub'*, *numsub*, *'-structure'*, *id*, *numnodes*, **snodes*, *'-largeSize'*, *level*, **llower*, **lupper*)
Create a background mesh.

`basicsize` (float)	basic mesh size
`lower` (list (float))	a list of coordinates of the lower point of the background region.
`upper` (list (float))	a list of coordinates of the uuper point of the background region.
`tol` (float)	tolerance for intri check. (optional, default 1e-10)
`meshtol` (float)	tolerance for cell boundary check. (optional, default 0.1)
`wavefilename` (str)	a filename to record wave heights and velocities (optional)
`numl` (int)	number of locations to record wave (optional)
`locations` (list (float))	coordinates of the locations (optional)
`id` (int)	structural id > 0, same meaning as *triangular mesh* (optional)
`numsnodes` (int)	number of structural nodes (optional)
`sNodes` (list (int))	a list of structural nodes (optional)
`level` (int)	some regions can have larger mesh size with larger `level`. `level = 1` means same as basic mesh size.
`llower` (list (float))	a list of coordinates of the lower point of the region with larger mesh size (optional)
`lupper` (list (float))	a list of coordinates of the upper point of the region with larger mesh size(optional)

1.8.2 remesh command

remesh (*alpha=-1.0*)

- $\alpha \geq 0$ for updating moving mesh.
- $\alpha < 0$ for updating background mesh.

If there are nodes shared by different mesh in the domain, the principles to decide what element should be used for a triangle:

1. If all 3 nodes share the same mesh, use that mesh for the triangle.
2. If all 3 nodes share more than one mesh, use the mesh with `eleArgs` defined and lowest `id`.
3. If all 3 nodes are in different mesh, use the mesh with lowest `id`.
4. If the selected mesh `id` >= 0, skip the triangle.
5. If the selected mesh has no `eleArgs`, skip the triangle.

`alpha` (float)	Parameter for the α method to construct a mesh from the node cloud of moving meshes. (optional) • $\alpha = 0$: no elements are created • large α : all elements in the convex hull are created • $1.0 < \alpha < 2.0$: usually gives a good shape

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.

`tolv` (float)	Tolerance for velocity increments
`tolp` (float)	Tolerance for pressure increments
`tolrv` (float)	Tolerance for velocity residual
`tolrp` (float)	Tolerance for pressure residual
`tolrv` (float)	Tolerance for relative velocity increments
`tolrp` (float)	Tolerance for relative pressure increments
`iter` (int)	Max number of iterations to check
`maxincr` (int)	Max times for error increasing
`pFlag` (int)	Print flag (optional): • 0 print nothing. • 1 print information on norms each time `test()` is invoked. • 2 print information on norms and number of iterations at end of successful test. • 4 at each step it will print the norms and also the ΔU and $R(U)$ vectors. • 5 if it fails to converge at end of `numIter` it will print an error message **but return a successfull test**.
`nType` (int)	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.

`dtmax` (float)	Maximum time steps.
`dtmin` (float)	Mimimum time steps.
`gravity` (float)	Gravity acceleration used to move isolated particles.
`ratio` (float)	The ratio to reduce time steps if it was not converged. (optional)

1.9 Sensitivity Commands

These commands are for sensitivity analysis in OpenSees.

1.9.1 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.

`tag` (int)	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

newValue (float)	the updated value to which the parameter needs to be set.
eleTags (list (int))	a list of element tags
start (int)	start element tag
end (int)	end element tag
args (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, • '-computeAtEachStep' automatically compute at the end of each step • '-compuateByCommand' compute by calling computeGradients.

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.

eleTag (int)	element tag
secNum (int)	section number (optional)
dof (int)	specific dof at the element (1 through element force ndf)
paramTag (int)	parameter tag

1.9.14 sensNodePressure command

sensNodePressure (*nodeTag*, *paramTag*)
Returns the current pressure sensitivity to a parameter at a specified node.

nodeTag (int)	node tag
paramTag (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

`tag` (int)	random variable tag
`dist` (str)	random variable distribution • `'normal'` • `'lognormal'` • `'gamma'` • `'shiftedExponential'` • `'shiftedRayleigh'` • `'exponential'` • `'rayleigh'` • `'uniform'` • `'beta'` • `'type1LargestValue'` • `'type1SmallestValue'` • `'type2LargestValue'` • `'type3SmallestValue'` • `'chiSquare'` • `'gumbel'` • `'weibull'` • `'laplace'` • `'pareto'`
`mean` (float)	mean value
`stdv` (float)	standard deviation
`startPoint` (float)	starting point of the distribution
`params` (list (int))	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*
13. *partition command*

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.

`pid` (int)	ID of processor where data is sent to
`data` (list (int))	can be a list of integers
`data` (list (float))	can be a list of floats
`data` (str)	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 *mesh command*

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.

1.11.9 partition command

partition (*'-ncuts'*, *ncuts*, *'-niter'*, *niters*, *'-ufactor'*, *ufactor*, *'-info'*)
In a parallel environment, this command partitions the model. It requires that all processors have the exact same model to be partitioned.

`ncuts` (int)	Specifies the number of different partitionings that it will compute. The final partitioning is the one that achieves the best edge cut or communication volume. (Optional default is 1).
`niters` (int)	Specifies the number of iterations for the refinement algorithms at each stage of the uncoarsening process. (Optional default is 10).
`ufactor` (int)	Specifies the maximum allowed load imbalance among the partitions. (Optional default is 30, indicating a load imbalance of 1.03).
`'-info'` (str)	print information. (optional)

1.12 Preprocessing Commands

The *mesh command* and *remesh command* should be called as

```
import openseespy.opensees as ops
ops.mesh()
ops.remesh()
```

The *DiscretizeMember command* should be called as

```
import openseespy.preprocessing.DiscretizeMember as opsdm

opsdm.DiscretizeMember()
```

1. *mesh command*
2. *remesh command*
3. *DiscretizeMember command*

1.12.1 DiscretizeMember command

`preprocessing.DiscretizeMember.`**`DiscretizeMember`**(*ndI*, *ndJ*, *numEle*, *eleType*, *integrTag*, *transfTag*, *nodeTag*, *eleTag*)

Discretize beam elements between two nodes.

`ndI` (int)	node tag at I end
`ndJ` (int)	node tag at J end
`numEle` (int)	number of element to discretize
`eleType` (str)	the element type
`integrTag` (int)	beam integration tag (*beamIntegration commands*)
`transfTag` (int)	geometric transformation tag (*geomTransf commands*)
`nodeTag` (int)	starting node tag
`eleTag` (int)	starting element tag

1.13 Postprocessing Modules

1.13.1 Get_Rendering Commands

The source code is maintained by Anurag Upadhyay from University of Utah.

For Developers : If you are interested in developing visualization functions and contributing to `Get_Rendering` library, please go through *Visualization Development Guide* to understand the existing code. This will be helpful in reducing changes while merging the new codes. There is a dedicated branch of OpenSeesPy for developement of Get_Rendering library (https://github.com/u-anurag/OpenSeesPy). All the new code should be added to this dedicated branch, and not the main OpenSeesPy branch.

Model visualization is an ongoing development to make OpenSeesPy more user friendly. It utilizes Matplotlib 3.0 library to plot 2D and 3D models in a dedicated interactive window. You can use click-and-hold to change the view angle and zoom the plot. The model image can be saved with the desired orientation directly from the interactive plot window. If you did not install matplotlib using Anaconda, you will have to install PyQt or PySide to enable an interactive window (Matplotlib Dependencies).

Animation: To save the animation movie as .mp4 file, FFmpeg codecs are required.

When using Spyder IDE and Jupyter notebook, the default setting is to produce a static, inline plot which is not interactive. To change that, write the command **%matplotlib qt** in the Ipython console and then execute the model plotting commands. This will produce an interactive plot in a dedicated window.

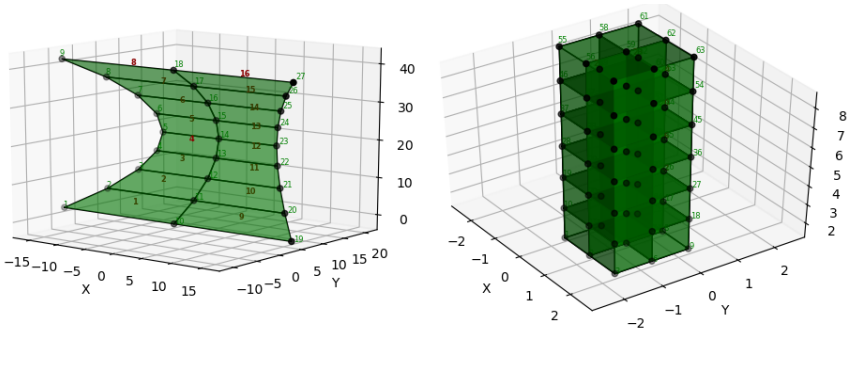
See the example *A Procedure to Render 2D or 3D OpenSees Model and Mode Shapes* for a sample script.

Following elements are supported:

- 2D and 3D Beam-Column Elements
- 2D and 3D Quad Elements
- 2D and 3D Tri Elements
- 8 Node Brick Elements
- Tetrahedron Elements (to be added)

The following two commands are needed to visualize the model, as shown below:

```
#Change plot backend to Qt. ONLY if you are using an Ipython console (e.g. Spyder)
%matplotlib qt

#Change plot backend to 'Nbagg' if using in Jupyter notebook to get an interactive,␣
↪inline plot.
%matplotlib notebook

# import OpenSeesPy rendering module
import openseespy.postprocessing.Get_Rendering as opsplt

# render the model after defining all the nodes and elements
opsplt.plot_model()

# plot mode shape
opsplt.plot_modeshape(3)
```



Following are commands and development guide related to model visualization:

1. *Create Output Database*
2. *saveFiberData2D command*
3. *plot_model command*
4. *plot_modeshape command*
5. *plot_deformedshape command*
6. *plot_fiberResponse2D command*
7. *animate_deformedshape command*
8. *animate_fiberResponse2D command*
9. *Plotting OpenSees Tcl Output*
10. *Visualization Development Guide*

Create Output Database

`postprocessing.Get_Rendering.`**`createODB`**(*ModelName*, *<LoadCaseName>*, *<Nmodes=0>*, *<deltaT=0.0>*, *<recorders=[]>*)

This command creates an Output Database for the active model with an option to save a specific load case output. The command **must** be called while the model is built, but before the main analysis is run. An output database with name ModelName_ODB is created in the current directory with node and element information in it. See the example below.

Note: The support to visualize stress, strain and element forces using `recorders=[]` argument in this command is not yet provided. These functionalities will be provided in the next release of the `Get_Rendering plotting` library. Input arguments are as follows,

`ModelName` (str)	Name of the model the user wants to save database with.
`LoadCaseName` (str)	Name of the subfolder to save load case output data.(Optional)
`Nmodes` (int)	Number of modes to be saved for visualization.(Optional)
`deltaT` (float)	Timesteps at which output to be saved. Default is 0.0. (optional)
`recorders` (list)	(NOT AVAILABLE YET) List of recorders to be saved for the loadcase. (optional)

Here is a simple example:

```
createODB("TwoSpan_Bridge", Nmodes=3)
```

The above command will create,

- a folder named **TwoSpan_Bridge_ODB** containing the information on the nodes and elements to visualize the structure in future without using a OpenSeesPy model file.
- a sub-folder named **ModeShapes** containing information on modeshapes and modal periods.
- no output from any loadcase will be saved saved in this case even if the analysis is run in the script since there is no argument for "LoadCaseName" is provided.

Here is another example command to be used when recording data from a load case.

```
createODB("TwoSpan_Bridge","Dynamic_GM1", Nmodes=3, deltaT=0.5)
```

The above command should be used right before running a load case analysis in the OpenSeesPy script and will create,

- a folder named **TwoSpan_Bridge_ODB** containing the information on the nodes and elements to visualize the structure in future without using a OpenSeesPy model file.
- a sub-folder named **Dynamic_GM1** containing the information on node displacement data to plot deformed shape.
- a sub-folder named **ModeShapes** containing information on modeshapes and modal periods.
- the node displacement data will be saved at closest time-steps at each 0.05 sec interval.

saveFiberData2D command

postprocessing.Get_Rendering.**saveFiberData2D**(*ModelName*, *LoadCaseName*, *eleNumber*, *sectionNumber*, *deltaT = 0.0*)

This command saves the fiber output of the specified section in a non-linear beam-column element to a sub-folder named "LoadCaseName" inside the output database folder "ModelName_ODB". This output data can be visualized using `plot_fiberResponse2D()` and `animate_fiberResponse2D()` commands.

`ModelName` (str)	Name of the model the user wants to save database with.
`LoadCaseName` (str)	Name of the subfolder to save load case output data.
`eleNumber` (int)	Tag of the element with a fiber section assigned.
`sectionNumber` (float)	Tag of the section where the fiber response is to be recorded.
`deltaT` (list)	Timesteps at which output to be saved. Default is 0.0. (optional)

Here is a simple example:

```
saveFiberData2D("TwoSpan_Bridge", "Pushover", 101, 2)
```

The above command will:

- create a folder named **TwoSpan_Bridge_ODB** and a sub-folder named **Pushover** if they are not already there.
- save fiber response of the section with a tag 2 of the element with a tag 101 in the Pushover folder.

plot_model command

postprocessing.Get_Rendering.**plot_model**(*<"nodes">*, *<"elements">*, *<Model="ModelName">*)

Once the model is built, it can be visualized using this command. By default Node and element tags are not displayed. Matplotlib and Numpy are required. No analysis is required in order to visualize the model.

To visualize an OpenSees model from an existing database (using `createdODB()` command), the optional argument Model="ModelName" should be used. The command will read the model data from folder **ModelName_ODB**.

"nodes" (str)	Displays the node tags on the model. (optional)
"elements" (str)	Displays the element tags on the model. (optional)
ModelName (str)	Displays the model saved in a database named "ModelName_ODB" (optional)

Input arguments to diaplay node and element tags can be used in any combination.

plot_model() Displays the model using data from the active OpenSeesPy model with no node and element tags on it.

plot_model("nodes") Displays the model using data from the active OpenSeesPy model with only node tags on it.

plot_model("elements") Displays the model using data from the active OpenSeesPy model with only element tags on it.

plot_model("nodes","elements") Displays the model using data from the active OpenSeesPy model with both node and element tags on it.

plot_model("nodes",Model="ModelName") Displays the model using data from an existing database "ModelName_ODB" with only node tags on it.

plot_modeshape command

postprocessing.Get_Rendering.**plot_modeshape**(*modeNumber*, *<scale>*, *<Model="modelName">*)

Any modeshape can be visualized using this command. There is no need to perform an eigen analysis exclusively since the command runs an eigen analysis internally. Matplotlib and Numpy are required.

mode_number (int)	Mode number to visualize. For example: plot_modeshape(3).
scale (int)	Scale factor for to display mode shape. (optional, default is 200)
ModelName (str)	Displays the model saved in a database named "ModelName_ODB". (optional)

Example:

plot_modeshape(3, 300) Displays the 3rd modeshape using data from the active OpenSeesPy model with a scale factor of 300. This command should be called from an OpenSeesPy model script once the model is completed. The model should successfully work for Eigen value analysis. There is no need to create a database with *createODB()* command to use this option.

plot_modeshape(3, 300, Model="TwoSpan_Bridge") Displays the 3rd modeshape using data from *TwoSpan_Bridge_ODB* database with a scale factor of 300. This command can be called from cmd, Ipython notebook or any Python IDE. An output database using *createODB()* has to be present in the current directory to use this command.

plot_deformedshape command

postprocessing.Get_Rendering.**plot_deformedshape**(*Model="ModelName"*, *LoadCase="LoadCaseName"*, *<tstep = -1>*, *<scale = 10>*, *<overlap='no'>*)

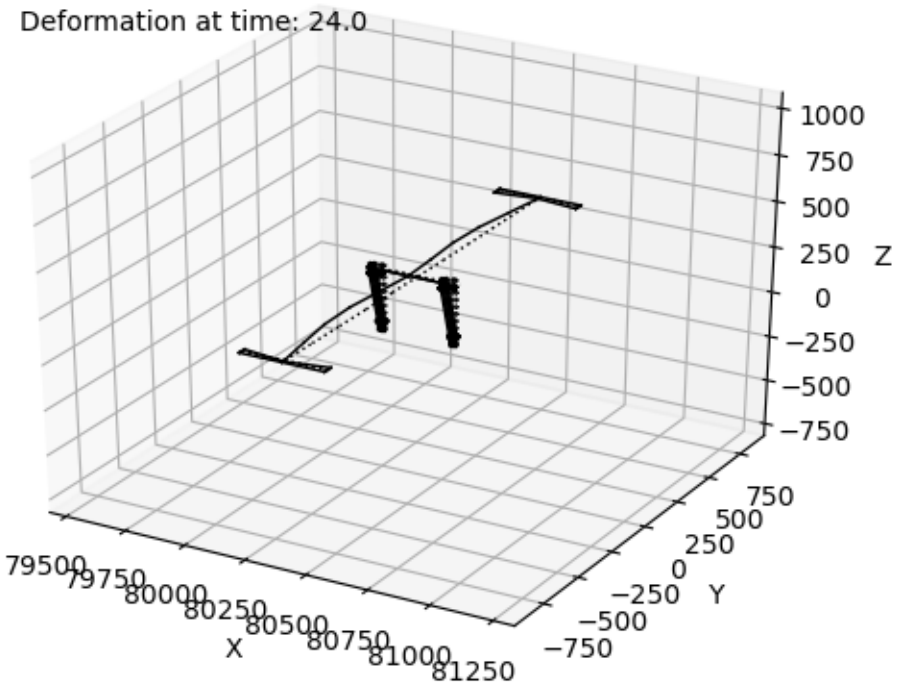
Displays the deformed shape of the structure by reading data from a saved output database.

`ModelName` (str)	Name of the model to read data from output database, created with *createODB()* command.
`LoadCaseName` (str)	Name of the subfolder with load case output data.
`tstep` (float)	Approximate time of the analysis at which deformed shape is to be displaced. (optional, default is the last step)
`scale` (int)	Scale factor for to display mode shape. (optional, default is 10)
`overlap` (str)	"yes" to overlap the deformed shape with the original structure. (optional, default is "no")

Examples:

`plot_deformedshape(Model="TwoSpan_Bridge", LoadCase="Dynamic_GM1")`
Displays the deformedshape of structure by reading data from *TwoSpan_Bridge_ODB* with a sub-folder *Dynamic_GM1* at the last analysis step (default) with a default scale factor of 10.

`plot_deformedshape(Model="TwoSpan_Bridge", LoadCase="Dynamic_GM1", tstep=24.0, scale=5`
Displays the deformedshape of structure by reading data from *TwoSpan_Bridge_ODB* with a sub-folder *Dynamic_GM1* at the analysis time closest to 24.0 sec with a scale factor of 50, overlapped with the original structure shape.



plot_fiberResponse2D command

postprocessing.Get_Rendering.**plot_fiberResponse2D**(*ModelName*, *LoadCaseName*, *element*, *section*, *LocalAxis = 'y'*, *InputType = 'stress'*, *tstep = -1*)

Plots the fibre stress or strain distribution along the local Z or Y-axis of a 2D fiber section. The local axis appears on the x axis, while the fiber response appears on the y axis.

`ModelName` (str)	Name of the model to read data from output database, created with *createODB()* command.
`LoadCaseName` (str)	Name of the subfolder with load case output data.
`element` (int)	Tag of the element where the section to be plotted is located.
`section` (int)	Tag of the section to be plotted.
`LocalAxis` (str)	Local axis of the section, based on a user defined axes transformation. (optional, default is "Y")
`InputType` (str)	Type of the fiber response to be plotted, `"stress"` or `"strain"`. (optional, default is "stress")
`tstep` (float)	Approximate time of the analysis at which fiber response is to be plotted. The program will find the closed time step to the input value.(optional, default is the last step).

Examples:

`plot_fiberResponse2D("TwoSpan_Bridge", "Dynamic_GM1", 101, 2)` Plots the fiber stress (default) distribution of section 2 in element 101 of structure by reading data from *TwoSpan_Bridge_ODB* with a sub-folder *Dynamic_GM1* at the last analysis step (default).

animate_deformedshape command

`postprocessing.Get_Rendering.animate_deformedshape(Model='ModelName', LoadCase='LoadCaseNa`
`dt = dT, <tStart = 0>, <tEnd = 0>, <scale = 10>, fps = 24, FrameInterval = 0,`
`timeScale = 1, Movie='none')`

Displays an animation of the deformed structure by reading data from a saved output database. The animation object should be stored as an variable in order for the animation to run. The input file should have approximately the same number of time between each step or the animation will appear to speed up or slow down. The time step for the input data can be controlled by passing the a recorder time step.

For many larger models, the runtime of the code for each frame likely be larger than the frame interval, this will result in the animation running slower than the target fps. **ffmpeg** codecs are required to save the animation as a movie (.mp4).

`ModelName` (str)	Name of the model to read data from output database, created with *createODB()* command.
`LoadCaseName` (str)	Name of the subfolder with load case output data.
`dT` (float)	The time step between frames in the input file.
`tStart` (float)	The start time for animation. It can be approximate value and the program will find the closest matching time step. (optional, default is 0)
`tEnd` (float)	The end time for animation. It can be approximate value and the program will find the closest matching time step. (optional, default is last step)
`scale` (int)	Scale factor for to display mode shape. (optional, default is 10)
`fps` (int)	The target frames per second to be displayed. (optional, The default is 24)
`FrameInterval` (int)	The time interval between frames to be used. Used to update at intervals different than 1/fps. The default is 0. (optional)
`timeScale` (int)	A scale factor that increase or decrease the time between animation frames. Will not improve results if the animation speed is governed by performance limited.(optional, default is 1)
`Movie` (str)	Name of the movie file in the *LoadCadeName* folder if the user wants to save the animation as .mp4 file. (optional, default is "none")

Examples:

```
ani = animate_deformedshape(Model="TwoSpan_Bridge", LoadCase="Dynamic_GM1", dt=0.01)
```

The above command animates the deformedshape of structure by reading data from *TwoSpan_Bridge_ODB* with a sub-folder *Dynamic_GM1* at dt=0.01.

```
ani = animate_deformedshape(Model="TwoSpan_Bridge", LoadCase="Dynamic_GM1",  dt=0.01,␣
↪tStart=10.0, tEnd=20.0,
                                                                    scale=50, Movie=
↪"Bridge_Dynamic")
```

The above command animates the deformedshape of structure by reading data from *TwoSpan_Bridge_ODB* with a sub-folder *Dynamic_GM1* at dt=0.01, starting at t=10.0 s and ending at t=20.0 s of the earthquake input, using a scale factor of 50. The animation movie will be saved as *Bridge_Dynamic.mp4* in the *Dynamic_GM1* sub-folder.

animate_fiberResponse2D command

postprocessing.Get_Rendering.animate_fiberResponse2D(Model, LoadCase, element, section, LocalAxis='y', InputType='stress', skipStart=0, skipEnd=0, rFactor=1, outputFrames=0, fps=24, Xbound=[], Ybound=[])

Animates fibre stress or strain distribution along the local Z or Y-axis of a 2D fiber section. The local axis appears on the x axis, while the fiber response appears on the y axis. The animation object should be stored as an variable in order for the animation to run.

ModelName (str)	Name of the model to read data from output database, created with *createODB()* command.
LoadCaseName (str)	Name of the subfolder with load case output data.
element (int)	Tag of the element where the section to be plotted is located.
section (int)	Tag of the section to be plotted.
LocalAxis (str)	Local axis of the section to appear on the x axis. (optional, default is "Y")
InputType (str)	Type of the fiber response to be plotted, possible values are "stress" or "strain". (optional, default is "stress")
skipStart (int)	If specified, this many datapoints will be skipped from the data start. (optional, default is 0)
skipEnd (int)	If specified, this many datapoints will be skipped from the data end. (optional, default is 0)
rFactor (int)	If specified, only every "x" frames will be output by this factor. For example, if x=2, every other frame will be used in the animation. (optional, default is 0)
outputFrames (int)	The number of frames to be included after all other reductions. (optional, default is 0)
fps (str)	Number of animation frames to be displayed per second. (optional, default is 24)
Xbound (list)	‘‘[xmin, xmax]‘ The domain of the chart. (optional, default is 1.1 the max and min values)
Ybound (float)	‘‘[ymin, ymax]‘ The domain of the chart. (optional, default is 1.1 the max and min values)

Examples:

ani = animate_fiberResponse2D("TwoSpan_Bridge", "Dynamic_GM1", 101, 2)
Animates the fiber stress (default) distribution of section 2 in element 101 of structure by reading data from *TwoSpan_Bridge_ODB* with a sub-folder *Dynamic_GM1* at the last analysis step (default).

Plotting OpenSees Tcl Output

OpenSees Tcl users can also take advantage of the plotting functions of OpenSeesPy *Get_Rendering* library. In order to do that, a Tcl script Get_Rendering.tcl to create an output database is used. First the user need to source *Get_Rendering.tcl* into the OpenSees tcl model file and then call the procedure to create an output database. This procedure does what createODB() does in OpenSeesPy. Once the output database is created, users can call plot_model(), plot_modeshape(), plot_deformedshape() and animate_deformedshape() commands. Compatibility with other commands will be added in the next release.

Download the Tcl script here Get_Rendering.tcl.

createODB "ModelName" "LoadCaseName" Nmodes

ModelName (str)	Name of the model the user wants to save database with. Folder name will be *ModelName_ODB*
LoadCaseName (str)	"none" or "LoadCaseName". Name of the subfolder to save load case output data.
Nmodes (int)	0 or Nmodes (int). Number of modes to be saved for visualization.

Note: To record modeshape data, this procedure utilizes an internal Eigenvalue analysis. Make sure your model is well defined to avoid errors.

Example: Here is a minimal example of how to use Get_Rendering.tcl.

```
# source the script in the beginning of the Tcl script.
source Get_Rendering.tcl

# create model here.
# define nodes, elements etc.
# Once the model definition is finished, call the procedure to record the first 3␣
↪modeshapes.
# When recording modeshapes, use "none" for the loadCaseName.

createODB "3DBuilding" "none"  3

# The above command will save all the data in a folder named "3DBuilding_ODB" and ...
# ... a sub-folder "Modeshapes".

# Now to record data from a dynamic loadcase, assign a name for load case folder and .
↪..
# ... the number 0 to Nmodes to avoid performing Eigenvalue analysis again.

createODB "3DBuilding" "Dynamic" 0

# The above command will save the node displacement data to a sub-folder "Dynamic" in␣
↪...
# ... the "3DBuilding_ODB" folder.
```

Now open a python terminal or Jupyter notebook and type the following. Make sure you install the latest version of OpenSeesPy first. Or, put the following lines in a Python script and run.

```
import openseespy.postprocessing.Get_Rendering as opsplt

# render the model with node and element tags on it
opsplt.plot_model("nodes","elements",Model=3DBuilding")

# plot mode shape 2 with a scale factor of 100
opsplt.plot_modeshape(2, 100, Model="3DBuilding")

# animate the deformed shape for dynaic analysis and save it as a 3DBuilding.mp4 file.
opsplt.animate_deformedshape(Model="3DBuilding", LoadCase="Dynamic", dt=0.01, Movie=
↪"3DBuilding")
```

All figures are interactive and can be saved as a .png file from the plot window.

Visualization Development Guide

You are welcome to contribute to the plotting/post-processing commands. This documentation is to explain what is going on inside the plotting library "Get_Rendering" and how you can enhance the functons. As of now, Get_Rendering has functions to plot a structure, mode shapes and shape of a displaced structure and works for 2D (beam-column elements, tri, and quad) and 3D (beam-column, tri, 4-node shell, and 8-node brick) elements.

As of now, all plotting functions should use Matplotlib only and should be able to produce interactive plots. Developers should test the new functions extensively , including on Jupyter Notebook, before submitting a pull request.

Note: A list of test examples will be available soon.

The source code of all the plotting functions is located in OpenSeesPy repository.

As an object oriented approach and to reduce the code repetition, Get_Rendering uses two types of functions.

Internal Database functions

These functions get, write to and read data from output database.

`_getNodesandElements()` : Gets node and element tags from the active model, stores node tags with coordinates and element tags with connected nodes in numpy arrays.

`_saveNodesandElements()` : Saves the node and element arrays from _getNodesandElements() to text files with ".out" extension.

`_readNodesandElements()` : Reads the node and element data into numpy arrays from the saved files.

`_getModeShapeData()` : Gets node deflection for a particular mode from the active model, stores node tags with modeshape data in numpy arrays.

`_saveModeShapeData()` : Saves the modeshape data arrays from _getModeShapeData() to text files with ".out" extension.

`_readModeShapeData()` : Reads the modeshape data into numpy arrays from the saved files.

`_readNodeDispData()` : Reads the node displacement data into numpy arrays from the saved files (from createODB() command).

`_readFiberData2D()` : Reads the section fiber output data into numpy arrays from the saved files (from saveFiberData2D() command).

Internal Plotting functions

These functions are helper functions that are called by the user functions once the updated node coordinates are calculated.

`_plotBeam2D()` : A procedure to plot a 2D beam-column (or any element with 2 nodes) using iNode, jNode and some internal variables as input.

`_plotBeam3D()` : A procedure to plot a 3D beam-column (or any element with 2 nodes) using iNode, jNode and some internal variables as input.

`_plotTri2D()` : A procedure to render a 2D, three node shell (Tri) element using iNode, jNode, kNode in counter-clockwise order and some internal variables as input.

`_plotTri3D()` : A procedure to render a 3D, three node shell (Tri) element using iNode, jNode, kNode in counter-clockwise order and some internal variables as input.

`_plotQuad2D()` : A procedure to render a 2D, four node shell (Quad, ShellDKGQ etc.) element using iNode, jNode, kNode, lNode in counter-clockwise and some internal variables as input.

`_plotQuad3D()` : A procedure to render a 3D, four node shell (Quad, ShellDKGQ etc.) element using iNode, jNode, kNode, lNode in counter-clockwise and some internal variables as input.

`_plotCubeSurf()` : This procedure is called by the *plotCubeVol()* command to render each surface in a cube using four corner nodes.

`_plotCubeVol()` : A procedure to render a 8-node brick element using a list of eight element nodes in bottom and top faces and in counter-clockwise order, and internal variables as input.

`_plotEle_2D()` : A procedure to plot any 2D element by calling other internal plotting commands for 2D elements.

`_plotEle_3D()` : A procedure to plot any 3D element by calling other internal plotting commands for 3D elements.

`_initializeFig()` : Initializes a matplotlib.pyplot figure for each of the user plotting commands. This procedure reduced the code repetition.

`_setStandardViewport()` : Sets a standard viewport for matplotlib.pyplot figure for each of the user plotting commands. This procedure reduced the code repetition.

User functions

These are the functions available to users to call through OpenSeesPy script. The following table describes what they are and how they work.

`createODB()` : Redords the model and loadcase data to be used by other plotting functions in a user defined output folder.

`saveFiberData2D()` : Redords the output data from all the fibers in a particular section to plot the distribution.

`plot_model()` : Gets the number of nodes and elements in lists by calling *getNodeTags()* and *getEleTags()*. Then plots the elements in a loop by checking if the model is 2D or 3D, and calling the *nodecoord()* command for each node to get its original coordinates and internal function *_plotBeam3D*.

`plot_modeshapes()` : Gets the number of nodes and elements in lists by calling *getNodeTags()* and *getEleTags()*. In a loop, calls *nodecoord()* and *nodeEigenvector()* for each node to get original and eigen coordinates respectively. Then plots the mode shape by calling the internal functions.

`plot_deformedshape()`: Reads the displacement data from the output of *createODB()* function and original coordinates using *nodecoord()* function. Then plots the displaced shape of the structure using the internal functions in a manner similar to *plot_modeshapes()*.

`plot_fiberResponse2D()`: Reads the fiber output data from the output of *saveFiberData2D()* function and plots the distribution across the section.

`animate_deformedshape()`: Reads the displacement data from the output of *createODB()* function and original coordinates using *nodecoord()* function. Then animates the displaced shape of the structure.

`animate_fiberResponse2D()`: Reads the fiber output data from the output of *saveFiberData2D()* function and animates the stress/strain distribution across the section.

Example of an internal function

Here is an example of `_plotQuad3D()` internal function:

```
_plotQuad3D(iNode, jNode, kNode, lNode, ax, show_element_tags, element, eleStyle,␣
↪fillSurface)
```

This function uses the following inputs:

*Nodes	iNode, jNode, kNode, lNode : A list of four nodes in counter-clockwise order.
ax	Reference to the Matplotlib fixure axes space. This should not be changed.
show_element_tags	**The default is set to "yes" for plotting the model with `plot_model()`** or deformed shapes. This should not be changed.
element	This is the tag of that particular element as a string which is displayed when show_element_tags="yes".
eleStyle	**Use eleStyle = "wire" for a wire frame, and "solid" for solid element lin** overlapping the original shape of the structure with the mode shape or displaced shape.
fillSurface	Use fillSurface = "yes" for color fill in the elements. fillSurface="no" for wireframe.

Naming conventions

The names of classes, variables, and functions should be self-explanatory. Look for names that give useful information about the meaning of the variable or function. All the internal functions should start with "_" and be in camelCase (for example `_plotCubeSurf`) to distinguish them from the user functions. There are two type of user functions, 1) recorder functions and 2) plot functions. All the "recorder" functions should start with "record" and use camelCase (example: `recordNodeDisp`) and the plotting functions should start with "`plot_`" use snake_case (example: `plot_deformedshape`). Try to keep the internal variables (see 3. Example) consistant. New internal variable should be defined only if necessary.

Wish List

Some functions helpful to users might include, but not limited to,

- plot_stress() : Record stress in all the elements of a function and plot. This will be useful in visualization of shear walls, fluid-structure interaction and soil modeled as brick elements.
- plot_strain() : Similar functionality as above.
- plot_sectionfiber() : Visualize stress-strain distribution accross a fiber section in a non-linear beam-column element.
- plot_elementforces() : Element forces such as moment, shear, axial force.
- animate_elementstress() : Animation of the stress or strain in a structure.

Feel free to comment and discuss how to streamline your plotting code with the existing `Get_Rendering` library. Or contact me Anurag Upadhyay.

1.13.2 ops_vis module

plot_model (ops_vis)

ops_vis.**plot_model**(*node_labels=1*, *element_labels=1*, *offset_nd_label=False*, *axis_off=0*, *az_el=(-60.0*, *30.0)*, *fig_wi_he=(16.0*, *10.0)*, *fig_lbrt=(0.04*, *0.04*, *0.96*, *0.96)*)

Plot defined model of the structure.

Parameters

- **node_labels** (*int*) – 1 - plot node labels, 0 - do not plot them; (default: 1)
- **element_labels** (*int*) – 1 - plot element labels, 0 - do not plot them; (default: 1)
- **offset_nd_label** (*bool*) – False - do not offset node labels from the actual node location. This option can enhance visibility.
- **axis_off** (*int*) – 0 - turn off axes, 1 - display axes; (default: 0)
- **az_el** (*tuple*) – contains azimuth and elevation for 3d plots. For 2d plots this parameter is neglected.
- **fig_wi_he** (*tuple*) – contains width and height of the figure
- **fig_lbrt** (*tuple*) – a tuple contating left, bottom, right and top offsets

Usage:

`plot_model()` - plot model with node and element labels.

`plot_model(node_labels=0, element_labels=0)` - plot model without node element labels

`plot_model(fig_wi_he=(20., 14.))` - plot model in a window 20 cm long, and 14 cm high.

plot_defo (ops_vis)

ops_vis.**plot_defo**(*sfac=False*, *nep=17*, *unDefoFlag=1*, *fmt_undefo='g–'*, *interpFlag=1*, *endDispFlag=0*, *fmt_interp='b-'*, *fmt_nodes='rs'*, *Eo=0*, *az_el=(-60.0*, *30.0)*, *fig_wi_he=(16.0*, *10.0)*, *fig_lbrt=(0.04*, *0.04*, *0.96*, *0.96)*)

Plot deformed shape of the structure.

Parameters

- **sfac** (*float*) – scale factor to increase/decrease displacements obtained from FE analysis. If not specified (False), sfac is automatically calculated based on the maximum overall displacement and this maximum displacement is plotted as 20 percent (hordcoded) of the maximum model dimension.
- **interpFlag** (*int*) – 1 - use interpolated deformation using shape function, 0 - do not use interpolation, just show displaced element nodes (default is 1)
- **nep** (*int*) – number of evaluation points for shape function interpolation (default: 17)

Returns

the automatically calculated scale factor can be returned.

Return type sfac (float)

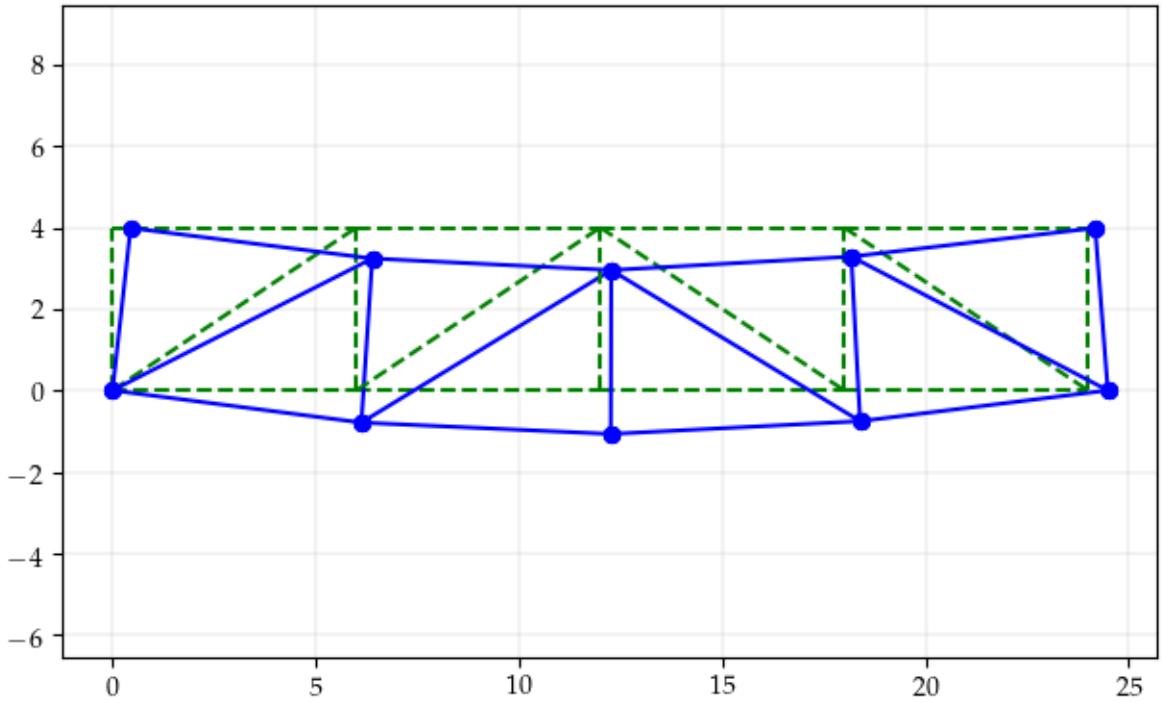
Usage:

`sfac = plot_defo()` - plot deformed shape with default parameters and automatically calcutated scale factor.

`plot_defo(interpFlag=0)` - plot simplified deformation by displacing the nodes connected with straight lines (shape function interpolation)

`plot_defo(sfac=1.5)` - plot with specified scale factor

`plot_defo(unDefoFlag=0, endDispFlag=0)` - plot without showing undeformed (original) mesh and without showing markers at the element ends.



plot_mode_shape (ops_vis)

`ops_vis.`**`plot_mode_shape`**(*modeNo*, *sfac=False*, *nep=17*, *unDefoFlag=1*, *fmt_undefo='g–'*, *interpFlag=1*, *endDispFlag=1*, *fmt_interp='b-'*, *fmt_nodes='rs'*, *Eo=0*, *az_el=(-60.0, 30.0)*, *fig_wi_he=(16.0, 10.0)*, *fig_lbrt=(0.04, 0.04, 0.96, 0.96)*)

Plot mode shape of the structure obtained from eigenvalue analysis.

Parameters

- **`modeNo`** (*`int`*) – indicates which mode shape to plot
- **`sfac`** (*`float`*) – scale factor to increase/decrease displacements obtained from FE analysis. If not specified (False), sfac is automatically calculated based on the maximum overall displacement and this maximum displacement is plotted as 20 percent (hordcoded) of the maximum model dimension.
- **`interpFlag`** (*`int`*) – 1 - use interpolated deformation using shape function, 0 - do not use interpolation, just show displaced element nodes (default is 1)
- **`nep`** (*`int`*) – number of evaluation points for shape function interpolation (default: 17)

Usage:

`plot_mode_shape(1)` - plot the first mode shape with default parameters and automatically calcutated scale factor.

`plot_mode_shape(2, interpFlag=0)` - plot the 2nd mode shape by displacing the nodes connected with straight lines (shape function interpolation)

`plot_mode_shape(3, sfac=1.5)` - plot the 3rd mode shape with specified scale factor

`plot_mode_shape(4, unDefoFlag=0, endDispFlag=0)` - plot the 4th mode shape without showing undeformed (original) mesh and without showing markers at the element ends.

Examples:

Notes:

See also:

section_force_diagram_2d (ops_vis)

ops_vis.**section_force_diagram_2d**(*sf_type*, *Ew*, *sfac=1.0*, *nep=17*, *fmt_secforce='b-'*)

Display section forces diagram for 2d beam column model.

This function plots a section forces diagram for 2d beam column elements with or without element loads. For now only '-beamUniform' constant transverse or axial element loads are supported.

Parameters

- **sf_type** (*str*) – type of section force: 'N' - normal force, 'V' - shear force, 'M' - bending moments.
- **Ew** (*dict*) – Ew Python dictionary contains information on non-zero element loads, therfore each item of the Python dictionary is in the form: 'ele_tag: ['-beamUniform', Wy, Wx]'.
- **sfac** (*float*) – scale factor by wich the values of section forces are multiplied.
- **nep** (*int*) – number of evaluation points including both end nodes (default: 17)
- **fmt_secforce** (*str*) – format line string for section force distribution curve. The format contains information on line color, style and marks as in the standard matplotlib plot function. (default: fmt_secforce = 'b-' # blue solid line)

Usage:

```
Wy, Wx = -10.e+3, 0.
Ew = {3: ['-beamUniform', Wy, Wx]}
sfacM = 5.e-5
plt.figure()
minVal, maxVal = opsv.section_force_diagram_2d('M', Ew, sfacM)
plt.title('Bending moments')
```

Todo:

Add support for other element loads available in OpenSees: partial (trapezoidal) uniform element load, and 'beamPoint' element load.

See example *Statics of a 2d Portal Frame (ops_vis)*

section_force_diagram_3d (ops_vis)

ops_vis.**section_force_diagram_3d**(*sf_type*, *Ew*, *sfac=1.0*, *nep=17*, *fmt_secforce='b-'*)

Display section forces diagram of a 3d beam column model.

This function plots section forces diagrams for 3d beam column elements with or without element loads. For now only '-beamUniform' constant transverse or axial element loads are supported.

Parameters

- **sf_type** (*str*) – type of section force: 'N' - normal force, 'Vy' or 'Vz' - shear force, 'My' or 'Mz' - bending moments, 'T' - torsional moment.
- **Ew** (*dict*) – Ew Python dictionary contains information on non-zero element loads, therfore each item of the Python dictionary is in the form: 'ele_tag: ['-beamUniform', Wy, Wz, Wx]'.
- **sfac** (*float*) – scale factor by wich the values of section forces are multiplied.
- **nep** (*int*) – number of evaluation points including both end nodes (default: 17)
- **fmt_secforce** (*str*) – format line string for section force distribution curve. The format contains information on line color, style and marks as in the standard matplotlib plot function. (default: fmt_secforce = 'b-' # blue solid line)

Usage:

```
Wy, Wz, Wx = -5., 0., 0.
Ew = {3: ['-beamUniform', Wy, Wz, Wx]}
sfacMz = 1.e-1
plt.figure()
minY, maxY = opsv.section_force_diagram_3d('Mz', Ew, sfacMz)
plt.title(f'Bending moments Mz, max = {maxY:.2f}, min = {minY:.2f}')
```

Todo:

Add support for other element loads available in OpenSees: partial (trapezoidal) uniform element load, and 'beamPoint' element load.

