STATIC ANALYSIS

OpenSeesPy Tutorial 2

TUTORIAL 2. INTRODUCTION

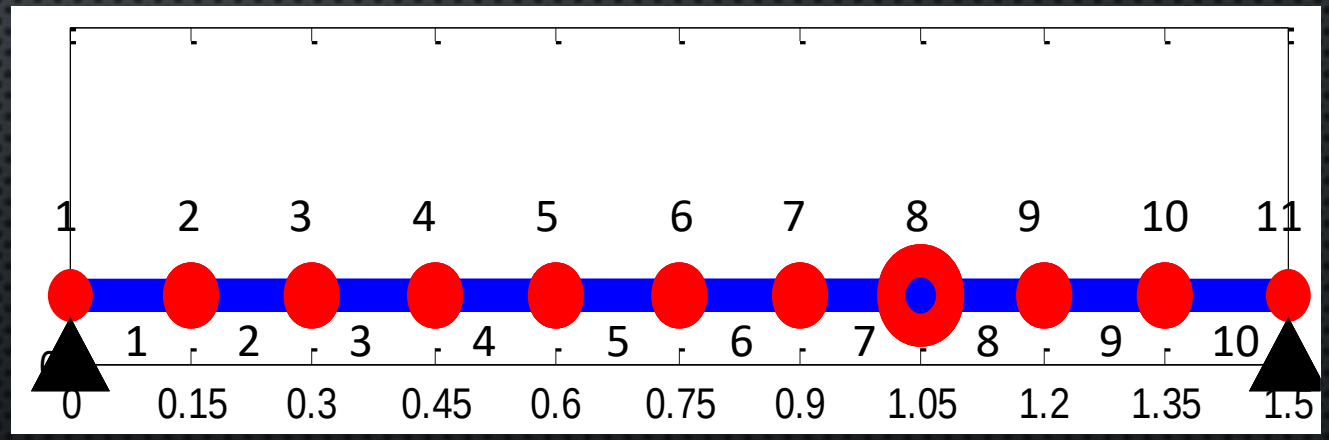
In this tutorial, we continued our series on OpenSeesPy with a static analysis that builds upon the modal analysis we performed in the previous tutorial. Our goal with this series of tutorials is to provide a clear understanding of the fundamental framework of OpenSeesPy.

This example is straightforward enough to grasp fundamental concepts in structural engineering, and it also provides the opportunity to dive deeper into more complex analyses, as you will discover in subsequent tutorials.

CASE OF STUDY: DYNAMIC ANALYSIS OF A SHAFT WITH MASS IMBALANCE

The figure illustrates the model of a rotodynamic equipment consisting of a steel shaft with a diameter of ½ inch and a length of 1.5 meters, along with a 40 kg mass disk attached to it. The rotor is supported at its ends by infinitely rigid simple supports.

For the static analysis consider the value of gravity as 10 m/s^2.



CASE OF STUDY: DYNAMIC ANALYSIS OF A SHAFT WITH MASS IMBALANCE (CONTINUED)

Length (L): 1.5 m

Diameter (D): 0.5 inches

Mass (m): 40 kg

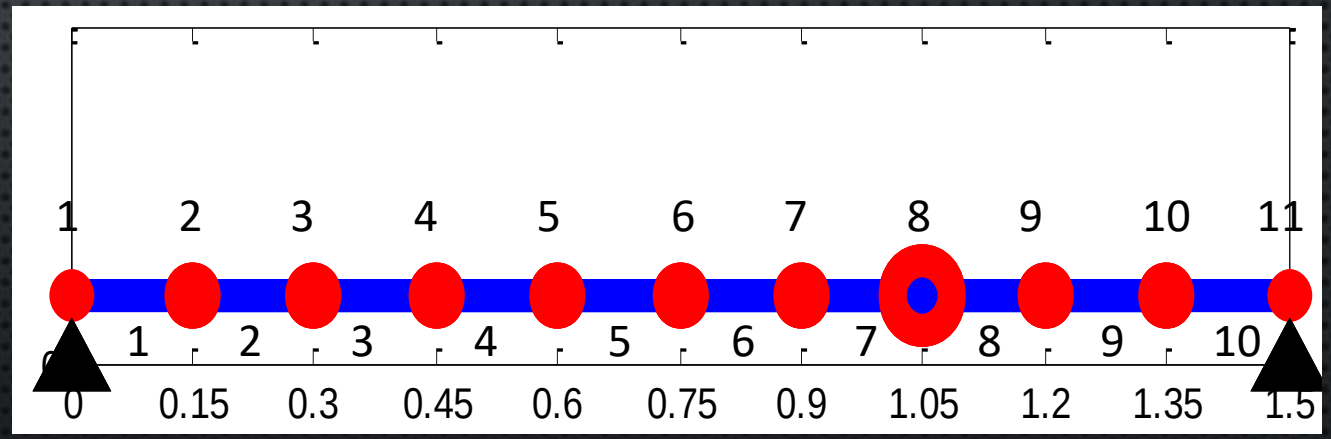
Density (rho): 7850 kg/m³

Young's Modulus (E): 2.1 GPa

Simply supported shaft.

Mass excess at 0.7L from left.

g = 10 m/s^2



WIPE COMMAND

```python
import openseespy.opensees as op

op.wipe()
# General model definition (2 simensions and 3 deegrees of freedom)
op.model('Basic', '-ndm', 2, '-ndf', 3)

############################ PHYSICAL PROPERTIES ##############################
L = 1.5                  # (m)
D = 0.5 * 0.0254         # (m)
m = 40                   # (kg)
import math
pi = math.acos(-1.0)
E = 2.1e11               # (Pa)
ro = 7850                # (kg/m^3)
A = 0.25 * pi * D**2     # (m^2)
Iz = pi * D**4 / 64      # (m^4)
transTag = 1             # (kg)

############################### DEFINE NODES ##################################
Nmax = 11
for i in range(1, Nmax+1):
    op.node(i, (i - 1) * L / (Nmax - 1), 0, 0)

############################## DOF CONSTRAINTS ################################
op.fix(   1, 1, 1, 0)
op.fix(Nmax, 1, 1, 0)

###################### ADDITION OF CONCENTRATION MASS #########################
node_loaded = 8
op.mass(node_loaded, 0.0, m, 0.0)
op.geomTransf('Linear', transTag)
```

The wipe command ensures that **all OpenSeesPy variables** and previously defined commands are no longer active in the current model.

