

OPENSEESPY TUTORIAL

STATIC ANALYSIS

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

Length (L): 1.5 m

Diameter (D): 0.5 inches

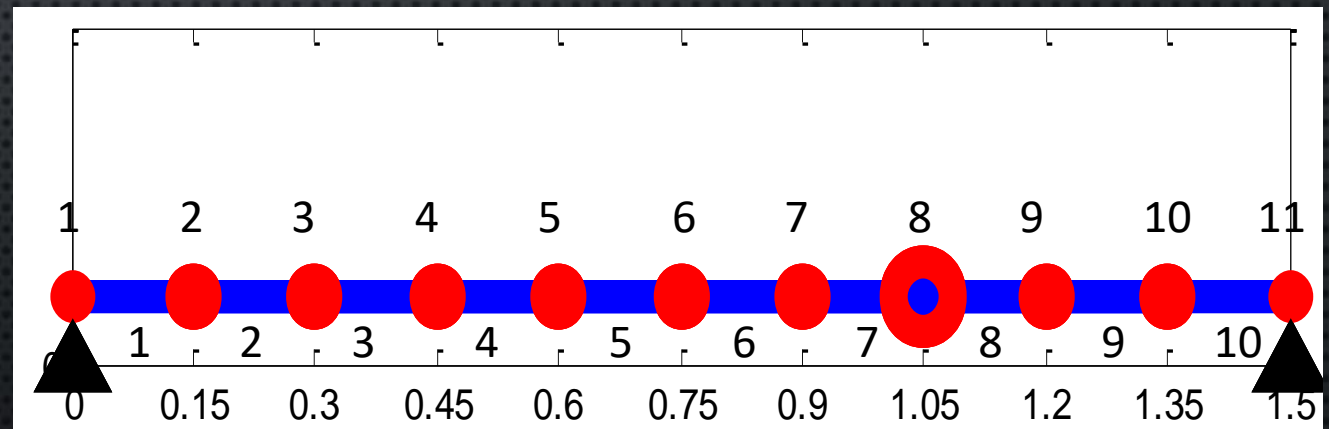
Mass (m): 40 kg

Density (ρ): 7850 kg/m³

Young's Modulus (E): 2.1 GPa

Simply supported shaft.

Mass excess at 0.7L from left.



MODEL DEFINITION

```
import openseespy.opensees as op
op.wipe()
# General model definition (2 dimensions and 3 degrees 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)
```

`wipe()` command ensures that **all openseespy variables** and previously defined commands are no longer active in the current model.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 degrees 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)
rho = 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 follow:

`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

```
import openseespy.opensees as op

op.wipe()
# General model definition (2 dimensions and 3 degrees 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)

##### 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)

transTag = 1 # (kg)
op.geomTransf('Linear', transTag)
```

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

Even though this analysis is relative 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 dimensions and 3 degrees 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)

##### DEFINE 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)

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

transTag = 1          # (kg)
op.geomTransf('Linear', transTag)
```

After defining the model, the next crucial 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)

NODE DEFINITION

```
import openseespy.opensees as op

op.wipe()
# General model definition (2 dimensions and 3 degrees 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)

##### 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 CONCENTRATED MASS #####
node_loaded = 8
op.mass(node_loaded, 0.0, m, 0.0)

transTag = 1           # (kg)
op.geomTransf('Linear', transTag)
```

The node definitions were accomplished iteratively using a for command. It's important to note that this approach is equivalent to defining each node individually, as illustrated in the figure.

Furthermore, it is worth mentioning that, in this 2D model (as specified in the model command), only two coordinates are required for each node.

```
op.node(1, 0.0, 0)
op.node(2, 0.15, 0)
op.node(3, 0.3, 0)
op.node(4, 0.45, 0)
op.node(5, 0.6, 0)
op.node(6, 0.75, 0)
op.node(7, 0.9, 0)
op.node(8, 1.05, 0)
op.node(9, 1.2, 0)
op.node(10, 1.35, 0)
op.node(11, 1.5, 0)
```

RESTRAINTS

```
import openseespy.opensees as op

op.wipe()
# General model definition (2 dimensions and 3 degrees 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)

##### DEFINE 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)

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

transTag = 1 # (kg)
op.geomTransf('Linear', transTag)
```

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

```
import openseespy.opensees as op

op.wipe()
# General model definition (2 dimensions and 3 degrees 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)