See example *Statics of a 3d 3-element cantilever beam (ops_vis)*

plot_stress_2d (ops_vis)

ops_vis.**plot_stress_2d**(*nds_val*, *mesh_outline=1*, *cmap='turbo'*, *levels=50*)

Plot stress distribution of a 2d elements of a 2d model.

Parameters

- **nds_val** (*ndarray*) – the values of a stress component, which can be extracted from sig_out array (see sig_out_per_node function)
- **mesh_outline** (*int*) – 1 - mesh is plotted, 0 - no mesh plotted.
- **cmap** (*str*) – Matplotlib color map (default is 'turbo')

Usage:

```
sig_out = opsv.sig_out_per_node()
j, jstr = 3, 'vmis'
nds_val = sig_out[:, j]
opsv.plot_stress_2d(nds_val)
plt.xlabel('x [m]')
plt.ylabel('y [m]')
plt.title(f'{jstr}')
plt.show()
```

See also:

sig_out_per_node (ops_vis)

See example *Plot stress distribution of plane stress quad model (ops_vis)*

anim_defo (ops_vis)

ops_vis.**anim_defo**(*Eds, timeV, sfac, nep=17, unDefoFlag=1, fmt_undefo='g–', interpFlag=1, endDispFlag=1, fmt_interp='b-', fmt_nodes='b-', az_el=(-60.0, 30.0), fig_lbrt=(0.04, 0.04, 0.96, 0.96), fig_wi_he=(16.0, 10.0), xlim=[0, 1], ylim=[0, 1]*)

Make animation of the deformed shape computed by transient analysis

Parameters

- **Eds** (*ndarray*) – A 3d array (n_steps x n_eles x n_dof_per_element) containing the collected displacements per element for all time steps.
- **timeV** (*1darray*) – vector of discretized time values
- **sfac** (*float*) – scale factor
- **nep** (*integer*) – number of evaluation points inside the element and including both element ends
- **unDefoFlag** (*integer*) – 1 - plot the undeformed model (mesh), 0 - do not plot the mesh
- **interpFlag** (*integer*) – 1 - interpolate deformation inside element, 0 - no interpolation
- **endDispFlag** (*integer*) – 1 - plot marks at element ends, 0 - no marks
- **fmt_interp** (*string*) – format line string for interpolated (continuous) deformated shape. The format contains information on line color, style and marks as in the standard matplotlib plot function.
- **fmt_nodes** (*string*) – format string for the marks of element ends
- **az_el** (*tuple*) – a tuple containing the azimuth and elevation
- **fig_lbrt** (*tuple*) – a tuple contating left, bottom, right and top offsets
- **fig_wi_he** (*tuple*) – contains width and height of the figure

Examples:

Notes:

See also:

anim_mode (ops_vis)

ops_vis.**anim_mode**(*modeNo, sfac=False, nep=17, unDefoFlag=1, fmt_undefo='g–', interpFlag=1, endDispFlag=1, fmt_interp='b-', fmt_nodes='b-', Eo=0, az_el=(-60.0, 30.0), fig_wi_he=(16.0, 10.0), fig_lbrt=(0.04, 0.04, 0.96, 0.96), xlim=[0, 1], ylim=[0, 1], lw=3.0*)

Make animation of a mode shape obtained from eigenvalue solution.

Parameters

- **modeNo** (*int*) – indicates which mode shape to animate.
- **sfac** (*float*) – scale factor
- **nep** (*integer*) – number of evaluation points inside the element and including both element ends
- **unDefoFlag** (*integer*) – 1 - plot the undeformed model (mesh), 0 - do not plot the mesh
- **interpFlag** (*integer*) – 1 - interpolate deformation inside element, 0 - no interpolation

- **endDispFlag** (*integer*) – 1 - plot marks at element ends, 0 - no marks
- **fmt_interp** (*string*) – format line string for interpolated (continuous) deformated shape. The format contains information on line color, style and marks as in the standard matplotlib plot function.
- **fmt_nodes** (*string*) – format string for the marks of element ends
- **az_el** (*tuple*) – a tuple containing the azimuth and elevation
- **fig_lbrt** (*tuple*) – a tuple contating left, bottom, right and top offsets
- **fig_wi_he** (*tuple*) – contains width and height of the figure

Examples

sfac_a = 100. Eds = np.zeros((n_steps, nel, 6)) timeV = np.zeros(n_steps)

for step in range(n_steps): ops.analyze(1, dt) timeV[step] = ops.getTime() # collect disp for element nodes for el_i, ele_tag in enumerate(el_tags):

nd1, nd2 = ops.eleNodes(ele_tag) # uAll[step, inode, 0] = ops.nodeDisp() Eds[step, el_i, :] = np.array([ops.nodeDisp(nd1)[0],

ops.nodeDisp(nd1)[1], ops.nodeDisp(nd1)[2], ops.nodeDisp(nd2)[0], ops.nodeDisp(nd2)[1], ops.nodeDisp(nd2)[2]])

fw = 20. fh = 1.2 * 4./6. * fw

anim = opsv.anim_defo(Eds, timeV, sfac_a, interpFlag=1, xlim=[-1, 7], ylim=[-1, 5], fig_wi_he=(fw, fh))

Notes:

See also:

anim_mode()

plot_fiber_section (ops_vis)

ops_vis.**plot_fiber_section** (*fib_sec_list, fillflag=1, matcolor=['y', 'b', 'r', 'g', 'm', 'k']*)
Plot fiber cross-section.

Parameters

- **fib_sec_list** (*list*) – list of lists in the format similar to the input given for in
- **fillflag** (*int*) – 1 - filled fibers with color specified in matcolor list, 0 - no color, only the outline of fibers
- **matcolor** (*list*) – sequence of colors for various material tags assigned to fibers

Examples

```
fib_sec_1 = [['section', 'Fiber', 1, '-GJ', 1.0e6],
             ['patch', 'quad', 1, 4, 1,  0.032, 0.317, -0.311, 0.067, -0.266, 0.
↪005, 0.077, 0.254],  # noqa: E501
             ['patch', 'quad', 1, 1, 4,  -0.075, 0.144, -0.114, 0.116, 0.075, -0.
↪144, 0.114, -0.116],  # noqa: E501
             ['patch', 'quad', 1, 4, 1,  0.266, -0.005,  -0.077, -0.254,  -0.032,␣
↪-0.317,  0.311, -0.067]  # noqa: E501
```

(continues on next page)

(continued from previous page)

```
            ]
opsv.fib_sec_list_to_cmds(fib_sec_1)
matcolor = ['r', 'lightgrey', 'gold', 'w', 'w', 'w']
opsv.plot_fiber_section(fib_sec_1, matcolor=matcolor)
plt.axis('equal')
# plt.savefig('fibsec_rc.png')
plt.show()
```

Notes

`fib_sec_list` can be reused by means of a python helper function `ops_vis.fib_sec_list_to_cmds(fib_sec_list_1)`

See also:

`ops_vis.fib_sec_list_to_cmds()`

See example *Plot steel and reinforced concrete fiber sections (ops_vis)*

fib_sec_list_to_cmds (ops_vis)

`ops_vis.`**`fib_sec_list_to_cmds`**(*fib_sec_list*)

Reuses fib_sec_list to define fiber section in OpenSees.

At present it is not possible to extract fiber section data from the OpenSees domain, this function is a workaround. The idea is to prepare data similar to the one the regular OpenSees commands (`section('Fiber', ...)`, `fiber()`, `patch()` and/or `layer()`) require.

Parameters

- **`fib_sec_list`** (*`list`*) – is a list of fiber section data. First sub-list
- **`defines the torsional stiffness`** (*`also`*) –

Warning:

If you use this function, do not issue the regular OpenSees: section, Fiber, Patch or Layer commands.

See also:

`ops_vis.plot_fiber_section()`

See example *Plot steel and reinforced concrete fiber sections (ops_vis)*

sig_out_per_node (ops_vis)

`ops_vis.`**`sig_out_per_node`**(*how_many='all'*)

Return a 2d numpy array of stress components per OpenSees node.

Three first stress components (sxx, syy, sxy) are calculated and extracted from OpenSees, while the rest svm (Huber-Mieses-Hencky), two principal stresses (s1, s2) and directional angle are calculated as postprocessed quantities.

Parameters **`how_many`** (*`str`*) – supported options are: 'all' - all components, 'sxx', 'syy', 'sxy', 'svm' (or 'vmis'), 's1', 's2', 'angle'.

Returns

a 2d array of stress components per node with the following components: sxx, syy, sxy, svm, s1, s2, angle. Size (n_nodes x 7).

Return type sig_out (ndarray)

Examples

sig_out = opsv.sig_out_per_node()

Notes

s1, s2: principal stresses angle: angle of the principal stress s1

See example *Plot stress distribution of plane stress quad model (ops_vis)*

`ops_vis` is an OpenSeesPy postprocessing and plotting module written by Seweryn Kokot (Opole University of Technology, Poland).

This module can be mainly useful for students when learning the fundamentals of structural analysis (interpolated deformation of frame structures (static images or animations), section force distribution of frame structures, stress distribution in triangle, quadrilateral 2d elements, orientation of frame members in 3d space, fibers of a cross section, static and animated eigenvalue mode shapes etc.). This way, we can lower the bar in teaching and learning OpenSees at earlier years of civil engineering studies. However the visualization features for OpenSees can also be helpful for research studies.

Note that OpenSeesPy contains another plotting module called `Get_Rendering`, however `ops_vis` is an alternative with some distinct features (on the other hand the `Get_Rendering` has other features that `ops_vis` does not have), which for example allow us to plot:

- interpolated deformation of frame structures,
- stresses of triangular and (four, eight and nine-node) quadrilateral 2d elements (calculation of Huber-Mieses-Hencky equivalent stress, principal stresses),
- fibers of cross-sections,
- models with extruded cross sections
- animation of mode shapes.

To use `ops_vis` in OpenSees Python scripts, your `.py` file should start as follows:

```
import openseespy.opensees as ops
import openseespy.postprocessing.ops_vis as opsv
import matplotlib.pyplot as plt
# ... your OpenSeesPy model and analysis commands ...
opsv.plot_model()
sfac = opsv.plot_defo()
```

The main commands related to various aspects of OpenSees model visualization are as follows:

1. *plot_model (ops_vis)*
2. *plot_defo (ops_vis)*
3. *plot_mode_shape (ops_vis)*
4. *section_force_diagram_2d (ops_vis)*
5. *section_force_diagram_3d (ops_vis)*
6. *plot_stress_2d (ops_vis)*

7. ops_vis_plot_extruded_model_rect_section_3d
8. *anim_defo (ops_vis)*
9. *anim_mode (ops_vis)*
10. *plot_fiber_section (ops_vis)*

Helper functions include:

1. *fib_sec_list_to_cmds (ops_vis)*
2. *sig_out_per_node (ops_vis)*

Notes:

- matplotlib's `plt.axis('equal')` does not work for 3d plots therefore right angles are not guaranteed to be 90 degrees on the plots
- `plot_fiber_section` is inspired by Matlab `plotSection.zip` written by D. Vamvatsikos available at http://users.ntua.gr/divamva/software.html

This is the main page for listing OpenSeesPy postprocessing modules. For commands and documentation check their corresponding subpages.

There are two OpenSees plotting modules:

1. *Get_Rendering Commands*
2. *ops_vis module*

These modules also contain helper functions and other postprocessing commands preparing data for plotting.

Note that you can use both modules at the same time (even the same function name e.g. `plot_model()`) in a single OpenSeesPy script by distinguishing them as follows:

```
import openseespy.opensees as ops
import openseespy.postprocessing.Get_Rendering as opsplt
import openseespy.postprocessing.ops_vis as opsv
# ...
opsplt.plot_model()  # command from Get_Rendering module
opsv.plot_model()  # command from ops_vis module
```

1.14 Examples

1. *Structural Examples*
2. *Earthquake Examples*
3. *Tsunami Examples*
4. *GeoTechnical Examples*
5. *Thermal Examples*
6. *Parallel Examples*
7. *Plotting Examples*

1.14.1 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*
8. *Cantilever FRP-Confined Circular Reinforced Concrete Column under Cyclic Lateral Loading*
9. *Reinforced Concrete Shear Wall with Special Boundary Elements*

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.

```
from openseespy.opensees import *

import numpy as np
import matplotlib.pyplot as plt

# ------------------------------
# Start of model generation
# ------------------------------

# remove existing model
wipe()

# set modelbuilder
model('basic', '-ndm', 2, '-ndf', 2)

# create nodes
node(1, 0.0, 0.0)
node(2, 144.0,  0.0)
node(3, 168.0,  0.0)
node(4,  72.0, 96.0)

# set boundary condition
fix(1, 1, 1)
fix(2, 1, 1)
fix(3, 1, 1)

# define materials
uniaxialMaterial("Elastic", 1, 3000.0)

# define elements
element("Truss",1,1,4,10.0,1)
element("Truss",2,2,4,5.0,1)
element("Truss",3,3,4,5.0,1)

```

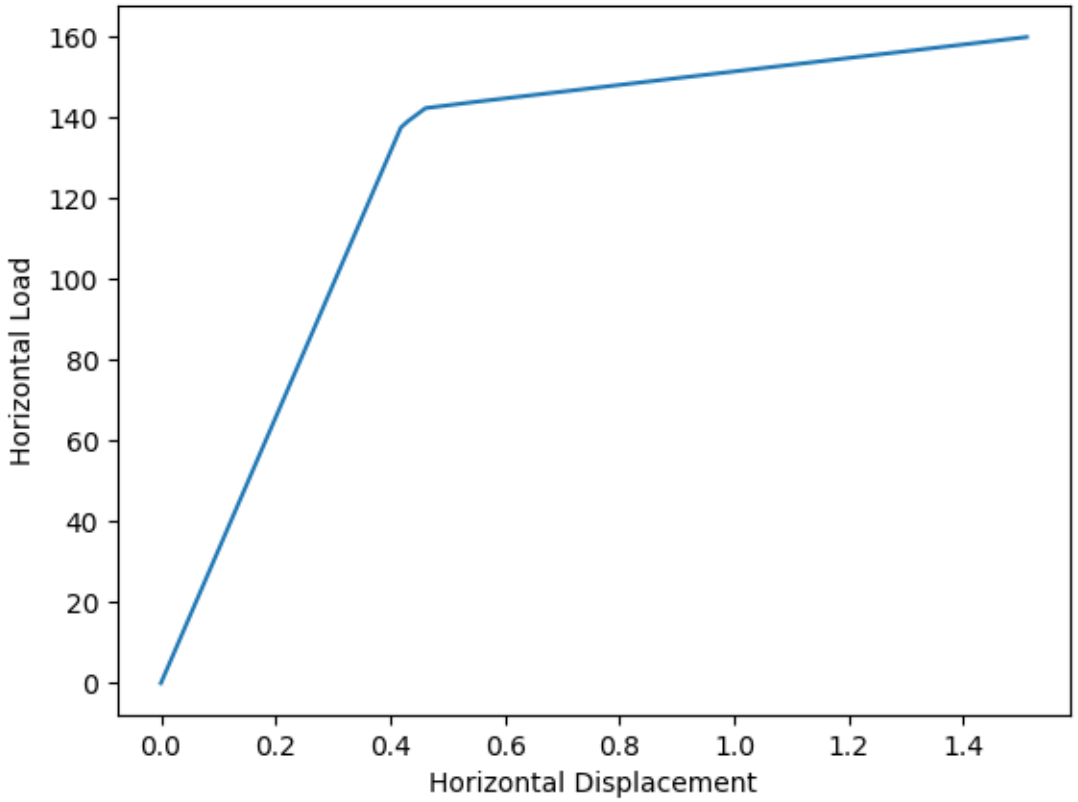
(continues on next page)

(continued from previous page)

```
# create TimeSeries
timeSeries("Linear", 1)

# create a plain load pattern
pattern("Plain", 1, 1)

# Create the nodal load - command: load nodeID xForce yForce
load(4, 100.0, -50.0)

# ------------------------------
# Start of analysis generation
# ------------------------------

# create SOE
system("BandSPD")

# create DOF number
numberer("RCM")

# create constraint handler
constraints("Plain")

# create integrator
integrator("LoadControl", 1.0)

# create algorithm
algorithm("Linear")

# create analysis object
analysis("Static")

# perform the analysis
analyze(1)

ux = nodeDisp(4,1)
uy = nodeDisp(4,2)
if abs(ux-0.5300927771322837 5450)<1e-12 and abs(uy+0.17789363846931768864)<1e-12:
    print("Passed!")
else:
    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




```python
from openseespy.opensees import *

import numpy as np
import matplotlib.pyplot as plt

# ----------------------------
# Start of model generation
# ----------------------------

# set modelbuilder
wipe()
model('basic', '-ndm', 2, '-ndf', 2)

# variables
A = 4.0
E = 29000.0
alpha = 0.05
sY = 36.0
udisp = 2.5
Nsteps = 1000
Px = 160.0
Py = 0.0

# create nodes
node(1, 0.0, 0.0)
```

(continues on next page)

(continued from previous page)

```
node(2, 72.0, 0.0)
node(3, 168.0, 0.0)
node(4, 48.0, 144.0)

# set boundary condition
fix(1, 1, 1)
fix(2, 1, 1)
fix(3, 1, 1)

# define materials
uniaxialMaterial("Hardening", 1, E, sY, 0.0, alpha/(1-alpha)*E)

# define elements
element("Truss",1,1,4,A,1)
element("Truss",2,2,4,A,1)
element("Truss",3,3,4,A,1)

# create TimeSeries
timeSeries("Linear", 1)

# create a plain load pattern
pattern("Plain", 1, 1)

# Create the nodal load
load(4, Px, Py)

# ------------------------------
# Start of analysis generation
# ------------------------------

# create SOE
system("ProfileSPD")

# create DOF number
numberer("Plain")

# create constraint handler
constraints("Plain")

# create integrator
integrator("LoadControl", 1.0/Nsteps)

# create algorithm
algorithm("Newton")

# create test
test('NormUnbalance',1e-8, 10)

# create analysis object
analysis("Static")

# ------------------------------
# Finally perform the analysis
# ------------------------------

# perform the analysis
data = np.zeros((Nsteps+1,2))
```

(continues on next page)

(continued from previous page)

```
for j in range(Nsteps):
    analyze(1)
    data[j+1,0] = nodeDisp(4,1)
    data[j+1,1] = getLoadFactor(1)*Px

plt.plot(data[:,0], data[:,1])
plt.xlabel('Horizontal Displacement')
plt.ylabel('Horizontal Load')
plt.show()

```

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


 tSatic 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
```

```
from openseespy.opensees import *

from math import asin, sqrt

# Two dimensional Frame: Eigenvalue & Static Loads


# REFERENCES:
# used in verification by SAP2000:
# SAP2000 Integrated Finite Element Analysis and Design of Structures, Verification␣
↪Manual,
# Computers and Structures, 1997. Example 1.
# and seismo-struct (Example 10)
# SeismoStruct, Verification Report For Version 6, 2012. Example 11.


# set some properties
wipe()

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  iColumn = ['W14X287', 'W14X287', 'W14X287', 'W14X246', 'W14X246', 'W14X211', 'W14X211
    ↪']
40  columns = [eColumn, iColumn, eColumn]
41
42  WSection = {
43      'W14X176': [51.7, 2150.],
44      'W14X211': [62.1, 2670.],
45      'W14X246': [72.3, 3230.],
46      'W14X287': [84.4, 3910.],
47      'W24X110': [32.5, 3330.],
48      'W24X130': [38.3, 4020.],
49      'W24X160': [47.1, 5120.]
50  }
51
52  nodeTag = 1
53
54
55  # procedure to read
56  def ElasticBeamColumn(eleTag, iNode, jNode, sectType, E, transfTag, M, massType):
57      found = 0
58
59      prop = WSection[sectType]
60
61      A = prop[0]
62      I = prop[1]
63      element('elasticBeamColumn', eleTag, iNode, jNode, A, E, I, transfTag, '-mass', M,
    ↪ massType)
64
65
66  # add the nodes
67  #  - floor at a time
68  yLoc = 0.
69  for j in range(0, numFloor + 1):
70
71      xLoc = 0.
72      for i in range(0, numBay + 1):
73          node(nodeTag, xLoc, yLoc)
```

(continues on next page)

(continued from previous page)

```
        xLoc += bayWidth
        nodeTag += 1

    if j < numFloor:
        storyHeight = storyHeights[j]

    yLoc += storyHeight

# fix first floor
fix(1, 1, 1, 1)
fix(2, 1, 1, 1)
fix(3, 1, 1, 1)

# rigid floor constraint & masses
nodeTagR = 5
nodeTag = 4
for j in range(1, numFloor + 1):
    for i in range(0, numBay + 1):

        if nodeTag != nodeTagR:
            equalDOF(nodeTagR, nodeTag, 1)
        else:
            mass(nodeTagR, massX, 1.0e-10, 1.0e-10)

        nodeTag += 1

    nodeTagR += numBay + 1

# add the columns
# add column element
geomTransf(coordTransf, 1)
eleTag = 1
for j in range(0, numBay + 1):

    end1 = j + 1
    end2 = end1 + numBay + 1
    thisColumn = columns[j]

    for i in range(0, numFloor):
        secType = thisColumn[i]
        ElasticBeamColumn(eleTag, end1, end2, secType, E, 1, M, massType)
        end1 = end2
        end2 += numBay + 1
        eleTag += 1

# add beam elements
for j in range(1, numFloor + 1):
    end1 = (numBay + 1) * j + 1
    end2 = end1 + 1
    secType = beams[j - 1]
    for i in range(0, numBay):
        ElasticBeamColumn(eleTag, end1, end2, secType, E, 1, M, massType)
        end1 = end2
        end2 = end1 + 1
        eleTag += 1

# calculate eigenvalues & print results
```

(continues on next page)

(continued from previous page)

```
numEigen = 7
eigenValues = eigen(numEigen)
PI = 2 * asin(1.0)

#
# apply loads for static analysis & perform analysis
#

timeSeries('Linear', 1)
pattern('Plain', 1, 1)
load(22, 20.0, 0., 0.)
load(19, 15.0, 0., 0.)
load(16, 12.5, 0., 0.)
load(13, 10.0, 0., 0.)
load(10, 7.5, 0., 0.)
load(7, 5.0, 0., 0.)
load(4, 2.5, 0., 0.)

integrator('LoadControl', 1.0)
algorithm('Linear')
analysis('Static')
analyze(1)

# determine PASS/FAILURE of test
ok = 0

#
# print pretty output of comparisons
#

#               SAP2000   SeismoStruct
comparisonResults = [[1.2732, 0.4313, 0.2420, 0.1602, 0.1190, 0.0951, 0.0795],
                     [1.2732, 0.4313, 0.2420, 0.1602, 0.1190, 0.0951, 0.0795]]
print("\n\nPeriod Comparisons:")
print('{:>10}{:>15}{:>15}{:>15}'.format('Period', 'OpenSees', 'SAP2000', 'SeismoStruct
↪'))

# formatString {%10s%15.5f%15.4f%15.4f}
for i in range(0, numEigen):
    lamb = eigenValues[i]
    period = 2 * PI / sqrt(lamb)
    print('{:>10}{:>15.5f}{:>15.4f}{:>15.4f}'.format(i + 1, period,␣
↪comparisonResults[0][i], comparisonResults[1][i]))
    resultOther = comparisonResults[0][i]
    if abs(period - resultOther) > 9.99e-5:
        ok = 1

# print table of comparision
#       Parameter          SAP2000   SeismoStruct
comparisonResults = [["Disp Top", "Axial Force Bottom Left", "Moment Bottom Left"],
                     [1.45076, 69.99, 2324.68],
                     [1.451, 70.01, 2324.71]]
tolerances = [9.99e-6, 9.99e-3, 9.99e-3]

print("\n\nSatic Analysis Result Comparisons:")
print('{:>30}{:>15}{:>15}{:>15}'.format('Parameter', 'OpenSees', 'SAP2000',
↪'SeismoStruct'))
```

(continues on next page)

(continued from previous page)

```
for i in range(3):
    response = eleResponse(1, 'forces')
    if i == 0:
        result = nodeDisp(22, 1)
    elif i == 1:
        result = abs(response[1])
    else:
        result = response[2]

    print('{:>30}{:>15.3f}{:>15.2f}{:>15.2f}'.format(comparisonResults[0][i],
                                                    result,
                                                    comparisonResults[1][i],
                                                    comparisonResults[2][i]))
    resultOther = comparisonResults[1][i]
    tol = tolerances[i]
    if abs(result - resultOther) > tol:
        ok = 1
        print("failed-> ", i, abs(result - resultOther), tol)

if ok == 0:
    print("PASSED Verification Test PortalFrame2d.py \n\n")
else:
    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!
==========================
```

```
from openseespy.opensees import *

def MomentCurvature(secTag, axialLoad, maxK, numIncr=100):

    # Define two nodes at (0,0)
    node(1, 0.0, 0.0)
    node(2, 0.0, 0.0)

    # Fix all degrees of freedom except axial and bending
    fix(1, 1, 1, 1)
    fix(2, 0, 1, 0)

    # Define element
    #                             tag ndI ndJ  secTag
    element('zeroLengthSection',  1,   1,   2,  secTag)

    # Define constant axial load
    timeSeries('Constant', 1)
    pattern('Plain', 1, 1)
```

(continues on next page)

(continued from previous page)

```
    load(2, axialLoad, 0.0, 0.0)

    # Define analysis parameters
    integrator('LoadControl', 0.0)
    system('SparseGeneral', '-piv')
    test('NormUnbalance', 1e-9, 10)
    numberer('Plain')
    constraints('Plain')
    algorithm('Newton')
    analysis('Static')

    # Do one analysis for constant axial load
    analyze(1)

    # Define reference moment
    timeSeries('Linear', 2)
    pattern('Plain',2, 2)
    load(2, 0.0, 0.0, 1.0)

    # Compute curvature increment
    dK = maxK / numIncr

    # Use displacement control at node 2 for section analysis
    integrator('DisplacementControl', 2,3,dK,1,dK,dK)

    # Do the section analysis
    analyze(numIncr)


wipe()
print("Start MomentCurvature.py example")

# Define model builder
# --------------------
model('basic','-ndm',2,'-ndf',3)

# Define materials for nonlinear columns
# ------------------------------------------
# CONCRETE                  tag   f'c        ec0   f'cu        ecu
# Core concrete (confined)
uniaxialMaterial('Concrete01',1, -6.0,  -0.004,  -5.0,  -0.014)

# Cover concrete (unconfined)
uniaxialMaterial('Concrete01',2, -5.0,  -0.002,  0.0,  -0.006)

# STEEL
# Reinforcing steel
fy = 60.0      # Yield stress
E = 30000.0    # Young's modulus

#                       tag  fy E0    b
uniaxialMaterial('Steel01', 3, fy, E, 0.01)

# Define cross-section for nonlinear columns
# ------------------------------------------

# set some paramaters
```

(continues on next page)

(continued from previous page)

```
colWidth = 15
colDepth = 24

cover = 1.5
As = 0.60;      # area of no. 7 bars

# some variables derived from the parameters
y1 = colDepth/2.0
z1 = colWidth/2.0


section('Fiber', 1)

# Create the concrete core fibers
patch('rect',1,10,1 ,cover-y1, cover-z1, y1-cover, z1-cover)

# Create the concrete cover fibers (top, bottom, left, right)
patch('rect',2,10,1 ,-y1, z1-cover, y1, z1)
patch('rect',2,10,1 ,-y1, -z1, y1, cover-z1)
patch('rect',2,2,1 ,-y1, cover-z1, cover-y1, z1-cover)
patch('rect',2,2,1 ,y1-cover, cover-z1, y1, z1-cover)

# Create the reinforcing fibers (left, middle, right)
layer('straight', 3, 3, As, y1-cover, z1-cover, y1-cover, cover-z1)
layer('straight', 3, 2, As, 0.0     , z1-cover, 0.0      , cover-z1)
layer('straight', 3, 3, As, cover-y1, z1-cover, cover-y1, cover-z1)

# Estimate yield curvature
# (Assuming no axial load and only top and bottom steel)
# d -- from cover to rebar
d = colDepth-cover
# steel yield strain
epsy = fy/E
Ky = epsy/(0.7*d)

# Print estimate to standard output
print("Estimated yield curvature: ", Ky)

# Set axial load
P = -180.0

# Target ductility for analysis
mu = 15.0

# Number of analysis increments
numIncr = 100

# Call the section analysis procedure
MomentCurvature(1, P, Ky*mu, numIncr)

results = open('results.out','a+')

u = nodeDisp(2,3)
if abs(u-0.0019047619047619541)<1e-12:
    results.write('PASSED : MomentCurvature.py\n');
    print("Passed!")
else:
```

(continues on next page)

(continued from previous page)

```
    results.write('FAILED : MomentCurvature.py\n');
    print("Failed!")

results.close()

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.

```
print("==========================")

from openseespy.opensees import *

print("Starting RCFrameGravity example")

# Create ModelBuilder (with two-dimensions and 3 DOF/node)
model('basic', '-ndm', 2, '-ndf', 3)

# Create nodes
# ------------

# Set parameters for overall model geometry
width = 360.0
height = 144.0

# Create nodes
#    tag, X, Y
node(1, 0.0, 0.0)
node(2, width, 0.0)
node(3, 0.0, height)
node(4, width, height)

# Fix supports at base of columns
#   tag, DX, DY, RZ
fix(1, 1, 1, 1)
fix(2, 1, 1, 1)

# Define materials for nonlinear columns
# ------------------------------------------
# CONCRETE                  tag  f'c    ec0    f'cu   ecu
# Core concrete (confined)
uniaxialMaterial('Concrete01', 1, -6.0, -0.004, -5.0, -0.014)

# Cover concrete (unconfined)
uniaxialMaterial('Concrete01', 2, -5.0, -0.002, 0.0, -0.006)

# STEEL
# Reinforcing steel
fy = 60.0;  # Yield stress
E = 30000.0;  # Young's modulus
#                          tag  fy E0    b
```

(continues on next page)

(continued from previous page)

```
uniaxialMaterial('Steel01', 3, fy, E, 0.01)

# Define cross-section for nonlinear columns
# ------------------------------------------

#  some parameters
colWidth = 15
colDepth = 24

cover = 1.5
As = 0.60  # area of no. 7 bars

# some variables derived from the parameters
y1 = colDepth / 2.0
z1 = colWidth / 2.0

section('Fiber', 1)

# Create the concrete core fibers
patch('rect', 1, 10, 1, cover - y1, cover - z1, y1 - cover, z1 - cover)

# Create the concrete cover fibers (top, bottom, left, right)
patch('rect', 2, 10, 1, -y1, z1 - cover, y1, z1)
patch('rect', 2, 10, 1, -y1, -z1, y1, cover - z1)
patch('rect', 2, 2, 1, -y1, cover - z1, cover - y1, z1 - cover)
patch('rect', 2, 2, 1, y1 - cover, cover - z1, y1, z1 - cover)

# Create the reinforcing fibers (left, middle, right)
layer('straight', 3, 3, As, y1 - cover, z1 - cover, y1 - cover, cover - z1)
layer('straight', 3, 2, As, 0.0, z1 - cover, 0.0, cover - z1)
layer('straight', 3, 3, As, cover - y1, z1 - cover, cover - y1, cover - z1)

# Define column elements
# ----------------------

# Geometry of column elements
#                tag

geomTransf('PDelta', 1)

# Number of integration points along length of element
np = 5

# Lobatto integratoin
beamIntegration('Lobatto', 1, 1, np)

# Create the coulumns using Beam-column elements
#               e            tag ndI ndJ transfTag integrationTag
eleType = 'forceBeamColumn'
element(eleType, 1, 1, 3, 1, 1)
element(eleType, 2, 2, 4, 1, 1)

# Define beam elment
# ------------------------------

# Geometry of column elements
#                tag
```

(continues on next page)

(continued from previous page)

```
geomTransf('Linear', 2)

# Create the beam element
#                           tag, ndI, ndJ, A,      E,     Iz, transfTag
element('elasticBeamColumn', 3, 3, 4, 360.0, 4030.0, 8640.0, 2)

# Define gravity loads
# --------------------

#  a parameter for the axial load
P = 180.0;  # 10% of axial capacity of columns

# Create a Plain load pattern with a Linear TimeSeries
timeSeries('Linear', 1)
pattern('Plain', 1, 1)

# Create nodal loads at nodes 3 & 4
#    nd  FX,  FY, MZ
load(3, 0.0, -P, 0.0)
load(4, 0.0, -P, 0.0)

# ------------------------------
# End of model generation
# ------------------------------


# ------------------------------
# Start of analysis generation
# ------------------------------

# Create the system of equation, a sparse solver with partial pivoting
system('BandGeneral')

# Create the constraint handler, the transformation method
constraints('Transformation')

# Create the DOF numberer, the reverse Cuthill-McKee algorithm
numberer('RCM')

# Create the convergence test, the norm of the residual with a tolerance of
# 1e-12 and a max number of iterations of 10
test('NormDispIncr', 1.0e-12, 10, 3)

# Create the solution algorithm, a Newton-Raphson algorithm
algorithm('Newton')

# Create the integration scheme, the LoadControl scheme using steps of 0.1
integrator('LoadControl', 0.1)

# Create the analysis object
analysis('Static')

# ------------------------------
# End of analysis generation
# ------------------------------


```

(continues on next page)

(continued from previous page)

```
# ------------------------------
# Finally perform the analysis
# ------------------------------

# perform the gravity load analysis, requires 10 steps to reach the load level
analyze(10)

# Print out the state of nodes 3 and 4
# print node 3 4

# Print out the state of element 1
# print ele 1

u3 = nodeDisp(3, 2)
u4 = nodeDisp(4, 2)

results = open('results.out', 'a+')

if abs(u3 + 0.0183736) < 1e-6 and abs(u4 + 0.0183736) < 1e-6:
    results.write('PASSED : RCFrameGravity.py\n')
    print("Passed!")
else:
    results.write('FAILED : RCFrameGravity.py\n')
    print("Failed!")

results.close()

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.

```
print("============================")
print("Start RCFramePushover Example")

# Units: kips, in, sec
#
# Written: GLF/MHS/fmk
# Date: January 2001
from openseespy.opensees import *

wipe()
# ----------------------------------------------------
# Start of Model Generation & Initial Gravity Analysis
# ----------------------------------------------------

# Do operations of Example3.1 by sourcing in the tcl file
import RCFrameGravity
print("Gravity Analysis Completed")

# Set the gravity loads to be constant & reset the time in the domain
```

(continues on next page)

(continued from previous page)

```
loadConst('-time', 0.0)

# ------------------------------------------------
# End of Model Generation & Initial Gravity Analysis
# ------------------------------------------------


# ------------------------------------------------
# Start of additional modelling for lateral loads
# ------------------------------------------------

# Define lateral loads
# --------------------

# Set some parameters
H = 10.0  # Reference lateral load

# Set lateral load pattern with a Linear TimeSeries
pattern('Plain', 2, 1)

# Create nodal loads at nodes 3 & 4
#    nd    FX  FY  MZ
load(3, H, 0.0, 0.0)
load(4, H, 0.0, 0.0)

# ------------------------------------------------
# End of additional modelling for lateral loads
# ------------------------------------------------


# ------------------------------------------------
# Start of modifications to analysis for push over
# ------------------------------------------------

# Set some parameters
dU = 0.1  # Displacement increment

# Change the integration scheme to be displacement control
#                             node dof init Jd min max
integrator('DisplacementControl', 3, 1, dU, 1, dU, dU)

# ------------------------------------------------
# End of modifications to analysis for push over
# ------------------------------------------------


# ------------------------------
# Start of recorder generation
# ------------------------------

# Stop the old recorders by destroying them
# remove recorders

# Create a recorder to monitor nodal displacements
# recorder Node -file node32.out -time -node 3 4 -dof 1 2 3 disp

# Create a recorder to monitor element forces in columns
```

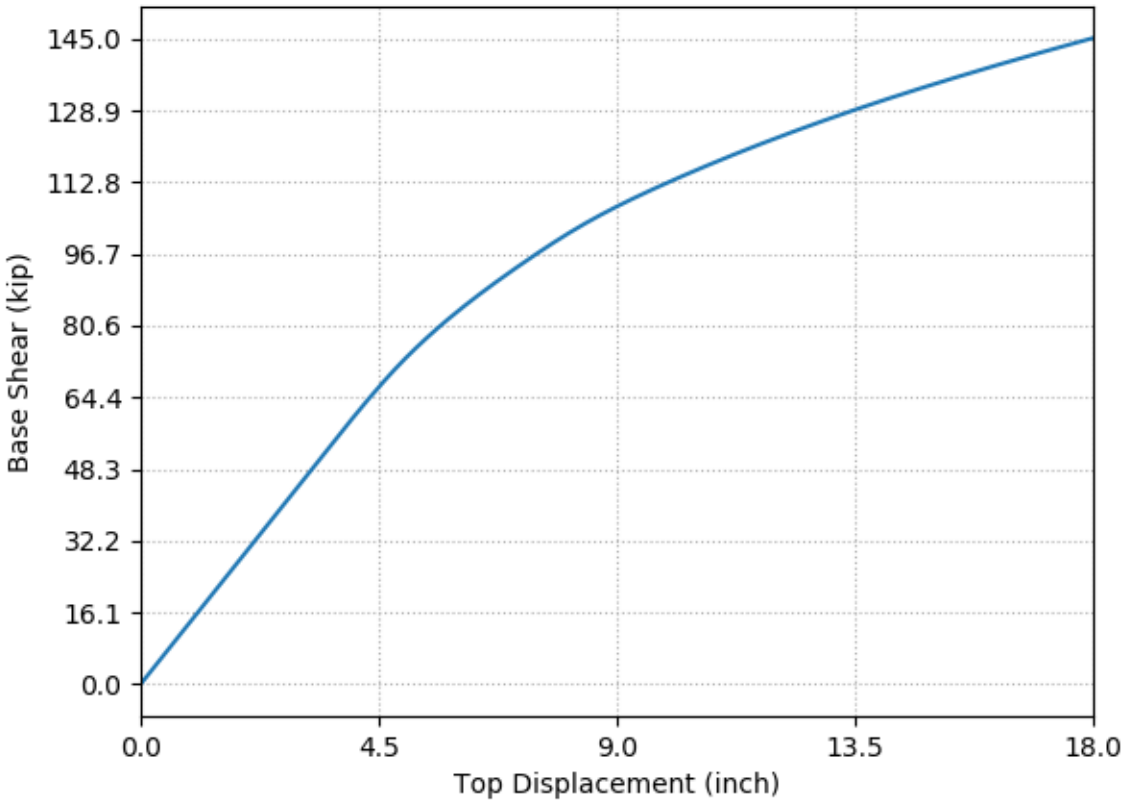
(continues on next page)

(continued from previous page)

```
# recorder EnvelopeElement -file ele32.out -time -ele 1 2 forces

# ------------------------------
# End of recorder generation
# ------------------------------


# ------------------------------
# Finally perform the analysis
# ------------------------------

# Set some parameters
maxU = 15.0  # Max displacement
currentDisp = 0.0
ok = 0

test('NormDispIncr', 1.0e-12, 1000)
algorithm('ModifiedNewton', '-initial')

while ok == 0 and currentDisp < maxU:

    ok = analyze(1)

    # if the analysis fails try initial tangent iteration
    if ok != 0:
        print("modified newton failed")
        break
    # print "regular newton failed .. lets try an initail stiffness for this step"
    # test('NormDispIncr', 1.0e-12,  1000)
    # # algorithm('ModifiedNewton', '-initial')
    # ok = analyze(1)
    # if ok == 0:
    #     print "that worked .. back to regular newton"

    # test('NormDispIncr', 1.0e-12,  10)
    # algorithm('Newton')

    currentDisp = nodeDisp(3, 1)

results = open('results.out', 'a+')

if ok == 0:
    results.write('PASSED : RCFramePushover.py\n')
    print("Passed!")
else:
    results.write('FAILED : RCFramePushover.py\n')
    print("Failed!")

results.close()

# Print the state at node 3
# print node 3


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  ##########################################################################
3  ## 2D steel frame example.
4  ## 3 story steel building with rigid beam-column connections.
5  ## This script uses W-section command inOpensees to create steel..
6  ## .. beam-column fiber sections.
7  ##
8  ## By - Anurag Upadhyay, PhD Student, University of Utah.
9  ## Date - 08/06/2018
10 ##########################################################################
11
12 print("============================================================")
13 print("Start 2D Steel Frame Example")
14
15 from openseespy.opensees import *
16
17 import numpy as np
18 import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
import os

AnalysisType='Pushover'        ;                #  Pushover  Gravity

## ------------------------------
## Start of model generation
## ------------------------------
# remove existing model
wipe()

# set modelbuilder
model('basic', '-ndm', 2, '-ndf', 3)

import math

###########################################
### Units and Constants  ##################
###########################################

inch = 1;
kip = 1;
sec = 1;

# Dependent units
sq_in = inch*inch;
ksi = kip/sq_in;
ft = 12*inch;

# Constants
g = 386.2*inch/(sec*sec);
pi = math.acos(-1);

#######################################
##### Dimensions
#######################################

# Dimensions Input
H_story=10.0*ft;
W_bayX=16.0*ft;
W_bayY_ab=5.0*ft+10.0*inch;
W_bayY_bc=8.0*ft+4.0*inch;
W_bayY_cd=5.0*ft+10.0*inch;

# Calculated dimensions
W_structure=W_bayY_ab+W_bayY_bc+W_bayY_cd;

################
### Material
################

# Steel02 Material

matTag=1;
matConnAx=2;
matConnRot=3;

Fy=60.0*ksi;                # Yield stress
```

(continues on next page)

(continued from previous page)

```
Es=29000.0*ksi;                 # Modulus of Elasticity of Steel
v=0.2;                                  # Poisson's ratio
Gs=Es/(1+v);                  # Shear modulus
b=0.10;                                 # Strain hardening ratio
params=[18.0,0.925,0.15]                # R0,cR1,cR2
R0=18.0
cR1=0.925
cR2=0.15
a1=0.05
a2=1.00
a3=0.05
a4=1.0
sigInit=0.0
alpha=0.05

uniaxialMaterial('Steel02', matTag, Fy, Es, b, R0, cR1, cR2, a1, a2, a3, a4, sigInit)

# ##################
# ## Sections
# ##################

colSecTag1=1;
colSecTag2=2;
beamSecTag1=3;
beamSecTag2=4;
beamSecTag3=5;

# COMMAND: section('WFSection2d', secTag, matTag, d, tw, bf, tf, Nfw, Nff)

section('WFSection2d', colSecTag1, matTag, 10.5*inch, 0.26*inch, 5.77*inch, 0.44*inch,
↪ 15, 16)                # outer Column
section('WFSection2d', colSecTag2, matTag, 10.5*inch, 0.26*inch, 5.77*inch, 0.44*inch,
↪ 15, 16)                # Inner Column

section('WFSection2d', beamSecTag1, matTag, 8.3*inch, 0.44*inch, 8.11*inch, 0.
↪685*inch, 15, 15)               # outer Beam
section('WFSection2d', beamSecTag2, matTag, 8.2*inch, 0.40*inch, 8.01*inch, 0.
↪650*inch, 15, 15)               # Inner Beam
section('WFSection2d', beamSecTag3, matTag, 8.0*inch, 0.40*inch, 7.89*inch, 0.
↪600*inch, 15, 15)               # Inner Beam

# Beam size - W10x26
Abeam=7.61*inch*inch;
IbeamY=144.*(inch**4);                          # Inertia along horizontal axis
IbeamZ=14.1*(inch**4);                          # inertia along vertical axis

# BRB input data
Acore=2.25*inch;
Aend=10.0*inch;
LR_BRB=0.55;

# ############################
# ##### Nodes
# ############################

# Create All main nodes
node(1, 0.0, 0.0)
```