As a common practice to prevent interference from other analyses, it is recommended to initiate the model with this command.

MODEL DEFINITION

```
import openseespy.opensees as op

op.wipe()
# General model definition (2 dimensions and 3 deegrees of freedom)
op.model('Basic', '-ndm', 2, '-ndf', 3)

############################ PHYSICAL PROPERTIES ##############################
L = 1.5                  # (m)
D = 0.5 * 0.0254         # (m)
m = 40                   # (kg)
import math
pi = math.acos(-1.0)
E = 2.1e11               # (Pa)
ro = 7850                # (kg/m^3)
A = 0.25 * pi * D**2     # (m^2)
Iz = pi * D**4 / 64      # (m^4)
transTag = 1             # (kg)

############################## DEFINE NODES ###################################
Nmax = 11
for i in range(1, Nmax+1):
    op.node(i, (i - 1) * L / (Nmax - 1), 0, 0)

############################## DOF CONSTRAINTS ################################
op.fix(   1, 1, 1, 0)
op.fix(Nmax, 1, 1, 0)

###################### ADDITION OF CONCENTRATION MASS #########################
node_loaded = 8
op.mass(node_loaded, 0.0, m, 0.0)
op.geomTransf('Linear', transTag)
```

The model command is a necessary step to define the model domain. The syntax for this command is as follows:

model('basic', '-ndm', ndm, '-ndf', ndf)

where:

- 'basic' represents the model type.
- '-ndm' denotes the number of dimensions.
- ndm is an integer representing the dimension.
- '-ndf' indicates the number of degrees of freedom.
- ndf is the integer representing the number of degrees of freedom.

It is important to note that ndf is an optional argument, if not specified, its default is the same as ndm.

GEOMETRY VARIABLES AND MATERIAL ATTRIBUTES

```python
import openseespy.opensees as op

op.wipe()
# General model definition (2 simensions and 3 deegrees of freedom)
op.model('Basic', '-ndm', 2, '-ndf', 3)

############################## PHYSICAL PROPERTIES ################################
# Shaft length
L = 1.5                  # (m)
# Shaft diameter
D = 0.5 * 0.0254         # (m)
# Lumped mass at 0.7L from the left end
m = 40                   # (kg)
import math
pi = math.acos(-1.0)
# Young´s Modulus
E = 2.1e11               # (Pa)
# Shaft Density
ro = 7850                # (kg/m^3)
# Shaft cross-section area
A = 0.25 * pi * D**2     # (m^2)
# Shaft cross-section second moment of Inertia
Iz = pi * D**4 / 64      # (m^4)

################################ DEFINE NODES #####################################
# Total number of nodes
Nmax = 11
for i in range(1, Nmax + 1):
    op.node(i, (i - 1) * L / (Nmax - 1), 0)

############################### DOF CONSTRAINTS ###################################
op.fix(   1, 1, 1, 0)
op.fix(Nmax, 1, 1, 0)
```

For complex models or those with a significant number of nodes and elements, it is good practice to define variables containing geometry and material attributes. This helps both, you and others understand the analysis more effectively.

Even though this analysis is relatively simple, we have defined the physical properties of the model as provided in the problem description.

NODE DEFINITION

```
import openseespy.opensees as op

op.wipe()
# General model definition (2 simensions and 3 deegrees of freedom)
op.model('Basic', '-ndm', 2, '-ndf', 3)

############################ PHYSICAL PROPERTIES ################################
# Shaft length
L = 1.5                 # (m)
# Shaft diameter
D = 0.5 * 0.0254        # (m)
# Lumped mass at 0.7L from the left end
m = 40                  # (kg)
import math
pi = math.acos(-1.0)
# Young´s Modulus
E = 2.1e11               # (Pa)
# Shaft Density
ro = 7850                # (kg/m^3)
# Shaft cross-section area
A = 0.25 * pi * D**2     # (m^2)
# Shaft cross-section second moment of Inertia
Iz = pi * D**4 / 64      # (m^4)

################################ DEFINE NODES ###################################
# Total number of nodes
Nmax = 11
for i in range(1, Nmax + 1):
    op.node(i, (i - 1) * L / (Nmax - 1), 0)

############################## DOF CONSTRAINTS ##################################
op.fix(   1, 1, 1, 0)
op.fix(Nmax, 1, 1, 0)
```

After defining the model, the next critical step is to define the nodes. This is accomplished using the node command, which follows this syntax:

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

Where:

- nodeTag is a **unique** identifier for each node.
- *crds represents the spatial coordinates of the nodes (consistent with the dimension defined in the model command).
- The remaining command options are optional but refer to nodal properties, including nodal degree of freedom ('-ndf', ndf), nodal mass ('-mass', mass´), nodal displacement ('-disp', *disp), nodal velocity ('vel', *vel) and nodal acceleration ('-accel', *accel).

RESTRAINTS

```python
import openseespy.opensees as op

op.wipe()
# General model definition (2 simensions and 3 deegrees of freedom)
op.model('Basic', '-ndm', 2, '-ndf', 3)

############################ PHYSICAL PROPERTIES ################################
# Shaft length
L = 1.5                 # (m)
# Shaft diameter
D = 0.5 * 0.0254        # (m)
# Lumped mass at 0.7L from the left end
m = 40                  # (kg)
import math
pi = math.acos(-1.0)
# Young´s Modulus
E = 2.1e11               # (Pa)
# Shaft Density
ro = 7850                # (kg/m^3)
# Shaft cross-section area
A = 0.25 * pi * D**2     # (m^2)
# Shaft cross-section second moment of Inertia
Iz = pi * D**4 / 64      # (m^4)

############################### DEFINE NODES ####################################
# Total number of nodes
Nmax = 11
for i in range(1, Nmax + 1):
    op.node(i, (i - 1) * L / (Nmax - 1), 0)

############################### DOF CONSTRAINTS #################################
op.fix(   1, 1, 1, 0)
op.fix(Nmax, 1, 1, 0)
```

The fix command is used to impose constraints on a specific node in the model. The syntax for this command is as follows:

fix(nodeTag, *constrValues)

Where:

- Node tag represents the node's unique identifier, which was defined during node creation.
- *constrValues are Boolean constraint values (0 for free degrees of freedom (dof) and 1 for fixed dof).

