TRANSIENT ANALYSIS

OpenSeesPy Tutorial 3

TUTORIAL 3. INTRODUCTION

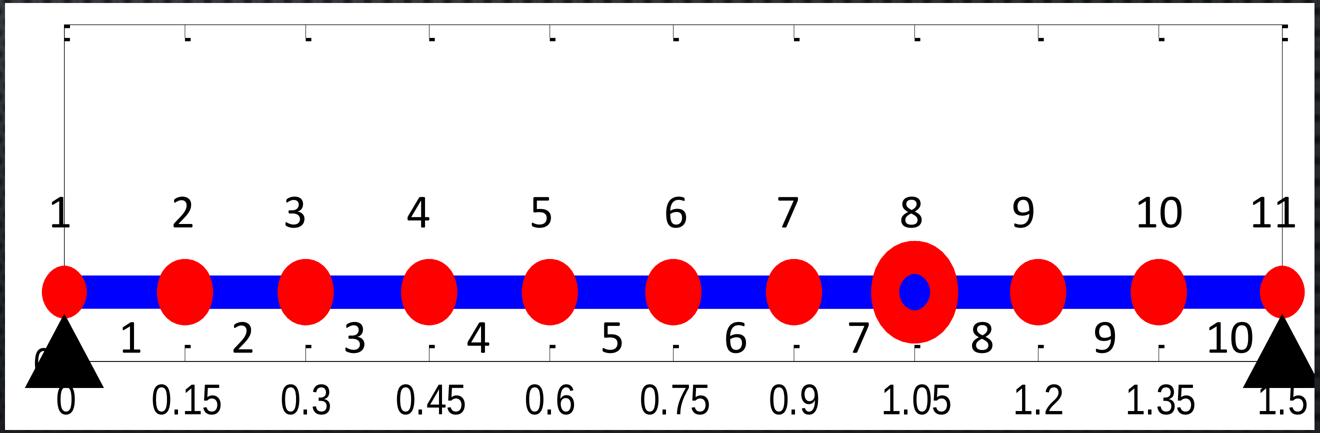
As part of out OpenSeesPy tutorial series, we have been demonstrating how to create simple models and perform structural analyses using OpenSeesPy library. In the previous tutorial, we completed a static analysis, and in this tutorial, we will be performing a transient analysis. Our goal is to provide a clear understanding of OpenSeesPy´s basic framework while offering you the opportunity to explore more complex analyses in subsequent tutorials.

This example is relatively straightforward, making it easy to grasp fundamental concepts in structural engineering. However, it also lays the foundation for more in-depth analyses, which we will cover in future tutorials.

CASE OF STUDY: TRANSIENT 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, 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