(continues on next page)

(continued from previous page)

```
node(2, W_bayX, 0.0)
node(3, 2*W_bayX, 0.0)

node(11, 0.0, H_story)
node(12, W_bayX, H_story)
node(13, 2*W_bayX, H_story)

node(21, 0.0, 2*H_story)
node(22, W_bayX, 2*H_story)
node(23, 2*W_bayX, 2*H_story)

node(31, 0.0, 3*H_story)
node(32, W_bayX, 3*H_story)
node(33, 2*W_bayX, 3*H_story)

# Beam Connection nodes

node(1101, 0.0, H_story)
node(1201, W_bayX, H_story)
node(1202, W_bayX, H_story)
node(1301, 2*W_bayX, H_story)

node(2101, 0.0, 2*H_story)
node(2201, W_bayX, 2*H_story)
node(2202, W_bayX, 2*H_story)
node(2301, 2*W_bayX, 2*H_story)

node(3101, 0.0, 3*H_story)
node(3201, W_bayX, 3*H_story)
node(3202, W_bayX, 3*H_story)
node(3301, 2*W_bayX, 3*H_story)

# ###############
#  Constraints
# ###############

fix(1, 1, 1, 1)
fix(2, 1, 1, 1)
fix(3, 1, 1, 1)

# ######################
# ### Elements
# ######################

# ### Assign beam-integration tags

ColIntTag1=1;
ColIntTag2=2;
BeamIntTag1=3;
BeamIntTag2=4;
BeamIntTag3=5;

beamIntegration('Lobatto', ColIntTag1, colSecTag1, 4)
beamIntegration('Lobatto', ColIntTag2, colSecTag2, 4)
beamIntegration('Lobatto', BeamIntTag1, beamSecTag1, 4)
beamIntegration('Lobatto', BeamIntTag2, beamSecTag2, 4)
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)

```
################
## Gravity Load
################
# create TimeSeries
timeSeries("Linear", 1)

# create a plain load pattern
pattern("Plain", 1, 1)

# Create the nodal load
load(11, 0.0, -5.0*kip, 0.0)
load(12, 0.0, -6.0*kip, 0.0)
load(13, 0.0, -5.0*kip, 0.0)

load(21, 0., -5.*kip, 0.0)
load(22, 0., -6.*kip,0.0)
load(23, 0., -5.*kip, 0.0)

load(31, 0., -5.*kip, 0.0)
load(32, 0., -6.*kip, 0.0)
load(33, 0., -5.*kip, 0.0)


# ------------------------------
# Start of analysis generation
# ------------------------------

NstepsGrav = 10

system("BandGEN")
numberer("Plain")
constraints("Plain")
integrator("LoadControl", 1.0/NstepsGrav)
algorithm("Newton")
test('NormUnbalance',1e-8, 10)
analysis("Static")


# perform the analysis
data = np.zeros((NstepsGrav+1,2))
for j in range(NstepsGrav):
    analyze(1)
    data[j+1,0] = nodeDisp(31,2)
    data[j+1,1] = getLoadFactor(1)*5

loadConst('-time', 0.0)

print("Gravity analysis complete")

wipeAnalysis()

##############################
### PUSHOVER ANALYSIS
##############################

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', '-
    ↪node', 2, '-dof',1, 'reaction')
326         recorder('Node', '-file', "PushoverOut/Node31Disp.out", '-closeOnWrite', '-
    ↪node', 31, '-dof',1, 'disp')
327         recorder('Element', '-file', "PushoverOut/BeamStress.out", '-closeOnWrite', '-
    ↪ele', 102, 'section', '4', 'fiber','1', 'stressStrain')
328
329         # analyze(NstepsPush)
330
331         # Perform pushover analysis
332         dataPush = np.zeros((NstepsPush+1,5))
333         for j in range(NstepsPush):
334                 analyze(1)
335                 dataPush[j+1,0] = nodeDisp(31,1)
336                 reactions()
337                 dataPush[j+1,1] = nodeReaction(1, 1) + nodeReaction(2, 1) +␣
    ↪nodeReaction(3, 1)
338
339         plt.plot(dataPush[:,0], -dataPush[:,1])
340         plt.xlim(0, MaxDisp)
341         plt.xticks(np.linspace(0,MaxDisp,5,endpoint=True))
342         plt.yticks(np.linspace(0, -int(dataPush[NstepsPush,1]),10,endpoint=True))
343         plt.grid(linestyle='dotted')
344         plt.xlabel('Top Displacement (inch)')
345         plt.ylabel('Base Shear (kip)')
346         plt.show()
347
348
349         print("Pushover analysis complete")
```

(continues on next page)

(continued from previous page)

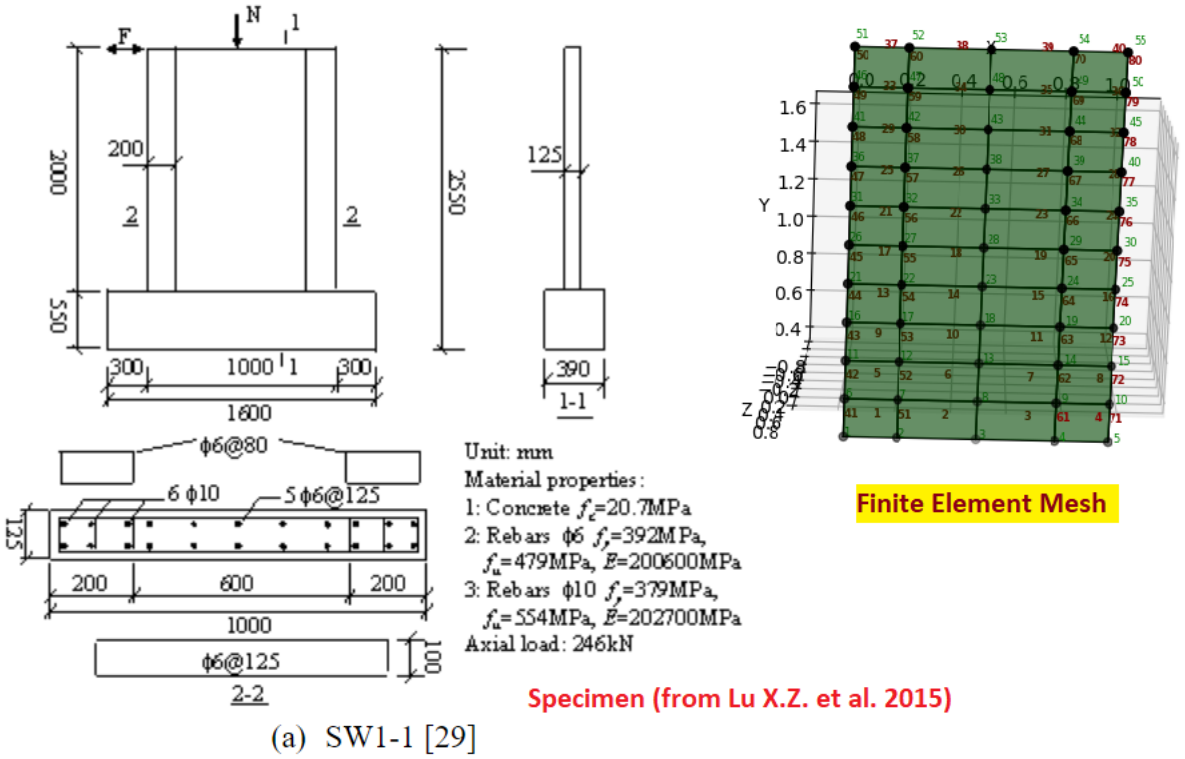
```
350
351
352
353
```

Cantilever FRP-Confined Circular Reinforced Concrete Column under Cyclic Lateral Loading

1. The source code is developed by Michael Haas & Konstantinos G. Megalooikonomou, German Research Centre for Geosciences (GFZ)
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.

Reinforced Concrete Shear Wall with Special Boundary Elements

1. The original code was written for OpenSees Tcl by *Lu X.Z. et al. (2015) <http://www.luxinzheng.net/download/OpenSEES/Examples_of_NLDKGQ_element.htm>*.
2. The source code is converted to OpenSeesPy by Anurag Upadhyay from University of Utah.
3. Four node shell elements with LayeredShell sections are used to model the shear wall.
4. The source code is shown below, which can be downloaded `here`.
5. Download the cyclic test load input and output files, `RCshearwall_Load_input`, `RCshearwall_TestOutput`.
6. The details of the shear wall specimen are shown in the figure below, along with the finite element mesh.
7. Run the source code and you should see the cyclic test plot overlaid by a pushover curve, shown at the end.



(a) SW1-1 [29]

```
1
2  # Converted to openseespy by: Anurag Upadhyay, University of Utah.
3  # Units: N and m to follow the originally published code.
4
5  from openseespy.postprocessing.Get_Rendering import *
6  from openseespy.opensees import *
7
8  import numpy as np
9  import matplotlib.pyplot as plt
10 import os
11 import math
12
13 pi = 3.1415
14
15 AnalysisType = "Pushover" # Cyclic   Pushover   Gravity
16
17 wipe()
18
19 model('basic','-ndm',3,'-ndf',6)
20
21 ###################################
22 ## Define Material
23 ###################################
24
25 # Define PSUMAT and convert it to plane stress material
26 nDMaterial('PlaneStressUserMaterial',1,40,7,20.7e6,2.07e6,-4.14e6,-0.002,-0.01,0.001,
   ↪0.3)
27 nDMaterial('PlateFromPlaneStress',4,1,1.25e10)
28
29 # Define material for rebar
```

(continues on next page)

(continued from previous page)

```
uniaxialMaterial('Steel02',7,379e6,202.7e9,0.01,18.5,0.925,0.15)
uniaxialMaterial('Steel02',8,392e6,200.6e9,0.01,18.5,0.925,0.15)

# Convert rebar material to plane stress/plate rebar
# Angle 0 is for vertical rebar and 90 is for horizontal rebar
nDMaterial('PlateRebar',9,7,90.0)
nDMaterial('PlateRebar',10,8,90.0)
nDMaterial('PlateRebar',11,8,0.0)

# Define LayeredShell sections. Section 1 is used for the special boundary elements␣
↪and section 2 is used for the unconfined interior wall portion
section('LayeredShell',1,10,4,0.0125,11,0.0002403,11,0.0003676,4,0.024696,4,0.024696,
↪4,0.024696,4,0.024696,11,0.0003676,11,0.0002403,4,0.0125)
section('LayeredShell',2,8,4,0.0125,11,0.0002403,10,0.0002356,4,0.0495241,4,0.0495241,
↪10,0.0002356,11,0.0002403,4,0.0125)

# ##################
# NODES
# ##################
#define nodes
node(1,0.0,0,0)
node(2,0.2,0,0)
node(3,0.5,0,0)
node(4,0.8,0,0)
node(5,1.0,0,0)

node(6,0.0,0.2,0)
node(7,0.2,0.2,0)
node(8,0.5,0.2,0)
node(9,0.8,0.2,0)
node(10,1.0,0.2,0)

node(11,0.0,0.4,0)
node(12,0.2,0.4,0)
node(13,0.5,0.4,0)
node(14,0.8,0.4,0)
node(15,1.0,0.4,0)

node(16,0.0,0.6,0)
node(17,0.2,0.6,0)
node(18,0.5,0.6,0)
node(19,0.8,0.6,0)
node(20,1.0,0.6,0)

node(21,0.0,0.8,0)
node(22,0.2,0.8,0)
node(23,0.5,0.8,0)
node(24,0.8,0.8,0)
node(25,1.0,0.8,0)

node(26,0.0,1.0,0)
node(27,0.2,1.0,0)
node(28,0.5,1.0,0)
node(29,0.8,1.0,0)
node(30,1.0,1.0,0)

node(31,0.0,1.2,0)
```

(continues on next page)

(continued from previous page)

```
node(32,0.2,1.2,0)
node(33,0.5,1.2,0)
node(34,0.8,1.2,0)
node(35,1.0,1.2,0)

node(36,0.0,1.4,0)
node(37,0.2,1.4,0)
node(38,0.5,1.4,0)
node(39,0.8,1.4,0)
node(40,1.0,1.4,0)

node(41,0.0,1.6,0)
node(42,0.2,1.6,0)
node(43,0.5,1.6,0)
node(44,0.8,1.6,0)
node(45,1.0,1.6,0)

node(46,0.0,1.8,0)
node(47,0.2,1.8,0)
node(48,0.5,1.8,0)
node(49,0.8,1.8,0)
node(50,1.0,1.8,0)

node(51,0.0,2.0,0)
node(52,0.2,2.0,0)
node(53,0.5,2.0,0)
node(54,0.8,2.0,0)
node(55,1.0,2.0,0)

##########################
# ELEMENTS
##########################

ShellType = "ShellNLDKGQ"
# ShellType = "ShellMITC4"

element(ShellType,1,1,2,7,6,1)
element(ShellType,2,2,3,8,7,2)
element(ShellType,3,3,4,9,8,2)
element(ShellType,4,4,5,10,9,1)

element(ShellType,5,6,7,12,11,1)
element(ShellType,6,7,8,13,12,2)
element(ShellType,7,8,9,14,13,2)
element(ShellType,8,9,10,15,14,1)

element(ShellType,9,11,12,17,16,1)
element(ShellType,10,12,13,18,17,2)
element(ShellType,11,13,14,19,18,2)
element(ShellType,12,14,15,20,19,1)

element(ShellType,13,16,17,22,21,1)
element(ShellType,14,17,18,23,22,2)
element(ShellType,15,18,19,24,23,2)
element(ShellType,16,19,20,25,24,1)

element(ShellType,17,21,22,27,26,1)
```

(continues on next page)

(continued from previous page)

```
element(ShellType,18,22,23,28,27,2)
element(ShellType,19,23,24,29,28,2)
element(ShellType,20,24,25,30,29,1)

element(ShellType,21,26,27,32,31,1)
element(ShellType,22,27,28,33,32,2)
element(ShellType,23,28,29,34,33,2)
element(ShellType,24,29,30,35,34,1)

element(ShellType,25,31,32,37,36,1)
element(ShellType,26,32,33,38,37,2)
element(ShellType,27,33,34,39,38,2)
element(ShellType,28,34,35,40,39,1)

element(ShellType,29,36,37,42,41,1)
element(ShellType,30,37,38,43,42,2)
element(ShellType,31,38,39,44,43,2)
element(ShellType,32,39,40,45,44,1)

element(ShellType,33,41,42,47,46,1)
element(ShellType,34,42,43,48,47,2)
element(ShellType,35,43,44,49,48,2)
element(ShellType,36,44,45,50,49,1)

element(ShellType,37,46,47,52,51,1)
element(ShellType,38,47,48,53,52,2)
element(ShellType,39,48,49,54,53,2)
element(ShellType,40,49,50,55,54,1)

# P-delta columns

element('truss',41,1,6,223.53e-6,7)
element('truss',42,6,11,223.53e-6,7)
element('truss',43,11,16,223.53e-6,7)
element('truss',44,16,21,223.53e-6,7)
element('truss',45,21,26,223.53e-6,7)
element('truss',46,26,31,223.53e-6,7)
element('truss',47,31,36,223.53e-6,7)
element('truss',48,36,41,223.53e-6,7)
element('truss',49,41,46,223.53e-6,7)
element('truss',50,46,51,223.53e-6,7)

element('truss',51,2,7,223.53e-6,7)
element('truss',52,7,12,223.53e-6,7)
element('truss',53,12,17,223.53e-6,7)
element('truss',54,17,22,223.53e-6,7)
element('truss',55,22,27,223.53e-6,7)
element('truss',56,27,32,223.53e-6,7)
element('truss',57,32,37,223.53e-6,7)
element('truss',58,37,42,223.53e-6,7)
element('truss',59,42,47,223.53e-6,7)
element('truss',60,47,52,223.53e-6,7)

element('truss',61,4,9,223.53e-6,7)
element('truss',62,9,14,223.53e-6,7)
element('truss',63,14,19,223.53e-6,7)
element('truss',64,19,24,223.53e-6,7)
```

(continues on next page)

(continued from previous page)

```
element('truss',65,24,29,223.53e-6,7)
element('truss',66,29,34,223.53e-6,7)
element('truss',67,34,39,223.53e-6,7)
element('truss',68,39,44,223.53e-6,7)
element('truss',69,44,49,223.53e-6,7)
element('truss',70,49,54,223.53e-6,7)

element('truss',71,5,10,223.53e-6,7)
element('truss',72,10,15,223.53e-6,7)
element('truss',73,15,20,223.53e-6,7)
element('truss',74,20,25,223.53e-6,7)
element('truss',75,25,30,223.53e-6,7)
element('truss',76,30,35,223.53e-6,7)
element('truss',77,35,40,223.53e-6,7)
element('truss',78,40,45,223.53e-6,7)
element('truss',79,45,50,223.53e-6,7)
element('truss',80,50,55,223.53e-6,7)

# Fix all bottom nodes
fixY(0.0,1,1,1,1,1,1)

# plot_model()

recorder('Node','-file','ReactionPY.txt','-time','-node',1,2,3,4,5,'-dof',1,'reaction
↪')

############################
# Gravity Analysis
############################

print("running gravity")

timeSeries("Linear", 1)                                          # create TimeSeries␣
↪for gravity analysis
pattern('Plain',1,1)
load(53,0,-246000.0,0.0,0.0,0.0,0.0)         # apply vertical load

recorder('Node','-file','Disp.txt','-time','-node',53,'-dof',1,'disp')

constraints('Plain')
numberer('RCM')
system('BandGeneral')
test('NormDispIncr',1.0e-4,200)
algorithm('BFGS','-count',100)
integrator('LoadControl',0.1)
analysis('Static')
analyze(10)

print("gravity analysis complete...")

loadConst('-time',0.0)                                           # Keep the gravity␣
↪loads for further analysis

wipeAnalysis()

##############################
### Cyclic ANALYSIS
```

(continues on next page)

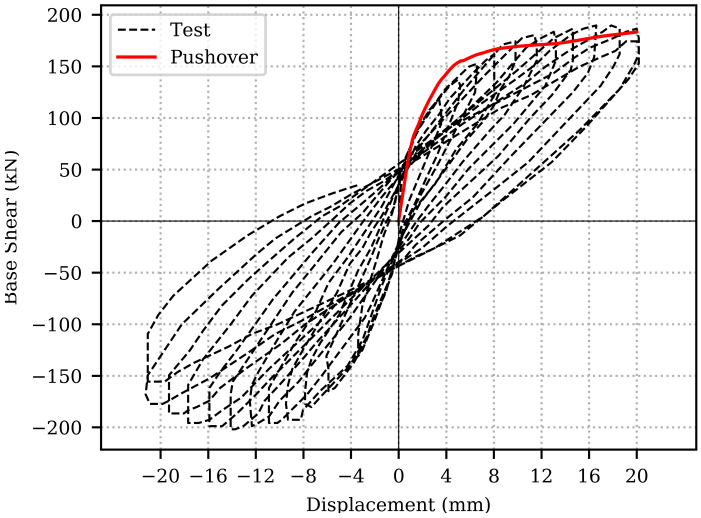
(continued from previous page)

```
################################

if(AnalysisType=="Cyclic"):

        # This is a load controlled analysis. The input load file "RCshearwall_Load_input.txt" should be in the
        # .. same folder as the model file.

        print("<<<< Running Cyclic Analysis >>>>")

        timeSeries('Path',2,'-dt',0.1,'-filePath','RCshearwall_Load_input.txt')
        pattern('Plain',2,2)
        sp(53,1,1)                                                                  # construct a single-point constraint object added to the LoadPattern.

        constraints('Penalty',1e20,1e20)
        numberer('RCM')
        system('BandGeneral')
        test('NormDispIncr',1e-05, 100, 1)
        algorithm('KrylovNewton')
        integrator('LoadControl',0.1)
        analysis('Static')
        analyze(700)


#######################
# PUSHOVER ANALYSIS
#######################

if(AnalysisType=="Pushover"):

        print("<<<< Running Pushover Analysis >>>>")

        # create a plain load pattern for pushover analysis
        pattern("Plain", 2, 1)

        ControlNode=53
        ControlDOF=1
        MaxDisp= 0.020
        DispIncr=0.00001
        NstepsPush=int(MaxDisp/DispIncr)

        load(ControlNode, 1.00, 0.0, 0.0, 0.0, 0.0, 0.0)        # Apply a unit reference load in DOF=1

        system("BandGeneral")
        numberer("RCM")
        constraints('Penalty',1e20,1e20)
        integrator("DisplacementControl", ControlNode, ControlDOF, DispIncr)
        algorithm('KrylovNewton')
        test('NormDispIncr',1e-05, 1000, 2)
        analysis("Static")

        # Create a folder to put the output
        PushDataDir = r'PushoverOut'
        if not os.path.exists(PushDataDir):
                os.makedirs(PushDataDir)
```

(continues on next page)

(continued from previous page)

```
        recorder('Node', '-file', "PushoverOut/React.out", '-closeOnWrite', '-node', 1, 2, 3, 4, 5, '-dof',1, 'reaction')
        recorder('Node', '-file', "PushoverOut/Disp.out", '-closeOnWrite', '-node', ControlNode, '-dof',1, 'disp')

        # Perform pushover analysis
        dataPush = np.zeros((NstepsPush+1,5))
        for j in range(NstepsPush):
                analyze(1)
                dataPush[j+1,0] = nodeDisp(ControlNode,1)*1000                  # Convert to mm
                dataPush[j+1,1] = -getLoadFactor(2)*0.001                       # Convert to kN

        # Read test output data to plot
        Test = np.loadtxt("RCshearwall_TestOutput.txt", delimiter="\t", unpack="False")

        ## Set parameters for the plot
        plt.rcParams.update({'font.size': 7})
        plt.figure(figsize=(4,3), dpi=100)
        plt.rc('font', family='serif')
        plt.plot(Test[0,:], Test[1,:], color="black", linewidth=0.8, linestyle="--", label='Test')
        plt.plot(dataPush[:,0], -dataPush[:,1], color="red", linewidth=1.2, linestyle="-", label='Pushover')
        plt.axhline(0, color='black', linewidth=0.4)
        plt.axvline(0, color='black', linewidth=0.4)
        plt.xlim(-25, 25)
        plt.xticks(np.linspace(-20,20,11,endpoint=True))
        plt.grid(linestyle='dotted')
        plt.xlabel('Displacement (mm)')
        plt.ylabel('Base Shear (kN)')
        plt.legend()
        plt.savefig("PushoverOut/RCshearwall_PushoverCurve.png",dpi=1200)
        plt.show()


        print("Pushover analysis complete")
```



1.14.2 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*

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 favorate Python program and should see results below

```
============================================================
Start cantilever 2D EQ ground motion with gravity example
u2 =  -0.07441860465116278
Passed!
========================================
```

```python
print("============================================================")
print("Start cantilever 2D EQ ground motion with gravity example")

from openseespy.opensees import *


# ----------------------------------------------------------------------------------------------
# Example 1. cantilever 2D
# EQ ground motion with gravity
# all units are in kip, inch, second
# elasticBeamColumn ELEMENT
#                 Silvia Mazzoni & Frank McKenna, 2006
#
#    ^Y
#    |
#    2       __
#    |         |
#    |         |
#    |         |
#  (1)      36'
#    |         |
#    |         |
#    |         |
#  =1=    ----  --------->X
#

# SET UP ---------------------------------------------------------------------------
wipe()                                                  # clear opensees model
model('basic', '-ndm', 2, '-ndf', 3)                # 2 dimensions, 3 dof per node
# file mkdir data                                     # create data directory

# define GEOMETRY -------------------------------------------------------------
# nodal coordinates:
node(1, 0., 0.)                                          # node#, X Y
node(2, 0., 432.)

# Single point constraints -- Boundary Conditions
fix(1, 1, 1, 1)                                       # node DX DY RZ

# nodal masses:
mass(2, 5.18, 0., 0.)                               # node#, Mx My Mz, Mass=Weight/g.

# Define ELEMENTS -------------------------------------------------------------
# define geometric transformation: performs a linear geometric transformation of beam stiffness and resisting force from the basic system to the global-coordinate system
geomTransf('Linear', 1)                             # associate a tag to transformation

# connectivity:
element('elasticBeamColumn', 1, 1, 2, 3600.0, 3225.0,1080000.0, 1)

```

(continues on next page)

(continued from previous page)

```
# define GRAVITY -------------------------------------------------------------
timeSeries('Linear', 1)
pattern('Plain', 1, 1,)
load(2, 0., -2000., 0.)                                  # node#, FX FY MZ -- superstructure-weight

constraints('Plain')                                     # how it handles boundary conditions
numberer('Plain')                                  # renumber dof's to minimize band-width (optimization), if you want to
system('BandGeneral')                           # how to store and solve the system of equations in the analysis
algorithm('Linear')                        # use Linear algorithm for linear analysis
integrator('LoadControl', 0.1)                           # determine the next time step for an analysis, # apply gravity in 10 steps
analysis('Static')                                             # define type of analysis static or transient
analyze(10)                                                 # perform gravity analysis
loadConst('-time', 0.0)                                  # hold gravity constant and restart time

# DYNAMIC ground-motion analysis -------------------------------------------------------
# create load pattern
G = 386.0
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)
pattern('UniformExcitation', 2, 1, '-accel', 2)                          # define where and how (pattern tag, dof) acceleration is applied

# set damping based on first eigen mode
freq = eigen('-fullGenLapack', 1)**0.5
dampRatio = 0.02
rayleigh(0., 0., 0., 2*dampRatio/freq)

# create the analysis
wipeAnalysis()                                 # clear previously-define analysis parameters
constraints('Plain')                # how it handles boundary conditions
numberer('Plain')    # renumber dof's to minimize band-width (optimization), if you want to
system('BandGeneral') # how to store and solve the system of equations in the analysis
algorithm('Linear')          # use Linear algorithm for linear analysis
integrator('Newmark', 0.5, 0.25)    # determine the next time step for an analysis
analysis('Transient')   # define type of analysis: time-dependent
analyze(3995, 0.01)          # apply 3995 0.01-sec time steps in analysis

u2 = nodeDisp(2, 2)
print("u2 = ", u2)


if abs(u2+0.07441860465116277579) < 1e-12:
    print("Passed!")
else:
    print("Failed!")

wipe()
```

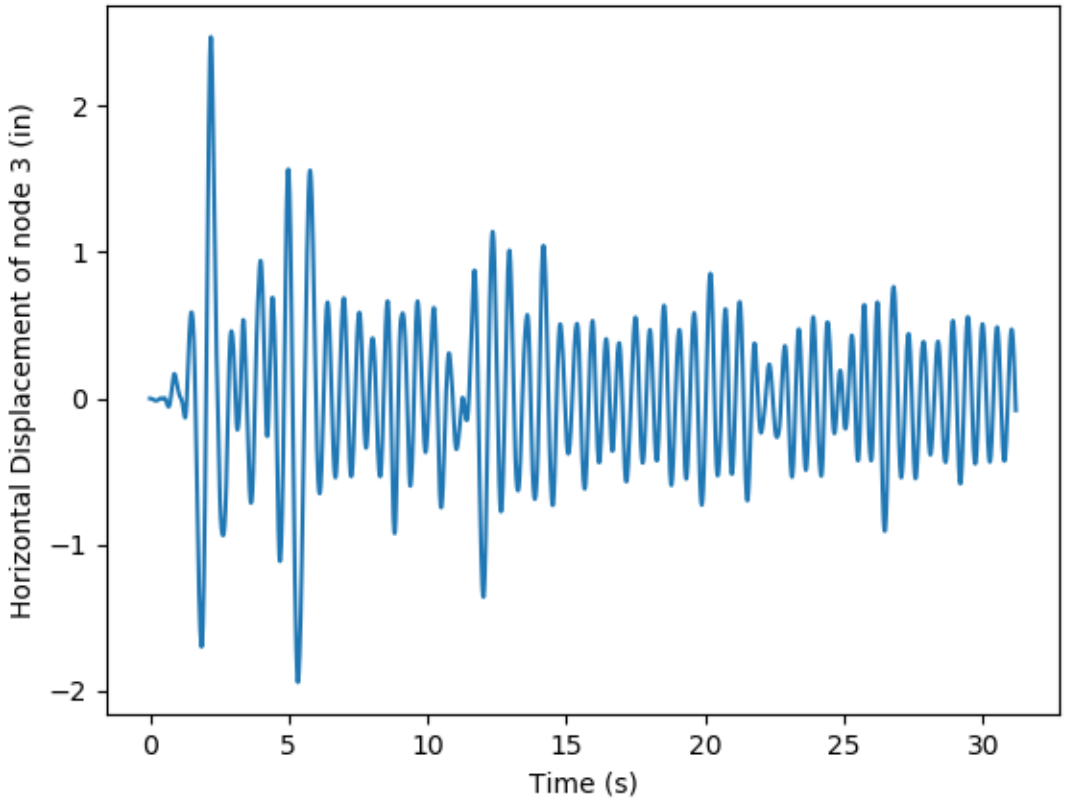
(continues on next page)

(continued from previous page)

```

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




```
print("==========================")
print("Start RCFrameEarthquake Example")

# Units: kips, in, sec
#
# Written: Minjie
```

(continues on next page)

(continued from previous page)

```
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
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 # Permform 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
```

(continues on next page)

(continued from previous page)

```
test('NormDispIncr', 1.0e-12,  10 )

# Create the solution algorithm, a Newton-Raphson algorithm
algorithm('Newton')

# Create the DOF numberer, the reverse Cuthill-McKee algorithm
numberer('RCM')

# Create the integration scheme, the Newmark with alpha =0.5 and beta =.25
integrator('Newmark',  0.5,  0.25 )

# Create the analysis object
analysis('Transient')

# Perform an eigenvalue analysis
numEigen = 2
eigenValues = eigen(numEigen)
print("eigen values at start of transient:",eigenValues)

# set some variables
tFinal = nPts*dt
tCurrent = getTime()
ok = 0

time = [tCurrent]
u3 = [0.0]

# Perform the transient analysis
while ok == 0 and tCurrent < tFinal:

    ok = analyze(1, .01)

    # if the analysis fails try initial tangent iteration
    if ok != 0:
        print("regular newton failed .. lets try an initail stiffness for this step")
        test('NormDispIncr', 1.0e-12,  100, 0)
        algorithm('ModifiedNewton', '-initial')
        ok =analyze( 1, .01)
        if ok == 0:
            print("that worked .. back to regular newton")
        test('NormDispIncr', 1.0e-12,  10 )
        algorithm('Newton')

    tCurrent = getTime()

    time.append(tCurrent)
    u3.append(nodeDisp(3,1))



# Perform an eigenvalue analysis
eigenValues = eigen(numEigen)
print("eigen values at end of transient:",eigenValues)

results = open('results.out','a+')

if ok == 0:
```

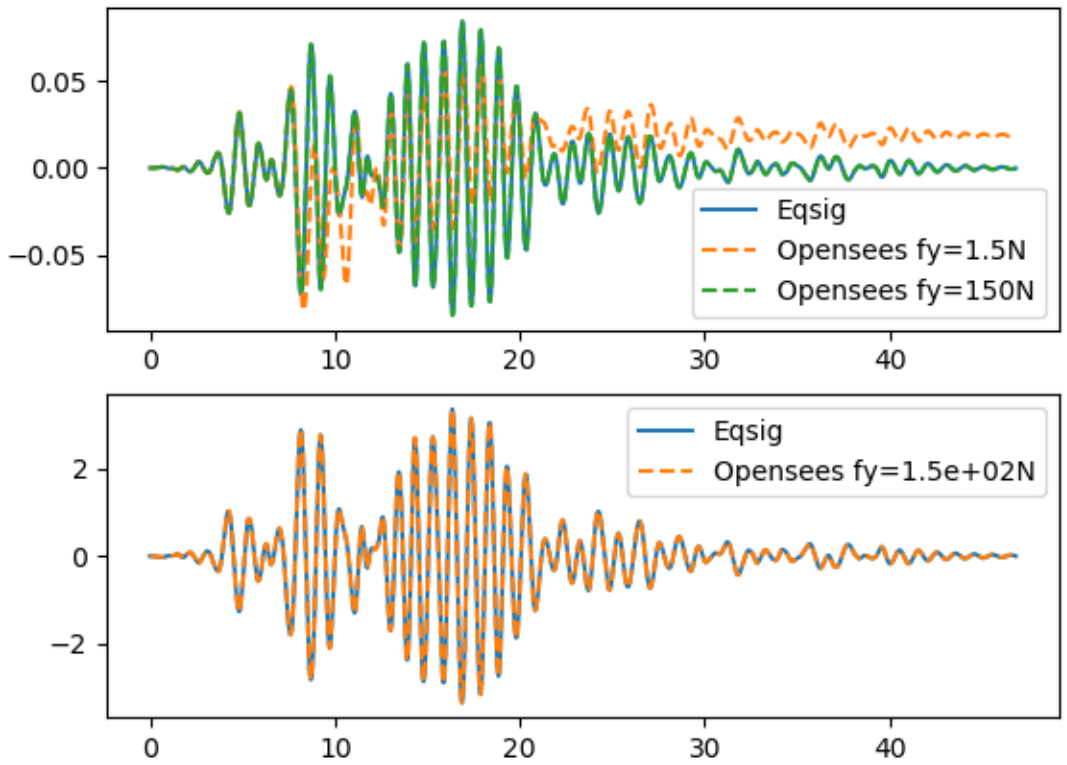
(continues on next page)

(continued from previous page)

```
    results.write('PASSED : RCFrameEarthquake.py\n');
    print("Passed!")
else:
    results.write('FAILED : RCFrameEarthquake.py\n');
    print("Failed!")

results.close()

plt.plot(time, u3)
plt.ylabel('Horizontal Displacement of node 3 (in)')
plt.xlabel('Time (s)')

plt.show()



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




```python
import eqsig
from eqsig import duhamels
import matplotlib.pyplot as plt
import numpy as np

import openseespy.opensees as op
import opensees_constants as opc  #opensees_constants.py should be close to main file␣
↪or use sys.path...   to its directory


def get_inelastic_response(mass, k_spring, f_yield, motion, dt, xi=0.05, r_post=0.0):
    """
    Run seismic analysis of a nonlinear SDOF

    :param mass: SDOF mass
    :param k_spring: spring stiffness
    :param f_yield: yield strength
    :param motion: list, acceleration values
    :param dt: float, time step of acceleration values
    :param xi: damping ratio
    :param r_post: post-yield stiffness
    :return:
    """

    op.wipe()
```

(continues on next page)

(continued from previous page)

```
    op.model('basic', '-ndm', 2, '-ndf', 3)  # 2 dimensions, 3 dof per node

    # Establish nodes
    bot_node = 1
    top_node = 2
    op.node(bot_node, 0., 0.)
    op.node(top_node, 0., 0.)

    # Fix bottom node
    op.fix(top_node, opc.FREE, opc.FIXED, opc.FIXED)
    op.fix(bot_node, opc.FIXED, opc.FIXED, opc.FIXED)
    # Set out-of-plane DOFs to be slaved
    op.equalDOF(1, 2, *[2, 3])

    # nodal mass (weight / g):
    op.mass(top_node, mass, 0., 0.)

    # Define material
    bilinear_mat_tag = 1
    mat_type = "Steel01"
    mat_props = [f_yield, k_spring, r_post]
    op.uniaxialMaterial(mat_type, bilinear_mat_tag, *mat_props)

    # Assign zero length element
    beam_tag = 1
    op.element('zeroLength', beam_tag, bot_node, top_node, "-mat", bilinear_mat_tag,
↪"-dir", 1, '-doRayleigh', 1)

    # Define the dynamic analysis
    load_tag_dynamic = 1
    pattern_tag_dynamic = 1

    values = list(-1 * motion)  # should be negative
    op.timeSeries('Path', load_tag_dynamic, '-dt', dt, '-values', *values)
    op.pattern('UniformExcitation', pattern_tag_dynamic, opc.X, '-accel', load_tag_
↪dynamic)

    # set damping based on first eigen mode
    angular_freq = op.eigen('-fullGenLapack', 1) ** 0.5
    alpha_m = 0.0
    beta_k = 2 * xi / angular_freq
    beta_k_comm = 0.0
    beta_k_init = 0.0

    op.rayleigh(alpha_m, beta_k, beta_k_init, beta_k_comm)

    # Run the dynamic analysis

    op.wipeAnalysis()

    op.algorithm('Newton')
    op.system('SparseGeneral')
    op.numberer('RCM')
    op.constraints('Transformation')
    op.integrator('Newmark', 0.5, 0.25)
    op.analysis('Transient')

```

(continues on next page)

(continued from previous page)

```
    tol = 1.0e-10
    iterations = 10
    op.test('EnergyIncr', tol, iterations, 0, 2)
    analysis_time = (len(values) - 1) * dt
    analysis_dt = 0.001
    outputs = {
        "time": [],
        "rel_disp": [],
        "rel_accel": [],
        "rel_vel": [],
        "force": []
    }

    while op.getTime() < analysis_time:
        curr_time = op.getTime()
        op.analyze(1, analysis_dt)
        outputs["time"].append(curr_time)
        outputs["rel_disp"].append(op.nodeDisp(top_node, 1))
        outputs["rel_vel"].append(op.nodeVel(top_node, 1))
        outputs["rel_accel"].append(op.nodeAccel(top_node, 1))
        op.reactions()
        outputs["force"].append(-op.nodeReaction(bot_node, 1))  # Negative since diff␣
↪node
    op.wipe()
    for item in outputs:
        outputs[item] = np.array(outputs[item])

    return outputs


def show_single_comparison():
    """
    Create a plot of an elastic analysis, nonlinear analysis and closed form elastic

    :return:
    """

    record_filename = 'test_motion_dt0p01.txt'
    motion_step = 0.01
    rec = np.loadtxt(record_filename)
    acc_signal = eqsig.AccSignal(rec, motion_step)
    period = 1.0
    xi = 0.05
    mass = 1.0
    f_yield = 1.5  # Reduce this to make it nonlinear
    r_post = 0.0

    periods = np.array([period])
    resp_u, resp_v, resp_a = duhamels.response_series(motion=rec, dt=motion_step,␣
↪periods=periods, xi=xi)

    k_spring = 4 * np.pi ** 2 * mass / period ** 2
    outputs = get_inelastic_response(mass, k_spring, f_yield, rec, motion_step, xi=xi,
↪ r_post=r_post)
    outputs_elastic = get_inelastic_response(mass, k_spring, f_yield * 100, rec,␣
↪motion_step, xi=xi, r_post=r_post)
    ux_opensees = outputs["rel_disp"]
```

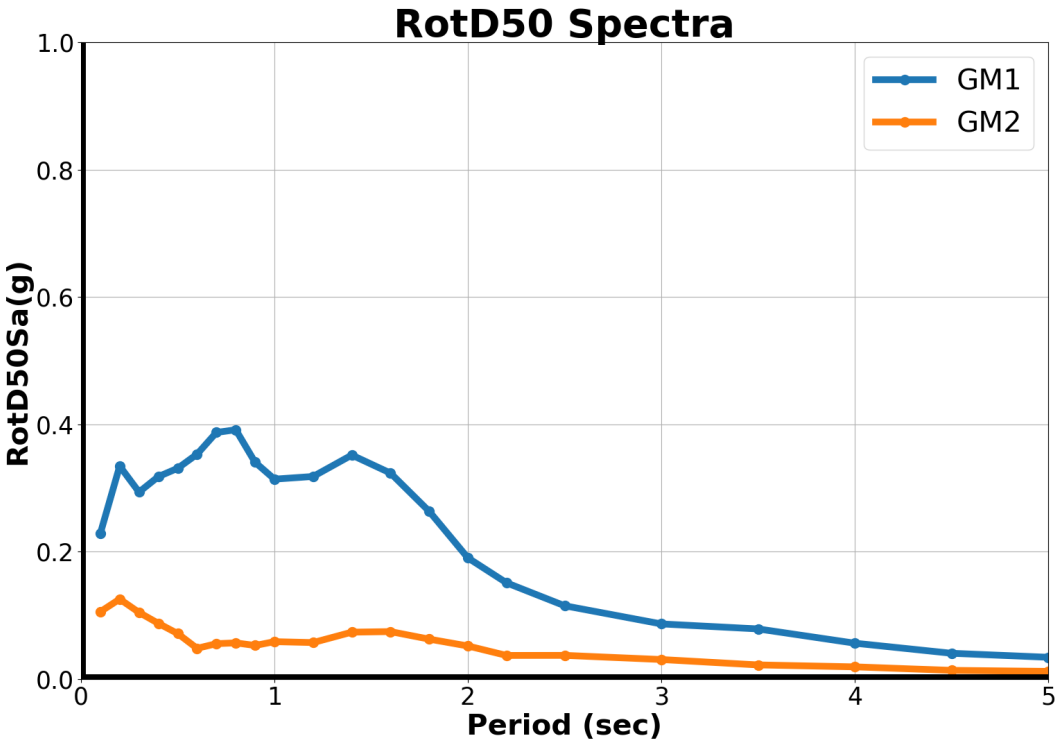
(continues on next page)

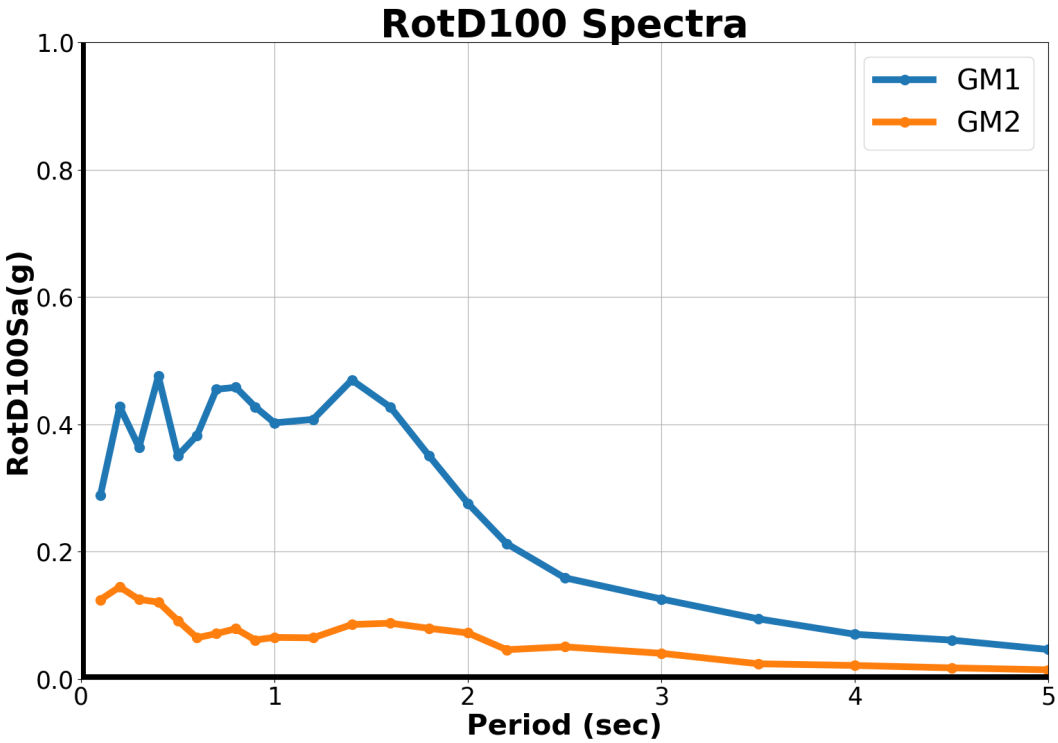
(continued from previous page)