In this example, only the initial node (i=1) and the last node (i = Nmax) are restrained. This means that both nodes have fixed displacements in the x and y directions, while allowing free rotation about the z-axis.

LUMPED MASS

```
###################### ADDITION OF CONCENTRATION MASS ##########################
node_loaded = 8
op.mass(node_loaded, 0.0, m, 0.0)

############################## DEFINE ELEMENTS ################################
# Geometric transformation Tag
transTag = 1
op.geomTransf('Linear', transTag)
for index in range(1, Nmax):
    op.element('elasticBeamColumn', index, *[index, index + 1], A, E, Iz,
               transTag, '-mass', ro*A)

########################### EIGENVALUES CALCULATION ###########################
# number of eigenvalues to calculate
eigenN = 6
# list containing lamda contaiing the first eigenN eigenvalues
lamda = op.eigen('-fullGenLapack', eigenN)    # (rad^2/s^2)
# list containing the angular frequencies of the system
freq_Ang = []                                 # (rad/s)
for eigenvalue in lamda:
    freq_Ang.append(eigenvalue**0.5)


# import openseespy.postprocessing.ops_vis as ops
import openseespy.postprocessing.Get_Rendering as ops

ops.plot_modeshape(1, 300)
ops.plot_modeshape(2, 300)
ops.plot_modeshape(3, 300)
ops.plot_modeshape(4, 300)
ops.plot_modeshape(5, 300)
ops.plot_modeshape(6, 300)
```

The mass command is necessary in this example due to the presence of a concentrated mass at one of the nodes. However, distributed mass related to the density of the shaft, will be defined along with the elements, as we will demonstrate later.

The syntax for the mass command is as follow:

mass(nodeTag, *massValues)

Where:

- node Tag refers to the corresponding node ID.
- *massValues are the values of the mass in the respective degree of freedom.

GEOMETRIC TRANSFORMATION

```
##################### ADDITION OF CONCENTRATION MASS ##########################
node_loaded = 8
op.mass(node_loaded, 0.0, m, 0.0)

############################## DEFINE ELEMENTS #################################
# Geometric transformation Tag
transTag = 1
op.geomTransf('Linear', transTag)
    op.element('elasticBeamColumn', index, *[index, index + 1], A, E, Iz,
               transTag, '-mass', ro*A)

########################### EIGENVALUES CALCULATION ###########################
# number of eigenvalues to calculate
eigenN = 6
# list containing lamda contaiing the first eigenN eigenvalues
lamda = op.eigen('-fullGenLapack', eigenN)     # (rad^2/s^2)
# list containing the angular frequencies of the system
freq_Ang = []                                  # (rad/s)
for eigenvalue in lamda:
        op.node(i, (i - 1) * L / (Nmax - 1), 0)


# import openseespy.postprocessing.ops_vis as ops
import openseespy.postprocessing.Get_Rendering as ops

ops.plot_modeshape(1, 300)
ops.plot_modeshape(2, 300)
ops.plot_modeshape(3, 300)
ops.plot_modeshape(4, 300)
ops.plot_modeshape(5, 300)
ops.plot_modeshape(6, 300)
```

Since we plan to use an elastic beam element to model the shaft, a geometric transformation is necessary to convert the local coordinate system to the global coordinate system. The syntax for this command is as follows:

geoTrasnf(transfType, transTag, *trasnfArgs)

Where:

- transfType represents the type of transformation (Linear, Pdelta or corotational)
- trasnfTag is a unique transformation ID
- transfArgs is a list of arguments for the geometric transformation.

GEOMETRIC TRANSFORMATION (CONTINUED)

```
##################### ADDITION OF CONCENTRATION MASS ##########################
node_loaded = 8
op.mass(node_loaded, 0.0, m, 0.0)

############################## DEFINE ELEMENTS #################################
# Geometric transformation Tag
transTag = 1
op.geomTransf('Linear', transTag)
for index in range(1, Nmax):
    op.element('elasticBeamColumn', index, *[index, index + 1], A, E, Iz,
               transTag, '-mass', ro*A)

########################### EIGENVALUES CALCULATION ###########################
# number of eigenvalues to calculate
eigenN = 6
# list containing lamda contaiing the first eigenN eigenvalues
lamda = op.eigen('-fullGenLapack', eigenN)     # (rad^2/s^2)
# list containing the angular frequencies of the system
freq_Ang = []                                  # (rad/s)
for eigenvalue in lamda:
    freq_Ang.append(eigenvalue**0.5)
        op.node(i, (i - 1) * L / (Nmax - 1), 0)


# import openseespy.postprocessing.ops_vis as ops
import openseespy.postprocessing.Get_Rendering as ops

ops.plot_modeshape(1, 300)
ops.plot_modeshape(2, 300)
ops.plot_modeshape(3, 300)
ops.plot_modeshape(4, 300)
ops.plot_modeshape(5, 300)
ops.plot_modeshape(6, 300)
```

In this example, a linear transformation was used. When using the geoTransf command, the general format resembles the following:

geoTransf('Linear', tranfTag, *vecxz, '-jntOffset', *dI, *dJ)

Where:
Vecxz represents the x, y and z components of the vector used to define the local x-z plane of the local coordinate system (applicable to 3D beam element)
dI and dJ are joint offset values (optional in this context but required for 3D models)