##### DEFINE 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)

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

transTag = 1 # (kg)
op.geomTransf('Linear', transTag)
```

The definition of mass is necessary in this example due to the presence of a concentrator mass at one of the nodes. However, for distributed mass related to the density of the shaft, it 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

```
import openseespy.opensees as op

op.wipe()
# General model definition (2 dimensions and 3 degrees 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)

##### DEFINE 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)

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

transTag = 1          # (kg)
op.geomTransf('Linear', transTag)
```

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)
- `transfTag` is a unique transformation ID
- `transfArgs` is a list of arguments for the geometric transformation.

GEOMETRIC TRANSFORMATION

```
import openseespy.opensees as op

op.wipe()
# General model definition (2 dimensions and 3 degrees 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)

##### DEFINE 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)

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

transTag = 1          # (kg)
op.geomTransf('Linear', transTag)
```

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

`geoTransf('Linear', tranfTag, *vecxz, '-jntOffset', *dl, *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)
dl and dJ are joint offset values (optional in this context but required for 3D models)

ELEMENT DEFINITION

```
##### DEFINE ELEMENTS #####
for index in range(1, Nmax):
    op.element('eElasticBeamColumn', index, *[index, index+1], A, E, Iz,
              transTag, -mass, ro*A)
import openseespy.postprocessing.ops_vis as ops
ops.plot_model("nodes")

##### EIGENVALUES CALCULATION #####

lamda = op.eigen('-fullGenLapack', 6)
# op.record()
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)
```

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

The syntax for the used for the elasticBeamColumn is described as follow:

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

```
##### DEFINE ELEMENTS #####
for index in range(1, Nmax):
    op.element('eElasticBeamColumn', index, *[index, index+1], A, E, Iz,
              transTag, -mass, ro*A)
import openseespy.postprocessing.ops_vis as ops
ops.plot_model("nodes")

##### EIGENVALUES CALCULATION #####

lamda = op.eigen('-fullGenLapack', 6)
# op.record()
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)
```

- Area, E_mod and Iz are the cross sectional area, Young's modulus and second moment of Inertia, respectively.
- transTag 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 represent the release conditions.

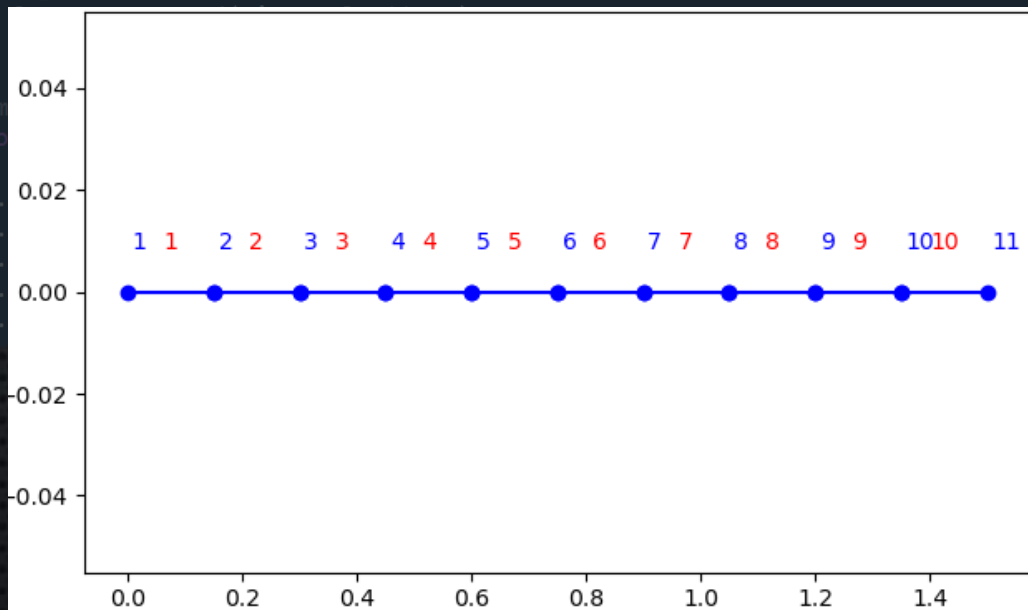
NODES & ELEMENTS VISUALIZATION

```
##### DEFINE ELEMENTS #####
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 #####
```

```
lamda = op.eigen('-fullGenLapack', 6)
# op.record()
freq_Ang = [] # (rad/s)
for eigenvalue in lamda:
```