```
    ux_opensees_elastic = outputs_elastic["rel_disp"]

    bf, sps = plt.subplots(nrows=2)
    sps[0].plot(acc_signal.time, resp_u[0], label="Eqsig")
    sps[0].plot(outputs["time"], ux_opensees, label="Opensees fy=%.3gN" % f_yield, ls="--")
    sps[0].plot(outputs["time"], ux_opensees_elastic, label="Opensees fy=%.3gN" % (f_yield * 100), ls="--")
    sps[1].plot(acc_signal.time, resp_a[0], label="Eqsig")  # Elastic solution
    time = acc_signal.time
    acc_opensees_elastic = np.interp(time, outputs_elastic["time"], outputs_elastic["rel_accel"]) - rec
    print("diff", sum(acc_opensees_elastic - resp_a[0]))
    sps[1].plot(time, acc_opensees_elastic, label="Opensees fy=%.2gN" % (f_yield * 100), ls="--")
    sps[0].legend()
    sps[1].legend()
    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


RotD50 Spectra
GM1
GM2
RotD50Sa(g)
Period (sec)
0.0
0.2
0.4
0.6
0.8
1.0
0
1
2
3
4
5


RotD100 Spectra
GM1
GM2
RotD100Sa(g)
Period (sec)
0.0
0.2
0.4
0.6
0.8
1.0
0
1
2
3
4
5

```
"""
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␣
↪Multinition
        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-define␣
↪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_Disp = 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_Disp[theta,0] = np.max(np.matmul(DISP_X_Y,Rot_Matrix)[:,0])

        # Storing Spectra
        Rot_Acc = np.dot(Rot_Disp,(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\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)

```
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 22 17:29:26 2019

@author: pchi893
"""
# Converted to openseespy by: Pavan Chigullapally
#                         University of Auckland
#                         Email: pchi893@aucklanduni.ac.nz
# Example 1b. portal frame in 2D
#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
# all units are in kip, inch, second
#To run EQ ground-motion analysis (BM68elc.acc needs to be downloaded into the same directory).
#the detailed problem description can be found here: http://opensees.berkeley.edu/wiki/index.php/Examples_Manual  (example: 1b)
# ----------------------------------------------------------------------------------------------
# elasticBeamColumn ELEMENT
#        OpenSees (Tcl) code by:        Silvia Mazzoni & Frank McKenna, 2006

#
#    ^Y
#    |
#    3__________(3)_________4       __
#    |                                   |           |
#    |                                   |           |
#    |                                   |           |
```

(continues on next page)

(continued from previous page)

```
#  (1)                                    (2)        LCol
#    |                                      |          |
#    |                                      |          |
#    |                                      |          |
#  =1=                                   =2=       _|_   -------->X
#    |----------LBeam------------|
#

# SET UP ---------------------------------------------------------------------------

import openseespy.opensees as op
#import the os module
import os
op.wipe()

##########################################################################################
##########################################################################################

##########################################################################################
##########################################################################################
op.model('basic', '-ndm', 2, '-ndf', 3)

#to create a directory at specified path with name "Data"
os.chdir('C:\\Opensees Python\\OpenseesPy examples')

#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
dir = "C:\\Opensees Python\\OpenseesPy examples\\Data-1b"
if not os.path.exists(dir):
    os.makedirs(dir)

#this will create just 'Data' folder
#os.mkdir("Data-1b")

#detect the current working directory
#path1 = os.getcwd()
#print(path1)

h = 432.0
w  =  504.0

op.node(1, 0.0, 0.0)
op.node(2, h, 0.0)
op.node(3, 0.0, w)
op.node(4, h, w)

op.fix(1, 1,1,1)
op.fix(2, 1,1,1)
op.fix(3, 0,0,0)
op.fix(4, 0,0,0)

op.mass(3, 5.18, 0.0, 0.0)
op.mass(4, 5.18, 0.0, 0.0)

op.geomTransf('Linear', 1)
A = 3600000000.0
E = 4227.0
```

(continues on next page)

(continued from previous page)

```
Iz = 1080000.0

A1 = 5760000000.0
Iz1 = 4423680.0
op.element('elasticBeamColumn', 1, 1, 3, A, E, Iz, 1)
op.element('elasticBeamColumn', 2, 2, 4, A, E, Iz, 1)
op.element('elasticBeamColumn', 3, 3, 4, A1, E, Iz1, 1)

op.recorder('Node', '-file', 'Data-1b/DFree.out','-time', '-node', 3,4, '-dof', 1,2,3, 'disp')
op.recorder('Node', '-file', 'Data-1b/DBase.out','-time', '-node', 1,2, '-dof', 1,2,3, 'disp')
op.recorder('Node', '-file', 'Data-1b/RBase.out','-time', '-node', 1,2, '-dof', 1,2,3, 'reaction')
#op.recorder('Drift', '-file', 'Data-1b/Drift.out','-time', '-node', 1, '-dof', 1,2,3, 'disp')
op.recorder('Element', '-file', 'Data-1b/FCol.out','-time', '-ele', 1,2, 'globalForce')
op.recorder('Element', '-file', 'Data-1b/DCol.out','-time', '-ele', 3, 'deformations')

#defining gravity loads
op.timeSeries('Linear', 1)
op.pattern('Plain', 1, 1)
op.eleLoad('-ele', 3, '-type', '-beamUniform', -7.94)

op.constraints('Plain')
op.numberer('Plain')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-8, 6)
op.algorithm('Newton')
op.integrator('LoadControl', 0.1)
op.analysis('Static')
op.analyze(10)

op.loadConst('-time', 0.0)

#applying Dynamic Ground motion analysis
op.timeSeries('Path', 2, '-dt', 0.01, '-filePath', 'BM68elc.acc', '-factor', 1.0)
op.pattern('UniformExcitation', 2, 1, '-accel', 2) #how to give accelseriesTag?

eigen = op. eigen('-fullGenLapack', 1)
import math
power = math.pow(eigen, 0.5)
betaKcomm = 2 * (0.02/power)

op.rayleigh(0.0, 0.0, 0.0, betaKcomm)

op.wipeAnalysis()
op.constraints('Plain')
op.numberer('Plain')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-8, 10)
op.algorithm('Newton')
op.integrator('Newmark', 0.5, 0.25)
op.analysis('Transient')
op.analyze(1000, 0.02)

```

(continues on next page)

(continued from previous page)

```
u3 = op.nodeDisp(3, 1)
print("u2 = ", u3)

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`.

```
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 22 15:12:06 2019

@author: pchi893
"""
# Converted to openseespy by: Pavan Chigullapally
#                         University of Auckland
#                         Email: pchi893@aucklanduni.ac.nz

# EQ ground motion with gravity- uniform excitation of structure
# all units are in kip, inch, second
##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
#the OpenSees input and also to run various tests and algorithms at once to increase the chances of convergence
# Example 2a. 2D cantilever column, dynamic eq ground motion
#To run EQ ground-motion analysis (BM68elc.acc needs to be downloaded into the same directory)
#the detailed problem description can be found here: http://opensees.berkeley.edu/wiki/index.php/Examples_Manual  (example:2a)
# ------------------------------------------------------------------------------------------
#        OpenSees (Tcl) code by:        Silvia Mazzoni & Frank McKenna, 2006


#
#    ^Y
#    |
#    2       __
#    |           |
#    |           |
```

(continues on next page)

(continued from previous page)

```
#    |           |
#  (1)        LCol
#    |           |
#    |           |
#    |           |
#  =1=       _|_   -------->X
#

# SET UP ---------------------------------------------------------------------------
import openseespy.opensees as op
#import the os module
import os
import math
op.wipe()

##########################################################################################
##########################################################################################

##########################################################################################
##########################################################################################
op.model('basic', '-ndm', 2, '-ndf', 3)

#to create a directory at specified path with name "Data"
os.chdir('C:\\Opensees Python\\OpenseesPy examples')

#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
dir = "C:\\Opensees Python\\OpenseesPy examples\\Data-2a"
if not os.path.exists(dir):
    os.makedirs(dir)

#this will create just 'Data' folder
#os.mkdir("Data")

#detect the current working directory
#path1 = os.getcwd()
#print(path1)

LCol = 432.0 # column length
Weight = 2000.0 # superstructure weight

# define section geometry
HCol = 60.0 # Column Depth
BCol = 60.0 # Column Width

PCol =Weight  # nodal dead-load weight per column
g = 386.4
Mass =  PCol/g

ACol = HCol*BCol*1000  # cross-sectional area, make stiff
IzCol = (BCol*math.pow(HCol,3))/12 # Column moment of inertia

op.node(1, 0.0, 0.0)
op.node(2, 0.0, LCol)

op.fix(1, 1, 1, 1)

```

(continues on next page)

(continued from previous page)

```
op.mass(2, Mass, 1e-9, 0.0)

ColTransfTag = 1
op.geomTransf('Linear', ColTransfTag)
#A = 3600000000.0
#E = 4227.0
#Iz = 1080000.0

fc = -4.0 # CONCRETE Compressive Strength (+Tension, -Compression)
Ec = 57*math.sqrt(-fc*1000) # Concrete Elastic Modulus (the term in sqr root needs to be in psi

op.element('elasticBeamColumn', 1, 1, 2, ACol, Ec, IzCol, ColTransfTag)

op.recorder('Node', '-file', 'Data-2a/DFree.out','-time', '-node', 2, '-dof', 1,2,3, 'disp')
op.recorder('Node', '-file', 'Data-2a/DBase.out','-time', '-node', 1, '-dof', 1,2,3, 'disp')
op.recorder('Node', '-file', 'Data-2a/RBase.out','-time', '-node', 1, '-dof', 1,2,3, 'reaction')
#op.recorder('Drift', '-file', 'Data-2a/Drift.out','-time', '-node', 1, '-dof', 1,2,3, 'disp')
op.recorder('Element', '-file', 'Data-2a/FCol.out','-time', '-ele', 1, 'globalForce')
op.recorder('Element', '-file', 'Data-2a/DCol.out','-time', '-ele', 1, 'deformations')

#defining gravity loads
op.timeSeries('Linear', 1)
op.pattern('Plain', 1, 1)
op.load(2, 0.0, -PCol, 0.0)

Tol = 1e-8 # convergence tolerance for test
NstepGravity = 10
DGravity = 1/NstepGravity
op.integrator('LoadControl', DGravity) # determine the next time step for an analysis
op.numberer('Plain') # renumber dof's to minimize band-width (optimization), if you want to
op.system('BandGeneral') # how to store and solve the system of equations in the analysis
op.constraints('Plain') # how it handles boundary conditions
op.test('NormDispIncr', Tol, 6) # determine if convergence has been achieved at the end of an iteration step
op.algorithm('Newton') # use Newton's solution algorithm: updates tangent stiffness at every iteration
op.analysis('Static') # define type of analysis static or transient
op.analyze(NstepGravity) # apply gravity

op.loadConst('-time', 0.0) #maintain constant gravity loads and reset time to zero

#applying Dynamic Ground motion analysis
GMdirection = 1
GMfile = 'BM68elc.acc'
GMfact = 1.0



Lambda = op.eigen('-fullGenLapack', 1) # eigenvalue mode 1
import math
```

(continues on next page)

(continued from previous page)

```
Omega = math.pow(Lambda, 0.5)
betaKcomm = 2 * (0.02/Omega)

xDamp = 0.02                                    # 2% damping ratio
alphaM = 0.0                                    # M-prop. damping; D = alphaM*M
betaKcurr = 0.0                 # K-proportional damping;       +beatKcurr*KCurrent
betaKinit = 0.0 # initial-stiffness proportional damping      +beatKinit*Kini

op.rayleigh(alphaM,betaKcurr, betaKinit, betaKcomm) # RAYLEIGH damping

# Uniform EXCITATION: acceleration input
IDloadTag = 400                             # load tag
dt = 0.01                           # time step for input ground motion
GMfatt = 1.0                           # data in input file is in g Unifts --␣
↪ACCELERATION TH
maxNumIter = 10
op.timeSeries('Path', 2, '-dt', dt, '-filePath', GMfile, '-factor', GMfact)
op.pattern('UniformExcitation', IDloadTag, GMdirection, '-accel', 2)

op.wipeAnalysis()
op.constraints('Transformation')
op.numberer('Plain')
op.system('BandGeneral')
op.test('EnergyIncr', Tol, maxNumIter)
op.algorithm('ModifiedNewton')

NewmarkGamma = 0.5
NewmarkBeta = 0.25
op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
op.analysis('Transient')

DtAnalysis = 0.01
TmaxAnalysis = 10.0

Nsteps =  int(TmaxAnalysis/ DtAnalysis)

ok = op.analyze(Nsteps, DtAnalysis)
tCurrent = op.getTime()

# for gravity analysis, load control is fine, 0.1 is the load factor increment (http:/
↪/opensees.berkeley.edu/wiki/index.php/Load_Control)

test = {1:'NormDispIncr', 2: 'RelativeEnergyIncr', 4: 'RelativeNormUnbalance',5:
↪'RelativeNormDispIncr', 6: 'NormUnbalance'}
algorithm = {1:'KrylovNewton', 2: 'SecantNewton' , 4: 'RaphsonNewton',5:
↪'PeriodicNewton', 6: 'BFGS', 7: 'Broyden', 8: 'NewtonLineSearch'}

for i in test:
    for j in algorithm:

        if ok != 0:
            if j < 4:
                op.algorithm(algorithm[j], '-initial')

            else:
                op.algorithm(algorithm[j])

```

(continues on next page)

(continued from previous page)

```
            op.test(test[i], Tol, 1000)
            ok = op.analyze(Nsteps, DtAnalysis)
            print(test[i], algorithm[j], ok)
            if ok == 0:
                break
        else:
            continue

u2 = op.nodeDisp(2, 1)
print("u2 = ", u2)

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 indepenently 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`.

```
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 22 15:12:06 2019

@author: pchi893
"""
# Converted to openseespy by: Pavan Chigullapally
#                         University of Auckland
#                         Email: pchi893@aucklanduni.ac.nz
# Example 2b. 2D cantilever column, dynamic eq ground motion
# EQ ground motion with gravity- uniform excitation of structure
#he 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,
#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
```

(continues on next page)

(continued from previous page)

```
#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.
#Note:In this example, both the axial behavior (typically elastic) and the flexural behavior (moment curvature) are defined indepenently and are then "aggregated" into a section.
#This is a characteristic of the uniaxial section: there is no coupling of behaviors.

#To run EQ ground-motion analysis (BM68elc.acc needs to be downloaded into the same directory)
#the problem description can be found here: http://opensees.berkeley.edu/wiki/index.php/Examples_Manual(example:2b)
# ------------------------------------------------------------------------------------------
#         OpenSees (Tcl) code by:        Silvia Mazzoni & Frank McKenna, 2006
#
#    ^Y
#    |
#    2       __
#    |           |
#    |           |
#    |           |
#  (1)        LCol
#    |           |
#    |           |
#    |           |
#  =1=      _|_   --------->X
#

# SET UP ---------------------------------------------------------------------------
import openseespy.opensees as op
#import the os module
import os
import math
op.wipe()
#############################################################################################

#to create a directory at specified path with name "Data"
os.chdir('C:\\Opensees Python\\OpenseesPy examples')

#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
dir = "C:\\Opensees Python\\OpenseesPy examples\\Data-2b"
if not os.path.exists(dir):
    os.makedirs(dir)
#this will create just 'Data' folder
#os.mkdir("Data")
#detect the current working directory
#path1 = os.getcwd()
#print(path1)
#############################################################################################

#############################################################################################
op.model('basic', '-ndm', 2, '-ndf', 3)
LCol = 432.0 # column length
```

(continues on next page)

(continued from previous page)

```
Weight = 2000.0 # superstructure weight

# define section geometry
HCol = 60.0 # Column Depth
BCol = 60.0 # Column Width

PCol =Weight  # nodal dead-load weight per column
g = 386.4
Mass =  PCol/g

ACol = HCol*BCol*1000  # cross-sectional area, make stiff
IzCol = (BCol*math.pow(HCol,3))/12 # Column moment of inertia

op.node(1, 0.0, 0.0)
op.node(2, 0.0, LCol)

op.fix(1, 1, 1, 1)

op.mass(2, Mass, 1e-9, 0.0)

#Define Elements and Sections
ColMatTagFlex  = 2
ColMatTagAxial = 3
ColSecTag = 1
BeamSecTag = 2

fc = -4.0 # CONCRETE Compressive Strength (+Tension, -Compression)
Ec = 57*math.sqrt(-fc*1000) # Concrete Elastic Modulus (the term in sqr root needs to␣
↪be in psi

#Column Section
EICol = Ec*IzCol # EI, for moment-curvature relationship
EACol = Ec*ACol # EA, for axial-force-strain relationship
MyCol = 130000.0 #yield Moment calculated
PhiYCol = 0.65e-4        # yield curvature
EIColCrack = MyCol/PhiYCol        # cracked section inertia
b = 0.01 # strain-hardening ratio (ratio between post-yield tangent and initial␣
↪elastic tangent)

op.uniaxialMaterial('Steel01', ColMatTagFlex, MyCol, EIColCrack, b) #steel moment␣
↪curvature isused for Mz of the section only, # bilinear behavior for flexure
op.uniaxialMaterial('Elastic', ColMatTagAxial, EACol) # this is not used as a␣
↪material, this is an axial-force-strain response
op.section('Aggregator', ColSecTag, ColMatTagAxial, 'P', ColMatTagFlex, 'Mz')  #␣
↪combine axial and flexural behavior into one section (no P-M interaction here)

ColTransfTag = 1
op.geomTransf('Linear', ColTransfTag)
numIntgrPts = 5
eleTag = 1
op.element('nonlinearBeamColumn', eleTag, 1, 2, numIntgrPts, ColSecTag, ColTransfTag)

op.recorder('Node', '-file', 'Data-2b/DFree.out','-time', '-node', 2, '-dof', 1,2,3,
↪'disp')
op.recorder('Node', '-file', 'Data-2b/DBase.out','-time', '-node', 1, '-dof', 1,2,3,
↪'disp')
op.recorder('Node', '-file', 'Data-2b/RBase.out','-time', '-node', 1, '-dof', 1,2,3,
↪'reaction')
```

(continues on next page)

(continued from previous page)

```
#op.recorder('Drift', '-file', 'Data-2b/Drift.out','-time', '-node', 1, '-dof', 1,2,3, 'disp')
op.recorder('Element', '-file', 'Data-2b/FCol.out','-time', '-ele', 1, 'globalForce')
op.recorder('Element', '-file', 'Data-2b/ForceColSec1.out','-time', '-ele', 1, 'section', 1, 'force')
#op.recorder('Element', '-file', 'Data-2b/DCol.out','-time', '-ele', 1, 'deformations')

#defining gravity loads
op.timeSeries('Linear', 1)
op.pattern('Plain', 1, 1)
op.load(2, 0.0, -PCol, 0.0)

Tol = 1e-8 # convergence tolerance for test
NstepGravity = 10
DGravity = 1/NstepGravity
op.integrator('LoadControl', DGravity) # determine the next time step for an analysis
op.numberer('Plain') # renumber dof's to minimize band-width (optimization), if you want to
op.system('BandGeneral') # how to store and solve the system of equations in the analysis
op.constraints('Plain') # how it handles boundary conditions
op.test('NormDispIncr', Tol, 6) # determine if convergence has been achieved at the end of an iteration step
op.algorithm('Newton') # use Newton's solution algorithm: updates tangent stiffness at every iteration
op.analysis('Static') # define type of analysis static or transient
op.analyze(NstepGravity) # apply gravity

op.loadConst('-time', 0.0) #maintain constant gravity loads and reset time to zero

#applying Dynamic Ground motion analysis
GMdirection = 1
GMfile = 'BM68elc.acc'
GMfact = 1.0



Lambda = op.eigen('-fullGenLapack', 1) # eigenvalue mode 1
import math
Omega = math.pow(Lambda, 0.5)
betaKcomm = 2 * (0.02/Omega)

xDamp = 0.02                                # 2% damping ratio
alphaM = 0.0                                # M-prop. damping; D = alphaM*M
betaKcurr = 0.0                 # K-proportional damping;      +beatKcurr*KCurrent
betaKinit = 0.0 # initial-stiffness proportional damping      +beatKinit*Kini

op.rayleigh(alphaM,betaKcurr, betaKinit, betaKcomm) # RAYLEIGH damping

# Uniform EXCITATION: acceleration input
IDloadTag = 400                             # load tag
dt = 0.01                           # time step for input ground motion
GMfatt = 1.0                            # data in input file is in g Unifts -- ACCELERATION TH
maxNumIter = 10
op.timeSeries('Path', 2, '-dt', dt, '-filePath', GMfile, '-factor', GMfact)
```

(continues on next page)

(continued from previous page)

```
op.pattern('UniformExcitation', IDloadTag, GMdirection, '-accel', 2)

op.wipeAnalysis()
op.constraints('Transformation')
op.numberer('Plain')
op.system('BandGeneral')
op.test('EnergyIncr', Tol, maxNumIter)
op.algorithm('ModifiedNewton')

NewmarkGamma = 0.5
NewmarkBeta = 0.25
op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
op.analysis('Transient')

DtAnalysis = 0.01
TmaxAnalysis = 10.0

Nsteps =  int(TmaxAnalysis/ DtAnalysis)

ok = op.analyze(Nsteps, DtAnalysis)

tCurrent = op.getTime()

# for gravity analysis, load control is fine, 0.1 is the load factor increment (http:/
↪/opensees.berkeley.edu/wiki/index.php/Load_Control)

test = {1:'NormDispIncr', 2: 'RelativeEnergyIncr', 4: 'RelativeNormUnbalance',5:
↪'RelativeNormDispIncr', 6: 'NormUnbalance'}
algorithm = {1:'KrylovNewton', 2: 'SecantNewton' , 4: 'RaphsonNewton',5:
↪'PeriodicNewton', 6: 'BFGS', 7: 'Broyden', 8: 'NewtonLineSearch'}

for i in test:
    for j in algorithm:

        if ok != 0:
            if j < 4:
                op.algorithm(algorithm[j], '-initial')

            else:
                op.algorithm(algorithm[j])

            op.test(test[i], Tol, 1000)
            ok = op.analyze(Nsteps, DtAnalysis)
            print(test[i], algorithm[j], ok)
            if ok == 0:
                break
        else:
            continue

u2 = op.nodeDisp(2, 1)
print("u2 = ", u2)

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`.

```
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 22 15:12:06 2019

@author: pchi893
"""
# Converted to openseespy by: Pavan Chigullapally
#                         University of Auckland
#                         Email: pchi893@aucklanduni.ac.nz
# Example 2c. 2D cantilever column, dynamic eq ground motion
# EQ ground motion with gravity- uniform excitation of structure
#In this example, the Uniaxial Section of Example 2b is replaced by a fiber section. Inelastic uniaxial materials are used in this example,
#which are assigned to each fiber, or patch of fibers, in the section.
#In this example the axial and flexural behavior are coupled, a characteristic of the fiber section.
#The nonlinear/inelastic behavior of a fiber section is defined by the stress-strain response of the uniaxial materials used to define it.

#To run EQ ground-motion analysis (BM68elc.acc needs to be downloaded into the same directory)
#the problem description can be found here: http://opensees.berkeley.edu/wiki/index.php/Examples_Manual(example: 2c)
# ------------------------------------------------------------------------------------------
#         OpenSees (Tcl) code by:        Silvia Mazzoni & Frank McKenna, 2006

#
#    ^Y
#    |
#    2       __
#    |          |
#    |          |
#    |          |
#  (1)       LCol
#    |          |
```

(continues on next page)

(continued from previous page)

```
#    |           |
#    |           |
#  =1=       _|_   -------->X
#

# SET UP ---------------------------------------------------------------------
import openseespy.opensees as op
#import the os module
import os
import math
op.wipe()
##############################################################################
##############################################################################
#to create a directory at specified path with name "Data"
os.chdir('C:\\Opensees Python\\OpenseesPy examples')

#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
dir = "C:\\Opensees Python\\OpenseesPy examples\\Data-2c"
if not os.path.exists(dir):
    os.makedirs(dir)
#this will create just 'Data' folder
#os.mkdir("Data")
#detect the current working directory
#path1 = os.getcwd()
#print(path1)
##############################################################################
##############################################################################

##############################################################################
##############################################################################
op.model('basic', '-ndm', 2, '-ndf', 3)
LCol = 432.0 # column length
Weight = 2000.0 # superstructure weight

# define section geometry
HCol = 60.0 # Column Depth
BCol = 60.0 # Column Width

PCol =Weight  # nodal dead-load weight per column
g = 386.4
Mass =  PCol/g

ACol = HCol*BCol*1000  # cross-sectional area, make stiff
IzCol = (BCol*math.pow(HCol,3))/12 # Column moment of inertia

op.node(1, 0.0, 0.0)
op.node(2, 0.0, LCol)

op.fix(1, 1, 1, 1)

op.mass(2, Mass, 1e-9, 0.0)

ColSecTag = 1                           # assign a tag number to the column section
coverCol = 5.0   # Column cover to reinforcing steel NA.
numBarsCol = 16  # number of longitudinal-reinforcement bars in column. (symmetric top & bot)
```

(continues on next page)

(continued from previous page)

```
barAreaCol = 2.25  # area of longitudinal-reinforcement bars

# MATERIAL parameters
IDconcU = 1                          # material ID tag -- unconfined cover concrete (here used for complete section)
IDreinf = 2                                  # material ID tag -- reinforcement

# nominal concrete compressive strength
fc = -4.0                                  # CONCRETE Compressive Strength (+Tension, -Compression)
Ec = 57*math.sqrt(-fc*1000) # Concrete Elastic Modulus (the term in sqr root needs to be in psi

# unconfined concrete
fc1U = fc                           # UNCONFINED concrete (todeschini parabolic model), maximum stress
eps1U = -0.003                           # strain at maximum strength of unconfined concrete
fc2U =  0.2*fc1U                  # ultimate stress
eps2U = -0.01                            # strain at ultimate stress
Lambda = 0.1                                  # ratio between unloading slope at $eps2 and initial slope $Ec

# tensile-strength properties
ftU = -0.14* fc1U                  # tensile strength +tension
Ets = ftU/0.002                         # tension softening stiffness

Fy = 66.8                           # STEEL yield stress
Es = 29000.0                                # modulus of steel
Bs = 0.01                                 # strain-hardening ratio
R0 = 18.0                                 # control the transition from elastic to plastic branches
cR1 = 0.925                                 # control the transition from elastic to plastic branches
cR2 = 0.15                                  # control the transition from elastic to plastic branches

op.uniaxialMaterial('Concrete02', IDconcU, fc1U, eps1U, fc2U, eps2U, Lambda, ftU, Ets) # build cover concrete (unconfined)
op.uniaxialMaterial('Steel02', IDreinf, Fy, Es, Bs, R0,cR1,cR2) # build reinforcement material
# FIBER SECTION properties -------------------------------------------------------------
# symmetric section
#                         y
#                         ^
#                         |
#              ---------------------      --   --
#              |   o     o     o    |     |    -- cover
#              |                        |     |
#              |                        |     |
#     z <--- |          +             |     H
#              |                        |     |
#              |                        |     |
#              |   o     o     o    |     |    -- cover
#              ---------------------      --   --
#              |-------- B --------|
```

(continues on next page)

(continued from previous page)

```
#
# RC section:

coverY = HCol/2.0        # The distance from the section z-axis to the edge of the cover concrete -- outer edge of cover concrete
coverZ = BCol/2.0        # The distance from the section y-axis to the edge of the cover concrete -- outer edge of cover concrete
coreY = coverY-coverCol
coreZ = coverZ-coverCol
nfY = 16  # number of fibers for concrete in y-direction
nfZ = 4                         # number of fibers for concrete in z-direction

op.section('Fiber', ColSecTag)
op.patch('quad', IDconcU, nfZ, nfY, -coverY,coverZ, -coverY,-coverZ, coverY,-coverZ, coverY,coverZ) # Define the concrete patch
op.layer('straight', IDreinf, numBarsCol, barAreaCol, -coreY,coreZ,-coreY,-coreZ)
op.layer('straight', IDreinf, numBarsCol, barAreaCol, coreY,coreZ, coreY,-coreZ)
ColTransfTag = 1
op.geomTransf('Linear', ColTransfTag)
numIntgrPts = 5
eleTag = 1

#import InelasticFiberSection

op.element('nonlinearBeamColumn', eleTag, 1, 2, numIntgrPts, ColSecTag, ColTransfTag)

op.recorder('Node', '-file', 'Data-2c/DFree.out','-time', '-node', 2, '-dof', 1,2,3, 'disp')
op.recorder('Node', '-file', 'Data-2c/DBase.out','-time', '-node', 1, '-dof', 1,2,3, 'disp')
op.recorder('Node', '-file', 'Data-2c/RBase.out','-time', '-node', 1, '-dof', 1,2,3, 'reaction')
#op.recorder('Drift', '-file', 'Data-2c/Drift.out','-time', '-node', 1, '-dof', 1,2,3, 'disp')
op.recorder('Element', '-file', 'Data-2c/FCol.out','-time', '-ele', 1, 'globalForce')
op.recorder('Element', '-file', 'Data-2c/ForceColSec1.out','-time', '-ele', 1, 'section', 1, 'force')
#op.recorder('Element', '-file', 'Data-2c/DCol.out','-time', '-ele', 1, 'deformations')

#defining gravity loads
op.timeSeries('Linear', 1)
op.pattern('Plain', 1, 1)
op.load(2, 0.0, -PCol, 0.0)

Tol = 1e-8 # convergence tolerance for test
NstepGravity = 10
DGravity = 1/NstepGravity
op.integrator('LoadControl', DGravity) # determine the next time step for an analysis
op.numberer('Plain') # renumber dof's to minimize band-width (optimization), if you want to
op.system('BandGeneral') # how to store and solve the system of equations in the analysis
op.constraints('Plain') # how it handles boundary conditions
op.test('NormDispIncr', Tol, 6) # determine if convergence has been achieved at the end of an iteration step
op.algorithm('Newton') # use Newton's solution algorithm: updates tangent stiffness at every iteration
```

(continues on next page)

(continued from previous page)

```
op.analysis('Static') # define type of analysis static or transient
op.analyze(NstepGravity) # apply gravity

op.loadConst('-time', 0.0) #maintain constant gravity loads and reset time to zero

#applying Dynamic Ground motion analysis
GMdirection = 1
GMfile = 'BM68elc.acc'
GMfact = 1.0



Lambda = op.eigen('-fullGenLapack', 1) # eigenvalue mode 1
import math
Omega = math.pow(Lambda, 0.5)
betaKcomm = 2 * (0.02/Omega)

xDamp = 0.02                                # 2% damping ratio
alphaM = 0.0                                # M-prop. damping; D = alphaM*M
betaKcurr = 0.0                 # K-proportional damping;       +beatKcurr*KCurrent
betaKinit = 0.0 # initial-stiffness proportional damping      +beatKinit*Kini

op.rayleigh(alphaM,betaKcurr, betaKinit, betaKcomm) # RAYLEIGH damping

# Uniform EXCITATION: acceleration input
IDloadTag = 400                         # load tag
dt = 0.01                           # time step for input ground motion
GMfatt = 1.0                            # data in input file is in g Unifts --␣
↪ACCELERATION TH
maxNumIter = 10
op.timeSeries('Path', 2, '-dt', dt, '-filePath', GMfile, '-factor', GMfact)
op.pattern('UniformExcitation', IDloadTag, GMdirection, '-accel', 2)

op.wipeAnalysis()
op.constraints('Transformation')
op.numberer('Plain')
op.system('BandGeneral')
op.test('EnergyIncr', Tol, maxNumIter)
op.algorithm('ModifiedNewton')

NewmarkGamma = 0.5
NewmarkBeta = 0.25
op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
op.analysis('Transient')

DtAnalysis = 0.01
TmaxAnalysis = 10.0

Nsteps =  int(TmaxAnalysis/ DtAnalysis)

ok = op.analyze(Nsteps, DtAnalysis)

tCurrent = op.getTime()

# for gravity analysis, load control is fine, 0.1 is the load factor increment (http:/
↪/opensees.berkeley.edu/wiki/index.php/Load_Control)

```

(continues on next page)

(continued from previous page)

```
test = {1:'NormDispIncr', 2: 'RelativeEnergyIncr', 4: 'RelativeNormUnbalance',5:
↪'RelativeNormDispIncr', 6: 'NormUnbalance'}
algorithm = {1:'KrylovNewton', 2: 'SecantNewton' , 4: 'RaphsonNewton',5:
↪'PeriodicNewton', 6: 'BFGS', 7: 'Broyden', 8: 'NewtonLineSearch'}

for i in test:
    for j in algorithm:

        if ok != 0:
            if j < 4:
                op.algorithm(algorithm[j], '-initial')

            else:
                op.algorithm(algorithm[j])

            op.test(test[i], Tol, 1000)
            ok = op.analyze(Nsteps, DtAnalysis)
            print(test[i], algorithm[j], ok)
            if ok == 0:
                break
        else:
            continue

u2 = op.nodeDisp(2, 1)
print("u2 = ", u2)

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`.

```
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 22 15:12:06 2019

@author: pchi893
"""
# Converted to openseespy by: Pavan Chigullapally
#                         University of Auckland
```

(continues on next page)

(continued from previous page)

```
#                         Email: pchi893@aucklanduni.ac.nz
# Example 3. 2D Cantilever -- EQ ground motion
#To run Uniaxial Inelastic Material, Fiber Section, Nonlinear Mode, Uniform␣
↪Earthquake Excitation:First import the InelasticFiberSection.py(upto gravity␣
↪loading is already in this script)
#and run the current script
#To run EQ ground-motion analysis (BM68elc.acc needs to be downloaded into the same␣
↪directory)
# Same acceleration input at all nodes restrained in specified direction (uniform␣
↪acceleration input at all support nodes)
#the detailed problem description can be found here: http://opensees.berkeley.edu/
↪wiki/index.php/Examples_Manual  (example: 3)
# ------------------------------------------------------------------------------------
↪--------------
#        OpenSees (Tcl) code by:        Silvia Mazzoni & Frank McKenna, 2006
##########################################################################################
↪##########################################################################################
import openseespy.opensees as op
#import the os module
#import os
import math
op.wipe()
##########################################################################################
↪##########################################################################################
import InelasticFiberSection
#applying Dynamic Ground motion analysis
Tol = 1e-8
GMdirection = 1
GMfile = 'BM68elc.acc'
GMfact = 1.0
Lambda = op.eigen('-fullGenLapack', 1) # eigenvalue mode 1
Omega = math.pow(Lambda, 0.5)
betaKcomm = 2 * (0.02/Omega)

xDamp = 0.02                                # 2% damping ratio
alphaM = 0.0                                # M-prop. damping; D = alphaM*M
betaKcurr = 0.0                 # K-proportional damping;      +beatKcurr*KCurrent
betaKinit = 0.0 # initial-stiffness proportional damping      +beatKinit*Kini

op.rayleigh(alphaM,betaKcurr, betaKinit, betaKcomm) # RAYLEIGH damping

# Uniform EXCITATION: acceleration input
IDloadTag = 400                         # load tag
dt = 0.01                          # time step for input ground motion
GMfatt = 1.0                          # data in input file is in g Unifts --␣
↪ACCELERATION TH
maxNumIter = 10
op.timeSeries('Path', 2, '-dt', dt, '-filePath', GMfile, '-factor', GMfact)
op.pattern('UniformExcitation', IDloadTag, GMdirection, '-accel', 2)

op.wipeAnalysis()
op.constraints('Transformation')
op.numberer('Plain')
op.system('BandGeneral')
op.test('EnergyIncr', Tol, maxNumIter)
op.algorithm('ModifiedNewton')

```

(continues on next page)

(continued from previous page)

```
NewmarkGamma = 0.5
NewmarkBeta = 0.25
op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
op.analysis('Transient')

DtAnalysis = 0.01 # time-step Dt for lateral analysis
TmaxAnalysis = 10.0 # maximum duration of ground-motion analysis

Nsteps =  int(TmaxAnalysis/ DtAnalysis)

ok = op.analyze(Nsteps, DtAnalysis)

tCurrent = op.getTime()

# for gravity analysis, load control is fine, 0.1 is the load factor increment (http:/
↪/opensees.berkeley.edu/wiki/index.php/Load_Control)

test = {1:'NormDispIncr', 2: 'RelativeEnergyIncr', 4: 'RelativeNormUnbalance',5:
↪'RelativeNormDispIncr', 6: 'NormUnbalance'}
algorithm = {1:'KrylovNewton', 2: 'SecantNewton' , 4: 'RaphsonNewton',5:
↪'PeriodicNewton', 6: 'BFGS', 7: 'Broyden', 8: 'NewtonLineSearch'}

for i in test:
    for j in algorithm:

        if ok != 0:
            if j < 4:
                op.algorithm(algorithm[j], '-initial')

            else:
                op.algorithm(algorithm[j])

            op.test(test[i], Tol, 1000)
            ok = op.analyze(Nsteps, DtAnalysis)
            print(test[i], algorithm[j], ok)
            if ok == 0:
                break
        else:
            continue

u2 = op.nodeDisp(2, 1)
print("u2 = ", u2)

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`.

```
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 22 15:12:06 2019

@author: pchi893
"""
# Converted to openseespy by: Pavan Chigullapally
#                         University of Auckland
#                         Email: pchi893@aucklanduni.ac.nz
# Example 3. 2D Cantilever -- Static Pushover
#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)
#and run the current script
#the detailed problem description can be found here: http://opensees.berkeley.edu/wiki/index.php/Examples_Manual  (example: 3)
# ------------------------------------------------------------------------------------------
#        OpenSees (Tcl) code by:        Silvia Mazzoni & Frank McKenna, 2006
# characteristics of pushover analysis
#############################################################################################
import openseespy.opensees as op
#import the os module
#import os
import math
op.wipe()

from InelasticFiberSection import *
Dmax = 0.05*LCol
Dincr = 0.001*LCol
Hload = Weight
maxNumIter = 6
tol = 1e-8

op.timeSeries('Linear', 2)
op.pattern('Plain', 200, 2)
op.load(2, Hload, 0.0,0.0)

op.wipeAnalysis()
op.constraints('Plain')
op.numberer('Plain')
op.system('BandGeneral')
op.test('EnergyIncr', Tol, maxNumIter)
op.algorithm('Newton')

op.integrator('DisplacementControl', IDctrlNode, IDctrlDOF, Dincr)
op.analysis('Static')


```

(continues on next page)

(continued from previous page)

```
Nsteps =  int(Dmax/ Dincr)

ok = op.analyze(Nsteps)
print(ok)

# for gravity analysis, load control is fine, 0.1 is the load factor increment (http:/
↪/opensees.berkeley.edu/wiki/index.php/Load_Control)

test = {1:'NormDispIncr', 2: 'RelativeEnergyIncr', 4: 'RelativeNormUnbalance',5:
↪'RelativeNormDispIncr', 6: 'NormUnbalance'}
algorithm = {1:'KrylovNewton', 2: 'SecantNewton' , 4: 'RaphsonNewton',5:
↪'PeriodicNewton', 6: 'BFGS', 7: 'Broyden', 8: 'NewtonLineSearch'}

for i in test:
    for j in algorithm:

        if ok != 0:
            if j < 4:
                op.algorithm(algorithm[j], '-initial')

            else:
                op.algorithm(algorithm[j])

            op.test(test[i], Tol, 1000)
            ok = op.analyze(Nsteps)
            print(test[i], algorithm[j], ok)
            if ok == 0:
                break
        else:
            continue

u2 = op.nodeDisp(2, 1)
print("u2 = ", u2)

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`.