ELEMENT DEFINITION

```
##################### ADDITION OF CONCENTRATION MASS ##########################
node_loaded = 8
op.mass(node_loaded, 0.0, m, 0.0)

############################## DEFINE ELEMENTS #################################
# Geometric transformation Tag
transTag = 1
op.geomTransf('Linear', transTag)
for index in range(1, Nmax):
    op.element('elasticBeamColumn', index, *[index, index + 1], A, E, Iz,
               transTag, '-mass', ro*A)

############################## EIGENVALUES CALCULATION ###########################
# number of eigenvalues to calculate
eigenN = 6
# list containing lamda contaiing the first eigenN eigenvalues
lamda = op.eigen('-fullGenLapack', eigenN)     # (rad^2/s^2)
# list containing the angular frequencies of the system
freq_Ang = []                                  # (rad/s)
for eigenvalue in lamda:
    freq_Ang.append(eigenvalue**0.5)


# import openseespy.postprocessing.ops_vis as ops
import openseespy.postprocessing.Get_Rendering as ops

ops.plot_modeshape(1, 300)
ops.plot_modeshape(2, 300)
ops.plot_modeshape(3, 300)
ops.plot_modeshape(4, 300)
ops.plot_modeshape(5, 300)
ops.plot_modeshape(6, 300)
```

The preceding steps were essential to define the element. In this example, we have defined elastic BeamColumn elements. However, it is worth noticing that the list of elements can vary including zero-length, truss, joint, link elements, and more.

The syntax used for the elasticBeamColumn is described as follows:

Element('elasticBeamColumn', eleTag, *eleNodes, Area, E_mod, Iz, transfTag, <'-mass', mass>, <'-cMass'>, <'-release', releaseCode>)

Where:

- eleTag refers to the element ID.
- *eleNodes are the two nodes that define the element.

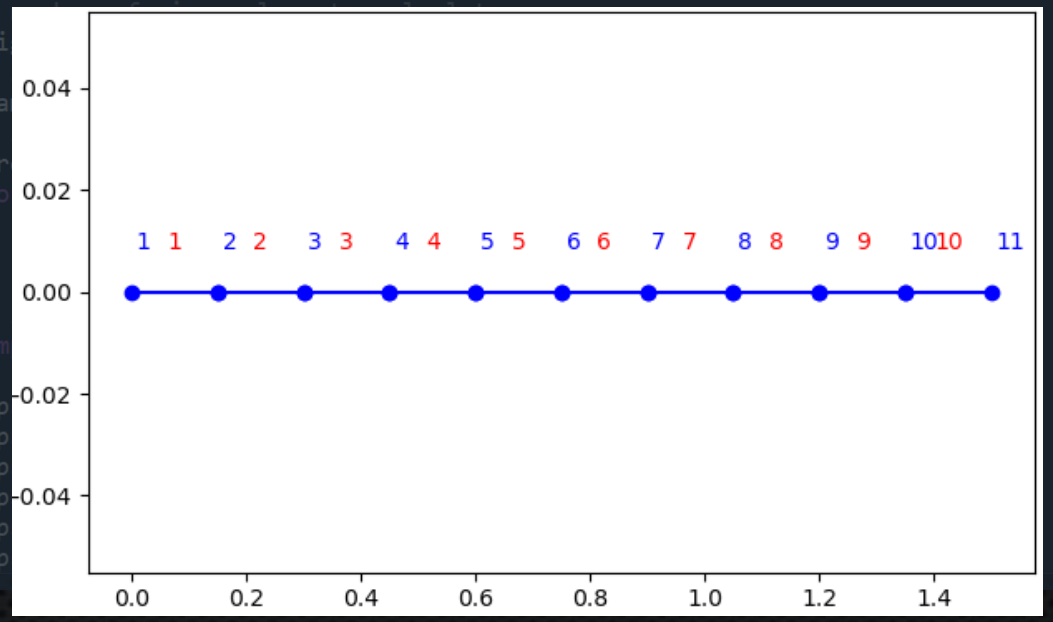
ELEMENT DEFINITION (CONTINUED)

```
##################### ADDITION OF CONCENTRATION MASS ##########################
node_loaded = 8
op.mass(node_loaded, 0.0, m, 0.0)

############################## DEFINE ELEMENTS #################################
# Geometric transformation Tag
transTag = 1
op.geomTransf('Linear', transTag)
for index in range(1, Nmax):
    op.element('elasticBeamColumn', index, *[index, index + 1], A, E, Iz,
               transTag, '-mass', ro*A)

########################### EIGENVALUES CALCULATION ###########################
# number of eigenvalues to calculate
eigenN = 6
# list containing lamda contaiing the first eigenN eigenvalues
lamda = op.eigen('-fullGenLapack', eigenN)    # (rad^2/s^2)
# list containing the angular frequencies of the system
freq_Ang = []                                 # (rad/s)
for eigenvalue in lamda:
    freq_Ang.append(eigenvalue**0.5)


# import openseespy.postprocessing.ops_vis as ops
import openseespy.postprocessing.Get_Rendering as ops

ops.plot_modeshape(1, 300)
ops.plot_modeshape(2, 300)
ops.plot_modeshape(3, 300)
ops.plot_modeshape(4, 300)
ops.plot_modeshape(5, 300)
ops.plot_modeshape(6, 300)
```

- Area, E_mod and Iz are the cross sectional area, Young's modulus and second moment of Inertia, respectively.
- transfTag is the transformation tag.
- '-mass' is the identifier for the mass per unit length.
- mass is the value of the mass per unit length.
- Release is an optional value that represents the release conditions.

NODES & ELEMENTS VISUALIZATION

```
##################### ADDITION OF CONCENTRATION MASS ##########################
node_loaded = 8
op.mass(node_loaded, 0.0, m, 0.0)

############################## DEFINE ELEMENTS #################################
# Geometric transformation Tag
transTag = 1
op.geomTransf('Linear', transTag)
for index in range(1, Nmax):
    op.element('elasticBeamColumn', index, *[index, index + 1], A, E, Iz,
               transTag, '-mass', ro*A)
import openseespy.postprocessing.ops_vis as ops
ops.plot_model("nodes")
############################ EIGENVALUES CALCULATION ###########################
```



Now that we have already defined the elements and nodes in the model, it might be valuable to obtain a graphical representation to confirm whether the intended model definition was achieved as intended.

To achieve this, we can import the **openseespostprocessing** library. Then, we can use the plot_model() command to generate a graphical representation of the model.