```
# in
imp
ops.
ops.
ops.
ops.
ops.
```



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

```
##### DEFINE ELEMENTS #####
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 #####

lamda = op.eigen('-fullGenLapack', 6)
# op.record()
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)
```

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 follow:

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.

EIGENVALUES CALCULATION AND MODES RECORDING

```
##### DEFINE ELEMENTS #####
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")

##### CALCULATION OF EIGENVALUES #####
lamda = op.eigen('-fullGenLapack', 6)
freq_Ang = [] # (rad/s)
for eigenvalue in lamda:
    freq_Ang.append(eigenvalue**0.5)

##### RECORDING OF EIGENVALUES #####
# 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()
```

Once we have obtained the eigenvalues, we proceed to calculate the angular frequencies.

For visualizing the eigenmodes, it can be used the recorder command, with the following syntax:

`recorder(recorderType, *recorderArgs)`

Where:

- `recorderType` correspond to the type of the recorder
- `recorderArgs` a list of recorder arguments.

RECORDER COMMAND

```
##### DEFINE ELEMENTS #####
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")

##### CALCULATION OF EIGENVALUES #####
lamda = op.eigen('-fullGenLapack', 6)
freq_Ang = [] # (rad/s)
for eigenvalue in lamda:
    freq_Ang.append(eigenvalue**0.5)

##### RECORDING OF EIGENVALUES #####
# 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()
```

In order to

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
- '-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
- 'dof' is the specified dof
- respType is a string indicating the required

RECORDER COMMAND

```
##### DEFINE ELEMENTS #####
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")

##### CALCULATION OF EIGENVALUES #####
lamda = op.eigen('-fullGenLapack', 6)
freq_Ang = [] # (rad/s)
for eigenvalue in lamda:
    freq_Ang.append(eigenvalue**0.5)

##### RECORDING OF EIGENVALUES #####
# 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()
```

- 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.

Notice that it were defined only the some of the arguments (the ones required). However, in the opensseespy documentation, more details of the rest of the terms are provided.

In addition, the record command was used to activety the recorder previously defined.

CONSTRAINTS COMMAND

```
##### ANALYSIS DEFINITION #####

op.timeSeries('Linear', 1)

op.pattern('Plain', 1, 1)

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('ProfileSPD')
op.test('NormDispIncr', 1e-6, 100)
op.algorithm('Newton')
op.integrator('LoadControl', 0.1)
op.analysis('Static')

op.analyze(10)

##### PLOTTING #####
import numpy as np
import matplotlib.pyplot as plt
f1 = np.linspace(0, L, Nmax)

##### SHAPE MODES #####
for i in range(6):
    V = np.loadtxt("mode%sVMA.out"%(i+1))
    plt.plot(f1, V)
plt.legend(['mode 1','mode 2','mode 3','mode 4','mode 5', 'mode 6'])
plt.show()
ops.plot_model()
```

The constraints command defines how the constraint equations are incorporated into the analysis

OpenSeesPy provided several types of constraints, including:

- Plain: enforces homogeneous single-point constraints and construct 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.

The one used in this example was the plain constraint, that enforce homogeneous single points constraints and multi-point constraints are constructed where the constraint matrix is equal to the identity.

NUMBERER COMMAND

```
##### ANALYSIS DEFINITION #####

op.timeSeries('Linear', 1)
op.pattern('Plain', 1, 1)

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('ProfilesFD')
op.test('NormDispIncr', 1e-6, 100)
op.algorithm('Newton')
op.integrator('LoadControl', 0.1)
op.analysis('Static')

op.analyze(10)

##### PLOTTING #####
import numpy as np
import matplotlib.pyplot as plt
f1 = np.linspace(0, L, Nmax)

##### SHAPE MODES #####
for i in range(6):
    V = np.loadtxt("mode%sVMA.out"%(i+1))
    plt.plot(f1, V)
plt.legend(['mode 1', 'mode 2', 'mode 3', 'mode 4', 'mode 5', 'mode 6'])
plt.show()
ops.plot_model()
```

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

The numbered can be different types:

- Plain
- RDM
- AMD
- Parallel plain
- Parallel RCM

The used in this example was the RCM numberer. RCM stand for Reverse Cuthill-McKee.

SYSTEM COMMAND

```
##### ANALYSIS DEFINITION #####

op.timeSeries('Linear', 1)

op.pattern('Plain', 1, 1)