```
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 22 15:12:06 2019

@author: pchi893
"""
# Converted to openseespy by: Pavan Chigullapally
#                          University of Auckland
#                          Email: pchi893@aucklanduni.ac.nz
# Example4. 2D Portal Frame--  Dynamic EQ input analysis

#To run Uniaxial Inelastic Material, Fiber Section, Nonlinear Mode, Uniform Earthquake Excitation:First import the InelasticFiberSectionPortal2Dframe.py
#(upto gravity loading is already in this script) and run the current script
#To run EQ ground-motion analysis (ReadRecord.py, H-E12140.AT2 needs to be downloaded into the same directory)
#Same acceleration input at all nodes restrained in specified direction (uniform acceleration input at all support nodes)
#the problem description can be found here:
#http://opensees.berkeley.edu/wiki/index.php/Examples_Manual and http://opensees.berkeley.edu/wiki/index.php/OpenSees_Example_4._Portal_Frame(example: 4)
# ------------------------------------------------------------------------------
#        OpenSees (Tcl) code by:        Silvia Mazzoni & Frank McKenna, 2006
################################################################################
import openseespy.opensees as op
#import the os module
#import os
import math
op.wipe()
################################################################################

from InelasticFiberSectionPortal2Dframe import *
#applying Dynamic Ground motion analysis
Tol = 1e-8
maxNumIter = 10
GMdirection = 1
GMfact = 1.5
GMfatt = g*GMfact
DtAnalysis = 0.01*sec # time-step Dt for lateral analysis
TmaxAnalysis = 10.0*sec # maximum duration of ground-motion analysis


Lambda = op.eigen('-fullGenLapack', 1) # eigenvalue mode 1
Omega = math.pow(Lambda, 0.5)
betaKcomm = 2 * (0.02/Omega)

xDamp = 0.02                                # 2% damping ratio
alphaM = 0.0                                # M-prop. damping; D = alphaM*M
betaKcurr = 0.0                 # K-proportional damping;      +beatKcurr*KCurrent
betaKinit = 0.0 # initial-stiffness proportional damping      +beatKinit*Kini

op.rayleigh(alphaM,betaKcurr, betaKinit, betaKcomm) # RAYLEIGH damping

# Set some parameters
```

(continues on next page)

(continued from previous page)

```
record = 'H-E12140'

import ReadRecord
# Permform the conversion from SMD record to OpenSees record
dt, nPts = ReadRecord.ReadRecord(record+'.at2', record+'.dat')
#print(dt, nPts)

# Uniform EXCITATION: acceleration input
IDloadTag = 400                         # load tag
op.timeSeries('Path', 2, '-dt', dt, '-filePath', record+'.dat', '-factor', GMfatt)
op.pattern('UniformExcitation', IDloadTag, GMdirection, '-accel', 2)

op.wipeAnalysis()
op.constraints('Transformation')
op.numberer('RCM')
op.system('BandGeneral')
#op.test('EnergyIncr', Tol, maxNumIter)
#op.algorithm('ModifiedNewton')
#NewmarkGamma = 0.5
#NewmarkBeta = 0.25
#op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
#op.analysis('Transient')


#Nsteps =  int(TmaxAnalysis/ DtAnalysis)
#
#ok = op.analyze(Nsteps, DtAnalysis)

tCurrent = op.getTime()

# for gravity analysis, load control is fine, 0.1 is the load factor increment (http:/
↪/opensees.berkeley.edu/wiki/index.php/Load_Control)

test = {1:'NormDispIncr', 2: 'RelativeEnergyIncr', 3:'EnergyIncr', 4:
↪'RelativeNormUnbalance',5: 'RelativeNormDispIncr', 6: 'NormUnbalance'}
algorithm = {1:'KrylovNewton', 2: 'SecantNewton' , 3:'ModifiedNewton' , 4:
↪'RaphsonNewton',5: 'PeriodicNewton', 6: 'BFGS', 7: 'Broyden', 8: 'NewtonLineSearch'}

tFinal = nPts*dt

#tFinal = 10.0*sec
time = [tCurrent]
u3 = [0.0]
u4 = [0.0]
ok = 0
while tCurrent < tFinal:
#    ok = op.analyze(1, .01)
    for i in test:
        for j in algorithm:
            if j < 4:
                op.algorithm(algorithm[j], '-initial')

            else:
                op.algorithm(algorithm[j])
            while ok == 0 and tCurrent < tFinal:

                op.test(test[i], Tol, maxNumIter)
```

(continues on next page)

(continued from previous page)

```
                NewmarkGamma = 0.5
                NewmarkBeta = 0.25
                op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
                op.analysis('Transient')
                ok = op.analyze(1, .01)

                if ok == 0 :
                    tCurrent = op.getTime()
                    time.append(tCurrent)
                    u3.append(op.nodeDisp(3,1))
                    u4.append(op.nodeDisp(4,1))
                    print(test[i], algorithm[j], 'tCurrent=', tCurrent)


import matplotlib.pyplot as plt
plt.figure(figsize=(8,8))
plt.plot(time, u3)
plt.ylabel('Horizontal Displacement of node 3 (in)')
plt.xlabel('Time (s)')
plt.savefig('Horizontal Disp at Node 3 vs time.jpeg', dpi = 500)
plt.show()

plt.figure(figsize=(8,8))
plt.plot(time, u4)
plt.ylabel('Horizontal Displacement of node 4 (in)')
plt.xlabel('Time (s)')
plt.savefig('Horizontal Disp at Node 4 vs time.jpeg', dpi = 500)
plt.show()


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`.

```
# -*- coding: utf-8 -*-
"""
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␣
    ↪excitation using acceleration timeseries
11
12  #To run Uniaxial Inelastic Material, Fiber Section, Nonlinear Mode, MultipleSupport␣
    ↪Earthquake ground motion:First import the InelasticFiberSectionPortal2Dframe.py
13  #(upto gravity loading is already in this script) and run the current script
14  #To run EQ ground-motion analysis (ReadRecord.py, H-E12140.AT2 needs to be downloaded␣
    ↪into the same directory)
15  # MultipleSupport Earthquake ground motion (different acceleration input at specified␣
    ↪support nodes) -- two nodes here
16  #the problem description can be found here:
17  #http://opensees.berkeley.edu/wiki/index.php/Examples_Manual and http://opensees.
    ↪berkeley.edu/wiki/index.php/OpenSees_Example_4._Portal_Frame(example: 4)
18  # ------------------------------------------------------------------------------------
    ↪--------------
19  #        OpenSees (Tcl) code by:        Silvia Mazzoni & Frank McKenna, 2006
20  ######################################################################################
    ↪######################################################################################
21  import openseespy.opensees as op
22  #import the os module
23  #import os
24  import math
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␣
    ↪support nodes) -- two nodes here
33
34  #applying Dynamic Ground motion analysis
35  iSupportNode = [1, 2]
36  iGMfact = [1.5, 1.5]
37  iGMdirection = [1, 1]
38  iGMfile = ['H-E12140', 'H-E12140']
39  DtAnalysis = 0.01*sec # time-step Dt for lateral analysis
40  TmaxAnalysis = 10.0*sec # maximum duration of ground-motion analysis
41  Tol = 1e-8
42
43  # define DAMPING-------------------------------------------------------------------------
    ↪----------------
44  # apply Rayleigh DAMPING from $xDamp
45  # D=$alphaM*M + $betaKcurr*Kcurrent + $betaKcomm*KlastCommit + $beatKinit*$Kinitial
46  Lambda = op.eigen('-fullGenLapack', 1) # eigenvalue mode 1
47  Omega = math.pow(Lambda, 0.5)
48  betaKcomm = 2 * (0.02/Omega)
49
50  xDamp = 0.02                                # 2% damping ratio
```

(continues on next page)

(continued from previous page)

```
alphaM = 0.0                                    # M-prop. damping; D = alphaM*M
betaKcurr = 0.0                 # K-proportional damping;       +beatKcurr*KCurrent
betaKinit = 0.0 # initial-stiffness proportional damping      +beatKinit*Kini

op.rayleigh(alphaM,betaKcurr, betaKinit, betaKcomm) # RAYLEIGH damping
#-------------------------------------------------------------------------------
#  ---------------------------------    perform Dynamic Ground-Motion Analysis
# the following commands are unique to the Multiple-Support Earthquake excitation
# Set some parameters
IDloadTag = 400                         # load tag
IDgmSeries = 500 # for multipleSupport Excitation

# read a PEER strong motion database file, extracts dt from the header and converts the file
# to the format OpenSees expects for Uniform/multiple-support ground motions
record = ['H-E12140', 'H-E12140']
#dt =[]
#nPts = []

import ReadRecord
# Permform the conversion from SMD record to OpenSees record
#dt, nPts = ReadRecord.ReadRecord(record+'.at2', record+'.dat')
#print(dt, nPts)
count = 2
#use displacement series, create time series('Path'), then create multi-support excitation patter (gmtag, 'Plain'), then create imposed ground motion
#using groundmotion('nodetag', gmtag), run this in a loop for each support or node where the earthquake load is going to be applied.
op.pattern('MultipleSupport', IDloadTag)
for i in range(len(iSupportNode)):
    record_single = record[i]
    GMfatt = (iGMfact[i])*g
    dt, nPts = ReadRecord.ReadRecord(record_single+'.AT2', record_single+'.dat')
    op.timeSeries('Path', count, '-dt', dt, '-filePath', record_single+'.dat', '-factor', GMfatt)
    op.groundMotion(IDgmSeries+count, 'Plain', '-accel', count)
    op.imposedMotion(iSupportNode[i], iGMdirection[i], IDgmSeries+count)
    count = count + 1

maxNumIter = 10
op.wipeAnalysis()
op.constraints('Transformation')
op.numberer('RCM')
op.system('BandGeneral')
#op.test('EnergyIncr', Tol, maxNumIter)
#op.algorithm('ModifiedNewton')
#NewmarkGamma = 0.5
#NewmarkBeta = 0.25
#op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
#op.analysis('Transient')
#
#
#Nsteps =  int(TmaxAnalysis/ DtAnalysis)
#
#ok = op.analyze(Nsteps, DtAnalysis)

```

(continues on next page)

(continued from previous page)

```
tCurrent = op.getTime()

# for gravity analysis, load control is fine, 0.1 is the load factor increment (http:/
↪/opensees.berkeley.edu/wiki/index.php/Load_Control)

test = {1:'NormDispIncr', 2: 'RelativeEnergyIncr', 3:'EnergyIncr', 4:
↪'RelativeNormUnbalance',5: 'RelativeNormDispIncr', 6: 'NormUnbalance'}
algorithm = {1:'KrylovNewton', 2: 'SecantNewton' , 3:'ModifiedNewton' , 4:
↪'RaphsonNewton',5: 'PeriodicNewton', 6: 'BFGS', 7: 'Broyden', 8: 'NewtonLineSearch'}

#tFinal = TmaxAnalysis
tFinal = nPts*dt
time = [tCurrent]
u3 = [0.0]
u4 = [0.0]
ok = 0

while tCurrent < tFinal:
#    ok = op.analyze(1, .01)
    for i in test:
        for j in algorithm:
            if j < 4:
                op.algorithm(algorithm[j], '-initial')

            else:
                op.algorithm(algorithm[j])
            while ok == 0 and tCurrent < tFinal:

                op.test(test[i], Tol, maxNumIter)
                NewmarkGamma = 0.5
                NewmarkBeta = 0.25
                op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
                op.analysis('Transient')
                ok = op.analyze(1, .01)

                if ok == 0 :
                    tCurrent = op.getTime()
                    time.append(tCurrent)
                    u3.append(op.nodeDisp(3,1))
                    u4.append(op.nodeDisp(4,1))
                    print(test[i], algorithm[j], 'tCurrent=', tCurrent)

import matplotlib.pyplot as plt
plt.figure(figsize=(8,8))
plt.plot(time, u3)
plt.ylabel('Horizontal Displacement of node 3 (in)')
plt.xlabel('Time (s)')
plt.savefig('Horizontal Disp at Node 3 vs time-multiple support excitation-acctime.
↪jpeg', dpi = 500)
plt.show()

plt.figure(figsize=(8,8))
plt.plot(time, u4)
plt.ylabel('Horizontal Displacement of node 4 (in)')
plt.xlabel('Time (s)')
plt.savefig('Horizontal Disp at Node 4 vs time-multiple support excitation-acctime.
↪jpeg', dpi = 500)
```

(continues on next page)

(continued from previous page)

```
plt.show()


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`.

```
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 22 15:12:06 2019

@author: pchi893
"""
# Converted to openseespy by: Pavan Chigullapally
#                         University of Auckland
#                         Email: pchi893@aucklanduni.ac.nz
# Example4. 2D Portal Frame--  Dynamic EQ input analysis-- multiple-support excitation using displacement timeseries

#To run Uniaxial Inelastic Material, Fiber Section, Nonlinear Mode, MultipleSupport Earthquake ground motion:First import the InelasticFiberSectionPortal2Dframe.py
#(upto gravity loading is already in this script) and run the current script
#To run EQ ground-motion analysis (ReadRecord.py, H-E12140.DT2 needs to be downloaded into the same directory)
# MultipleSupport Earthquake ground motion (different displacement input at specified support nodes) -- two nodes here
#the problem description can be found here:
#http://opensees.berkeley.edu/wiki/index.php/Examples_Manual and http://opensees.berkeley.edu/wiki/index.php/OpenSees_Example_4._Portal_Frame(example: 4)
# ------------------------------------------------------------------------------------------------
#         OpenSees (Tcl) code by:        Silvia Mazzoni & Frank McKenna, 2006
############################################################################################################
import openseespy.opensees as op
#import the os module
#import os
import math
```

(continues on next page)

(continued from previous page)

```
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, 1.25]
iGMdirection = [1, 1]
iGMfile = ['H-E12140', '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 $xDamp
# D=$alphaM*M + $betaKcurr*Kcurrent + $betaKcomm*KlastCommit + $beatKinit*$Kinitial
Lambda = op.eigen('-fullGenLapack', 1) # eigenvalue mode 1
Omega = math.pow(Lambda, 0.5)
betaKcomm = 2 * (0.02/Omega)

xDamp = 0.02                                # 2% damping ratio
alphaM = 0.0                                # M-prop. damping; D = alphaM*M
betaKcurr = 0.0                 # K-proportional damping;      +beatKcurr*KCurrent
betaKinit = 0.0 # initial-stiffness proportional damping      +beatKinit*Kini

op.rayleigh(alphaM,betaKcurr, betaKinit, betaKcomm) # RAYLEIGH damping
#-------------------------------------------------------------------------------------------
#  ---------------------------------    perform Dynamic Ground-Motion Analysis
# the following commands are unique to the Multiple-Support Earthquake excitation
# Set some parameters
IDloadTag = 400                         # load tag
IDgmSeries = 500 # for multipleSupport Excitation

# read a PEER strong motion database file, extracts dt from the header and converts the file
# to the format OpenSees expects for Uniform/multiple-support ground motions
record = ['H-E12140', 'H-E12140']
#dt =[]
#nPts = []

import ReadRecord
# Permform the conversion from SMD record to OpenSees record
#dt, nPts = ReadRecord.ReadRecord(record+'.at2', record+'.dat')
#print(dt, nPts)
count = 2
#use displacement series, create time series('Path'), then create multi-support excitation patter (gmtag, 'Plain'), then create imposed ground motion
#using groundmotion('nodetag', gmtag), run this in a loop for each support or node where the earthquake load is going to be applied.
```

(continues on next page)

(continued from previous page)

```
op.pattern('MultipleSupport', IDloadTag)
for i in range(len(iSupportNode)):
    record_single = record[i]
    GMfatt = (iGMfact[i])*cm
    dt, nPts = ReadRecord.ReadRecord(record_single+'.DT2', record_single+'.dat')
    op.timeSeries('Path', count, '-dt', dt, '-filePath', record_single+'.dat', '-factor', GMfatt)
    op.groundMotion(IDgmSeries+count, 'Plain', '-disp', count)
    op.imposedMotion(iSupportNode[i], iGMdirection[i], IDgmSeries+count)
    count = count + 1

maxNumIter = 10
op.wipeAnalysis()
op.constraints('Transformation')
op.numberer('RCM')
op.system('BandGeneral')
#op.test('EnergyIncr', Tol, maxNumIter)
#op.algorithm('ModifiedNewton')
#NewmarkGamma = 0.5
#NewmarkBeta = 0.25
#op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
#op.analysis('Transient')
#
#
#Nsteps =  int(TmaxAnalysis/ DtAnalysis)
#
#ok = op.analyze(Nsteps, DtAnalysis)

tCurrent = op.getTime()

# for gravity analysis, load control is fine, 0.1 is the load factor increment (http://opensees.berkeley.edu/wiki/index.php/Load_Control)

test = {1:'NormDispIncr', 2: 'RelativeEnergyIncr', 3:'EnergyIncr', 4: 'RelativeNormUnbalance',5: 'RelativeNormDispIncr', 6: 'NormUnbalance'}
algorithm = {1:'KrylovNewton', 2: 'SecantNewton' , 3:'ModifiedNewton' , 4: 'RaphsonNewton',5: 'PeriodicNewton', 6: 'BFGS', 7: 'Broyden', 8: 'NewtonLineSearch'}

#tFinal = TmaxAnalysis
tFinal = nPts*dt
time = [tCurrent]
u3 = [0.0]
u4 = [0.0]
ok = 0

while tCurrent < tFinal:
#    ok = op.analyze(1, .01)
    for i in test:
        for j in algorithm:
            if j < 4:
                op.algorithm(algorithm[j], '-initial')

            else:
                op.algorithm(algorithm[j])
            while ok == 0 and tCurrent < tFinal:

                op.test(test[i], Tol, maxNumIter)
```

(continues on next page)

(continued from previous page)

```
                NewmarkGamma = 0.5
                NewmarkBeta = 0.25
                op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
                op.analysis('Transient')
                ok = op.analyze(1, .01)

                if ok == 0 :
                    tCurrent = op.getTime()
                    time.append(tCurrent)
                    u3.append(op.nodeDisp(3,1))
                    u4.append(op.nodeDisp(4,1))
                    print(test[i], algorithm[j], 'tCurrent=', tCurrent)

import matplotlib.pyplot as plt
plt.figure(figsize=(8,8))
plt.plot(time, u3)
plt.ylabel('Horizontal Displacement of node 3 (in)')
plt.xlabel('Time (s)')
plt.savefig('Horizontal Disp at Node 3 vs time-multiple support excitation-disptime.
↪jpeg', dpi = 500)
plt.show()

plt.figure(figsize=(8,8))
plt.plot(time, u4)
plt.ylabel('Horizontal Displacement of node 4 (in)')
plt.xlabel('Time (s)')
plt.savefig('Horizontal Disp at Node 4 vs time-multiple support excitation-disptime.
↪jpeg', dpi = 500)
plt.show()


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`.

```
# -*- coding: utf-8 -*-
"""
```

(continues on next page)

(continued from previous page)

```
Created on Mon Apr 22 15:12:06 2019

@author: pchi893
"""
# Converted to openseespy by: Pavan Chigullapally
#                         University of Auckland
#                         Email: pchi893@aucklanduni.ac.nz

# Example4. 2D Portal Frame--  Dynamic EQ input analysis-- Bidirectional-uniform support excitation using acceleration timeseries

#To run Uniaxial Inelastic Material, Fiber Section, Nonlinear Mode, Bidirectional-uniform earthquake ground motion:First import the InelasticFiberSectionPortal2Dframe.py
#(upto gravity loading is already in this script) and run the current script
#To run EQ ground-motion analysis (ReadRecord.py, H-E12140.AT2 and H-E01140.AT2 needs to be downloaded into the same directory)
# Bidirectional-uniform support excitation using acceleration timeseries (different accelerations are input at all support nodes in two directions) -- two support nodes here
#the problem description can be found here:
#http://opensees.berkeley.edu/wiki/index.php/Examples_Manual and http://opensees.berkeley.edu/wiki/index.php/OpenSees_Example_4._Portal_Frame(example: 4)
# ------------------------------------------------------------------------------------------------
#        OpenSees (Tcl) code by:        Silvia Mazzoni & Frank McKenna, 2006
# ------------------------------------------------------------------------------------------------
################################################################################################################################################################
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)

```
# D=$alphaM*M + $betaKcurr*Kcurrent + $betaKcomm*KlastCommit + $beatKinit*$Kinitial
Lambda = op.eigen('-fullGenLapack', 1) # eigenvalue mode 1
Omega = math.pow(Lambda, 0.5)
betaKcomm = 2 * (0.02/Omega)

xDamp = 0.02                                # 2% damping ratio
alphaM = 0.0                                # M-prop. damping; D = alphaM*M
betaKcurr = 0.0                 # K-proportional damping;      +beatKcurr*KCurrent
betaKinit = 0.0 # initial-stiffness proportional damping      +beatKinit*Kini

op.rayleigh(alphaM,betaKcurr, betaKinit, betaKcomm) # RAYLEIGH damping
#-------------------------------------------------------------------------------------------
#  ---------------------------------    perform Dynamic Ground-Motion Analysis
# the following commands are unique to the Multiple-Support Earthquake excitation
# Set some parameters
IDloadTag = 400                         # load tag


# read a PEER strong motion database file, extracts dt from the header and converts the file
# to the format OpenSees expects for Uniform/multiple-support ground motions
record = ['H-E01140', 'H-E12140']
#dt =[]
#nPts = []

import ReadRecord

#this is similar to uniform excitation in single direction
count = 2
for i in range(len(iGMdirection)):
    IDloadTag = IDloadTag+count
    record_single = record[i]
    GMfatt = (iGMfact[i])*g
    dt, nPts = ReadRecord.ReadRecord(record_single+'.AT2', record_single+'.dat')
    op.timeSeries('Path', count, '-dt', dt, '-filePath', record_single+'.dat', '-factor', GMfatt)
    op.pattern('UniformExcitation', IDloadTag, iGMdirection[i], '-accel', 2)
    count = count + 1

maxNumIter = 10
op.wipeAnalysis()
op.constraints('Transformation')
op.numberer('RCM')
op.system('BandGeneral')
#op.test('EnergyIncr', Tol, maxNumIter)
#op.algorithm('ModifiedNewton')
#NewmarkGamma = 0.5
#NewmarkBeta = 0.25
#op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
#op.analysis('Transient')


#Nsteps =  int(TmaxAnalysis/ DtAnalysis)
#
#ok = op.analyze(Nsteps, DtAnalysis)

```

(continues on next page)

(continued from previous page)

```
tCurrent = op.getTime()

# for gravity analysis, load control is fine, 0.1 is the load factor increment (http:/
↪/opensees.berkeley.edu/wiki/index.php/Load_Control)

test = {1:'NormDispIncr', 2: 'RelativeEnergyIncr', 3:'EnergyIncr', 4:
↪'RelativeNormUnbalance',5: 'RelativeNormDispIncr', 6: 'NormUnbalance'}
algorithm = {1:'KrylovNewton', 2: 'SecantNewton' , 3:'ModifiedNewton' , 4:
↪'RaphsonNewton',5: 'PeriodicNewton', 6: 'BFGS', 7: 'Broyden', 8: 'NewtonLineSearch'}

#tFinal = TmaxAnalysis
tFinal = nPts*dt
time = [tCurrent]
u3 = [0.0]
u4 = [0.0]
ok = 0

while tCurrent < tFinal:
#    ok = op.analyze(1, .01)
    for i in test:
        for j in algorithm:
            if j < 4:
                op.algorithm(algorithm[j], '-initial')

            else:
                op.algorithm(algorithm[j])
            while ok == 0 and tCurrent < tFinal:

                op.test(test[i], Tol, maxNumIter)
                NewmarkGamma = 0.5
                NewmarkBeta = 0.25
                op.integrator('Newmark', NewmarkGamma, NewmarkBeta)
                op.analysis('Transient')
                ok = op.analyze(1, .01)

                if ok == 0 :
                    tCurrent = op.getTime()
                    time.append(tCurrent)
                    u3.append(op.nodeDisp(3,1))
                    u4.append(op.nodeDisp(4,1))
                    print(test[i], algorithm[j], 'tCurrent=', tCurrent)

import matplotlib.pyplot as plt
plt.figure(figsize=(8,8))
plt.plot(time, u3)
plt.ylabel('Horizontal Displacement of node 3 (in)')
plt.xlabel('Time (s)')
plt.savefig('Horizontal Disp at Node 3 vs time-uniform excitation-acctime.jpeg', dpi␣
↪= 500)
plt.show()

plt.figure(figsize=(8,8))
plt.plot(time, u4)
plt.ylabel('Horizontal Displacement of node 4 (in)')
plt.xlabel('Time (s)')
plt.savefig('Horizontal Disp at Node 4 vs time-uniform excitation-acctime.jpeg', dpi␣
↪= 500)
```

(continues on next page)

(continued from previous page)

```
plt.show()
#

op.wipe()
```

1.14.3 Tsunami Examples

1. *Moving Mesh*
2. *Background Mesh*

Moving Mesh

1. *Dambreak Analysis using moving mesh*
2. *Dambreak with Elastic Obstacle Analysis using moving mesh*

Dambreak Analysis using moving mesh

1. The source code is shown below, which can be downloaded `here`.
2. Run the source code in your favorite Python program.
3. 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.

```
import os
import openseespy.opensees as ops

# ------------------------------
# Start of model generation
# ------------------------------

# remove existing model
ops.wipe()

# set modelbuilder
ops.model('basic', '-ndm', 2, '-ndf', 2)

# geometric
L = 0.146
H = L*2
H2 = 0.3
h = 0.005
alpha = 1.4
tw =  3*h

# material
rho = 1000.0
mu = 0.0001
b1 = 0.0
b2 = -9.81
thk = 0.012
kappa = -1.0
```

(continues on next page)

(continued from previous page)

```

# time steps
dtmax = 1e-3
dtmin = 1e-6
totaltime = 1.0

# filename
filename = 'dambreak'

# recorder
if not os.path.exists(filename):
    os.makedirs(filename)
ops.recorder('PVD', filename, 'disp', 'vel', 'pressure')

# nodes
ops.node(1, 0.0, 0.0)
ops.node(2, L, 0.0)
ops.node(3, L, H)
ops.node(4, 0.0, H)
ops.node(5, 0.0, H2)
ops.node(6, 4*L, 0.0)
ops.node(7, 4*L, H2)
ops.node(8, -tw, H2)
ops.node(9, -tw, -tw)
ops.node(10, 4*L+tw, -tw)
ops.node(11, 4*L+tw, H2)

# ids for meshing
wall_id = 1
water_bound_id = -1
water_body_id = -2

# wall mesh
wall_tag = 3
ndf = 2
ops.mesh('line', 1, 9, 4,5,8,9,10,11,7,6,2, wall_id, ndf, h)
ops.mesh('line', 2, 3, 2,1,4, wall_id, ndf, h)
ops.mesh('tri', wall_tag, 2, 1,2, wall_id, ndf, h)

# fluid mesh
fluid_tag = 4
ops.mesh('line', 5, 3, 2,3,4, water_bound_id, ndf, h)

eleArgs = ['PFEMElementBubble',rho,mu,b1,b2,thk,kappa]
ops.mesh('tri', fluid_tag, 2, 2,5, water_body_id, ndf, h, *eleArgs)

for nd in ops.getNodeTags('-mesh', wall_tag):
    ops.fix(nd, 1,1)

# save the original modal
ops.record()

# create constraint object
ops.constraints('Plain')

# create numberer object
ops.numberer('Plain')
```

(continues on next page)

(continued from previous page)

```

# create convergence test object
ops.test('PFEM', 1e-5, 1e-5, 1e-5, 1e-5, 1e-15, 1e-15, 20, 3, 1, 2)

# create algorithm object
ops.algorithm('Newton')

# create integrator object
ops.integrator('PFEM')

# create SOE object
ops.system('PFEM', '-umfpack', '-print')

# create analysis object
ops.analysis('PFEM', dtmax, dtmin, b2)

# analysis
while ops.getTime() < totaltime:

    # analysis
    if ops.analyze() < 0:
        break

    ops.remesh(alpha)



```

Dambreak with Elastic Obstacle Analysis using moving mesh

1. The source code is shown below, which can be downloaded `here`.
2. Run the source code in your favorite Python program.
3. 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.

```
import os
import openseespy.opensees as ops

# ------------------------------
# Start of model generation
# ------------------------------

# remove existing model
ops.wipe()

# set modelbuilder
ops.model('basic', '-ndm', 2, '-ndf', 3)

# geometric
L = 0.146
H = 2*L
H2 = 0.3
b = 0.012
```

(continues on next page)

(continued from previous page)

```
h = 0.005
alpha = 1.4
Hb = 20.0*b/3.0
tw =  3*h

# material
rho = 1000.0
mu = 0.0001
b1 = 0.0
b2 = -9.81
thk = 0.012
kappa = -1.0

rhos = 2500.0
A = thk*thk
E = 1e6
Iz = thk*thk*thk*thk/12.0
bmass = A*Hb*rhos

# analysis
dtmax = 1e-3
dtmin = 1e-6
totaltime = 1.0

filename = 'obstacle'

# recorder
if not os.path.exists(filename):
    os.makedirs(filename)
ops.recorder('PVD', filename, 'disp', 'vel', 'pressure')

# nodes
ops.node(1, 0.0, 0.0)
ops.node(2, L, 0.0)
ops.node(3, L, H, '-ndf', 2)
ops.node(4, 0.0, H)
ops.node(5, 0.0, H2)
ops.node(6, 4*L, 0.0)
ops.node(7, 4*L, H2)
ops.node(8, -tw, H2)
ops.node(9, -tw, -tw)
ops.node(10, 4*L+tw, -tw)
ops.node(11, 4*L+tw, H2)
ops.node(12, 2*L, 0.0)
ops.node(13, 2*L, Hb)

# ids for meshing
wall_id = 1
beam_id = 2
water_bound_id = -1
water_body_id = -2

# transformation
transfTag = 1
ops.geomTransf('Corotational', transfTag)

# section
```

(continues on next page)

(continued from previous page)

```
secTag = 1
ops.section('Elastic', secTag, E, A, Iz)

# beam integration
inteTag = 1
numpts = 2
ops.beamIntegration('Legendre', inteTag, secTag, numpts)

# beam mesh
beamTag = 6
ndf = 3
ops.mesh('line', beamTag, 2, 12, 13, beam_id, ndf, h, 'dispBeamColumn', transfTag,␣
↪inteTag)

ndmass = bmass/len(ops.getNodeTags('-mesh', beamTag))

for nd in ops.getNodeTags('-mesh', beamTag):
    ops.mass(nd, ndmass, ndmass, 0.0)

# fluid mesh
fluidTag = 4
ndf = 2
ops.mesh('line', 1, 10, 4,5,8,9,10,11,7,6,12,2, wall_id, ndf, h)
ops.mesh('line', 2, 3, 2,1,4, wall_id, ndf, h)
ops.mesh('line', 3, 3, 2,3,4, water_bound_id, ndf, h)

eleArgs = ['PFEMElementBubble',rho,mu,b1,b2,thk,kappa]
ops.mesh('tri', fluidTag, 2, 2,3, water_body_id, ndf, h, *eleArgs)

# wall mesh
wallTag = 5
ops.mesh('tri', wallTag, 2, 1,2, wall_id, ndf, h)

for nd in ops.getNodeTags('-mesh', wallTag):
    ops.fix(nd, 1,1,1)

# save the original modal
ops.record()

# create constraint object
ops.constraints('Plain')

# create numberer object
ops.numberer('Plain')

# create convergence test object
ops.test('PFEM', 1e-5, 1e-5, 1e-5, 1e-5, 1e-15, 1e-15, 20, 3, 1, 2)

# create algorithm object
ops.algorithm('Newton')

# create integrator object
ops.integrator('PFEM')

# create SOE object
ops.system('PFEM')

```

(continues on next page)

(continued from previous page)

```
# create analysis object
ops.analysis('PFEM', dtmax, dtmin, b2)

# analysis
while ops.getTime() < totaltime:

    # analysis
    if ops.analyze() < 0:
        break

    ops.remesh(alpha)
```

Background Mesh

1. *Dambreak Analysis using background mesh*
2. *Dambreak with Elastic Obstacle Analysis using background mesh*

Dambreak Analysis using background mesh

1. The source code is shown below, which can be downloaded `here`.
2. Run the source code in your favorite Python program.
3. 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.

```
import os
import os.path
import openseespy.opensees as ops

# ------------------------------
# Start of model generation
# ------------------------------

# wipe all previous objects
ops.wipe()

# create a model with fluid
ops.model('basic', '-ndm', 2, '-ndf', 2)

# geometric
L = 0.146
H = L * 2
h = L / 40

# number of particles per cell in each direction
numx = 3.0
numy = 3.0

# material
rho = 1000.0
mu = 0.0001
b1 = 0.0
b2 = -9.81
```

(continues on next page)

(continued from previous page)

```
thk = 0.012
kappa = -1.0

# analysis
dtmax = 1e-3
dtmin = 1e-3
totaltime = 1.0
filename = 'dambreak-bg'

# recorder
ops.recorder('BgPVD', filename, 'disp', 'vel', 'pressure', '-dT', 1e-3)
if not os.path.exists(filename):
    os.makedirs(filename)

# fluid particles
ndf = 2

# total number of particles in each direction
nx = round(L / h * numx)
ny = round(H / h * numy)

# create particles
eleArgs = ['PFEMElementBubble', rho, mu, b1, b2, thk, kappa]
partArgs = ['quad', 0.0, 0.0, L, 0.0, L, H, 0.0, H, nx, ny]
parttag = 1
ops.mesh('part', parttag, *partArgs, *eleArgs, '-vel', 0.0, 0.0)

print('num particles =', nx * ny)

# wall
ops.node(1, 0.0, H)
ops.node(2, 0.0, 0.0)
ops.node(3, 4 * L, 0.0)
ops.node(4, 4 * L, H)

walltag = 2
wallid = 1
ops.mesh('line', walltag, 4, 1, 2, 3, 4, wallid, ndf, h)

wallnodes = ops.getNodeTags('-mesh', walltag)

for nd in wallnodes:
    ops.fix(nd, 1, 1)

# background mesh
lower = [-h, -h]
upper = [4 * L + L, H + L]

ops.mesh('bg', h, *lower, *upper,
         '-structure', wallid, len(wallnodes), *wallnodes)

# create constraint object
ops.constraints('Plain')

# create numberer object
ops.numberer('Plain')

```

(continues on next page)

(continued from previous page)

```
# create convergence test object
ops.test('PFEM', 1e-5, 1e-5, 1e-5, 1e-5, 1e-5, 1e-5, 10, 3, 1, 2)

# create algorithm object
ops.algorithm('Newton')

# create integrator object
ops.integrator('PFEM', 0.5, 0.25)

# create SOE object
ops.system('PFEM')
# ops.system('PFEM', '-mumps) Linux version can use mumps

# create analysis object
ops.analysis('PFEM', dtmax, dtmin, b2)

# analysis
while ops.getTime() < totaltime:

    # analysis
    if ops.analyze() < 0:
        break

    ops.remesh()

print("=========================================")
```

Dambreak with Elastic Obstacle Analysis using background mesh

1. The source code is shown below, which can be downloaded `here`.
2. Run the source code in your favorite Python program.
3. 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.

```
import os
import openseespy.opensees as ops


print("=======================================================")
print("Starting Dambreak with Obstacle Background Mesh example")

# ------------------------------
# Start of model generation
# ------------------------------

# wipe all previous objects
ops.wipe()

# create a model with fluid
ops.model('basic', '-ndm', 2, '-ndf', 3)

# geometric
L = 0.146
```

(continues on next page)

(continued from previous page)

```
H = L * 2
H2 = 0.3
b = 0.012
h = L / 40
Hb = 20.0 * b / 3.0

# number of particles per cell in each direction
numx = 3.0
numy = 3.0

# fluid properties
rho = 1000.0
mu = 0.0001
b1 = 0.0
b2 = -9.81
thk = 0.012
kappa = -1.0

# elastis structural material
rhos = 2500.0
A = thk * thk
E = 1e6
Iz = thk * thk * thk * thk / 12.0
bmass = A * Hb * rhos

# nonlinear structural material
E0 = 1e6
Fy = 5e4
hardening = 0.02

nonlinear = False

# analysis
dtmax = 1e-3
dtmin = 1e-3
totaltime = 1.0

if nonlinear:
    filename = 'obstaclenonlinear-bg'
else:
    filename = 'obstacle-bg'

# recorder
ops.recorder('BgPVD', filename, 'disp', 'vel', 'pressure', '-dT', 1e-3)
if not os.path.exists(filename):
    os.makedirs(filename)

# fluid mesh
ndf = 3

# total number of particles in each direction
nx = round(L / h * numx)
ny = round(H / h * numy)

# create particles
eleArgs = ['PFEMElementBubble', rho, mu, b1, b2, thk, kappa]
partArgs = ['quad', 0.0, 0.0, L, 0.0, L, H, 0.0, H, nx, ny]
```

(continues on next page)

(continued from previous page)

```
parttag = 1
ops.mesh('part', parttag, *partArgs, *eleArgs, '-vel', 0.0, 0.0)

# wall mesh
ops.node(1, 2 * L, 0.0)
ops.node(2, 2 * L, Hb)
ops.node(3, 0.0, H)
ops.node(4, 0.0, 0.0)
ops.node(5, 4 * L, 0.0)
ops.node(6, 4 * L, H)

sid = 1
walltag = 4
ops.mesh('line', walltag, 5, 3, 4, 1, 5, 6, sid, ndf, h)

wallNodes = ops.getNodeTags('-mesh', walltag)
for nd in wallNodes:
    ops.fix(nd, 1, 1, 1)

# structural mesh

# transformation
transfTag = 1
ops.geomTransf('Corotational', transfTag)

# section
secTag = 1
if nonlinear:
    matTag = 1
    ops.uniaxialMaterial('Steel01', matTag, Fy, E0, hardening)
    numfiber = 5
    ops.section('Fiber', secTag)
    ops.patch('rect', matTag, numfiber, numfiber, 0.0, 0.0, thk, thk)
else:
    ops.section('Elastic', secTag, E, A, Iz)

# beam integration
inteTag = 1
numpts = 2
ops.beamIntegration('Legendre', inteTag, secTag, numpts)

coltag = 3
eleArgs = ['dispBeamColumn', transfTag, inteTag]
ops.mesh('line', coltag, 2, 1, 2, sid, ndf, h, *eleArgs)

# mass
sNodes = ops.getNodeTags('-mesh', coltag)
bmass = bmass / len(sNodes)
for nd in sNodes:
    ops.mass(int(nd), bmass, bmass, 0.0)


# background mesh
lower = [-h, -h]
upper = [5 * L, 3 * L]

ops.mesh('bg', h, *lower, *upper,
```

(continues on next page)

(continued from previous page)

```
    '-structure', sid, len(sNodes), *sNodes,
    '-structure', sid, len(wallNodes), *wallNodes)

print('num nodes =', len(ops.getNodeTags()))
print('num particles =', nx * ny)

# create constraint object
ops.constraints('Plain')

# create numberer object
ops.numberer('Plain')

# create convergence test object
ops.test('PFEM', 1e-5, 1e-5, 1e-5, 1e-5, 1e-5, 1e-5, 100, 3, 1, 2)

# create algorithm object
ops.algorithm('Newton')

# create integrator object
ops.integrator('PFEM', 0.5, 0.25)

# create SOE object
ops.system('PFEM')
# system('PFEM', '-mumps') Linux version can use mumps

# create analysis object
ops.analysis('PFEM', dtmax, dtmin, b2)

# analysis
while ops.getTime() < totaltime:

    # analysis
    if ops.analyze() < 0:
        break

    ops.remesh()

print("=========================================")
```

1.14.4 GeoTechnical Examples

1. *Laterally-Loaded Pile Foundation*
2. *Effective Stress Site Response Analysis of a Layered Soil Column*

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), and shown below, which can be downloaded `here`.

```
# -*- coding: utf-8 -*-
"""
```

(continues on next page)

(continued from previous page)

```
Created on Thu Jan 10 18:24:47 2019

@author: pchi893
"""




############################################################
#                                                          #
# Procedure to compute ultimate lateral resistance, p_u,   #
#  and displacement at 50% of lateral capacity, y50, for   #
#  p-y springs representing cohesionless soil.             #
#   Converted to openseespy by: Pavan Chigullapally        #
#                               University of Auckland     #
#                                                          #
#   Created by:   Hyung-suk Shin                           #
#                 University of Washington                 #
#   Modified by:  Chris McGann                             #
#                 Pedro Arduino                            #
#                 Peter Mackenzie-Helnwein                 #
#                 University of Washington                 #
#                                                          #
############################################################

# references
#  American Petroleum Institute (API) (1987). Recommended Practice for Planning, Designing and
#   Constructing Fixed Offshore Platforms. API Recommended Practice 2A(RP-2A), Washington D.C,
#   17th edition.
#
# Brinch Hansen, J. (1961). "The ultimate resistance of rigid piles against transversal forces."
#  Bulletin No. 12, Geoteknisk Institute, Copenhagen, 59.
#
#  Boulanger, R. W., Kutter, B. L., Brandenberg, S. J., Singh, P., and Chang, D. (2003). Pile
#   Foundations in liquefied and laterally spreading ground during earthquakes: Centrifuge experiments
#   and analyses. Center for Geotechnical Modeling, University of California at Davis, Davis, CA.
#   Rep. UCD/CGM-03/01.
#
#  Reese, L.C. and Van Impe, W.F. (2001), Single Piles and Pile Groups Under Lateral Loading.
#    A.A. Balkema, Rotterdam, Netherlands.

import math

def get_pyParam ( pyDepth, gamma, phiDegree, b, pEleLength, puSwitch, kSwitch, gwtSwitch):

    #-----------------------------------------------------------
    #  define ultimate lateral resistance, pult
    #-----------------------------------------------------------

```

(continues on next page)

(continued from previous page)

```
    # pult is defined per API recommendations (Reese and Van Impe, 2001 or API, 1987) for puSwitch = 1
    #  OR per the method of Brinch Hansen (1961) for puSwitch = 2

    pi = 3.14159265358979
    phi = phiDegree * (pi/180)
    zbRatio = pyDepth / b

    #-------API recommended method-------

    if puSwitch == 1:

      # obtain loading-type coefficient A for given depth-to-diameter ratio zb
      #  ---> values are obtained from a figure and are therefore approximate
        zb = []
        dataNum = 41
        for i in range(dataNum):
            b1 = i * 0.125
            zb.append(b1)
        As = [2.8460, 2.7105, 2.6242, 2.5257, 2.4271, 2.3409, 2.2546, 2.1437, 2.0575, 1.9589, 1.8973, 1.8111, 1.7372, 1.6632, 1.5893, 1.5277, 1.4415, 1.3799, 1.3368, 1.2690, 1.2074, 1.1581,
            1.1211, 1.0780, 1.0349, 1.0164, 0.9979, 0.9733, 0.9610, 0.9487, 0.9363, 0.9117, 0.8994, 0.8994, 0.8871, 0.8871, 0.8809, 0.8809, 0.8809, 0.8809, 0.8809]

      # linear interpolation to define A for intermediate values of depth:diameter ratio
        for i in range(dataNum):
            if zbRatio >= 5.0:
                A = 0.88
            elif zb[i] <= zbRatio and zbRatio <= zb[i+1]:
                A = (As[i+1] - As[i])/(zb[i+1] - zb[i]) * (zbRatio-zb[i]) + As[i]

      # define common terms
        alpha = phi / 2
        beta = pi / 4 + phi / 2
        K0 = 0.4

        tan_1 = math.tan(pi / 4 - phi / 2)
        Ka = math.pow(tan_1 , 2)

      # terms for Equation (3.44), Reese and Van Impe (2001)
        tan_2 = math.tan(phi)
        tan_3 = math.tan(beta - phi)
        sin_1 = math.sin(beta)
        cos_1 = math.cos(alpha)
        c1 = K0 * tan_2 * sin_1 / (tan_3*cos_1)

        tan_4 = math.tan(beta)
        tan_5 = math.tan(alpha)
        c2 = (tan_4/tan_3)*tan_4 * tan_5

        c3 = K0 * tan_4 * (tan_2 * sin_1 - tan_5)

        c4 = tan_4 / tan_3 - Ka

        # terms for Equation (3.45), Reese and Van Impe (2001)
```

(continues on next page)

(continued from previous page)

```
        pow_1 = math.pow(tan_4,8)
        pow_2 = math.pow(tan_4,4)
        c5 = Ka * (pow_1-1)
        c6 = K0 * tan_2 * pow_2

      # Equation (3.44), Reese and Van Impe (2001)
        pst = gamma * pyDepth * (pyDepth * (c1 + c2 + c3) + b * c4)

      # Equation (3.45), Reese and Van Impe (2001)
        psd = b * gamma * pyDepth * (c5 + c6)

      # pult is the lesser of pst and psd. At surface, an arbitrary value is defined
        if pst <=psd:
            if pyDepth == 0:
                pu = 0.01

            else:
                pu = A * pst

        else:
            pu = A * psd

      # PySimple1 material formulated with pult as a force, not force/length,␣
↪multiply by trib. length
        pult = pu * pEleLength

    #-------Brinch Hansen method-------
    elif puSwitch == 2:
      # pressure at ground surface
        cos_2 = math.cos(phi)

        tan_6 = math.tan(pi/4+phi/2)

        sin_2 = math.sin(phi)
        sin_3 = math.sin(pi/4 + phi/2)

        exp_1 = math.exp((pi/2+phi)*tan_2)
        exp_2 = math.exp(-(pi/2-phi) * tan_2)

        Kqo = exp_1 * cos_2 * tan_6 - exp_2 * cos_2 * tan_1
        Kco = (1/tan_2) * (exp_1 * cos_2 * tan_6 - 1)

      # pressure at great depth
        exp_3 = math.exp(pi * tan_2)
        pow_3 = math.pow(tan_2,4)
        pow_4 = math.pow(tan_6,2)
        dcinf = 1.58 + 4.09 * (pow_3)
        Nc = (1/tan_2)*(exp_3)*(pow_4 - 1)
        Ko = 1 - sin_2
        Kcinf = Nc * dcinf
        Kqinf = Kcinf * Ko * tan_2

      # pressure at an arbitrary depth
        aq = (Kqo/(Kqinf - Kqo))*(Ko*sin_2/sin_3)
        KqD = (Kqo + Kqinf * aq * zbRatio)/(1 + aq * zbRatio)

      # ultimate lateral resistance
```

(continues on next page)

(continued from previous page)

```
        if pyDepth == 0:
            pu = 0.01
        else:
            pu = gamma * pyDepth * KqD * b

     # PySimple1 material formulated with pult as a force, not force/length, multiply by trib. length
        pult  = pu * pEleLength

    #---------------------------------------------------------
    #  define displacement at 50% lateral capacity, y50
    #---------------------------------------------------------

    # values of y50 depend of the coefficent of subgrade reaction, k, which can be defined in several ways.
    #  for gwtSwitch = 1, k reflects soil above the groundwater table
    #  for gwtSwitch = 2, k reflects soil below the groundwater table
    #  a linear variation of k with depth is defined for kSwitch = 1 after API (1987)
    #  a parabolic variation of k with depth is defined for kSwitch = 2 after Boulanger et al. (2003)

    # API (1987) recommended subgrade modulus for given friction angle, values obtained from figure (approximate)

    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, 40.0]

    # subgrade modulus above the water table
    if gwtSwitch == 1:
        k = [10, 23, 45, 61, 80, 100, 120, 140, 160, 182, 215, 250, 275]

    else:
        k = [10, 20, 33, 42, 50, 60, 70, 85, 95, 107, 122, 141, 155]

    dataNum = 13
    for i in range(dataNum):
        if ph[i] <= phiDegree and phiDegree <= ph[i+1]:
            khat = (k[i+1]-k[i])/(ph[i+1]-ph[i])*(phiDegree - ph[i]) + k[i]

    # change units from (lb/in^3) to (kN/m^3)
    k_SIunits = khat * 271.45

    # define parabolic distribution of k with depth if desired (i.e. lin_par switch == 2)
    sigV = pyDepth * gamma

    if sigV == 0:
         sigV = 0.01

    if kSwitch == 2:
       # Equation (5-16), Boulanger et al. (2003)
        cSigma = math.pow(50 / sigV , 0.5)
       # Equation (5-15), Boulanger et al. (2003)
        k_SIunits = cSigma * k_SIunits

    # define y50 based on pult and subgrade modulus k

```

(continues on next page)

(continued from previous page)

```
    # based on API (1987) recommendations, p-y curves are described using tanh␣
↪functions.
    #  tcl does not have the atanh function, so must define this specifically

    #  i.e.  atanh(x) = 1/2*ln((1+x)/(1-x)), |x| < 1

    # when half of full resistance has been mobilized, p(y50)/pult = 0.5
    x = 0.5
    log_1 = math.log((1+x)/(1-x))
    atanh_value = 0.5 * log_1

    # need to be careful at ground surface (don't want to divide by zero)
    if pyDepth == 0.0:
        pyDepth = 0.01

    y50 = 0.5 * (pu/ A)/(k_SIunits * pyDepth) * atanh_value
    # return pult and y50 parameters
    outResult = []
    outResult.append(pult)
    outResult.append(y50)

    return outResult

##############################################################################################
↪##############################################################################

##############################################################################################
↪##############################################################################

############################################################
#                                                          #
# Procedure to compute ultimate tip resistance, qult, and #
#  displacement at 50% mobilization of qult, z50, for     #
#  use in q-z curves for cohesionless soil.                #
#   Converted to openseespy by: Pavan Chigullapally        #
#                               University of Auckland     #
#   Created by:  Chris McGann                              #
#                Pedro Arduino                             #
#                University of Washington                  #
#                                                          #
############################################################

# references
#  Meyerhof G.G. (1976). "Bearing capacity and settlement of pile foundations."
#   J. Geotech. Eng. Div., ASCE, 102(3), 195-228.
#
#  Vijayvergiya, V.N. (1977). "Load-movement characteristics of piles."
#   Proc., Ports 77 Conf., ASCE, New York.
#
#  Kulhawy, F.H. ad Mayne, P.W. (1990). Manual on Estimating Soil Properties for
#   Foundation Design. Electrical Power Research Institute. EPRI EL-6800,
#   Project 1493-6 Final Report.

def get_qzParam (phiDegree, b, sigV, G):

    # define required constants; pi, atmospheric pressure (kPa), pa, and coeff. of␣
↪lat earth pressure, Ko
```

(continues on next page)

(continued from previous page)

```
    pi = 3.14159265358979
    pa = 101
    sin_4 = math.sin(phiDegree * (pi/180))
    Ko = 1 - sin_4

  # ultimate tip pressure can be computed by qult = Nq*sigV after Meyerhof (1976)
  #  where Nq is a bearing capacity factor, phi is friction angle, and sigV is eff. overburden
  #  stress at the pile tip.
    phi = phiDegree * (pi/180)

  # rigidity index
    tan_7 = math.tan(phi)
    Ir = G/(sigV * tan_7)
  # bearing capacity factor
    tan_8 = math.tan(pi/4+phi/2)
    sin_5 = math.sin(phi)
    pow_4 = math.pow(tan_8,2)
    pow_5 = math.pow(Ir,(4*sin_5)/(3*(1+sin_5)))
    exp_4 = math.exp(pi/2-phi)

    Nq = (1+2*Ko)*(1/(3-sin_5))*exp_4*(pow_4)*(pow_5)
  # tip resistance
    qu = Nq * sigV
  # QzSimple1 material formulated with qult as force, not stress, multiply by area of pile tip
    pow_6 = math.pow(b, 2)
    qult = qu * pi*pow_6/4

  # the q-z curve of Vijayvergiya (1977) has the form, q(z) = qult*(z/zc)^(1/3)
  #  where zc is critical tip deflection given as ranging from 3-9% of the
  #  pile diameter at the tip.