EIGENVALUES CALCULATION

```
##################### ADDITION OF CONCENTRATION MASS ##########################
node_loaded = 8
op.mass(node_loaded, 0.0, m, 0.0)

############################## DEFINE ELEMENTS #################################
# Geometric transformation Tag
transTag = 1
op.geomTransf('Linear', transTag)
for index in range(1, Nmax):
    op.element('elasticBeamColumn', index, *[index, index + 1], A, E, Iz,
               transTag, '-mass', ro*A)
import openseespy.postprocessing.ops_vis as ops
ops.plot_model("nodes")
############################ EIGENVALUES CALCULATION ###########################
# number of eigenvalues to calculate
eigenN = 6
# list containing lamda contaiing the first eigenN eigenvalues
lamda = op.eigen('-fullGenLapack', eigenN)     # (rad^2/s^2)
# list containing the angular frequencies of the system
freq_Ang = []                                  # (rad/s)
for eigenvalue in lamda:
    freq_Ang.append(eigenvalue**0.5)


# import openseespy.postprocessing.ops_vis as ops
import openseespy.postprocessing.Get_Rendering as ops

ops.plot_modeshape(1, 300)
ops.plot_modeshape(2, 300)
ops.plot_modeshape(3, 300)
ops.plot_modeshape(4, 300)
ops.plot_modeshape(5, 300)
ops.plot_modeshape(6, 300)
```

After the elements are properly defined, we proceed to calculate the eigenvalues using the eigen command. This command, in a general form, is represented as follows:

eigen(solver, numEigenValues)

Where:

- solver is the type of solver to use (optional parameter), with two available types (‘-genBandArpack’ and ‘-fullGenLapack’).
- numEigenValues is the number of eigenvalues to calculate.

RECORDER COMMAND

```
############################ DEFINE ELEMENTS #################################
# Geometric transformation Tag
transTag = 1
op.geomTransf('Linear', transTag)
for index in range(1, Nmax):
    op.element('elasticBeamColumn', index, *[index, index + 1], A, E, Iz,
               transTag, '-mass', ro*A)
import openseespy.postprocessing.ops_vis as ops
ops.plot_model("nodes")
########################### EIGENVALUES CALCULATION ##########################
# number of eigenvalues to calculate
eigenN = 6
# list containing lamda contaiing the first eigenN eigenvalues
lamda = op.eigen('-fullGenLapack', eigenN)    # (rad^2/s^2)
# list containing the angular frequencies of the system
freq_Ang = []                                 # (rad/s)
for eigenvalue in lamda:
    freq_Ang.append(eigenvalue**0.5)
op.recorder('Node', '-file', 'mode1VMA.out', '-nodeRange', 1 , Nmax, '-dof',
            2, 'eigen 1')
# recorder eigenvalue 2
op.recorder('Node', '-file', 'mode2VMA.out', '-nodeRange', 1 , Nmax, '-dof',
            2, 'eigen 2')
# recorder eigenvalue 3
op.recorder('Node', '-file', 'mode3VMA.out', '-nodeRange', 1 , Nmax, '-dof',
            2, 'eigen 3')
# recorder eigenvalue 4
op.recorder('Node', '-file', 'mode4VMA.out', '-nodeRange', 1 , Nmax, '-dof',
            2, 'eigen 4')
# recorder eigenvalue 5
op.recorder('Node', '-file', 'mode5VMA.out', '-nodeRange', 1 , Nmax, '-dof',
            2, 'eigen 5')
# recorder eigenvalue 6
op.recorder('Node', '-file', 'mode6VMA.out', '-nodeRange', 1 , Nmax, '-dof',
            2, 'eigen 6')
```

To record eigenvector modes and other structures responses, you can use the recorder command. The syntax for this command is as follows:

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)

Where:

- 'Node' is the identifier for the recorder.
- '-file' is the identificatory referring the file.
- filename is the name of the file to which output is sent.
- nodeRange is the identifier for the range node defined by the startNode and endNode.

RECORDER COMMAND (CONTINUED)

```
for eigenvalue in lamda:
    freq_Ang.append(eigenvalue**0.5)

########################## RECORDING OF EIGGENVALUES ############################
# recorder eigenvalue 1
op.recorder('Node', '-file', 'mode1VMA.out', '-nodeRange', 1 , Nmax, '-dof',
            2, 'eigen 1')
# recorder eigenvalue 2
op.recorder('Node', '-file', 'mode2VMA.out', '-nodeRange', 1 , Nmax, '-dof',
            2, 'eigen 2')
# recorder eigenvalue 3
op.recorder('Node', '-file', 'mode3VMA.out', '-nodeRange', 1 , Nmax, '-dof',
            2, 'eigen 3')
# recorder eigenvalue 4
op.recorder('Node', '-file', 'mode4VMA.out', '-nodeRange', 1 , Nmax, '-dof',
            2, 'eigen 4')
# recorder eigenvalue 5
op.recorder('Node', '-file', 'mode5VMA.out', '-nodeRange', 1 , Nmax, '-dof',
            2, 'eigen 5')
# recorder eigenvalue 6
op.recorder('Node', '-file', 'mode6VMA.out', '-nodeRange', 1 , Nmax, '-dof',
            2, 'eigen 6')

op.record()
```

- ‘dof’ is the specified dof.
- respType is a string indicating the required response.
- nodeRange is the identifier for the range node defined by the startNode and endNode.
- ‘dof’ is the specified dof.
- respType is a string indicating the required response.

After defining the recorder using the recorder command, you can activate it using the record command as shown in the example.