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('ProfileSPD')
op.test('NormDispIncr', 1e-6, 100)
op.algorithm('Newton')
op.integrator('LoadControl', 0.1)
op.analysis('Static')

op.analyze(10)

##### PLOTTING #####
import numpy as np
import matplotlib.pyplot as plt
f1 = np.linspace(0, L, Nmax)

##### SHAPE MODES #####
for i in range(6):
    V = np.loadtxt("mode%sVMA.out"%(i+1))
    plt.plot(f1, V)
plt.legend(['mode 1', 'mode 2', 'mode 3', 'mode 4', 'mode 5', 'mode 6'])
plt.show()
ops.plot_model()
```

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

The numbered can be different types:

BandGeneral, BandSPD, ProfileSPD, SuperLU, UmfPack, FullGeneral, SparseSYM, PFEM and MUMPS.

In the example we are discussing, the ProfileSPD type was used. ProfileSPD is designed for solving symmetric positive definite matrix system efficiently.

TEST COMMAND

```
##### ANALYSIS DEFINITION #####

op.timeSeries('Linear', 1)

op.pattern('Plain', 1, 1)

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('ProfileSPD')
op.test('NormDispIncr', 1e-6, 100)
op.algorithm('newton')
op.integrator('LoadControl', 0.1)
op.analysis('Static')

op.analyze(10)

##### PLOTTING #####
import numpy as np
import matplotlib.pyplot as plt
f1 = np.linspace(0, L, Nmax)

##### SHAPE MODES #####
for i in range(6):
    V = np.loadtxt("mode%sVMA.out"%(i+1))
    plt.plot(f1, V)
plt.legend(['mode 1', 'mode 2', 'mode 3', 'mode 4', 'mode 5', 'mode 6'])
plt.show()
ops.plot_model()
```

The test command is used to control the convergence criteria in a structural analysis. It constructs the LinearSOE, and LinearSolver object to store and solve the system of equation.

The test can be different types:

NormBalance, NormDispIncr, energyIncr, RelativeNormUnbalance, and more.

The NormDispIncr test type was used to evaluate convergence based on the norm of the left hand side solution vector of the matrix equation, providing a reliable measure to determine if convergence was reached.

ALGORITHM COMMAND

```
##### ANALYSIS DEFINITION #####

op.timeSeries('Linear', 1)

op.pattern('Plain', 1, 1)

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('ProfileSPD')
op.test('NormDispIncr', 1e-6, 100)
op.algorithm('Newton')
op.integrator('LoadControl', 0.1)
op.analysis('Static')

op.analyze(10)

##### PLOTTING #####
import numpy as np
import matplotlib.pyplot as plt
f1 = np.linspace(0, L, Nmax)

##### SHAPE MODES #####
for i in range(6):
    V = np.loadtxt("mode%sVMA.out"%(i+1))
    plt.plot(f1, V)
plt.legend(['mode 1', 'mode 2', 'mode 3', 'mode 4', 'mode 5', 'mode 6'])
plt.show()
ops.plot_model()
```

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 based on the displacement increment dU .

The test can be different types:

Linear, Newton, NewtonLineSearch, ModifiedNewton, KrylovNewton, etc.

The Newton type, as the name suggests, refers to Newton-Raphson method to solve the system of equations.

INTEGRATOR COMMAND

```
##### ANALYSIS DEFINITION #####

op.timeSeries('Linear', 1)

op.pattern('Plain', 1, 1)

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('ProfileSPD')
op.test('NormDispIncr', 1e-6, 100)
op.algorithm('Newton')
op.integrator('LoadControl', 0.1)
op.analysis('Static')

op.analyze(10)

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

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

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

The integrator command provides various types to suit 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.

ANALYSIS COMMAND

```
##### ANALYSIS DEFINITION #####

op.timeSeries('Linear', 1)
op.pattern('Plain', 1, 1)

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('ProfileSPD')
op.test('NormDispIncr', 1e-6, 100)
op.algorithm('Newton')
op.integrator('LoadControl', 0.1)
op.analysis('Static')

op.analyze(10)

##### PLOTTING #####
import numpy as np
import matplotlib.pyplot as plt
f1 = np.linspace(0, L, Nmax)

##### SHAPE MODES #####
for i in range(6):
    V = np.loadtxt("mode%sVMA.out"%(i+1))
    plt.plot(f1, V)
plt.legend(['mode 1', 'mode 2', 'mode 3', 'mode 4', 'mode 5', 'mode 6'])
plt.show()
ops.plot_model()
```