  # assume zc is 5% of pile diameter
    zc = 0.05 * b

  # based on Vijayvergiya (1977) curve, z50 = 0.125*zc
    z50 = 0.125 * zc

  # return values of qult and z50 for use in q-z material
    outResult = []
    outResult.append(qult)
    outResult.append(z50)

    return outResult

##########################################################################################
##########################################################################################
##########################################################
#                                                        #
# Procedure to compute ultimate resistance, tult, and    #
#  displacement at 50% mobilization of tult, z50, for    #
#  use in t-z curves for cohesionless soil.              #
#   Converted to openseespy by: Pavan Chigullapally      #
```

(continues on next page)

(continued from previous page)

```
#                                 University of Auckland    #
#   Created by:  Chris McGann                               #
#                University of Washington                   #
#                                                           #
#############################################################

def get_tzParam ( phi, b, sigV, pEleLength):

# references
#  Mosher, R.L. (1984). "Load transfer criteria for numerical analysis of
#   axial loaded piles in sand." U.S. Army Engineering and Waterways
#   Experimental Station, Automatic Data Processing Center, Vicksburg, Miss.
#
#  Kulhawy, F.H. (1991). "Drilled shaft foundations." Foundation engineering
#   handbook, 2nd Ed., Chap 14, H.-Y. Fang ed., Van Nostrand Reinhold, New York

    pi = 3.14159265358979

  # Compute tult based on tult = Ko*sigV*pi*dia*tan(delta), where
  #   Ko    is coeff. of lateral earth pressure at rest,
  #         taken as Ko = 0.4
  #   delta is interface friction between soil and pile,
  #         taken as delta = 0.8*phi to be representative of a
  #         smooth precast concrete pile after Kulhawy (1991)

    delta = 0.8 * phi * pi/180

  # if z = 0 (ground surface) need to specify a small non-zero value of sigV

    if sigV == 0.0:
        sigV = 0.01

    tan_9 = math.tan(delta)
    tu = 0.4 * sigV * pi * b * tan_9

  # TzSimple1 material formulated with tult as force, not stress, multiply by␣
↪tributary length of pile
    tult = tu * pEleLength

  # Mosher (1984) provides recommended initial tangents based on friction angle
        # values are in units of psf/in
    kf = [6000, 10000, 10000, 14000, 14000, 18000]
    fric = [28, 31, 32, 34, 35, 38]

    dataNum = len(fric)


        # determine kf for input value of phi, linear interpolation for intermediate␣
↪values
    if phi < fric[0]:
        k = kf[0]
    elif phi > fric[5]:
        k = kf[5]
    else:
        for i in range(dataNum):
            if fric[i] <= phi and phi <= fric[i+1]:
                k = ((kf[i+1] - kf[i])/(fric[i+1] - fric[i])) * (phi - fric[i]) +␣
↪kf[i]
```

(continues on next page)

(continued from previous page)

```
372
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 import openseespy.opensees as op
412
413 op.wipe()
414
415 ################################################################################
    ↪################################################################################
416
417 ################################################################################
    ↪################################################################################
418
419 # all the units are in SI units N and mm
420
421 #-----------------------------------------------------------
422 #  pile geometry and mesh
423 #-----------------------------------------------------------
424
```

(continues on next page)

(continued from previous page)

```
# length of pile head (above ground surface) (m)
L1 = 1.0
# length of embedded pile (below ground surface) (m)
L2 = 20.0
# pile diameter
diameter = 1.0

# number of pile elements
nElePile = 84
# pile element length
eleSize = (L1+L2)/nElePile

# number of total pile nodes
nNodePile =  1 + nElePile

#------------------------------------------------------------
#  create spring nodes
#------------------------------------------------------------
# spring nodes created with 3 dim, 3 dof
op.model('basic', '-ndm', 3, '-ndf', 3)

# counter to determine number of embedded nodes
count = 0

# create spring nodes

#1 to 85 are spring nodes

pile_nodes = dict()

for i in range(nNodePile):
    zCoord = eleSize * i
    if zCoord <= L2:
        op.node(i+1, 0.0, 0.0, zCoord)
        op.node(i+101, 0.0, 0.0, zCoord)
        pile_nodes[i+1] = (0.0, 0.0, zCoord)
        pile_nodes[i+101] = (0.0, 0.0, zCoord)
        count = count + 1

print("Finished creating all spring nodes...")

# number of embedded nodes
nNodeEmbed = count

# spring node fixities
for i in range(nNodeEmbed):
    op.fix(i+1, 1, 1, 1)
    op.fix(i+101, 0, 1, 1)

print("Finished creating all spring node fixities...")

#------------------------------------------------------------
#  soil properties
#------------------------------------------------------------

# soil unit weight (kN/m^3)
gamma = 17.0
```

(continues on next page)

(continued from previous page)

```
# soil internal friction angle (degrees)
phi = 36.0
# soil shear modulus at pile tip (kPa)
Gsoil = 150000.0

# select pult definition method for p-y curves
# API (default) --> 1
# Brinch Hansen --> 2
puSwitch = 1

# variation in coefficent of subgrade reaction with depth for p-y curves
# API linear variation (default)   --> 1
# modified API parabolic variation --> 2
kSwitch = 1

# effect of ground water on subgrade reaction modulus for p-y curves
# above gwt --> 1
# below gwt --> 2
gwtSwitch = 1

#---------------------------------------------------------
#  create spring material objects
#---------------------------------------------------------

# p-y spring material

for i in range(1 , nNodeEmbed+1):
    # depth of current py node
    pyDepth = L2 - eleSize * (i-1)
    # procedure to define pult and y50
    pyParam = get_pyParam(pyDepth, gamma, phi, diameter, eleSize, puSwitch, kSwitch,␣
↪gwtSwitch)
    pult = pyParam [0]
    y50 = pyParam [1]
    op.uniaxialMaterial('PySimple1', i, 2, pult, y50, 0.0)


# t-z spring material
for i in range(2, nNodeEmbed+1):
  # depth of current tz node
    pyDepth = eleSize * (i-1)
  # vertical effective stress at current depth
    sigV = gamma * pyDepth
  # procedure to define tult and z50
    tzParam = get_tzParam(phi, diameter, sigV, eleSize)
    tult = tzParam [0]
    z50 = tzParam [1]
    op.uniaxialMaterial('TzSimple1', i+100, 2, tult, z50, 0.0)


# q-z spring material

  # vertical effective stress at pile tip, no water table (depth is embedded pile␣
↪length)
sigVq = gamma * L2
  # procedure to define qult and z50
qzParam = get_qzParam (phi, diameter, sigVq, Gsoil)
```

(continues on next page)

(continued from previous page)

```
qult = qzParam [0]
z50q = qzParam [1]

#op.uniaxialMaterial('QzSimple1', 101, 2, qult, z50q) #, 0.0, 0.0
op.uniaxialMaterial('TzSimple1', 101, 2, qult, z50q, 0.0)

print("Finished creating all p-y, t-z, and z-z spring material objects...")


#-----------------------------------------------------------
#  create zero-length elements for springs
#-----------------------------------------------------------

# element at the pile tip (has q-z spring)
op.element('zeroLength', 1001, 1, 101, '-mat', 1, 101, '-dir', 1, 3)

# remaining elements
for i in range(2, nNodeEmbed+1):
    op.element('zeroLength', 1000+i, i, 100+i, '-mat', i, 100+i, '-dir', 1, 3)

print("Finished creating all zero-Length elements for springs...")

#-----------------------------------------------------------
#  create pile nodes
#-----------------------------------------------------------

# pile nodes created with 3 dimensions, 6 degrees of freedom
op.model('basic', '-ndm', 3, '-ndf', 6)

# create pile nodes
for i in range(1, nNodePile+1):
    zCoord = eleSize * i
    op.node(i+200, 0.0, 0.0, zCoord)

print("Finished creating all pile nodes...")

# create coordinate-transformation object
op.geomTransf('Linear', 1, 0.0, -1.0, 0.0)


# create fixity at pile head (location of loading)
op.fix(200+nNodePile, 0, 1, 0, 1, 0, 1)


# create fixities for remaining pile nodes
for i in range(201, 200+nNodePile):
    op.fix(i, 0, 1, 0, 1, 0, 1)

print("Finished creating all pile node fixities...")

#-----------------------------------------------------------
#  define equal dof between pile and spring nodes
#-----------------------------------------------------------

for i in range(1, nNodeEmbed+1):
    op.equalDOF(200+i, 100+i, 1, 3)

```

(continues on next page)

(continued from previous page)

```
print("Finished creating all equal degrees of freedom...")

#-----------------------------------------------------------
#  pile section
#-----------------------------------------------------------
##########################################################################################
↪##########################################################################################

##########################################################################################
↪##########################################################################################

#-----------------------------------------------------------
#  create elastic pile section
#-----------------------------------------------------------

secTag = 1
E = 25000000.0
A = 0.785
Iz = 0.049
Iy = 0.049
G = 9615385.0
J = 0.098

matTag = 3000
op.section('Elastic', 1, E, A, Iz, Iy, G, J)

# elastic torsional material for combined 3D section
op.uniaxialMaterial('Elastic', 3000, 1e10)

# create combined 3D section
secTag3D = 3
op.section('Aggregator', secTag3D, 3000, 'T', '-section', 1)


##########################################################################################
↪##########################################################################################

##########################################################################################
↪##########################################################################################

# elastic pile section
#import elasticPileSection

#-----------------------------------------------------------
#  create pile elements
#-----------------------------------------------------------
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)

for i in range(201, 201+nElePile):
    op.element('dispBeamColumn', i, i, i+1, 1, 1)

print("Finished creating all pile elements...")

#-----------------------------------------------------------
#  create recorders
```

(continues on next page)

(continued from previous page)

```
#-----------------------------------------------------------

# record information at specified increments
timeStep = 0.5

# record displacements at pile nodes
op.recorder('Node', '-file', 'pileDisp.out','-time', '-dT', timeStep, '-nodeRange', 201, 200 + nNodePile, '-dof', 1,2,3, 'disp')

# record reaction force in the p-y springs
op.recorder('Node', '-file', 'reaction.out','-time', '-dT', timeStep, '-nodeRange', 1, nNodePile, '-dof', 1, 'reaction')

# record element forces in pile elements
op.recorder('Element', '-file', 'pileForce.out','-time', '-dT', timeStep, '-eleRange', 201, 200+nElePile, 'globalForce')

print("Finished creating all recorders...")

#-----------------------------------------------------------
#  create the loading
#-----------------------------------------------------------

op.setTime(10.0)

# 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]

nodeTag = 200+nNodePile
loadValues = [3500.0, 0.0, 0.0, 0.0, 0.0, 0.0]
op.timeSeries('Path', 1, '-values', *values, '-time', *time, '-factor', 1.0)

op.pattern('Plain', 10, 1)
op.load(nodeTag, *loadValues)

print("Finished creating loading object...")

#-----------------------------------------------------------
#  create the analysis
#-----------------------------------------------------------
op.integrator('LoadControl', 0.05)
op.numberer('RCM')
op.system('SparseGeneral')
op.constraints('Transformation')
op.test('NormDispIncr', 1e-5, 20, 1)
op.algorithm('Newton')
op.analysis('Static')

print("Starting Load Application...")
op.analyze(201)

print("Load Application finished...")
#print("Loading Analysis execution time: [expr $endT-$startT] seconds.")

#op.wipe

```

(continues on next page)

(continued from previous page)

```
op.reactions()
Nodereactions = dict()
Nodedisplacements = dict()
for i in range(201,nodeTag+1):
    Nodereactions[i] = op.nodeReaction(i)
    Nodedisplacements[i] = op.nodeDisp(i)
print('Node Reactions are: ', Nodereactions)
print('Node Displacements are: ', Nodedisplacements)
```

Effective Stress Site Response Analysis of a Layered Soil Column

1. The original model can be found here.
2. The Python code is converted by **Harsh Mistry from The University of Manchester, UK** (harsh.mistry@manchester.ac.uk), and shown below, which can be downloaded `here`.
3. The Input motion (velocity-time) can be downloaded `here`.

```
# -*- coding: utf-8 -*-
"""
Created on Fri Jan 29 16:39:41 2021

@author: harsh
"""
import numpy as np
import math as mm
import opensees as op
import time as tt
############################################################
#                                                          #
# Effective stress site response analysis for a layered    #
# soil profile located on a 2% slope and underlain by an   #
# elastic half-space.  9-node quadUP elements are used.    #
# The finite rigidity of the elastic half space is         #
# considered through the use of a viscous damper at the    #
# base.                                                    #
#                                                          #
#    Converted to openseespy by: Harsh Mistry              #
#                                The University of Manchester  #
#                                                          #
#   Created by:  Chris McGann                              #
#                HyungSuk Shin                             #
#                Pedro Arduino                             #
#                Peter Mackenzie-Helnwein                  #
#              --University of Washington--                #
#                                                          #
# ---> Basic units are kN and m   (unless specified)       #
#                                                          #
############################################################
#-------------------------------------------------------------------------------
↪----
```

(continues on next page)

(continued from previous page)

```
#  1. DEFINE SOIL AND MESH GEOMETRY
#-----------------------------------------------------------------------------------------

op.wipe()
nodes_dict = dict()

#---SOIL GEOMETRY
# thicknesses of soil profile (m)
soilThick = 30.0
# number of soil layers
numLayers = 3
# layer thicknesses
layerThick=[20.0,8.0,2.0]

# depth of water table
waterTable = 2.0

# define layer boundaries
layerBound=np.zeros((numLayers,1))
layerBound[0]=layerThick[0];
for i in range(1,numLayers):
    layerBound[i]=layerBound[i-1]+layerThick[i]

#---MESH GEOMETRY
# number of elements in horizontal direction
nElemX = 1
# number of nodes in horizontal direction
nNodeX =2 * nElemX+1
# horizontal element size (m)
sElemX = 2.0

# number of elements in vertical direction for each layer
nElemY = [40,16,4]

# total number of elements in vertical direction
nElemT = 60

sElemY = np.zeros((numLayers,1))
# vertical element size in each layer
for i in range(numLayers):
    sElemY[i] = [layerThick[i-1]/nElemY[i-1]]
    print('size:',sElemY[i])

# number of nodes in vertical direction
nNodeY = 2 * nElemT+1

# total number of nodes
nNodeT = nNodeX * nNodeY

#-----------------------------------------------------------------------------------------
#  2. CREATE PORE PRESSURE NODES AND FIXITIES
#-----------------------------------------------------------------------------------------
op.model('basic', '-ndm', 2, '-ndf', 3)

```

(continues on next page)

(continued from previous page)

```
count = 1
layerNodeCount = 0
dry_Node=np.zeros((500,1))
node_save=np.zeros((500,1))
# loop over soil layers
for k in range(1,numLayers+1):
    # loop in horizontal direction
    for i in range(1,nNodeX+1,2):
       if k==1:
           bump = 1
       else:
           bump = 0
       j_end=2 * nElemY[k-1] + bump
       for j in range(1,j_end+1,2):
            xCoord  = (i-1) * (sElemX/2)
            yctr    = j + layerNodeCount
            yCoord  = (yctr-1) * (np.float(sElemY[k-1]))/2
            nodeNum = i + ((yctr-1) * nNodeX)
            op.node(nodeNum, xCoord, yCoord)

          # output nodal information to data file
            nodes_dict[nodeNum] = (nodeNum, xCoord, yCoord)
            node_save[nodeNum] = np.int(nodeNum)
          # designate nodes above water table
            waterHeight = soilThick - waterTable
            if yCoord >= waterHeight:
                dry_Node[count] = nodeNum
                count = count+1
    layerNodeCount = yctr + 1

dryNode=np.trim_zeros(dry_Node)
Node_d=np.unique(node_save)
Node_d=np.trim_zeros(Node_d)
np.savetxt('Node_record.txt',Node_d)
print('Finished creating all -ndf 3 nodes')
print('Number of Dry Nodes:',len(dryNode))

# define fixities for pore pressure nodes above water table
for i in range(count-1):
    n_dryNode=np.int(dryNode[i])
    op.fix(n_dryNode, 0, 0, 1)

op.fix(1, 0, 1, 0)
op.fix(3, 0, 1, 0)
print('Finished creating all -ndf 3 boundary conditions...')

# define equal degrees of freedom for pore pressure nodes
for i in range(1,((3*nNodeY)-2),6):
    op.equalDOF(i, i+2, 1, 2)

print("Finished creating equalDOF for pore pressure nodes...")

#-------------------------------------------------------------------------------------
#  3. CREATE INTERIOR NODES AND FIXITIES
#-------------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
op.model('basic', '-ndm', 2, '-ndf', 2)

xCoord = np.float(sElemX/2)

# loop over soil layers
layerNodeCount = 0

for k in range(1,numLayers+1):
    if k==1:
        bump = 1
    else:
        bump = 0
    j_end=2 * nElemY[k-1] + bump
    for j in range(1,j_end+1,1):
        yctr = j + layerNodeCount
        yCoord  = (yctr-1) * (np.float(sElemY[k-1]))/2
        nodeNum = (3*yctr) - 1
        op.node(nodeNum, xCoord, yCoord)
        # output nodal information to data file
        nodes_dict[nodeNum] = (nodeNum, xCoord, yCoord)

    layerNodeCount = yctr

# interior nodes on the element edges
# loop over layers
layerNodeCount = 0

for k in range(1,numLayers+1):
    # loop in vertical direction
    for j in range(1,((nElemY[k-1])+1)):
        yctr     = j + layerNodeCount;
        yCoord   = np.float(sElemY[k-1])*(yctr-0.5)
        nodeNumL = (6*yctr) - 2
        nodeNumR = nodeNumL + 2

        op.node(nodeNumL ,0.0, yCoord)
        op.node(nodeNumR , sElemX, yCoord)

        # output nodal information to data file
        nodes_dict[nodeNumL] = (nodeNumL ,0.0, yCoord)
        nodes_dict[nodeNumR] = (nodeNumR , sElemX, yCoord)
    layerNodeCount = yctr

print("Finished creating all -ndf 2 nodes...")

# define fixities for interior nodes at base of soil column
op.fix(2, 0, 1)
print('Finished creating all -ndf 2 boundary conditions...')

# define equal degrees of freedom which have not yet been defined
for i in range(1,((3*nNodeY)-6),6):
    op.equalDOF(i  , i+1, 1, 2)
    op.equalDOF(i+3, i+4, 1, 2)
    op.equalDOF(i+3, i+5, 1, 2)

op.equalDOF(nNodeT-2, nNodeT-1, 1, 2)
print('Finished creating equalDOF constraints...')
```

(continues on next page)

(continued from previous page)

```
199
200 #------------------------------------------------------------------------------------
    ↪----
201 #  4. CREATE SOIL MATERIALS
202 #------------------------------------------------------------------------------------
    ↪----
203
204 # define grade of slope (%)
205 grade = 2.0
206 slope = mm.atan(grade/100.0)
207 g     = -9.81
208
209 xwgt_var = g * (mm.sin(slope))
210 ywgt_var = g * (mm.cos(slope))
211 thick = [1.0,1.0,1.0]
212 xWgt  = [xwgt_var, xwgt_var, xwgt_var]
213 yWgt  = [ywgt_var, ywgt_var, ywgt_var]
214 uBulk = [6.88E6,  5.06E6, 5.0E-6]
215 hPerm = [1.0E-4, 1.0E-4, 1.0E-4]
216 vPerm = [1.0E-4, 1.0E-4, 1.0E-4]
217
218
219 # nDMaterial PressureDependMultiYield02
220 # nDMaterial('PressureDependMultiYield02', matTag, nd, rho, refShearModul,␣
    ↪refBulkModul,\
221 #    frictionAng, peakShearStra, refPress, pressDependCoe, PTAng,\
222 #        contrac[0], contrac[2], dilat[0], dilat[2], noYieldSurf=20.0,\
223 #            *yieldSurf=[], contrac[1]=5.0, dilat[1]=3.0, *liquefac=[1.0,0.0],e=0.6, \
224 #                *params=[0.9, 0.02, 0.7, 101.0], c=0.1)
225
226 op.nDMaterial('PressureDependMultiYield02',3, 2, 1.8, 9.0e4, 2.2e5, 32, 0.1, \
227                                       101.0, 0.5, 26, 0.067, 0.23, 0.06, \
228                                       0.27, 20, 5.0, 3.0, 1.0, \
229                                       0.0, 0.77, 0.9, 0.02, 0.7, 101.0)
230
231 op.nDMaterial('PressureDependMultiYield02', 2, 2, 2.24, 9.0e4, 2.2e5, 32, 0.1, \
232                                       101.0, 0.5, 26, 0.067, 0.23, 0.06, \
233                                       0.27, 20, 5.0, 3.0, 1.0, \
234                                       0.0, 0.77, 0.9, 0.02, 0.7, 101.0)
235
236 op.nDMaterial('PressureDependMultiYield02',1, 2, 2.45, 1.3e5, 2.6e5, 39, 0.1, \
237                                       101.0, 0.5, 26, 0.010, 0.0, 0.35, \
238                                       0.0, 20, 5.0, 3.0, 1.0, \
239                                       0.0, 0.47, 0.9, 0.02, 0.7, 101.0)
240
241 print("Finished creating all soil materials...")
242
243 #------------------------------------------------------------------------------------
    ↪----
244 #  5. CREATE SOIL ELEMENTS
245 #------------------------------------------------------------------------------------
    ↪----
246
247 for j in range(1,nElemT+1):
248     nI = ( 6*j) - 5
249     nJ = nI + 2
250     nK = nI + 8
```

(continues on next page)

(continued from previous page)

```
    nL = nI + 6
    nM = nI + 1
    nN = nI + 5
    nP = nI + 7
    nQ = nI + 3
    nR = nI + 4

    lowerBound = 0.0
    for i in range(1,numLayers+1):
        if j * sElemY[i-1] <= layerBound[i-1] and j * sElemY[i-1] > lowerBound:
            # permeabilities are initially set at 1.0 m/s for gravity analysis,
            op.element('9_4_QuadUP', j, nI, nJ, nK, nL, nM, nN, nP, nQ, nR, \
                        thick[i-1], i, uBulk[i-1], 1.0, 1.0, 1.0, xWgt[i-1],␣
↪yWgt[i-1])

        lowerBound = layerBound[i-1]

print("Finished creating all soil elements...")
#-------------------------------------------------------------------------------
↪----
#  6. LYSMER DASHPOT
#-------------------------------------------------------------------------------
↪----

# define dashpot nodes
dashF =  nNodeT+1
dashS =  nNodeT+2

op.node(dashF,  0.0, 0.0)
op.node(dashS,  0.0, 0.0)

# define fixities for dashpot nodes
op.fix(dashF, 1, 1)
op.fix(dashS, 0, 1)

# define equal DOF for dashpot and base soil node
op.equalDOF(1, dashS,  1)
print('Finished creating dashpot nodes and boundary conditions...')

# define dashpot material
colArea      = sElemX * thick[0]
rockVS       = 700.0
rockDen      = 2.5
dashpotCoeff = rockVS * rockDen

#uniaxialMaterial('Viscous', matTag, C, alpha)
op.uniaxialMaterial('Viscous', numLayers+1, dashpotCoeff * colArea, 1)

# define dashpot element
op.element('zeroLength', nElemT+1, dashF, dashS, '-mat', numLayers+1, '-dir', 1)

print("Finished creating dashpot material and element...")

#-------------------------------------------------------------------------------
↪----
#  7. CREATE GRAVITY RECORDERS
#-------------------------------------------------------------------------------
↪----
```

(continues on next page)

(continued from previous page)

```
304
305 # create list for pore pressure nodes
306 load_nodeList3=np.loadtxt('Node_record.txt')
307 nodeList3=[]
308
309 for i in range(len(load_nodeList3)):
310     nodeList3.append(np.int(load_nodeList3[i]))
311 # record nodal displacment, acceleration, and porepressure
312 op.recorder('Node','-file','Gdisplacement.txt','-time','-node',*nodeList3,'-dof', 1,␣
    ↪2, 'disp')
313 op.recorder('Node','-file','Gacceleration.txt','-time','-node',*nodeList3,'-dof', 1,␣
    ↪2, 'accel')
314 op.recorder('Node','-file','GporePressure.txt','-time','-node',*nodeList3,'-dof', 3,
    ↪'vel')
315
316 # record elemental stress and strain (files are names to reflect GiD gp numbering)
317 op.recorder('Element','-file','Gstress1.txt','-time','-eleRange', 1,nElemT,'material',
    ↪'1','stress')
318 op.recorder('Element','-file','Gstress2.txt','-time','-eleRange', 1,nElemT,'material',
    ↪'2','stress')
319 op.recorder('Element','-file','Gstress3.txt','-time','-eleRange', 1,nElemT,'material',
    ↪'3','stress')
320 op.recorder('Element','-file','Gstress4.txt','-time','-eleRange', 1,nElemT,'material',
    ↪'4','stress')
321 op.recorder('Element','-file','Gstress9.txt','-time','-eleRange', 1,nElemT,'material',
    ↪'9','stress')
322 op.recorder('Element','-file','Gstrain1.txt','-time','-eleRange', 1,nElemT,'material',
    ↪'1','strain')
323 op.recorder('Element','-file','Gstrain2.txt','-time','-eleRange', 1,nElemT,'material',
    ↪'2','strain')
324 op.recorder('Element','-file','Gstrain3.txt','-time','-eleRange', 1,nElemT,'material',
    ↪'3','strain')
325 op.recorder('Element','-file','Gstrain4.txt','-time','-eleRange', 1,nElemT,'material',
    ↪'4','strain')
326 op.recorder('Element','-file','Gstrain9.txt','-time','-eleRange', 1,nElemT,'material',
    ↪'9','strain')
327
328 print("Finished creating gravity recorders...")
329
330 #------------------------------------------------------------------------------------
    ↪----
331 #  8. DEFINE ANALYSIS PARAMETERS
332 #------------------------------------------------------------------------------------
    ↪----
333
334 #---GROUND MOTION PARAMETERS
335 # time step in ground motion record
336 motionDT = 0.005
337 # number of steps in ground motion record
338 motionSteps = 7990
339
340 #---RAYLEIGH DAMPING PARAMETERS
341 # damping ratio
342 damp = 0.02
343 # lower frequency
344 omega1 = 2 * np.pi * 0.2
345 # upper frequency
```

(continues on next page)

(continued from previous page)

```
omega2 = 2 * np.pi * 20
# damping coefficients
a0 = 2*damp*omega1*omega2/(omega1 + omega2)
a1 = 2*damp/(omega1 + omega2)
print("Damping Coefficients: a_0 = $a0;  a_1 = $a1")

#---DETERMINE STABLE ANALYSIS TIME STEP USING CFL CONDITION
# maximum shear wave velocity (m/s)
vsMax = 250.0
# duration of ground motion (s)
duration = motionDT*motionSteps
# minimum element size
minSize = sElemY[0]

for i in range(2,numLayers+1):
    if sElemY[i-1] <= minSize:
        minSize = sElemY[i-1]

# trial analysis time step
kTrial = minSize/(vsMax**0.5)
# define time step and number of steps for analysis
if motionDT <= kTrial:
    nSteps = motionSteps
    dT     = motionDT
else:
    nSteps = np.int(mm.floor(duration/kTrial)+1)
    dT     = duration/nSteps


print("Number of steps in analysis: $nSteps")
print("Analysis time step: $dT")

#---ANALYSIS PARAMETERS
# Newmark parameters
gamma = 0.5
beta  = 0.25

#-------------------------------------------------------------------------------------------
#  9. GRAVITY ANALYSIS
#-------------------------------------------------------------------------------------------
# update materials to ensure elastic behavior
op.updateMaterialStage('-material', 1, '-stage', 0)
op.updateMaterialStage('-material', 2, '-stage', 0)
op.updateMaterialStage('-material', 3, '-stage', 0)

op.constraints('Penalty', 1.0E14, 1.0E14)
op.test('NormDispIncr', 1e-4, 35, 1)
op.algorithm('KrylovNewton')
op.numberer('RCM')
op.system('ProfileSPD')
op.integrator('Newmark', gamma, beta)
op.analysis('Transient')

startT = tt.time()
op.analyze(10, 5.0E2)
```

(continues on next page)

(continued from previous page)

```
print('Finished with elastic gravity analysis...')

# update material to consider elastoplastic behavior
op.updateMaterialStage('-material', 1, '-stage', 1)
op.updateMaterialStage('-material', 2, '-stage', 1)
op.updateMaterialStage('-material', 3, '-stage', 1)

# plastic gravity loading
op.analyze(40, 5.0e2)

print('Finished with plastic gravity analysis...')

#------------------------------------------------------------------------------------------
#  10. UPDATE ELEMENT PERMEABILITY VALUES FOR POST-GRAVITY ANALYSIS
#------------------------------------------------------------------------------------------

# choose base number for parameter IDs which is higer than other tags used in analysis
ctr = 10000.0
# loop over elements to define parameter IDs
for i in range(1,nElemT+1):
    op.parameter(np.int(ctr+1.0), 'element', i, 'vPerm')
    op.parameter(np.int(ctr+2.0), 'element', i, 'hPerm')
    ctr   = ctr+2.0

# update permeability parameters for each element using parameter IDs
ctr = 10000.0
for j in range(1,nElemT+1):
    lowerBound = 0.0
    for i in range(1,numLayers+1):
        if j * sElemY[i-1] <= layerBound[i-1] and j*sElemY[i-1] > lowerBound:
            op.updateParameter(np.int(ctr+1.0), vPerm[i-1])
            op.updateParameter(np.int(ctr+2.0), hPerm[i-1])
        lowerBound = layerBound[i-1]
    ctr = ctr+2.0

print("Finished updating permeabilities for dynamic analysis...")

#------------------------------------------------------------------------------------------
#  11. CREATE POST-GRAVITY RECORDERS
#------------------------------------------------------------------------------------------

# reset time and analysis
op.setTime(0.0)
op.wipeAnalysis()
op.remove('recorders')

# recorder time step
recDT = 10*motionDT

# record nodal displacment, acceleration, and porepressure
op.recorder('Node','-file','displacement.txt','-time', '-dT',recDT,'-node',*nodeList3,
'-dof', 1, 2, 'disp')
op.recorder('Node','-file','acceleration.txt','-time', '-dT',recDT,'-node',*nodeList3,
'-dof', 1, 2, 'accel')
```

(continues on next page)

(continued from previous page)

```
op.recorder('Node','-file','porePressure.txt','-time', '-dT',recDT,'-node',*nodeList3,'-dof', 3, 'vel')

# record elemental stress and strain (files are names to reflect GiD gp numbering)
op.recorder('Element','-file','stress1.txt','-time', '-dT',recDT,'-eleRange', 1,nElemT,'material','1','stress')
op.recorder('Element','-file','stress2.txt','-time', '-dT',recDT,'-eleRange', 1,nElemT,'material','2','stress')
op.recorder('Element','-file','stress3.txt','-time', '-dT',recDT,'-eleRange', 1,nElemT,'material','3','stress')
op.recorder('Element','-file','stress4.txt','-time', '-dT',recDT,'-eleRange', 1,nElemT,'material','4','stress')
op.recorder('Element','-file','stress9.txt','-time', '-dT',recDT,'-eleRange', 1,nElemT,'material','9','stress')
op.recorder('Element','-file','strain1.txt','-time', '-dT',recDT,'-eleRange', 1,nElemT,'material','1','strain')
op.recorder('Element','-file','strain2.txt','-time', '-dT',recDT,'-eleRange', 1,nElemT,'material','2','strain')
op.recorder('Element','-file','strain3.txt','-time', '-dT',recDT,'-eleRange', 1,nElemT,'material','3','strain')
op.recorder('Element','-file','strain4.txt','-time', '-dT',recDT,'-eleRange', 1,nElemT,'material','4','strain')
op.recorder('Element','-file','strain9.txt','-time', '-dT',recDT,'-eleRange', 1,nElemT,'material','9','strain')

print("Finished creating all recorders...")

#-------------------------------------------------------------------------------------------
#  12. DYNAMIC ANALYSIS
#-------------------------------------------------------------------------------------------
op.model('basic', '-ndm', 2, '-ndf', 3)

# define constant scaling factor for applied velocity
cFactor = colArea * dashpotCoeff

# define velocity time history file
velocityFile='velocityHistory';
data_gm=np.loadtxt('velocityHistory.txt')
#motionSteps=len(data_gm)
#print('Number of point for GM:',motionSteps)

# timeseries object for force history
op.timeSeries('Path', 2, '-dt', motionDT, '-filePath', velocityFile+'.txt', '-factor', cFactor)
op.pattern('Plain', 10, 2)
op.load(1, 1.0, 0.0, 0.0)

print( "Dynamic loading created...")

op.constraints('Penalty', 1.0E16, 1.0E16)
op.test('NormDispIncr', 1e-3, 35, 1)
op.algorithm('KrylovNewton')
op.numberer('RCM')
op.system('ProfileSPD')
op.integrator('Newmark', gamma, beta)
```

(continues on next page)

(continued from previous page)

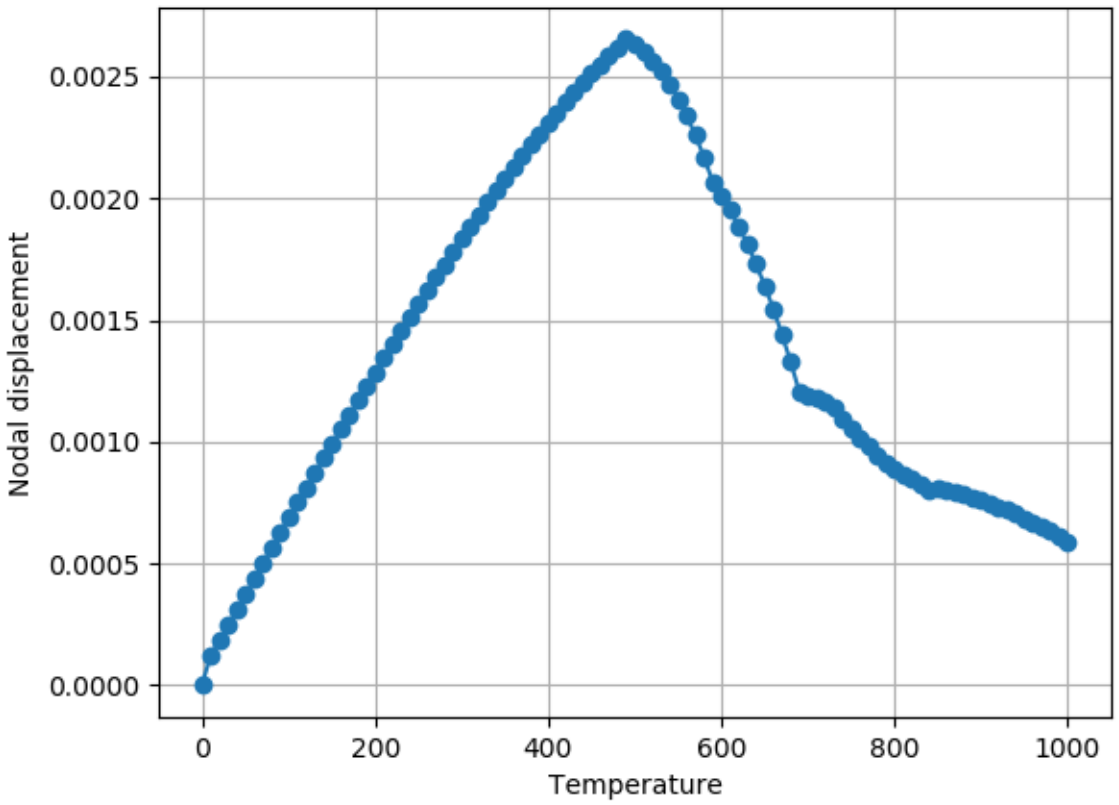
```
op.rayleigh(a0, a1, 0.0, 0.0)
op.analysis('Transient')

# perform analysis with timestep reduction loop
ok = op.analyze(nSteps,dT)

# if analysis fails, reduce timestep and continue with analysis
if ok !=0:
    print("did not converge, reducing time step")
    curTime = op.getTime()
    mTime = curTime
    print("curTime: ", curTime)
    curStep = curTime/dT
    print("curStep: ", curStep)
    rStep   = (nSteps-curStep)*2.0
    remStep = np.int((nSteps-curStep)*2.0)
    print("remStep: ", remStep)
    dT = dT/2.0
    print("dT: ", dT)

    ok = op.analyze(remStep, dT)
    # if analysis fails again, reduce timestep and continue with analysis
    if ok !=0:
        print("did not converge, reducing time step")
        curTime = op.getTime()
        print("curTime: ", curTime)
        curStep = (curTime-mTime)/dT
        print("curStep: ", curStep)
        remStep = np.int((rStep-curStep)*2.0)
        print("remStep: ", remStep)
        dT = dT/2.0
        print("dT: ", dT)

        ok = op.analyze(remStep, dT)

endT = tt.time()
print("Finished with dynamic analysis...")
print("Analysis execution time: ",(endT-startT))
op.wipe()
```

1.14.5 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 favorate Python program and should see




```
from openseespy.opensees import *

import numpy as np
import matplotlib.pyplot as plt

# define model
model('basic', '-ndm', 2, '-ndf', 3)

#define node
node(1, 0.0, 0.0)
node(2, 2.0, 0.0)
node(3, 1.0, 0.0)

#define boundary condition
fix(1, 1, 1, 1)
fix(2, 1, 1, 1)
fix(3, 0, 1, 1)

#define an elastic material with Tag=1 and E=2e11.
matTag = 1
uniaxialMaterial('Steel01Thermal', 1, 2e11, 2e11, 0.01)

#define fibred section Two fibres: fiber $yLoc $zLoc $A $matTag
secTag = 1
section('FiberThermal',secTag)
```

(continues on next page)

(continued from previous page)

```
fiber(-0.025, 0.0, 0.005, matTag)
fiber(0.025, 0.0, 0.005, matTag)

#define coordinate transforamtion
#three transformation types can be chosen: Linear, PDelta, Corotational)
transfTag = 1
geomTransf('Linear', transfTag)

# beam integration
np = 3
biTag = 1
beamIntegration('Lobatto',biTag, secTag, np)

#define beam element
element('dispBeamColumnThermal', 1, 1, 3, transfTag, biTag)
element('dispBeamColumnThermal', 2, 3, 2, transfTag, biTag)

# define time series
tsTag = 1
timeSeries('Linear',tsTag)

# define load pattern
patternTag = 1
maxtemp = 1000.0
pattern('Plain', patternTag, tsTag)
eleLoad('-ele', 1, '-type', '-beamThermal', 1000.0, -0.05, 1000.0, 0.05)
#eleLoad -ele 2 -type -beamThermal 0 -0.05 0 0.05

# define analysis
incrtemp = 0.01
system('BandGeneral')
constraints('Plain')
numberer('Plain')
test('NormDispIncr', 1.0e-3,  100, 1)
algorithm('Newton')
integrator('LoadControl', incrtemp)
analysis('Static')

# analysis
nstep = 100
temp = [0.0]
disp = [0.0]
for i in range(nstep):
    if analyze(1) < 0:
        break

    temp.append(getLoadFactor(patternTag)*maxtemp)
    disp.append(nodeDisp(3,1))


plt.plot(temp,disp,'-o')
plt.xlabel('Temperature')
plt.ylabel('Nodal displacement')
plt.grid()
plt.show()
```

1.14.6 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

```
import openseespy.opensees as ops

pid = ops.getPID()
np = ops.getNP()

print('Hello World Process:', pid)
if pid == 0:
    print('Total number of processes:', np)

```

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

```
import openseespy.opensees as ops

pid = ops.getPID()
np = ops.getNP()

# datatype = 'float'
# datatype = 'int'
datatype = 'str'

if pid == 0:
    print('Random: ')

    for i in range(1, np):
        data = ops.recv('-pid', 'ANY')
        print(data)
else:
    if datatype == 'str':
        ops.send('-pid', 0, 'Hello from {}'.format(pid))
    elif datatype == 'float':
        ops.send('-pid', 0, float(pid))
    elif datatype == 'int':
        ops.send('-pid', 0, int(pid))

ops.barrier()

if pid == 0:
    print('\nOrdered: ')

    for i in range(1, np):
        data = ops.recv('-pid', i)
        print(data)
else:
    if datatype == 'str':
        ops.send('-pid', 0, 'Hello from {}'.format(pid))
    elif datatype == 'float':
        ops.send('-pid', 0, float(pid))
    elif datatype == 'int':
        ops.send('-pid', 0, int(pid))

```

(continues on next page)

(continued from previous page)

```
ops.barrier()
if pid == 0:
    print('\nBroadcasting: ')
    if datatype == 'str':
        ops.Bcast('Hello from {}'.format(pid))
    elif datatype == 'float':
        ops.Bcast(float(pid))
    elif datatype == 'int':
        ops.Bcast(int(pid))
else:
    data = ops.Bcast()
    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

```
import openseespy.opensees as ops

pid = ops.getPID()
np = ops.getNP()
ops.start()
if np != 2:
    exit()

ops.model('basic', '-ndm', 2, '-ndf', 2)
ops.uniaxialMaterial('Elastic', 1, 3000.0)

if pid == 0:
    ops.node(1, 0.0, 0.0)
    ops.node(4, 72.0, 96.0)

    ops.fix(1, 1, 1)

    ops.element('Truss', 1, 1, 4, 10.0, 1)
    ops.timeSeries('Linear', 1)
```