TIMESERIES COMMAND

```
############################## ANALYSIS DEFINITION  ##############################
tsTag = 1
op.timeSeries('Constant', tsTag)
patternTag = 1
op.pattern('Plain', patternTag, tsTag)

op.eleLoad('-ele','-range', 1, Nmax-1, '-type', '-beamUniform',-ro*A*10)
op.load(8, 0.0, -m*10, 0.0)     # (N)

op.constraints('Plain')
op.numberer('RCM')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, 100)
op.algorithm('Linear')
op.integrator('LoadControl', 0.1)
op.analysis('Static')

op.analyze(10)

################################## PLOTTING ##################################
import numpy as np
import matplotlib.pyplot as plt
length = np.linspace(0, L, Nmax)

################################## SHAPE MODES ##################################
plt.figure()
for i in range(6):
    eigenMode = np.loadtxt("mode%sVMA.out"%(i+1))
    plt.plot(length, eigenMode)
plt.legend(['mode 1','mode 2','mode 3','mode 4','mode 5', 'mode 6'])
plt.show()
```

The timeSeries command is essential for constructing a timeseries object that defines the relationship between time(t) in the domain and the load factor(λ) applied to the loads within the associated load pattern, expressed as λ = F(t)

The syntax for using the timeSeries command is as follows:

timeSeries(tsType, tsTag, *tsArgs)

Where:

- tsType represents the type of time series being defined.
- tsTag specifies the time series tag, providing a unique identifier.
- tsArgs is a list of time series arguments, depending on the specific tsType chosen.

PATTERN COMMAND

```
############################## ANALYSIS DEFINITION  ##############################
tsTag = 1
op.timeSeries('Constant', tsTag)
patternTag = 1
op.pattern('Plain', patternTag, tsTag)

op.eleLoad('-ele','-range', 1, Nmax-1, '-type', '-beamUniform',-ro*A*10)
op.load(8, 0.0, -m*10, 0.0)     # (N)

op.constraints('Plain')
op.numberer('RCM')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, 100)
op.algorithm('Linear')
op.integrator('LoadControl', 0.1)
op.analysis('Static')

op.analyze(10)

################################ PLOTTING ###################################
import numpy as np
import matplotlib.pyplot as plt
length = np.linspace(0, L, Nmax)

############################## SHAPE MODES ##################################
plt.figure()
for i in range(6):
    eigenMode = np.loadtxt("mode%sVMA.out"%(i+1))
    plt.plot(length, eigenMode)
plt.legend(['mode 1','mode 2','mode 3','mode 4','mode 5', 'mode 6'])
plt.show()
```

The pattern command is employed to create a load pattern and add it to the structural domain. Each load pattern is associated with a TimeSeries and may include Element Loads, Nodal Loads, and Single Point Constraints. The syntax for using the pattern command is as follows:

pattern(patternType, patternTag, *patternArgs)

where:

- patternType specifies the type of load pattern to be created.
- patternTag assigns a unique tag to the load pattern for identification.
- patternArgs is a list of pattern arguments.

In the example we are discussing, a plain pattern was used for defining both nodal and element loads.

ELELOAD COMMAND

```
############################ ANALYSIS DEFINITION  ############################
tsTag = 1
op.timeSeries('Constant', tsTag)
patternTag = 1
op.pattern('Plain', patternTag, tsTag)

op.eleLoad('-ele','-range', 1, Nmax-1, '-type', '-beamUniform',-ro*A*10)
op.load(8, 0.0, -m*10, 0.0)     # (N)

op.constraints('Plain')
op.numberer('RCM')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, 100)
op.algorithm('Linear')
op.integrator('LoadControl', 0.1)
op.analysis('Static')

op.analyze(10)

################################## PLOTTING ##################################
import numpy as np
import matplotlib.pyplot as plt
length = np.linspace(0, L, Nmax)

############################### SHAPE MODES ##################################
plt.figure()
for i in range(6):
    eigenMode = np.loadtxt("mode%sVMA.out"%(i+1))
    plt.plot(length, eigenMode)
plt.legend(['mode 1','mode 2','mode 3','mode 4','mode 5', 'mode 6'])
plt.show()
```

eleLoad() is used to create an element load object and associate it with a specific load pattern.
The syntax for using the eleLoad command is:

eleLoad(‘ele’, *eleTags, ‘-range’, eleTag1, eleTag2, ‘-type’, ‘-beamUniform’, Wy, <>, Wx=0.0, ‘beamPoint’, Py, <Pz>, xL, Px, ‘-beamThermal’, *tempPts)

Where:
‘-ele’ / eleTags refers to the tags of the elements to which the load will be applied.
‘-range’ refers to the range of elements to consider from EleTag1 to eleTag2.
‘-beamUniform’ for beam elements, this option allows you to apply uniform loads, with parameters like Ey and Wx.
‘-beamPoint’ for beam elements, this option lets you apply point loads with parameters like Py, Pz, xL and Px.

CONSTRAINTS COMMAND

```
############################ ANALYSIS DEFINITION  ############################
tsTag = 1
op.timeSeries('Constant', tsTag)
patternTag = 1
op.pattern('Plain', patternTag, tsTag)

op.eleLoad('-ele','-range', 1, Nmax-1, '-type', '-beamUniform',-ro*A*10)
op.load(8, 0.0, -m*10, 0.0)     # (N)

op.constraints('Plain')
op.numberer('RCM')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, 100)
op.algorithm('Linear')
op.integrator('LoadControl', 0.1)
op.analysis('Static')

op.analyze(10)
```

The constraints command defines how constraint equations are integrated into the analysis. OpenSeesPy provided several types of constraints, including:

- Plain: enforces homogeneous single-point constraints and constructs multi-point constraints where the constraint matrix is equal to the identity.
- Lagrange Multipliers: allows for more complex constraint formulations.
- Penalty Method: penalize violations of constraints in the system.
- Transformation Method: provides flexibility in constraint definition.

Here, the Plain Constraint type was used. This constraint enforces homogeneous single-points and constructs multi-point constraints where the constraint matrix is equal to the identity.

NUMBERER COMMAND

```
############################ ANALYSIS DEFINITION  ############################
tsTag = 1
op.timeSeries('Constant', tsTag)
patternTag = 1
op.pattern('Plain', patternTag, tsTag)

op.eleLoad('-ele','-range', 1, Nmax-1, '-type', '-beamUniform',-ro*A*10)
op.load(8, 0.0, -m*10, 0.0)     # (N)

op.constraints('Plain')
op.numberer('RCM')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, 100)
op.algorithm('Linear')
op.integrator('LoadControl', 0.1)
op.analysis('Static')

op.analyze(10)
```

The numberer command determines how degrees of freedom are assigned numbers.