The analysis command is used to specify the type of analysis to perform defines the type of analysis to perform. The syntax is:

analysis(analysisType)

Where the analysis type can be:

- 'Static' Used for static structural analysis, where loads are constant and equilibrium is sought.
- 'Transient' Employed for transient dynamic analysis, considering time-dependent loading and behavior.
- 'VariableTransient' Suitable for variable transient analysis, accommodating 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 #####

op.timeSeries('Linear', 1)
op.pattern('Plain', 1, 1)

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('ProfileSPD')
op.test('NormDispIncr', 1e-6, 100)
op.algorithm('Newton')
op.integrator('LoadControl', 0.1)
op.analysis('Static')

op.analyze(10)

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

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

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 = 1, dt = 0.0, dtMin = 0.0, dtMax = 0.0, Jd = 0)`

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

ANALYSIS COMMAND

```
##### 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()
```

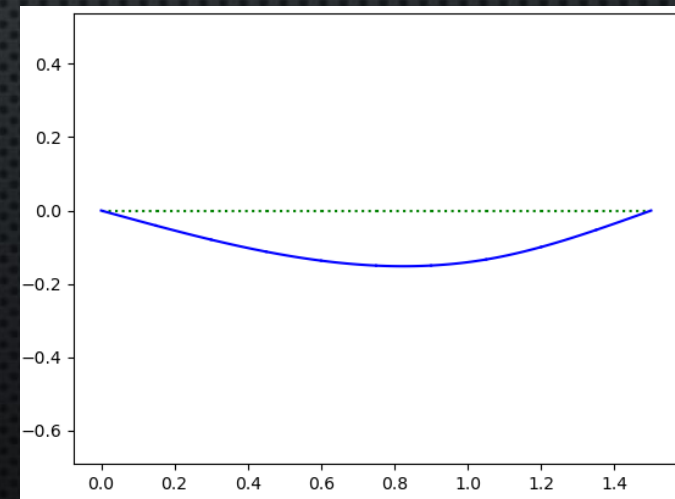
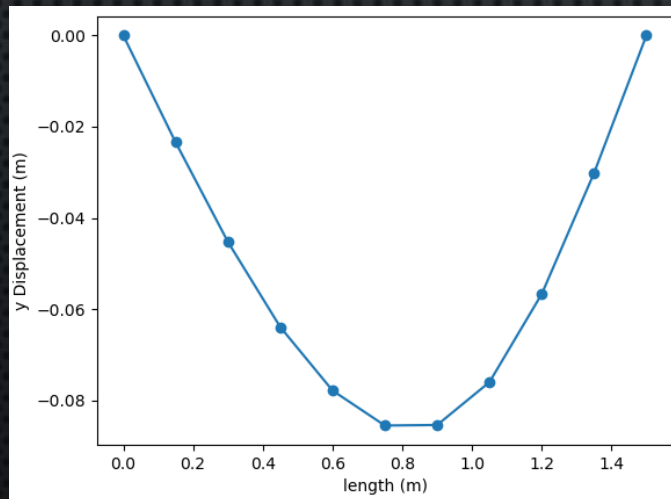
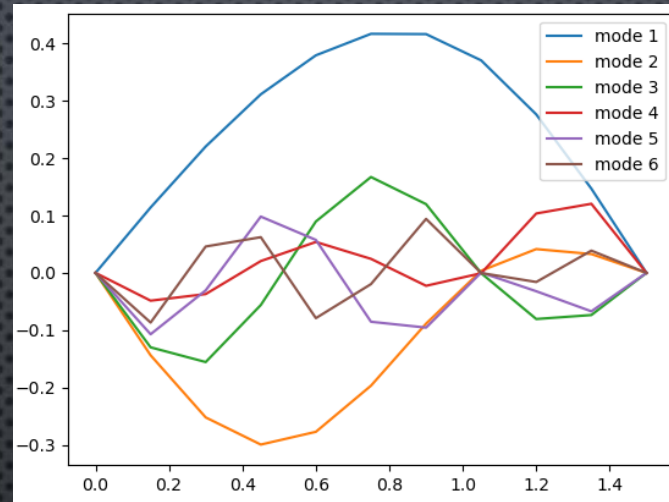
Once the analysis is completed, it was proceed to visualize and the results.

In the upper part of the slide, we detail 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 aplot with it.

EIGENMODES VISUALIZATION AND STATIC DISPLACEMENT



REFERENCES

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

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