(continues on next page)

(continued from previous page)

```
    ops.pattern('Plain', 1, 1)
    ops.load(4, 100.0, -50.0)

else:
    ops.node(2, 144.0, 0.0)
    ops.node(3, 168.0, 0.0)
    ops.node(4, 72.0, 96.0)

    ops.fix(2, 1, 1)
    ops.fix(3, 1, 1)

    ops.element('Truss', 2, 2, 4, 5.0, 1)
    ops.element('Truss', 3, 3, 4, 5.0, 1)

ops.constraints('Transformation')
ops.numberer('ParallelPlain')
ops.system('Mumps')
ops.test('NormDispIncr', 1e-6, 6, 2)
ops.algorithm('Newton')
ops.integrator('LoadControl', 0.1)
ops.analysis('Static')

ops.analyze(10)

print('Node 4: ', [ops.nodeCoord(4), ops.nodeDisp(4)])

ops.loadConst('-time', 0.0)

if pid == 0:
    ops.pattern('Plain', 2, 1)
    ops.load(4, 1.0, 0.0)

ops.domainChange()
ops.integrator('ParallelDisplacementControl', 4, 1, 0.1)
ops.analyze(10)

print('Node 4: ', [ops.nodeCoord(4), ops.nodeDisp(4)])
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

```
import openseespy.opensees as ops

pid = ops.getPID()
np = ops.getNP()
ops.start()

ops.model('basic', '-ndm', 2, '-ndf', 2)

L = 48.0
H = 4.0

Lp = L / np
ndf = 2
meshsize = 0.05

ops.node(pid, Lp * pid, 0.0)
ops.node(pid + 1, Lp * (pid + 1), 0.0)
ops.node(np + pid + 2, Lp * (pid + 1), H)
ops.node(np + pid + 1, Lp * pid, H)

sid = 1
ops.setStartNodeTag(2 * np + 2 + pid * int(H / meshsize + 10))
ops.mesh('line', 3, 2, pid, np + pid + 1, sid, ndf, meshsize)
ops.setStartNodeTag(2 * np + 2 + (pid + 1) * int(H / meshsize + 10))
ops.mesh('line', 4, 2, pid + 1, np + pid + 2, sid, ndf, meshsize)

ops.setStartNodeTag(int(2 * L / meshsize + (np + 1) * H / meshsize * 2) +
                    pid * int(H * L / meshsize ** 2 * 2))
ops.mesh('line', 1, 2, pid, pid + 1, sid, ndf, meshsize)
ops.mesh('line', 2, 2, np + pid + 1, np + pid + 2, sid, ndf, meshsize)

ops.nDMaterial('ElasticIsotropic', 1, 3000.0, 0.3)

eleArgs = ['tri31', 1.0, 'PlaneStress', 1]

ops.mesh('quad', 5, 4, 1, 4, 2, 3, sid, ndf, meshsize, *eleArgs)


```

(continues on next page)

(continued from previous page)

```
if pid == 0:
    ops.fix(pid, 1, 1)
    ops.fix(np+pid+1, 1, 1)
if pid == np-1:
    ops.timeSeries('Linear', 1)
    ops.pattern('Plain', 1, 1)
    ops.load(np + pid + 2, 0.0, -1.0)


ops.constraints('Transformation')
ops.numberer('ParallelPlain')
ops.system('Mumps')
ops.test('NormDispIncr', 1e-6, 6)
ops.algorithm('Newton')
ops.integrator('LoadControl', 1.0)
ops.analysis('Static')

ops.stop()
ops.start()
ops.analyze(1)

if pid == np-1:
    print('Node', pid+1, ops.nodeDisp(pid+1))


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

```
import openseespy.opensees as ops

pid = ops.getPID()
np = ops.getNP()
ops.start()
if np != 2:
```

(continues on next page)

(continued from previous page)

```
    exit()

ops.model('basic', '-ndm', 2, '-ndf', 2)

if pid == 0:
    E = 3000.0
else:
    E = 6000.0

ops.uniaxialMaterial('Elastic', 1, E)

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)

ops.fix(1, 1, 1)
ops.fix(2, 1, 1)
ops.fix(3, 1, 1)

ops.element('Truss', 1, 1, 4, 10.0, 1)
ops.timeSeries('Linear', 1)
ops.pattern('Plain', 1, 1)
ops.load(4, 100.0, -50.0)

ops.element('Truss', 2, 2, 4, 5.0, 1)
ops.element('Truss', 3, 3, 4, 5.0, 1)

ops.constraints('Transformation')
ops.numberer('ParallelPlain')
ops.system('Umfpack')
ops.test('NormDispIncr', 1e-6, 6)
ops.algorithm('Newton')
ops.integrator('LoadControl', 0.1)
ops.analysis('Static')

ops.analyze(10)


if pid == 0:
    print('Processor 0')
    print('Node 4 (E =', E, ') Disp :', ops.nodeDisp(4))

ops.barrier()

if pid == 1:
    print('Processor 1')
    print('Node 4 (E =', E, ') Disp :', ops.nodeDisp(4))


ops.stop()
```

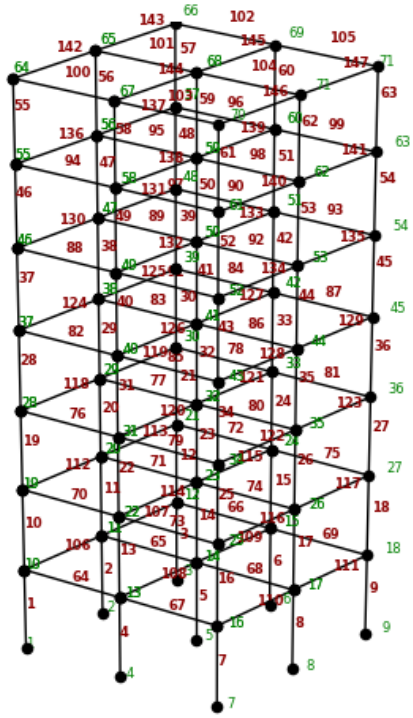
1.14.7 Plotting Examples

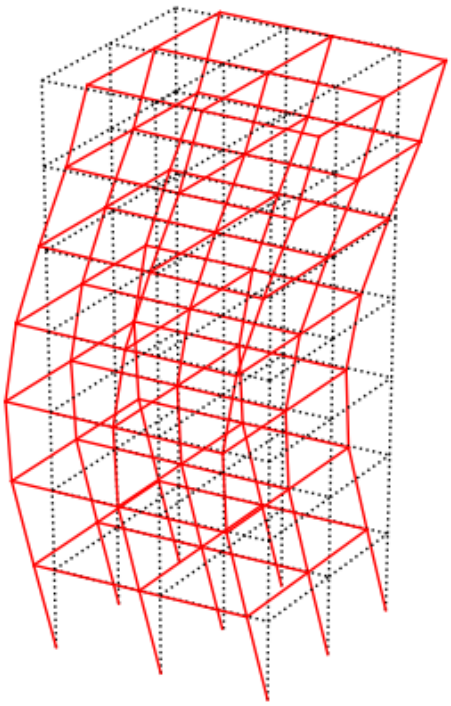
1. *A Procedure to Render 2D or 3D OpenSees Model and Mode Shapes*

2. *Statics of a 2d Portal Frame (ops_vis)*
3. *Statics of a 3d 3-element cantilever beam (ops_vis)*
4. *Plot stress distribution of plane stress quad model (ops_vis)*
5. *Plot steel and reinforced concrete fiber sections (ops_vis)*
6. *Animation of dynamic analysis and mode shapes of a 2d Portal Frame (ops_vis)*

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  ######################################################################
3  ## 3D frame example to show how to render opensees model and
4  ## plot mode shapes
5  ##
6  ## By - Anurag Upadhyay, PhD Candidate, University of Utah.
7  ## Updated - 09/10/2020
8  ######################################################################
9
10 import openseespy.postprocessing.Get_Rendering as opsplt
11 import openseespy.opensees as ops
12
13 import numpy as np
14
15 from math import asin, sqrt
16
17 # set some properties
18 ops.wipe()
19
20 ops.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)

```
numBayY = 2
numFloor = 7

bayWidthX = 120.0
bayWidthY = 120.0
storyHeights = [162.0, 162.0, 156.0, 156.0, 156.0, 156.0, 156.0, 156.0, 156.0, 156.0, 156.0]

E = 29500.0
massX = 0.49
M = 0.
coordTransf = "Linear"  # Linear, PDelta, Corotational
massType = "-lMass"  # -lMass, -cMass

nodeTag = 1

# add the nodes
#  - floor at a time
zLoc = 0.
for k in range(0, numFloor + 1):
        xLoc = 0.
        for i in range(0, numBayX + 1):
                yLoc = 0.
                for j in range(0, numBayY + 1):
                        ops.node(nodeTag, xLoc, yLoc, zLoc)
                        ops.mass(nodeTag, massX, massX, 0.01, 1.0e-10, 1.0e-10, 1.0e-10)
                        if k == 0:
                                ops.fix(nodeTag, 1, 1, 1, 1, 1, 1)

                        yLoc += bayWidthY
                        nodeTag += 1

                xLoc += bayWidthX

        if k < numFloor:
                storyHeight = storyHeights[k]

        zLoc += storyHeight

# add column element
ops.geomTransf(coordTransf, 1, 1, 0, 0)
ops.geomTransf(coordTransf, 2, 0, 0, 1)

eleTag = 1
nodeTag1 = 1

for k in range(0, numFloor):
        for i in range(0, numBayX+1):
                for j in range(0, numBayY+1):
                        nodeTag2 = nodeTag1 + (numBayX+1)*(numBayY+1)
                        iNode = ops.nodeCoord(nodeTag1)
                        jNode = ops.nodeCoord(nodeTag2)
                        ops.element('elasticBeamColumn', eleTag, nodeTag1, nodeTag2, 50., E, 1000., 1000., 2150., 2150., 1, '-mass', M, massType)
                        eleTag += 1
                        nodeTag1 += 1
```

(continues on next page)

(continued from previous page)

```
80
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 = ops.nodeCoord(nodeTag1)
89                          jNode = ops.nodeCoord(nodeTag2)
90                          ops.element('elasticBeamColumn', eleTag, nodeTag1, nodeTag2,␣
    ↪50., E, 1000., 1000., 2150., 2150., 2, '-mass', M, massType)
91                          eleTag += 1
92                          nodeTag1 += 1
93
94          nodeTag1 += (numBayY+1)
95
96  nodeTag1 = 1+ (numBayX+1)*(numBayY+1)
97  #add beam elements
98  for j in range(1, numFloor + 1):
99          for i in range(0, numBayY+1):
100                 for k in range(0, numBayX):
101                         nodeTag2 = nodeTag1 + 1
102                         iNode = ops.nodeCoord(nodeTag1)
103                         jNode = ops.nodeCoord(nodeTag2)
104                         ops.element('elasticBeamColumn', eleTag, nodeTag1, nodeTag2,␣
    ↪50., E, 1000., 1000., 2150., 2150., 2, '-mass', M, massType)
105                         eleTag += 1
106                         nodeTag1 += 1
107                 nodeTag1 += 1
108
109 # calculate eigenvalues & print results
110 numEigen = 7
111 eigenValues = ops.eigen(numEigen)
112 PI = 2 * asin(1.0)
113
114 ###################################
115 #### Display the active model with node tags only
116 opsplt.plot_model("nodes")
117
118 ####  Display specific mode shape with scale factor of 300 using the active model
119 opsplt.plot_modeshape(5, 300)
120
121 ###################################
122 # To save the analysis output for deformed shape, use createODB command before␣
    ↪running the analysis
123 # The following command saves the model data, and output for gravity analysis and the␣
    ↪first 3 modes
124 # in a folder "3DFrame_ODB"
125
126 opsplt.createODB("3DFrame", "Gravity", Nmodes=3)
127
128
129 # Define Static Analysis
130 ops.timeSeries('Linear', 1)
131 ops.pattern('Plain', 1, 1)
132 ops.load(72, 1, 0, 0, 0, 0, 0)
```

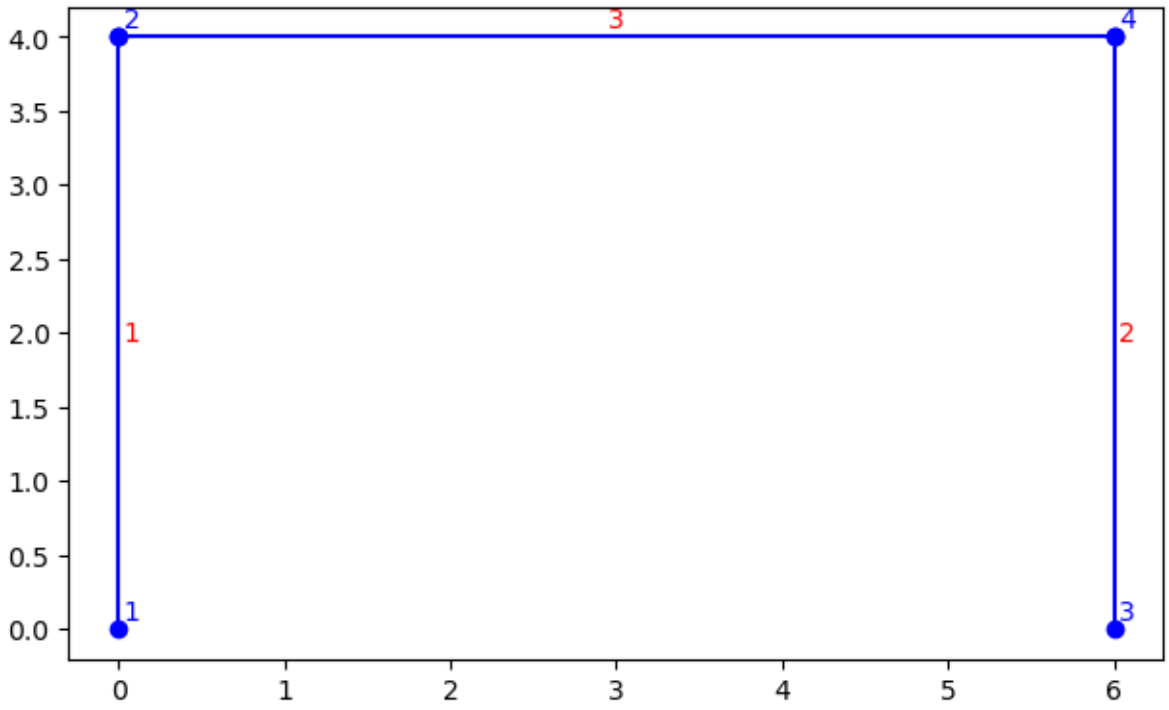
(continues on next page)

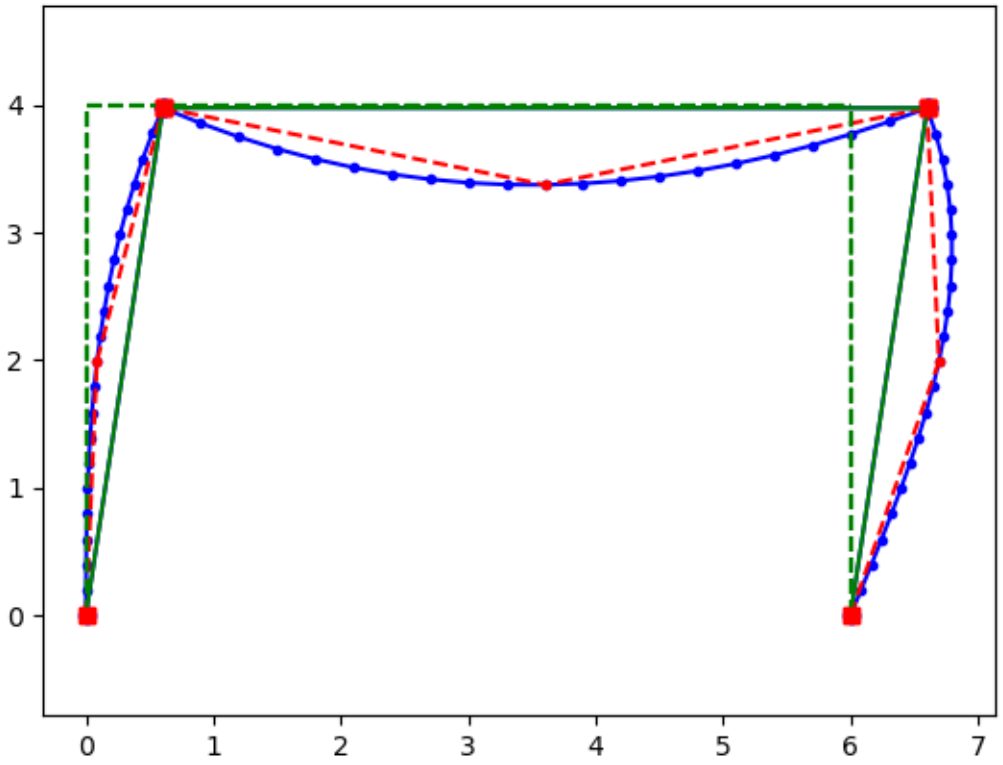
(continued from previous page)

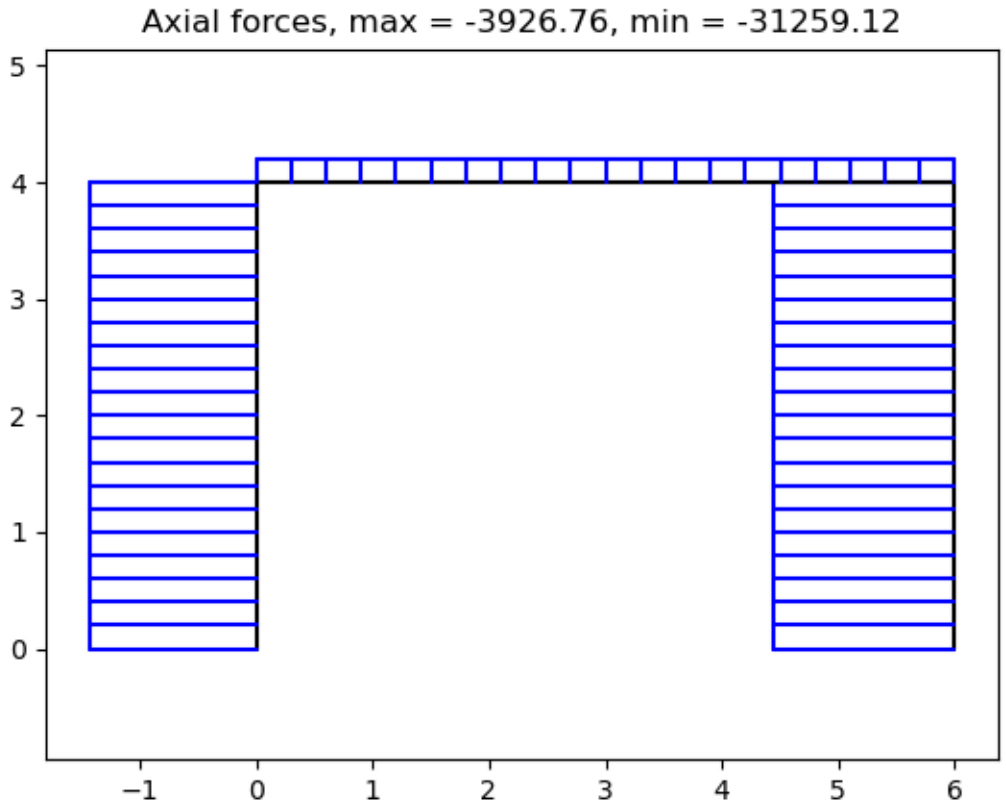
```
ops.analysis('Static')

# Run Analysis
ops.analyze(10)

# IMPORTANT: Make sure to issue a wipe() command to close all the recorders. Not issuing a wipe() command
# ... can cause errors in the plot_deformedshape() command.

ops.wipe()

###################################
### Now plot mode shape 2 with scale factor of 300 and the deformed shape using the recorded output data

opsplt.plot_modeshape(2, 300, Model="3DFrame")
opsplt.plot_deformedshape(Model="3DFrame", LoadCase="Gravity")
```

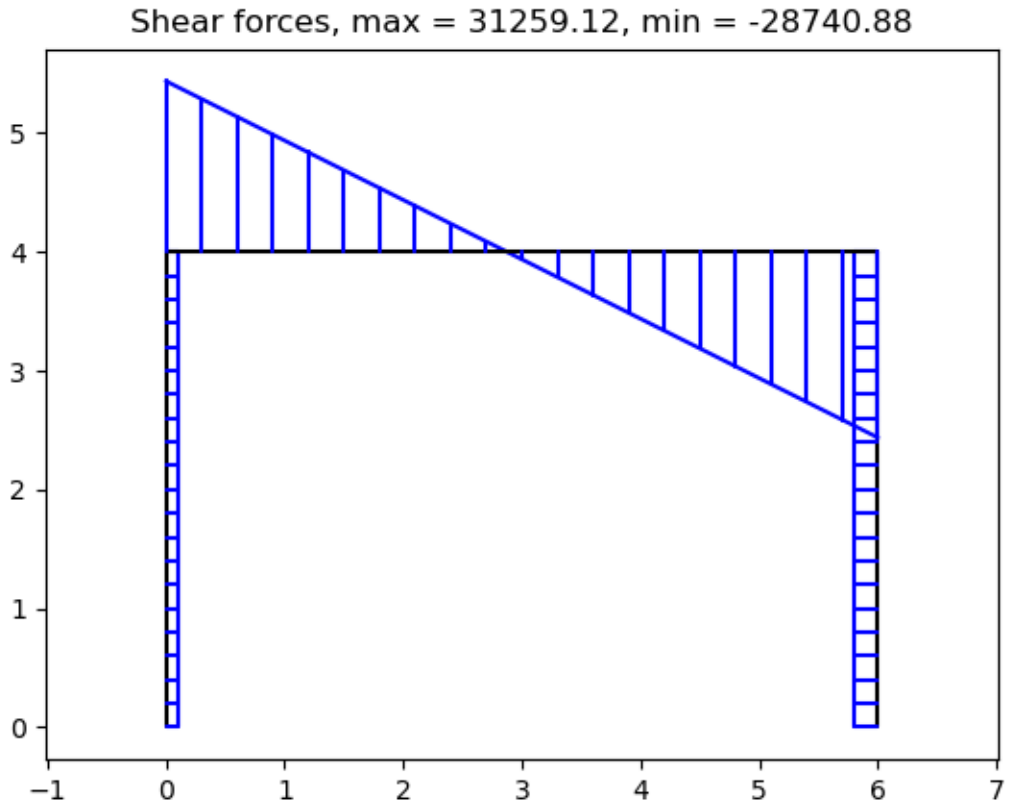
Statics of a 2d Portal Frame (ops_vis)

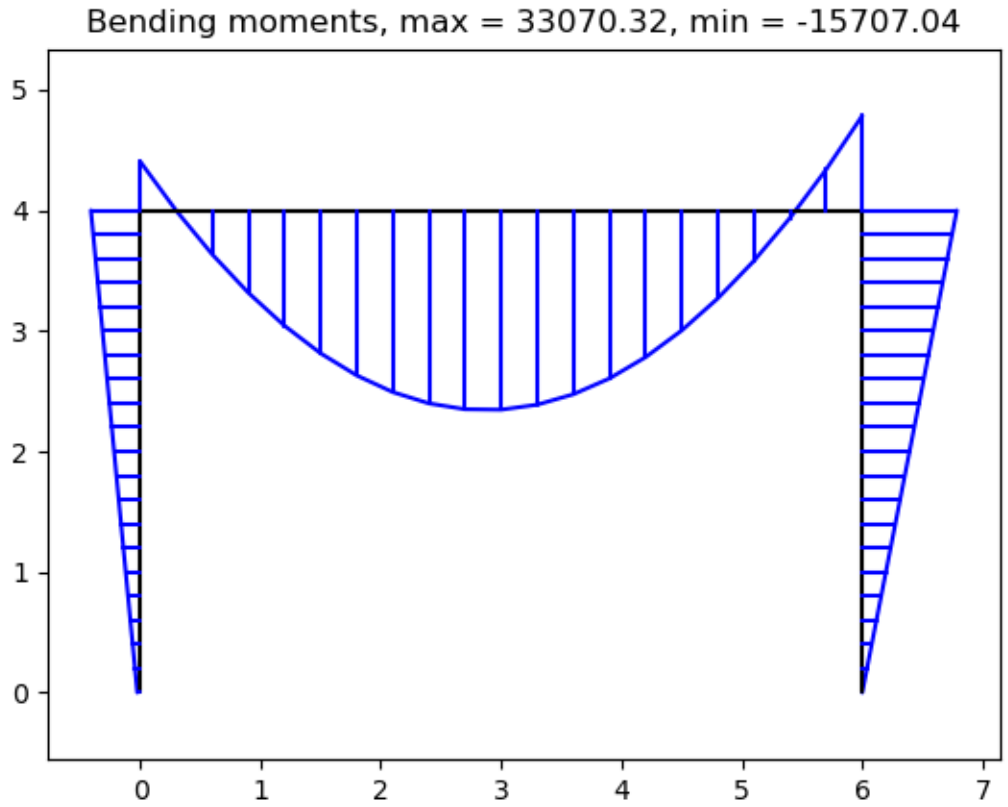
Example `.py` file can be downloaded `here`:




4
3
2
1
0
0
1
2
3
4
5
6
7


Axial forces, max = -3926.76, min = -31259.12
5
4
3
2
1
0
−1
0
1
2
3
4
5
6


Shear forces, max = 31259.12, min = -28740.88
5
4
3
2
1
0
−1
0
1
2
3
4
5
6
7




```
import openseespy.opensees as ops
import openseespy.postprocessing.ops_vis as opsv
# import opensees as ops  # local compilation
# import ops_vis as opsv  # local

import matplotlib.pyplot as plt

ops.wipe()
ops.model('basic', '-ndm', 2, '-ndf', 3)

colL, girL = 4., 6.

Acol, Agir = 2.e-3, 6.e-3
IzCol, IzGir = 1.6e-5, 5.4e-5

E = 200.e9

Ep = {1: [E, Acol, IzCol],
      2: [E, Acol, IzCol],
      3: [E, Agir, IzGir]}

ops.node(1, 0., 0.)
ops.node(2, 0., colL)
ops.node(3, girL, 0.)
ops.node(4, girL, colL)
```

(continues on next page)

(continued from previous page)

```

ops.fix(1, 1, 1, 1)
ops.fix(3, 1, 1, 0)

ops.geomTransf('Linear', 1)

# columns
ops.element('elasticBeamColumn', 1, 1, 2, Acol, E, IzCol, 1)
ops.element('elasticBeamColumn', 2, 3, 4, Acol, E, IzCol, 1)
# girder
ops.element('elasticBeamColumn', 3, 2, 4, Agir, E, IzGir, 1)

Px = 2.e+3
Wy = -10.e+3
Wx = 0.

Ew = {3: ['-beamUniform', Wy, Wx]}

ops.timeSeries('Constant', 1)
ops.pattern('Plain', 1, 1)
ops.load(2, Px, 0., 0.)

for etag in Ew:
    ops.eleLoad('-ele', etag, '-type', Ew[etag][0], Ew[etag][1],
                Ew[etag][2])

ops.constraints('Transformation')
ops.numberer('RCM')
ops.system('BandGeneral')
ops.test('NormDispIncr', 1.0e-6, 6, 2)
ops.algorithm('Linear')
ops.integrator('LoadControl', 1)
ops.analysis('Static')
ops.analyze(1)

ops.printModel()

# 1. plot model with tag lebels

szer, wys = 16., 10.

fig = plt.figure(figsize=(szer/2.54, wys/2.54))
fig.subplots_adjust(left=.08, bottom=.08, right=.985, top=.94)
ax1 = plt.subplot(111)

opsv.plot_model()

# 2. plot deformed model

sfac = 80.

plt.figure()
# plot_defo with optional arguments
# sfac = opsv.plot_defo()
opsv.plot_defo(sfac, fmt_interp='b.-')
opsv.plot_defo(sfac, 5, interpFlag=0, fmt_nodes='bo-')
opsv.plot_defo(sfac, 3, endDispFlag=0, fmt_interp='r.--')
```

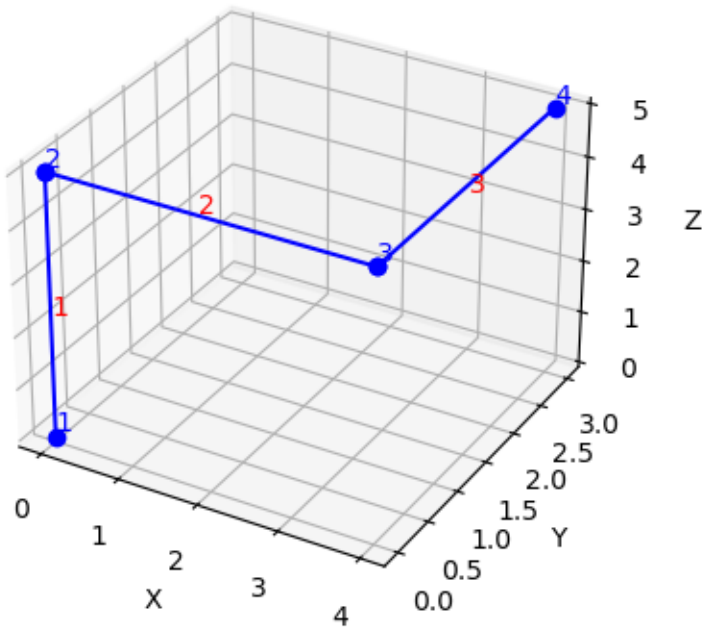
(continues on next page)

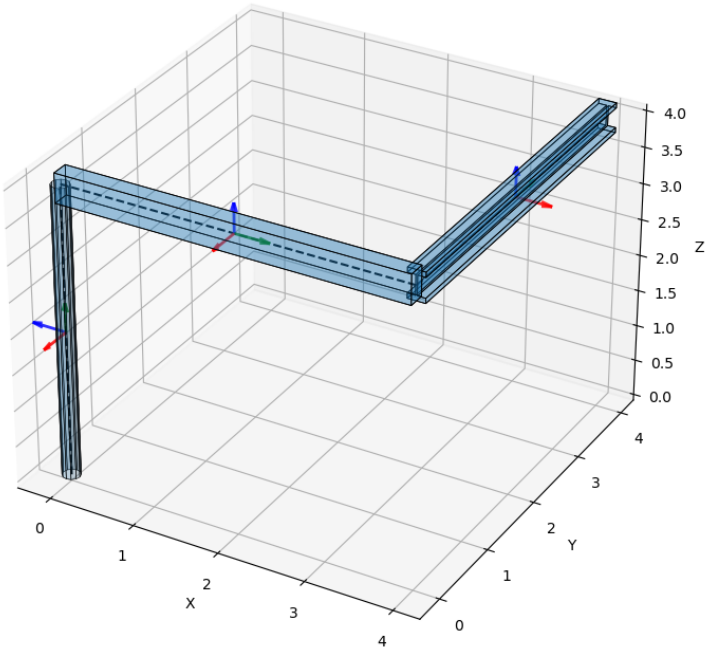
(continued from previous page)

```
opsv.plot_defo(sfac, 2, fmt_interp='g.-')

# print(f'sfac: {sfac}')  # return sfac if automatically calculated

# 3. plot N, V, M forces diagrams

sfacN, sfacV, sfacM = 5.e-5, 5.e-5, 5.e-5

plt.figure()
minVal, maxVal = opsv.section_force_diagram_2d('N', Ew, sfacN)
plt.title(f'Axial forces, max = {maxVal:.2f}, min = {minVal:.2f}')

plt.figure()
minVal, maxVal = opsv.section_force_diagram_2d('T', Ew, sfacV)
plt.title(f'Shear forces, max = {maxVal:.2f}, min = {minVal:.2f}')

plt.figure()
minVal, maxVal = opsv.section_force_diagram_2d('M', Ew, sfacM)
plt.title(f'Bending moments, max = {maxVal:.2f}, min = {minVal:.2f}')

plt.show()

exit()
```

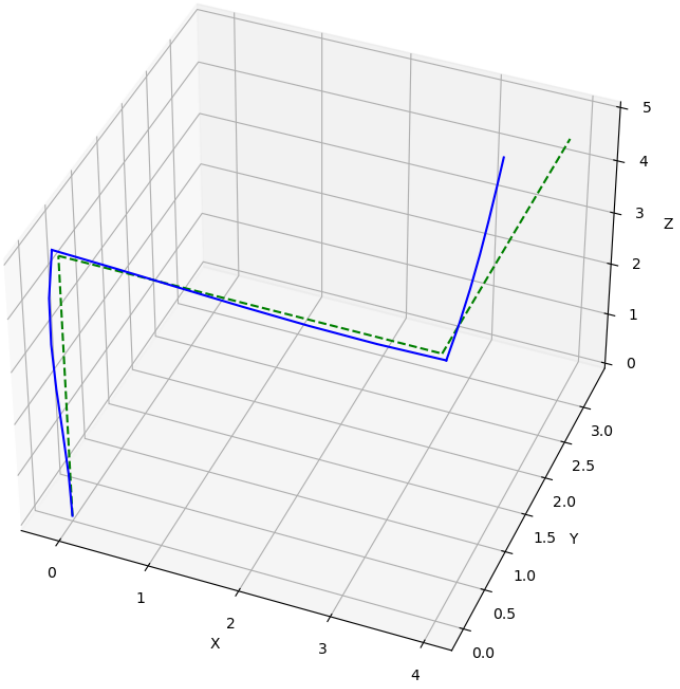
Statics of a 3d 3-element cantilever beam (ops_vis)

Example `.py` file can be downloaded `here`:

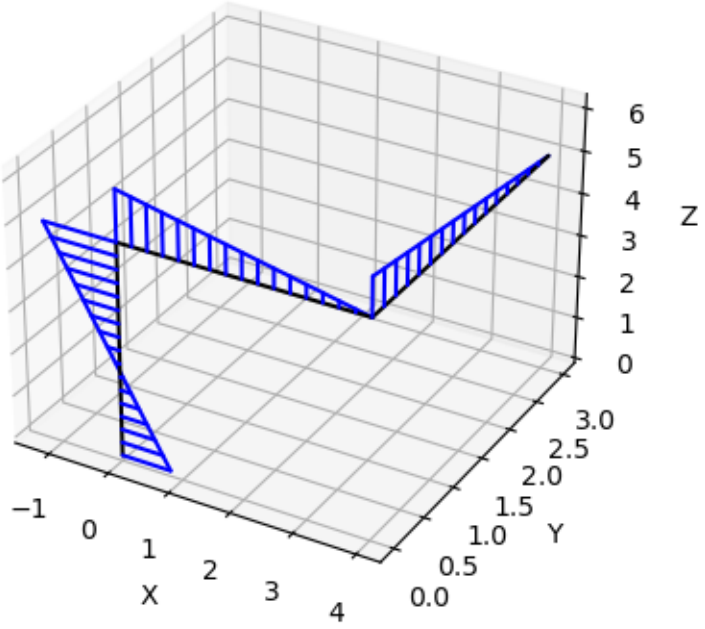




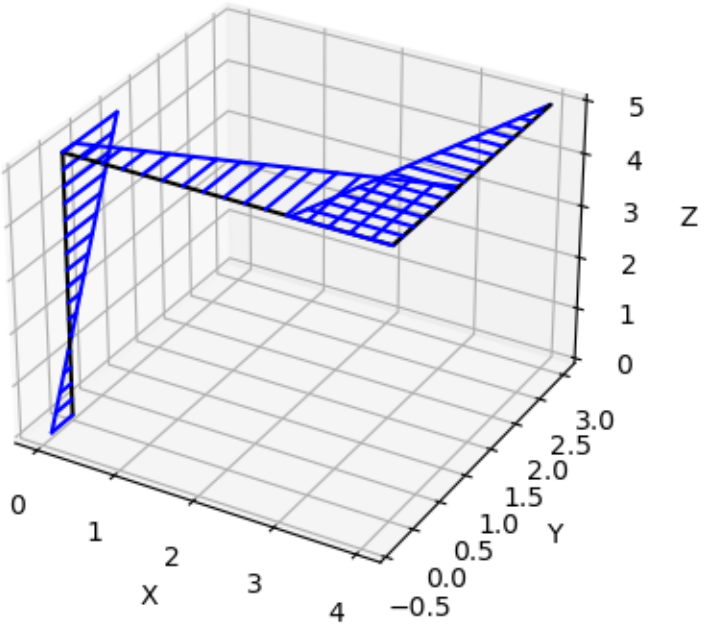
3d 3-element cantilever beam

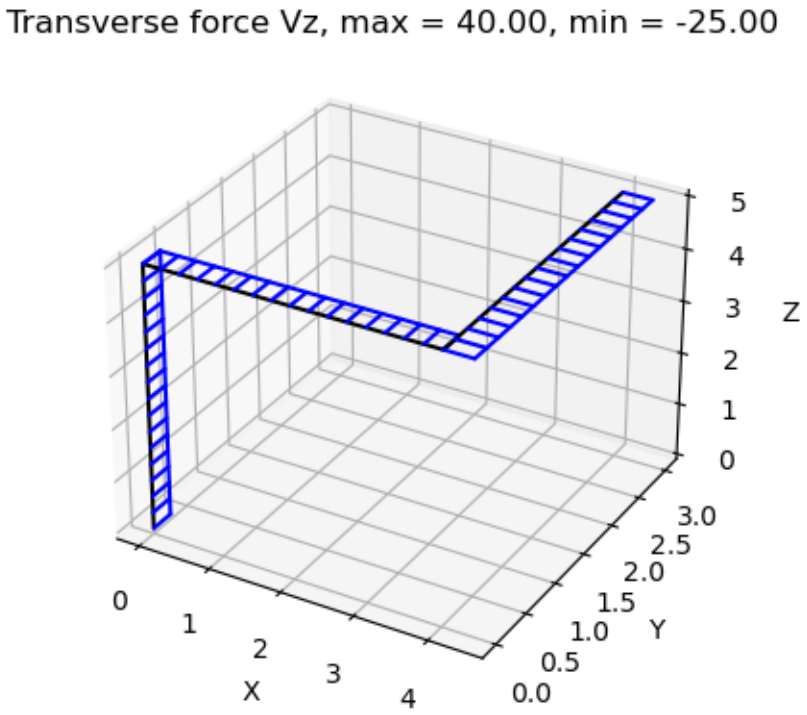


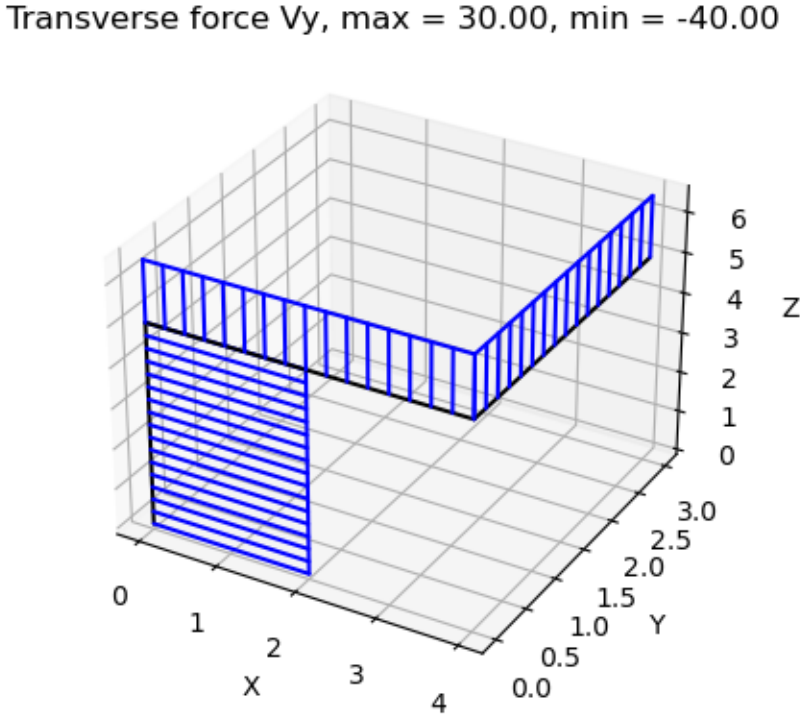
Bending moments Mz, max = 80.00, min = -120.00

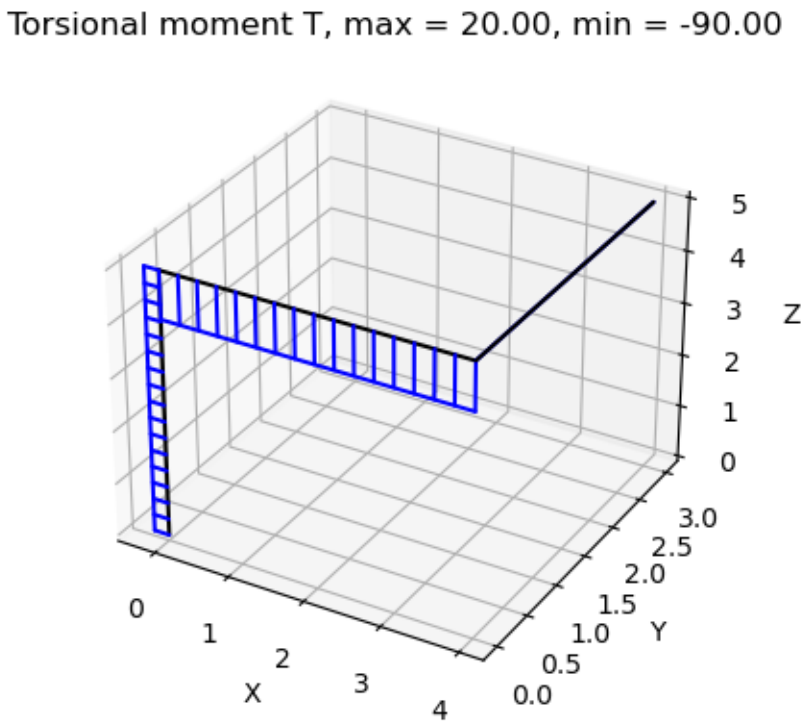


Bending moments My, max = 35.00, min = -120.00




Transverse force Vz, max = 40.00, min = -25.00
5
4
3
Z
2
1
0
3.0
2.5
2.0
1.5
1.0
Y
0.5
0.0
0
1
2
3
4
X


Transverse force Vy, max = 30.00, min = -40.00
6
5
4
Z
3
2
1
0
3.0
2.5
2.0
1.5
1.0
Y
0.5
0.0
0
1
2
3
4
X




```
import openseespy.opensees as ops
import openseespy.postprocessing.ops_vis as opsv
# import opensees as ops  # local compilation
# import ops_vis as opsv  # local

import matplotlib.pyplot as plt

ops.wipe()

ops.model('basic', '-ndm', 3, '-ndf', 6)

b = 0.2
h = 0.4

A, Iz, Iy, J = 0.04, 0.0010667, 0.0002667, 0.01172

E = 25.0e6
G = 9615384.6

# Lx, Ly, Lz = 4., 3., 5.
Lx, Ly, Lz = 4., 4., 4.

ops.node(1, 0., 0., 0.)
ops.node(2, 0., 0., Lz)
ops.node(3, Lx, 0., Lz)
ops.node(4, Lx, Ly, Lz)

ops.fix(1, 1, 1, 1, 1, 1, 1)

lmass = 200.