The numbered can be different types:

- Plain: assigns degrees of freedom sequentially wy specific reordering.
- RDM: assigns degrees of freedom randomly.
- AMD: uses the approximate minimum degree algorithm to optimize the degree of freedom numbering.
- Parallel plain: sequential numbering for parallel analysis.
- Parallel RCM: uses the Reverse Cuthill-McKee algorithm for parallel analysis, which can lead to improved performance.

SYSTEM COMMAND

```
############################ ANALYSIS DEFINITION  ############################
tsTag = 1
op.timeSeries('Constant', tsTag)
patternTag = 1
op.pattern('Plain', patternTag, tsTag)

op.eleLoad('-ele','-range', 1, Nmax-1, '-type', '-beamUniform',-ro*A*10)
op.load(8, 0.0, -m*10, 0.0)     # (N)

op.constraints('Plain')
op.numberer('RCM')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, 100)
op.algorithm('Linear')
op.integrator('LoadControl', 0.1)
op.analysis('Static')

op.analyze(10)
```

The system command is responsible for constructing the LinearSOE (Linear System of Equations), and LinearSolver object, both of which are essential components for storing and solving the system of equations in a structural analysis.

The system command provides a range of alternatives ´for different problems and computational needs, including:
BandGeneral, BandSPD, ProfileSPD, SuperLU, UmfPack, FullGeneral, SparseSYM, PFEM and MUMPS.

In the example under discussion, the ProfileSPD type was used. ProfileSPD is designed for solving symmetric positive definite matrix systems, making is well-suited for a wide range of structural analysis problems.

TEST COMMAND

```
############################ ANALYSIS DEFINITION  ############################
tsTag = 1
op.timeSeries('Constant', tsTag)
patternTag = 1
op.pattern('Plain', patternTag, tsTag)

op.eleLoad('-ele','-range', 1, Nmax-1, '-type', '-beamUniform',-ro*A*10)
op.load(8, 0.0, -m*10, 0.0)     # (N)

op.constraints('Plain')
op.numberer('RCM')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, 100)
op.algorithm('Linear')
op.integrator('LoadControl', 0.1)
op.analysis('Static')

op.analyze(10)
```

The test command is used to control the convergence criteria in a structural analysis. It also plays a role in constructing the LinearSOE, and LinearSolver object to store and solve the system of equations.

The test can be different types to ensure accuracy and efficiency. These include:
NormBalance, NormDispIncr, energyIncr, RelativeNormUnbalance, and more.

In the example being discussed, the NormDispIncr test type was used. With the syntax:

test('NormDispIncr', tol, iter, pFlag=0, nType=2)

Where:

- tol is the tolerance criteria used for convergence.
- iter is the maximum number of iterations to check.
- Pflag and nType are optional parameters that represent the print flag and Type or norm, respectively.

ALGORITHM COMMAND

```
############################ ANALYSIS DEFINITION  ############################
tsTag = 1
op.timeSeries('Constant', tsTag)
patternTag = 1
op.pattern('Plain', patternTag, tsTag)

op.eleLoad('-ele','-range', 1, Nmax-1, '-type', '-beamUniform',-ro*A*10)
op.load(8, 0.0, -m*10, 0.0)     # (N)

op.constraints('Plain')
op.numberer('RCM')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, 100)
op.algorithm('Linear')
op.integrator('LoadControl', 0.1)
op.analysis('Static')

op.analyze(10)
```

The algorithm command plays a crucial role in the analysis process. It determines:

- The time t+dt.
- The tangent matrix and residual vector at any iteration.
- The corrective step is based on the displacement increment dU.

The algorithm offers a range of option to tailor the analysis to specific requirements, including:
Linear, Newton, NewtonLineSearch, ModifiedNewton, KrylovNewton, etc.

Due to the simplicity of the analysis, in this case it was assign the linear approach to the algorithm command.

INTEGRATOR COMMAND

```
############################ ANALYSIS DEFINITION  ############################
tsTag = 1
op.timeSeries('Constant', tsTag)
patternTag = 1
op.pattern('Plain', patternTag, tsTag)

op.eleLoad('-ele','-range', 1, Nmax-1, '-type', '-beamUniform',-ro*A*10)
op.load(8, 0.0, -m*10, 0.0)     # (N)

op.constraints('Plain')
op.numberer('RCM')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, 100)
op.algorithm('Linear')
op.integrator('LoadControl', 0.1)
op.analysis('Static')

op.analyze(10)
```

The integrator command is used to determine the sequence of steps taken to solve the nonlinear equations during structural analyses.

The integrator command provides various types to accommodate different analysis scenarios, including:

- LoadControl: Controls the analysis by varying applied loads.
- DisplacementControl: Controls the analysis by adjusting displacements.
- Parallel DisplacementControl: Allows for parallel control of displacements.
- Minimum Unbalanced Displacement: Minimizes unbalanced displacements to achieve convergence.
- Arc-Length Control: Utilizes arc-length methods for load-displacement path tracing.

For a transient analysis, there are more options available.

ANALYSIS COMMAND

```
############################ ANALYSIS DEFINITION  ############################
tsTag = 1
op.timeSeries('Constant', tsTag)
patternTag = 1
op.pattern('Plain', patternTag, tsTag)

op.eleLoad('-ele','-range', 1, Nmax-1, '-type', '-beamUniform',-ro*A*10)
op.load(8, 0.0, -m*10, 0.0)     # (N)

op.constraints('Plain')
op.numberer('RCM')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, 100)
op.algorithm('Linear')
op.integrator('LoadControl', 0.1)
op.analysis('Static')

op.analyze(10)
```

The analysis command is used to specify the type of analysis to perform within the analysis process. The syntax for using the analysis command is as follows:

analysis(analysisType)

Where the analysisType parameter can take on various approaches, including:

- 'Static' used for static structural analyses, where loads are constant.
- 'Transient' employed for transient dynamic analysis, considering time-dependent loading and behavior over time.
- 'VariableTransient' is suitable for variable transient analysis, allowing for the modeling of changing parameters over time.
- 'PFEM' (Particle Finite Element Method): Used when modeling problems with fluid-structure interaction or particle-based simulations.