```

(continues on next page)

(continued from previous page)

```
ops.mass(2, lmass, lmass, lmass, 0.001, 0.001, 0.001)
ops.mass(3, lmass, lmass, lmass, 0.001, 0.001, 0.001)
ops.mass(4, lmass, lmass, lmass, 0.001, 0.001, 0.001)

gTTagz = 1
gTTagx = 2
gTTagy = 3

coordTransf = 'Linear'
ops.geomTransf(coordTransf, gTTagz, 0., -1., 0.)
ops.geomTransf(coordTransf, gTTagx, 0., -1., 0.)
ops.geomTransf(coordTransf, gTTagy, 1., 0., 0.)

ops.element('elasticBeamColumn', 1, 1, 2, A, E, G, J, Iy, Iz, gTTagz)
ops.element('elasticBeamColumn', 2, 2, 3, A, E, G, J, Iy, Iz, gTTagx)
ops.element('elasticBeamColumn', 3, 3, 4, A, E, G, J, Iy, Iz, gTTagy)

Ew = {}

Px = -4.e1
Py = -2.5e1
Pz = -3.e1

ops.timeSeries('Constant', 1)
ops.pattern('Plain', 1, 1)
ops.load(4, Px, Py, Pz, 0., 0., 0.)

ops.constraints('Transformation')
ops.numberer('RCM')
ops.system('BandGeneral')
ops.test('NormDispIncr', 1.0e-6, 6, 2)
ops.algorithm('Linear')
ops.integrator('LoadControl', 1)
ops.analysis('Static')
ops.analyze(1)


opsv.plot_model()

sfac = 2.0e0

# fig_wi_he = 22., 14.
fig_wi_he = 30., 20.

# - 1
nep = 9
opsv.plot_defo(sfac, nep, fmt_interp='b-', az_el=(-68., 39.),
               fig_wi_he=fig_wi_he, endDispFlag=0)

plt.title('3d 3-element cantilever beam')

# - 2
opsv.plot_defo(sfac, 19, fmt_interp='b-', az_el=(6., 30.),
               fig_wi_he=fig_wi_he)

plt.title('3d 3-element cantilever beam')

```

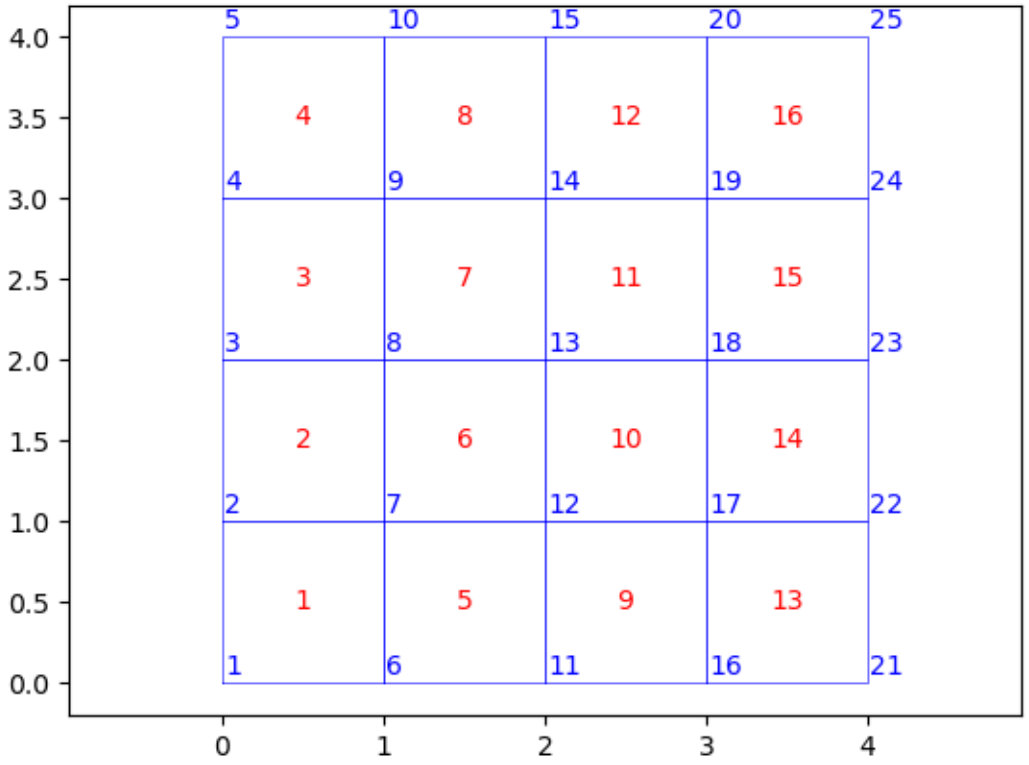
(continues on next page)

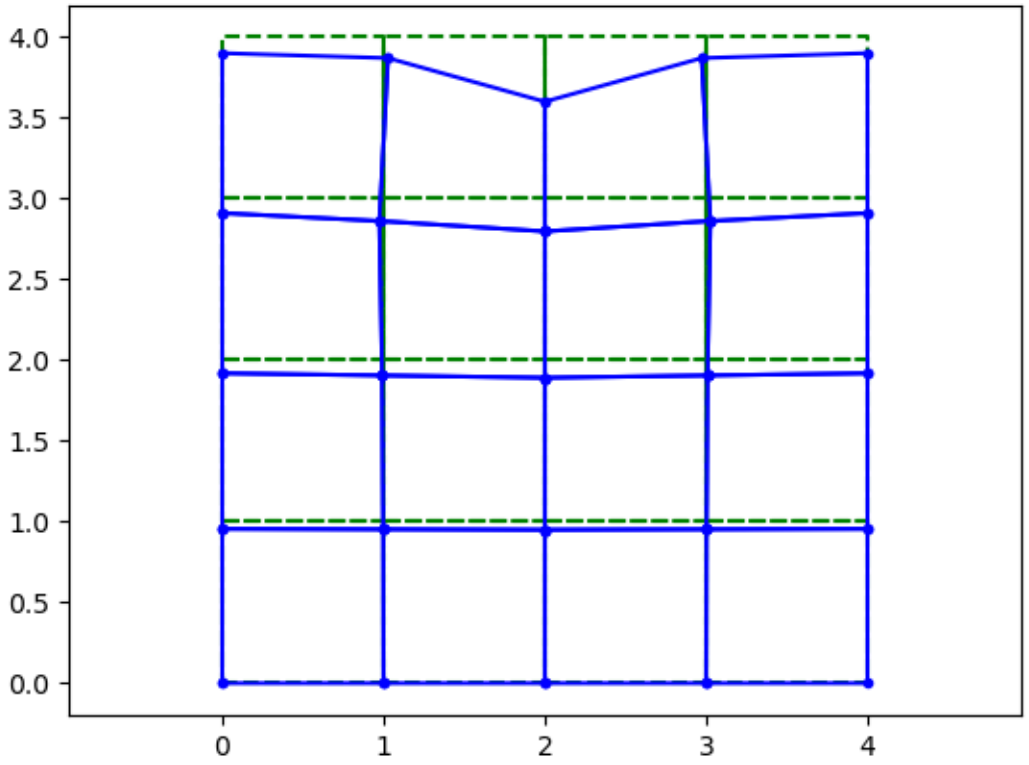
(continued from previous page)

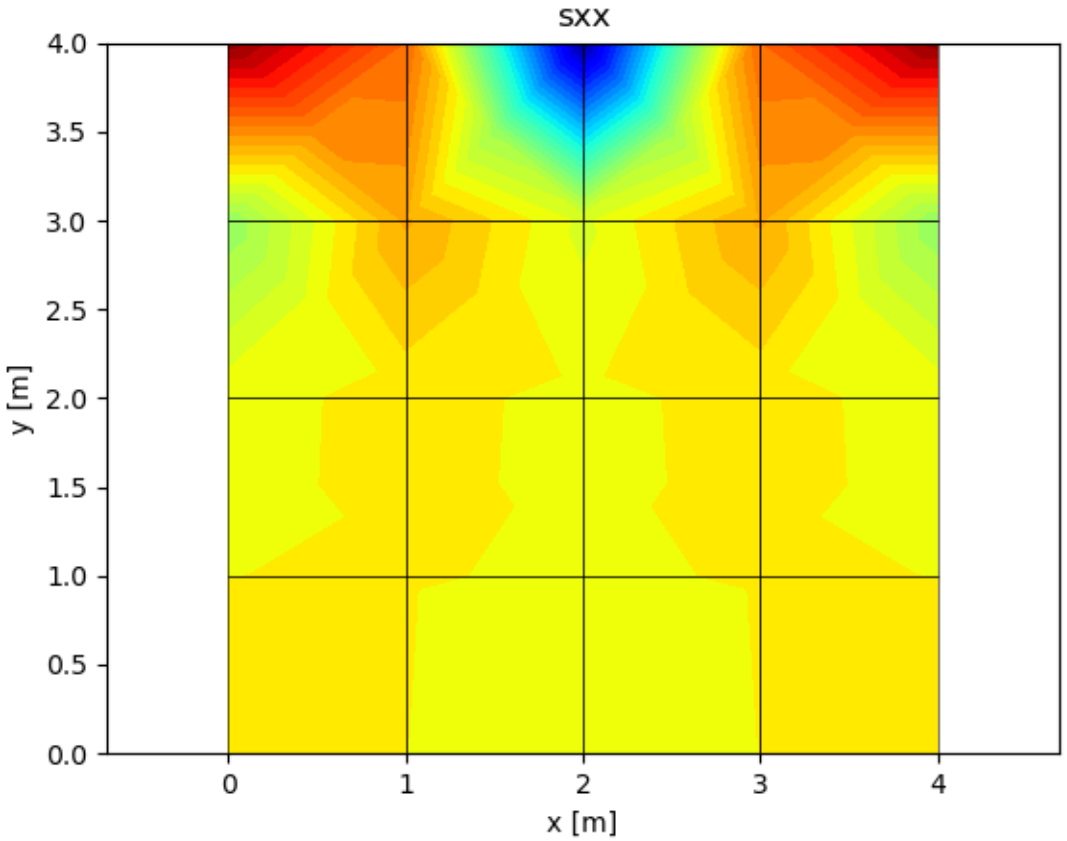
```
# - 3
nfreq = 6
eigValues = ops.eigen(nfreq)

modeNo = 6

sfac = 2.0e1
opsv.plot_mode_shape(modeNo, sfac, 19, fmt_interp='b-', az_el=(106., 46.),
                     fig_wi_he=fig_wi_he)
plt.title(f'Mode {modeNo}')

sfacN = 1.e-2
sfacVy = 5.e-2
sfacVz = 1.e-2
sfacMy = 1.e-2
sfacMz = 1.e-2
sfacT = 1.e-2

# plt.figure()
minY, maxY = opsv.section_force_diagram_3d('N', Ew, sfacN)
plt.title(f'Axial force N, max = {maxY:.2f}, min = {minY:.2f}')

minY, maxY = opsv.section_force_diagram_3d('Vy', Ew, sfacVy)
plt.title(f'Transverse force Vy, max = {maxY:.2f}, min = {minY:.2f}')

minY, maxY = opsv.section_force_diagram_3d('Vz', Ew, sfacVz)
plt.title(f'Transverse force Vz, max = {maxY:.2f}, min = {minY:.2f}')

minY, maxY = opsv.section_force_diagram_3d('My', Ew, sfacMy)
plt.title(f'Bending moments My, max = {maxY:.2f}, min = {minY:.2f}')

minY, maxY = opsv.section_force_diagram_3d('Mz', Ew, sfacMz)
plt.title(f'Bending moments Mz, max = {maxY:.2f}, min = {minY:.2f}')

minY, maxY = opsv.section_force_diagram_3d('T', Ew, sfacT)
plt.title(f'Torsional moment T, max = {maxY:.2f}, min = {minY:.2f}')

# just for demonstration,
# the section data below does not match the data in OpenSees model above
# For now it can be source of inconsistency because OpenSees has
# not got functions to return section dimensions.
# A workaround is to have own Python helper functions to reuse data
# specified once
ele_shapes = {1: ['circ', [h]],
              2: ['rect', [b, h]],
              3: ['I', [b, h, b/10., h/6.]]}
opsv.plot_extruded_shapes_3d(ele_shapes, fig_wi_he=fig_wi_he)

plt.show()

exit()
```

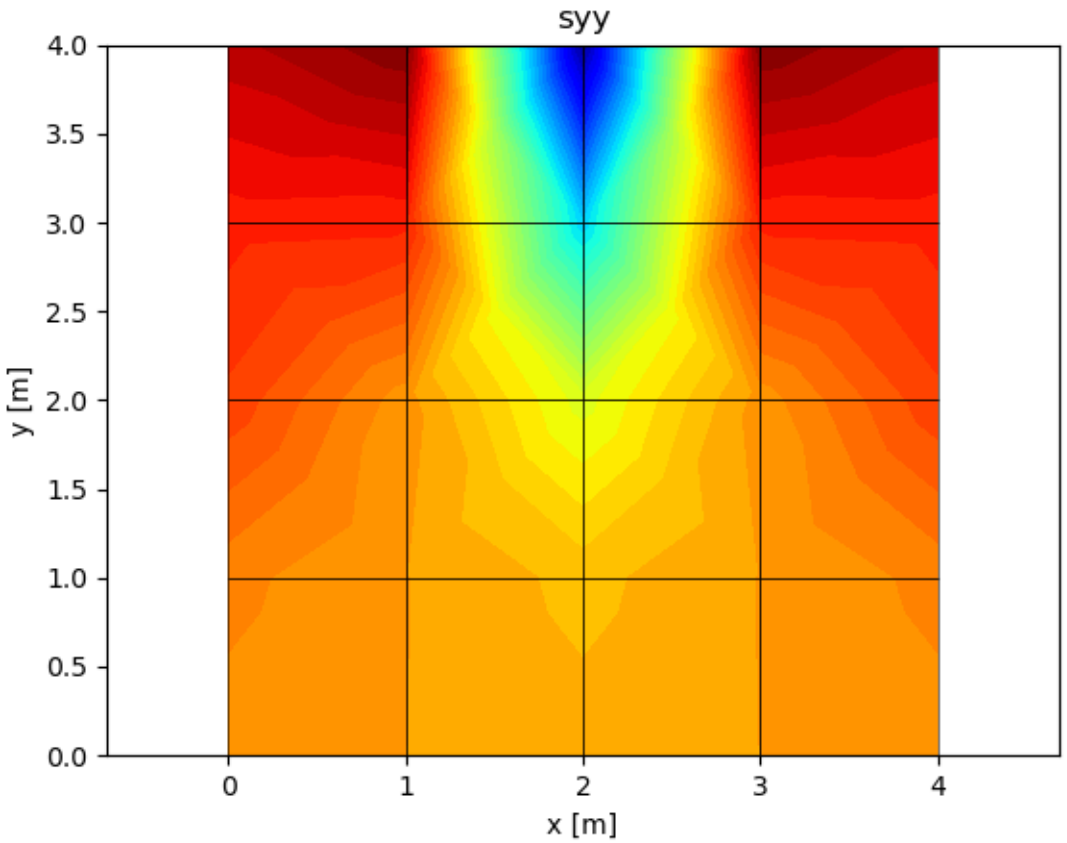
Plot stress distribution of plane stress quad model (ops_vis)

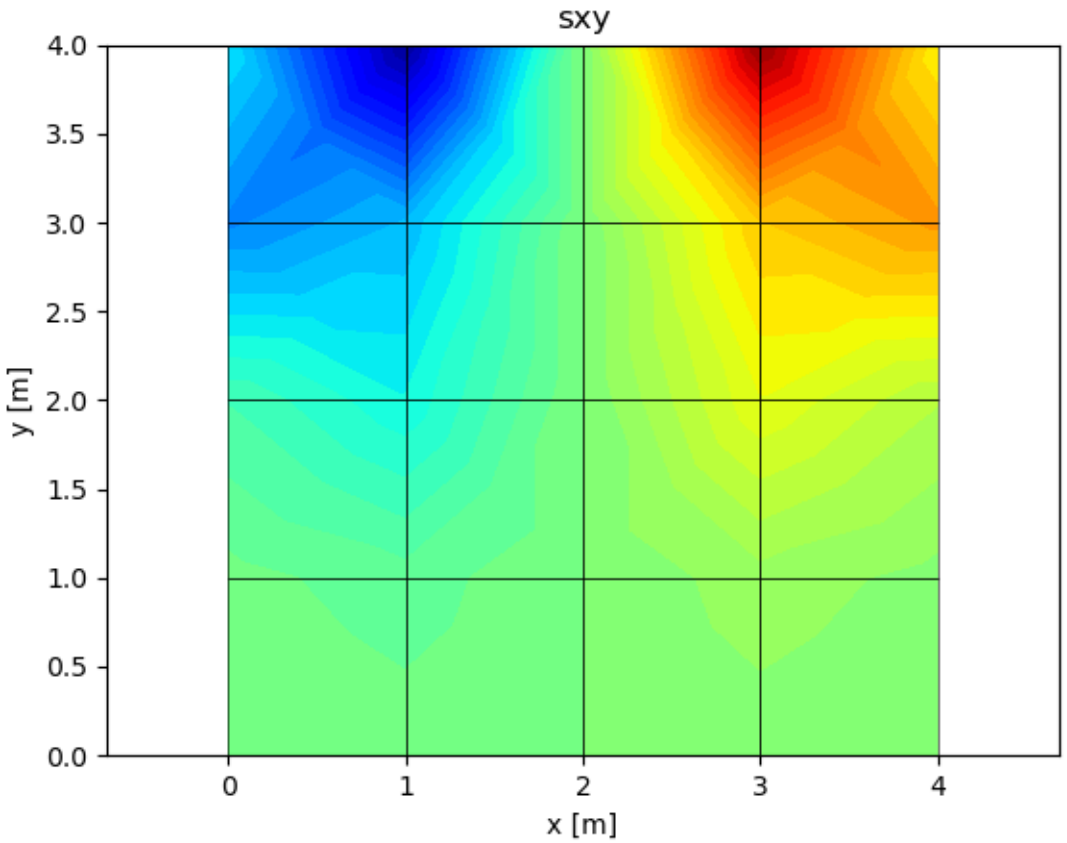
Example `.py` file can be downloaded `here`:

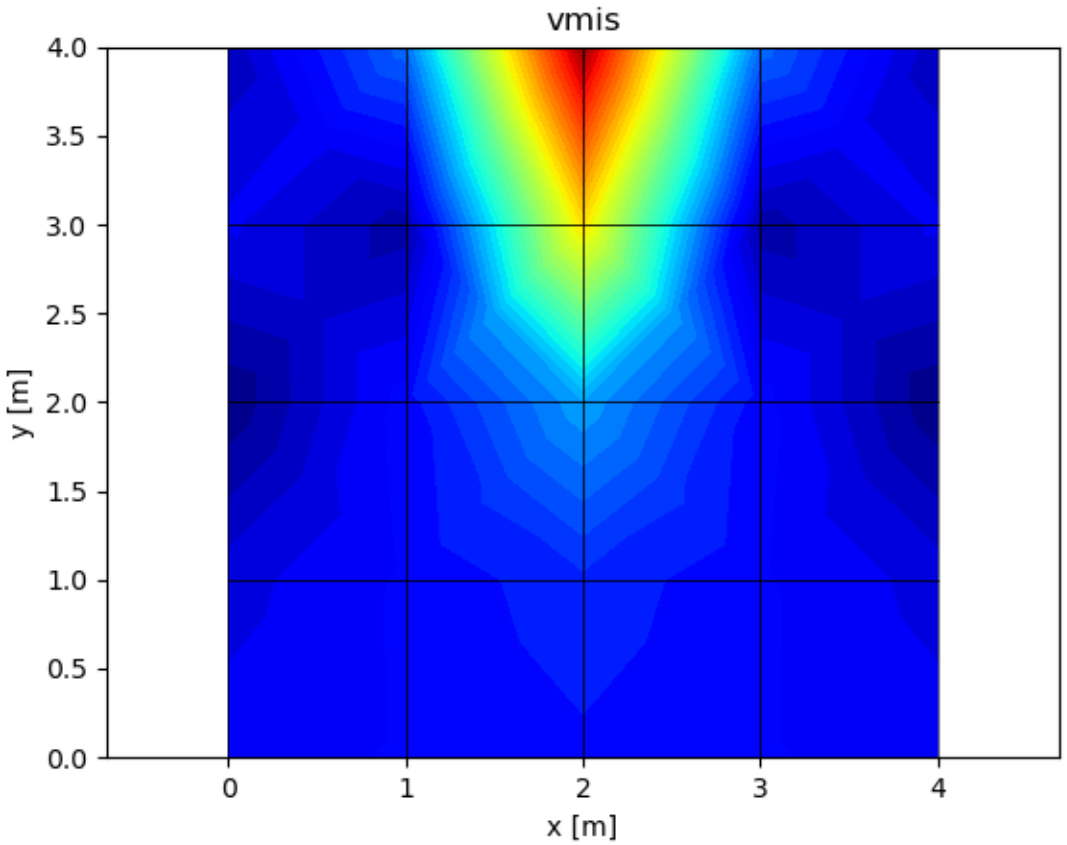

5
10
15
20
25
4
8
12
16
4
9
14
19
24
3
7
11
15
3
8
13
18
23
2
6
10
14
2
7
12
17
22
1
5
9
13
1
6
11
16
21
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
1
2
3
4

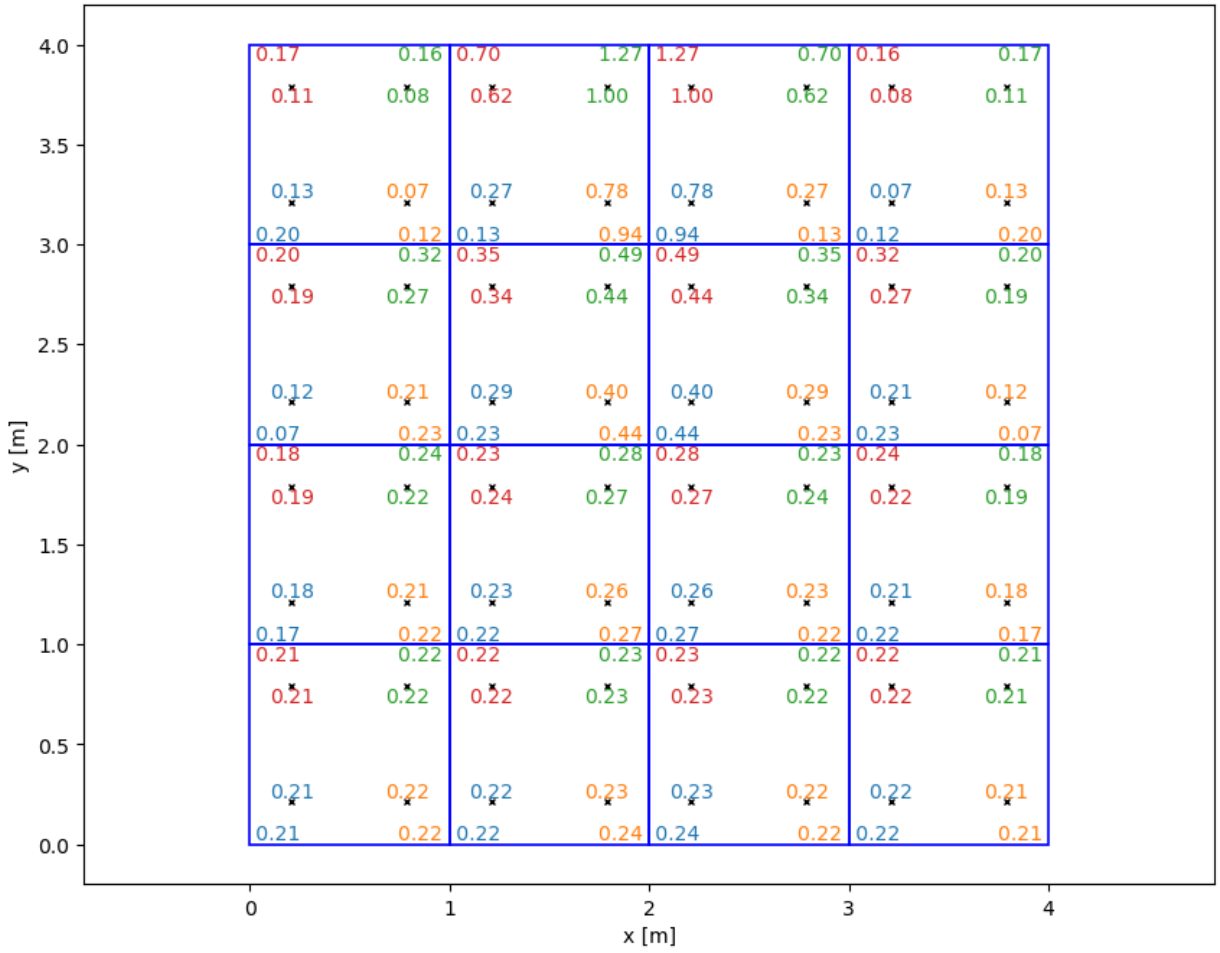

4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
1
2
3
4


sxx
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
y [m]
0
1
2
3
4
x [m]


syy
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
y [m]
0
1
2
3
4
x [m]


sxy
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
y [m]
0
1
2
3
4
x [m]


vmis
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
y [m]
0
1
2
3
4
x [m]




```python
import openseespy.opensees as ops
import openseespy.postprocessing.ops_vis as opsv
# import opensees as ops  # local compilation
# import ops_vis as opsv  # local

import matplotlib.pyplot as plt

ops.wipe()
ops.model('basic', '-ndm', 2, '-ndf', 2)
ops.node(1, 0., 0.)
ops.node(2, 0., 1.)
ops.node(3, 0., 2.)
ops.node(4, 0., 3.)
ops.node(5, 0., 4.)
ops.node(6, 1., 0.)
ops.node(7, 1., 1.)
ops.node(8, 1., 2.)
ops.node(9, 1., 3.)
ops.node(10, 1., 4.)
ops.node(11, 2., 0.)
ops.node(12, 2., 1.)
ops.node(13, 2., 2.)
```

(continues on next page)

(continued from previous page)

```
ops.node(14, 2., 3.)
ops.node(15, 2., 4.)
ops.node(16, 3., 0.)
ops.node(17, 3., 1.)
ops.node(18, 3., 2.)
ops.node(19, 3., 3.)
ops.node(20, 3., 4.)
ops.node(21, 4., 0.)
ops.node(22, 4., 1.)
ops.node(23, 4., 2.)
ops.node(24, 4., 3.)
ops.node(25, 4., 4.)

ops.nDMaterial('ElasticIsotropic', 1, 1000, 0.3)

ops.element('quad', 1, 1, 6, 7, 2, 1, 'PlaneStress', 1)
ops.element('quad', 2, 2, 7, 8, 3, 1, 'PlaneStress', 1)
ops.element('quad', 3, 3, 8, 9, 4, 1, 'PlaneStress', 1)
ops.element('quad', 4, 4, 9, 10, 5, 1, 'PlaneStress', 1)
ops.element('quad', 5, 6, 11, 12, 7, 1, 'PlaneStress', 1)
ops.element('quad', 6, 7, 12, 13, 8, 1, 'PlaneStress', 1)
ops.element('quad', 7, 8, 13, 14, 9, 1, 'PlaneStress', 1)
ops.element('quad', 8, 9, 14, 15, 10, 1, 'PlaneStress', 1)
ops.element('quad', 9, 11, 16, 17, 12, 1, 'PlaneStress', 1)
ops.element('quad', 10, 12, 17, 18, 13, 1, 'PlaneStress', 1)
ops.element('quad', 11, 13, 18, 19, 14, 1, 'PlaneStress', 1)
ops.element('quad', 12, 14, 19, 20, 15, 1, 'PlaneStress', 1)
ops.element('quad', 13, 16, 21, 22, 17, 1, 'PlaneStress', 1)
ops.element('quad', 14, 17, 22, 23, 18, 1, 'PlaneStress', 1)
ops.element('quad', 15, 18, 23, 24, 19, 1, 'PlaneStress', 1)
ops.element('quad', 16, 19, 24, 25, 20, 1, 'PlaneStress', 1)

ops.fix(1, 1, 1)
ops.fix(6, 1, 1)
ops.fix(11, 1, 1)
ops.fix(16, 1, 1)
ops.fix(21, 1, 1)

ops.equalDOF(2, 22, 1, 2)
ops.equalDOF(3, 23, 1, 2)
ops.equalDOF(4, 24, 1, 2)
ops.equalDOF(5, 25, 1, 2)

ops.timeSeries('Linear', 1)
ops.pattern('Plain', 1, 1)
ops.load(15, 0., -1.)

ops.analysis('Static')
ops.analyze(1)

# - plot model
plt.figure()
opsv.plot_model()
plt.axis('equal')

# - plot deformation
plt.figure()
```

(continues on next page)

(continued from previous page)

```
opsv.plot_defo()
# opsv.plot_defo(sfac, unDefoFlag=1, fmt_undefo='g:')
plt.axis('equal')

# get values at OpenSees nodes
sig_out = opsv.sig_out_per_node()
print(f'sig_out:\n{sig_out}')

# - visu stress map

# !!! select from sig_out: e.g. vmises
# j, jstr = 0, 'sxx'
# j, jstr = 1, 'syy'
# j, jstr = 2, 'sxy'
j, jstr = 3, 'vmis'
# j, jstr = 4, 's1'
# j, jstr = 5, 's2'
# j, jstr = 6, 'alfa'

nds_val = sig_out[:, j]
# print(f'nds_val:\n{nds_val}')

plt.figure()
opsv.plot_stress_2d(nds_val)

plt.xlabel('x [m]')
plt.ylabel('y [m]')
plt.title(f'{jstr}')

# plt.savefig(f'quads_4x4_{jstr}.png')

# for educational purposes show values at integration points and
# nodes which can finally be averaged at nodes
eles_ips_crd, eles_nds_crd, nds_crd, quads_conn = opsv.quad_crds_node_to_ip()

print(f'\neles_ips_crd:\n{eles_ips_crd}')
print(f'\neles_nds_crd:\n{eles_nds_crd}')
print(f'\nnds_crd:\n{nds_crd}')
print(f'\nquads_conn:\n{quads_conn}')

eles_ips_sig_out, eles_nds_sig_out = opsv.sig_out_per_ele_quad()

print(f'\neles_ips_sig_out:\n{eles_ips_sig_out}')
print(f'\neles_nds_sig_out:\n{eles_nds_sig_out}')

sig_out_indx = j  # same as j, jstr

fig = plt.figure(figsize=(22./2.54, 18./2.54))  # centimeter to inch conversion
fig.subplots_adjust(left=.08, bottom=.08, right=.985, top=.94)
opsv.plot_mesh_with_ips_2d(nds_crd, eles_ips_crd, eles_nds_crd, quads_conn,
                           eles_ips_sig_out, eles_nds_sig_out, sig_out_indx)

plt.xlabel('x [m]')
plt.ylabel('y [m]')

# plt.savefig(f'quads_4x4_{jstr}_ips_nds_vals.png')

```

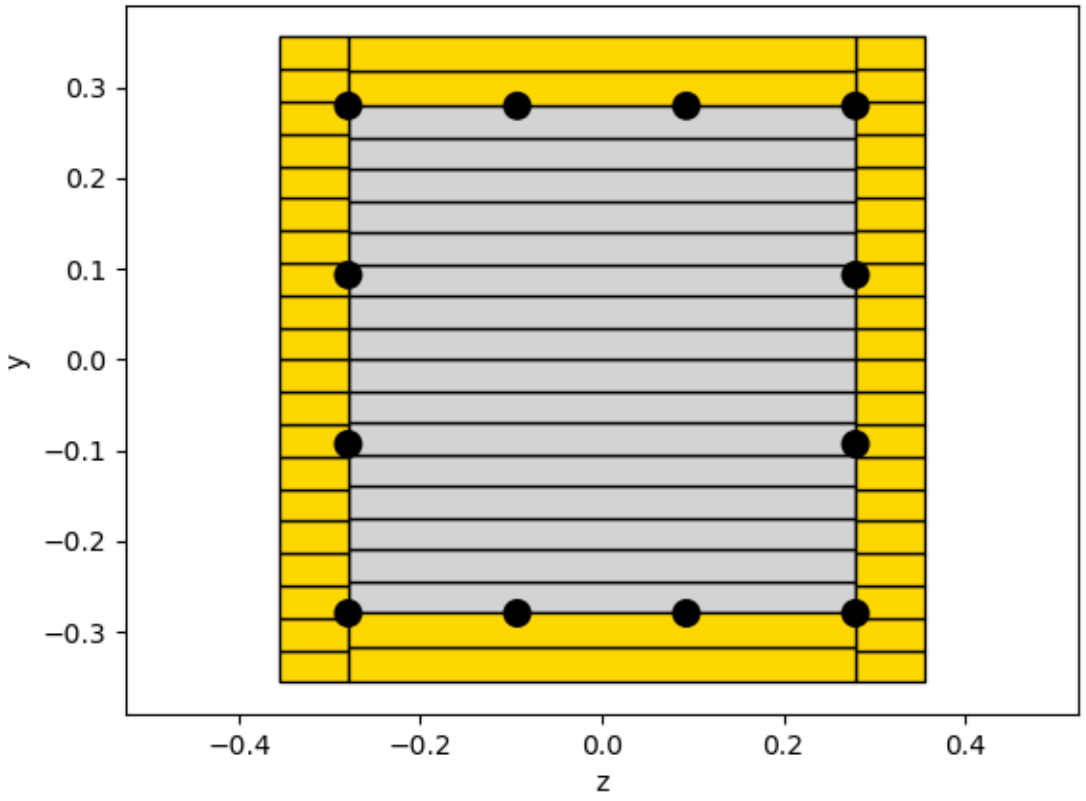
(continues on next page)

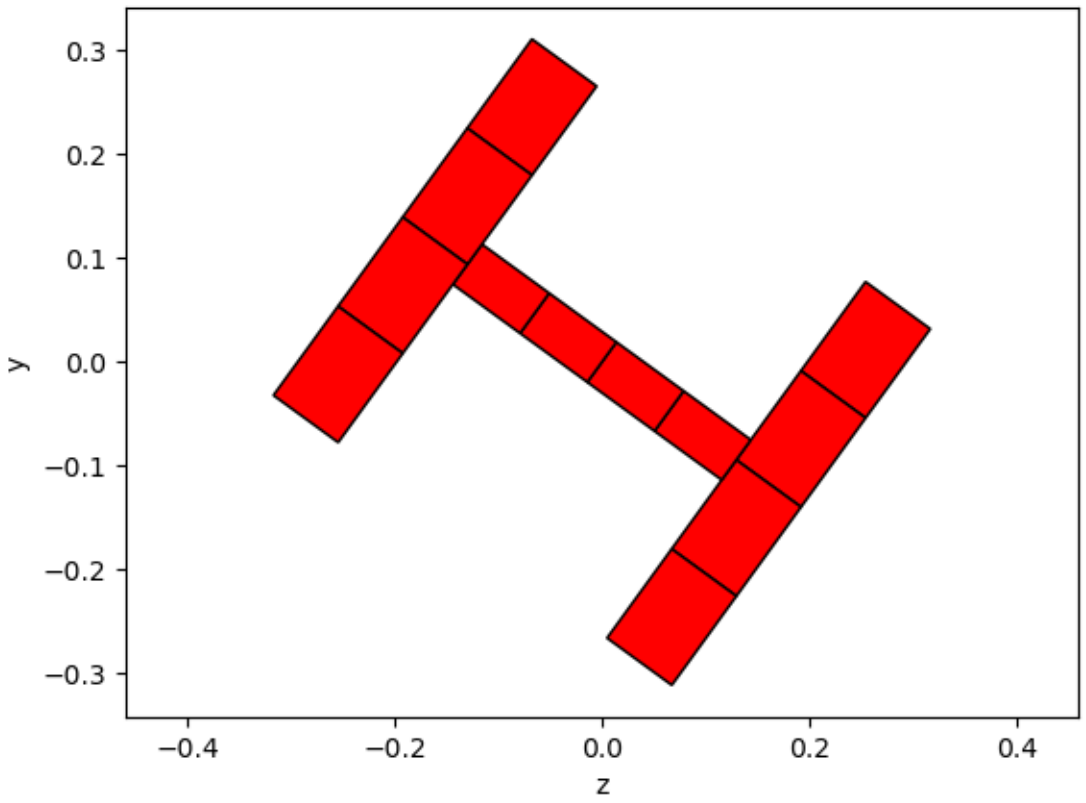
(continued from previous page)

```
plt.show()

exit()
```

Plot steel and reinforced concrete fiber sections (ops_vis)

Example `.py` file can be downloaded `here`:






```
import openseespy.opensees as ops
import openseespy.postprocessing.ops_vis as opsv
# import opensees as ops  # local compilation
# import ops_vis as opsv  # local

import matplotlib.pyplot as plt

ops.wipe()
ops.model('basic', '-ndm', 2, '-ndf', 3)  # frame 2D

# 1. rotated steel shape

fib_sec_1 = [['section', 'Fiber', 1, '-GJ', 1.0e6],
             ['patch', 'quad', 1, 4, 1,  0.032, 0.317, -0.311, 0.067, -0.266, 0.005, 0.077, 0.254],
             ['patch', 'quad', 1, 1, 4,  -0.075, 0.144, -0.114, 0.116, 0.075, -0.144, 0.114, -0.116],
             ['patch', 'quad', 1, 4, 1,  0.266, -0.005,  -0.077, -0.254,  -0.032, -0.317,  0.311, -0.067]
             ]

# fib_sec_1 list can be used both for plotting and OpenSees commands defining
# the actual fiber section in the OpenSees domain. Normally you would have to
# use regular section('Fiber', ...), fiber(), layer(), patch() commands with
# the same data to define the fiber section. However do not use both
```

(continues on next page)

(continued from previous page)

```
# ways in the same model.

# opsv.fib_sec_list_to_cmds(fib_sec_1)

# 2. RC section

Bcol = 0.711
Hcol = Bcol

c = 0.076  # cover

y1col = Hcol/2.0
z1col = Bcol/2.0

y2col = 0.5*(Hcol-2*c)/3.0

nFibZ = 1
nFib = 20
nFibCover, nFibCore = 2, 16
As9 = 0.0006446

fib_sec_2 = [['section', 'Fiber', 3, '-GJ', 1.0e6],
             ['patch', 'rect', 2, nFibCore, nFibZ, c-y1col, c-z1col, y1col-c, z1col-
↪c],
             ['patch', 'rect', 3, nFib, nFibZ, -y1col, -z1col, y1col, c-z1col],
             ['patch', 'rect', 3, nFib, nFibZ, -y1col, z1col-c, y1col, z1col],
             ['patch', 'rect', 3, nFibCover, nFibZ, -y1col, c-z1col, c-y1col, z1col-
↪c],
             ['patch', 'rect', 3, nFibCover, nFibZ, y1col-c, c-z1col, y1col, z1col-c],
             ['layer', 'straight', 4, 4, As9, y1col-c, z1col-c, y1col-c, c-z1col],
             ['layer', 'straight', 4, 2, As9, y2col, z1col-c, y2col, c-z1col],
             ['layer', 'straight', 4, 2, As9, -y2col, z1col-c, -y2col, c-z1col],
             ['layer', 'straight', 4, 4, As9, c-y1col, z1col-c, c-y1col, c-z1col]]


# opsv.fib_sec_list_to_cmds(fib_sec_2)

matcolor = ['r', 'lightgrey', 'gold', 'w', 'w', 'w']
opsv.plot_fiber_section(fib_sec_1, matcolor=matcolor)
plt.axis('equal')
# plt.savefig('fibsec_wshape.png')

matcolor = ['r', 'lightgrey', 'gold', 'w', 'w', 'w']
opsv.plot_fiber_section(fib_sec_2, matcolor=matcolor)
plt.axis('equal')
# plt.savefig('fibsec_rc.png')

plt.show()
```

Animation of dynamic analysis and mode shapes of a 2d Portal Frame (ops_vis)

Example `.py` file can be downloaded `here`:

```
import openseespy.opensees as ops
import openseespy.postprocessing.ops_vis as opsv
```

(continues on next page)

(continued from previous page)

```
# import opensees as ops  # local compilation
# import ops_vis as opsv  # local

import numpy as np
import matplotlib.pyplot as plt

# input_parameters = (3.8, 50., 100.)
# input_parameters = (53.5767, 50., 100.)
input_parameters = (20.8, 300., 8.)
# input_parameters = (70.0, 500., 2.)

pf, sfac_a, tkt = input_parameters

ops.wipe()
ops.model('basic', '-ndm', 2, '-ndf', 3)  # frame 2D

colL, girL = 4., 6.

Acol, Agir = 0.06, 0.06
IzCol, IzGir = 0.0002, 0.0002

E = 3.2e10
rho = 2400.
muCol = rho * Acol
muGir = rho * Agir
massCol = ['-mass', muCol, '-cMass']
massGir = ['-mass', muGir, '-cMass']

ops.node(0, 0.0, 0.0)
ops.node(1, 0.0, 2.0)
ops.node(2, 0.0, 4.0)
ops.node(3, 3.0, 4.0)
ops.node(4, 6.0, 4.0)
ops.node(5, 6.0, 2.0)
ops.node(6, 6.0, 0.0)

ops.fix(0, 1, 1, 1)
ops.fix(6, 1, 1, 0)

gTTag = 1
ops.geomTransf('Linear', gTTag)

# 1st column
ops.element('elasticBeamColumn', 1, 0, 1, Acol, E, IzCol, gTTag, *massCol)
ops.element('elasticBeamColumn', 2, 1, 2, Acol, E, IzCol, gTTag, *massCol)
# girder
ops.element('elasticBeamColumn', 3, 2, 3, Agir, E, IzGir, gTTag, *massGir)
ops.element('elasticBeamColumn', 4, 3, 4, Agir, E, IzGir, gTTag, *massGir)
# 2nd column
ops.element('elasticBeamColumn', 5, 4, 5, Acol, E, IzCol, gTTag, *massCol)
ops.element('elasticBeamColumn', 6, 5, 6, Acol, E, IzCol, gTTag, *massCol)

t0 = 0.
tk = 1.
Tp = 1/pf
P0 = 15000.
dt = 0.002
```

(continues on next page)

(continued from previous page)

```
n_steps = int((tk-t0)/dt)

tsTag = 1
ops.timeSeries('Trig', tsTag, t0, tk, Tp, '-factor', P0)

patTag = 1
ops.pattern('Plain', patTag, tsTag)
ops.load(1, 1., 0., 0.)

ops.constraints('Transformation')
ops.numberer('RCM')
ops.test('NormDispIncr', 1.0e-6, 10, 1)
ops.algorithm('Linear')
ops.system('ProfileSPD')
ops.integrator('Newmark', 0.5, 0.25)
ops.analysis('Transient')

el_tags = ops.getEleTags()

nels = len(el_tags)

Eds = np.zeros((n_steps, nels, 6))
timeV = np.zeros(n_steps)

# transient analysis loop and collecting the data
for step in range(n_steps):
    ops.analyze(1, dt)
    timeV[step] = ops.getTime()
    # collect disp for element nodes
    for el_i, ele_tag in enumerate(el_tags):
        nd1, nd2 = ops.eleNodes(ele_tag)
        Eds[step, el_i, :] = [ops.nodeDisp(nd1)[0],
                              ops.nodeDisp(nd1)[1],
                              ops.nodeDisp(nd1)[2],
                              ops.nodeDisp(nd2)[0],
                              ops.nodeDisp(nd2)[1],
                              ops.nodeDisp(nd2)[2]]

# 1. animate the deformed shape
anim = opsv.anim_defo(Eds, timeV, sfac_a, interpFlag=1, xlim=[-1, 7],
                      ylim=[-1, 5], fig_wi_he=(30., 22.))

plt.show()

# 2. after closing the window, animate the specified mode shape
eigVals = ops.eigen(5)

modeNo = 2  # specify which mode to animate
f_modeNo = np.sqrt(eigVals[modeNo-1])/(2*np.pi)  # i-th natural frequency

anim = opsv.anim_mode(modeNo, interpFlag=1, xlim=[-1, 7], ylim=[-1, 5],
                      fig_wi_he=(30., 22.))
plt.title(f'Mode {modeNo}, f_{modeNo}: {f_modeNo:.3f} Hz')

plt.show()
```

Index

P

R

S

T

U

V

W