ANALYZE COMMAND

```
############################ ANALYSIS DEFINITION  ############################
tsTag = 1
op.timeSeries('Constant', tsTag)
patternTag = 1
op.pattern('Plain', patternTag, tsTag)

op.eleLoad('-ele','-range', 1, Nmax-1, '-type', '-beamUniform',-ro*A*10)
op.load(8, 0.0, -m*10, 0.0)     # (N)

op.constraints('Plain')
op.numberer('RCM')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, 100)
op.algorithm('Linear')
op.integrator('LoadControl', 0.1)
op.analysis('Static')

op.analyze(10)
```

The analyze command is responsible for executing the structural analysis. It allows you to define various parameters to control the analysis process. The syntax is:

analyze(numIncr, dt, dtMin, dtMax, Jd)

Where the analysis type can be:

- numIncr specifies the number of analysis steps to perform.
- dt sets the time step increment (required for transient analysis).
- dtMin represents the minimum time allowed.
- dtMax specifies the maximum time steps allowed.
- Jd indicates the number of iterations performed at each step.

Notice that the last 3 parameters (dtMin, dtMax, Jd) are required for VariableTransient analysis

GRAPHIC REPRESENTATION OF RESULTS

```
################################ PLOTTING ###################################
import numpy as np
import matplotlib.pyplot as plt
length = np.linspace(0, L, Nmax)

############################## SHAPE MODES ##################################
plt.figure()
for i in range(6):
    eigenMode = np.loadtxt("mode%sVMA.out"%(i+1))
    plt.plot(length, eigenMode)
plt.legend(['mode 1','mode 2','mode 3','mode 4','mode 5', 'mode 6'])
plt.show()
```

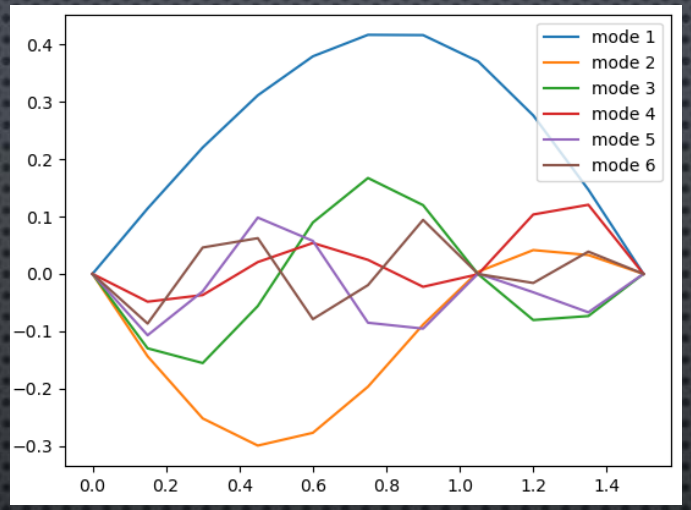
```
############################## STATIC RESPONSE ##############################
plt.figure()
ops.plot_defo()

staticDisplacement = []
for i in range(1, Nmax + 1):
    disp = op.nodeDisp(i)
    staticDisplacement.append(disp[1])
plt.figure()
plt.plot(length, staticDisplacement, marker='o')
plt.xlabel('length (m)')
plt.ylabel('y Displacement (m)')
plt.show()

op.wipe()
```

Once the analysis is completed, it was proceeded to visualize and the results.
In the upper part of the slide, we detailed how to plot and visualize modes that were previously recorded during the analysis.

In the lower part, we demonstrate how to obtain and visualize static deformations by using:

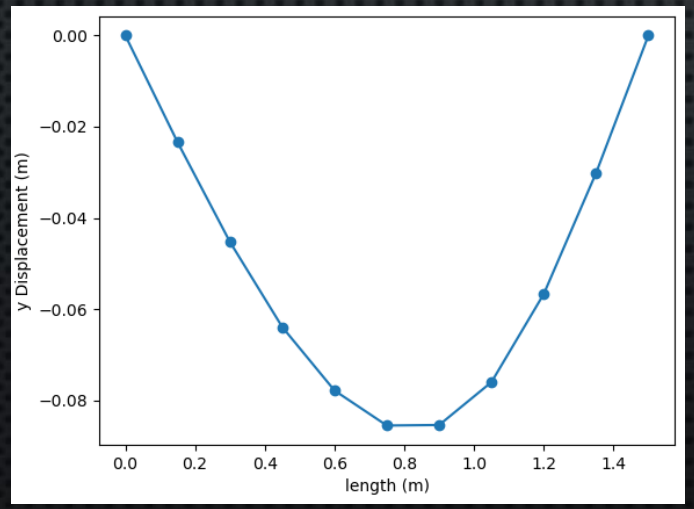
- plot_defo() command.
- creating a staticDisplacement vector plot making a plot with it.

EIGENMODES VISUALIZATION AND STATIC DEFORMATION

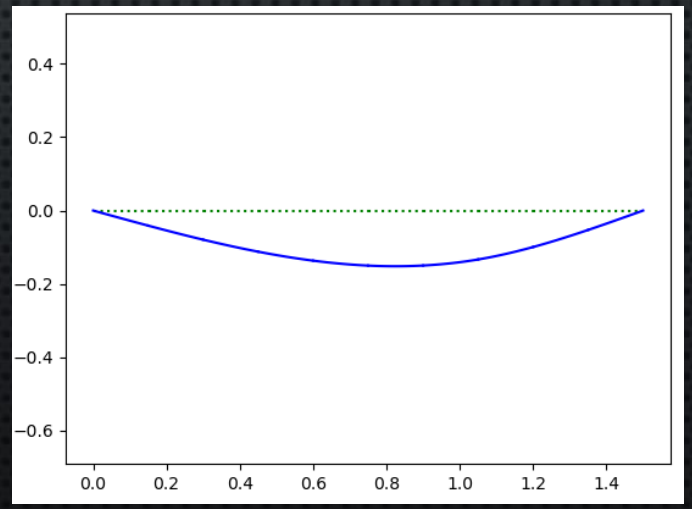


First 6 Eigenmodes

Static deformation



Static deformation



REFERENCES

For more detailed information about these commands, you can refer to the official documentation at:

https://openseespydoc.readthedocs.io/en/latest/index.html