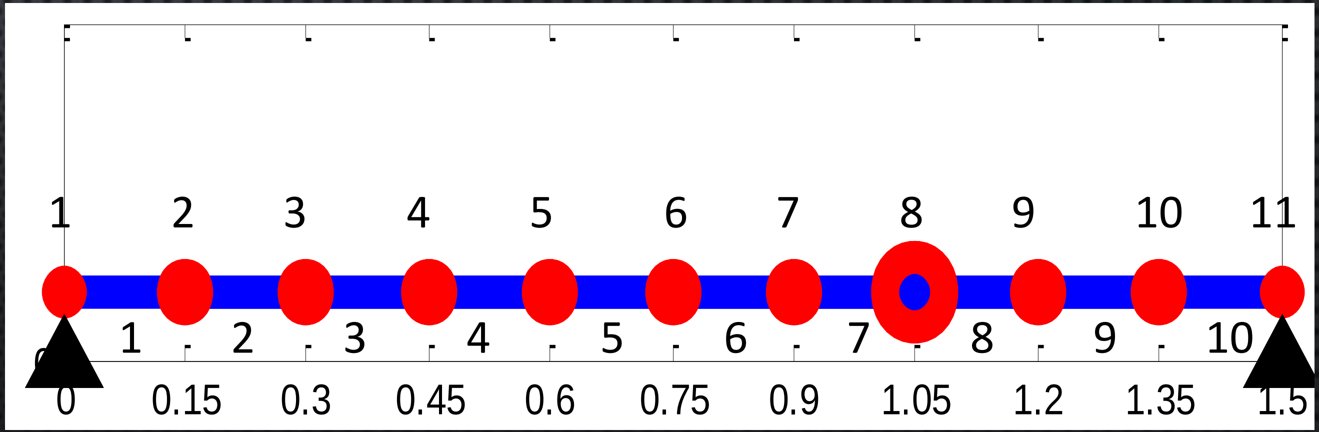
Density (rho): 7850 kg/m³

Young's Modulus (E): 2.1 GPa

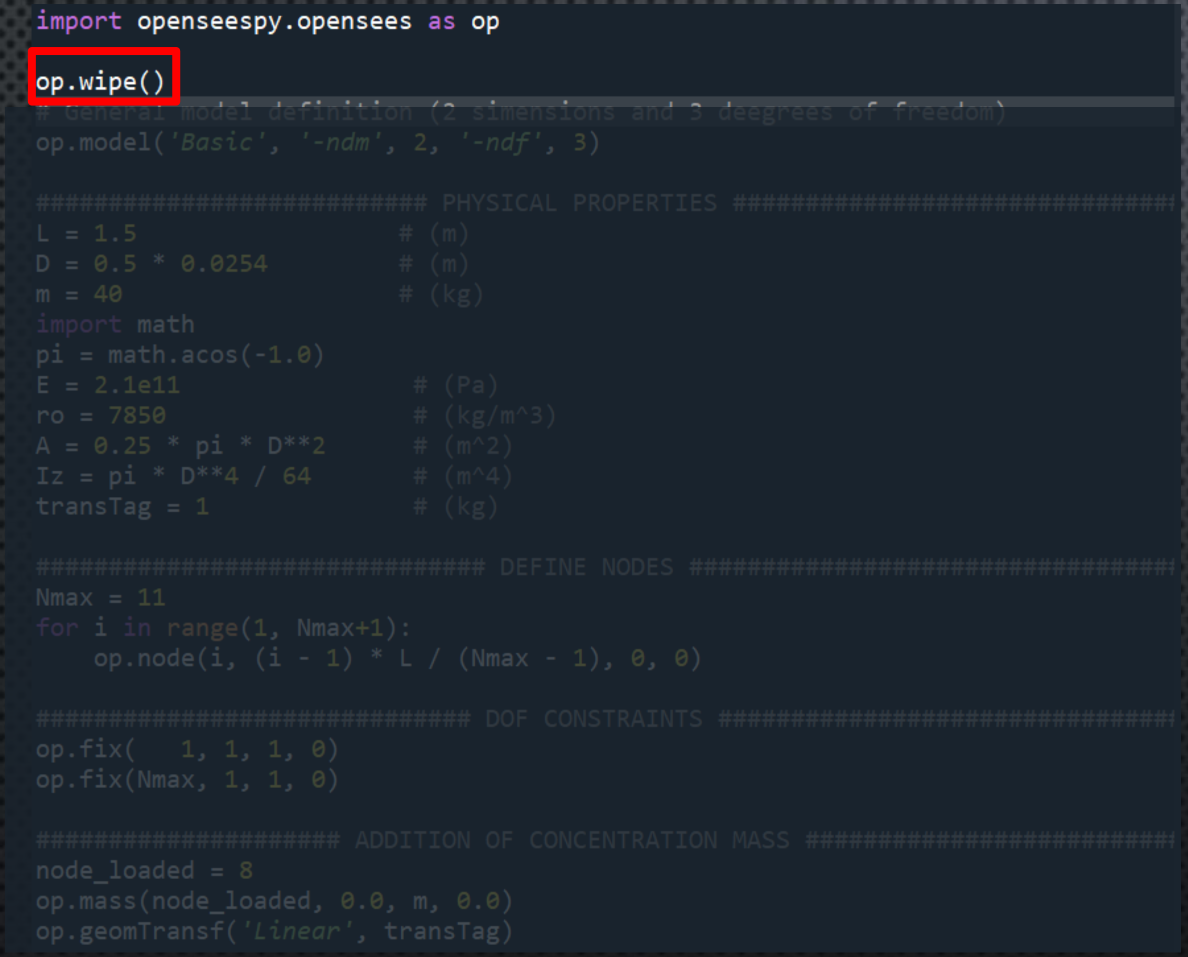
Simply supported shaft.

Mass excess at 0.7L from left.

g : 10 m/s^2



WIPE COMMAND

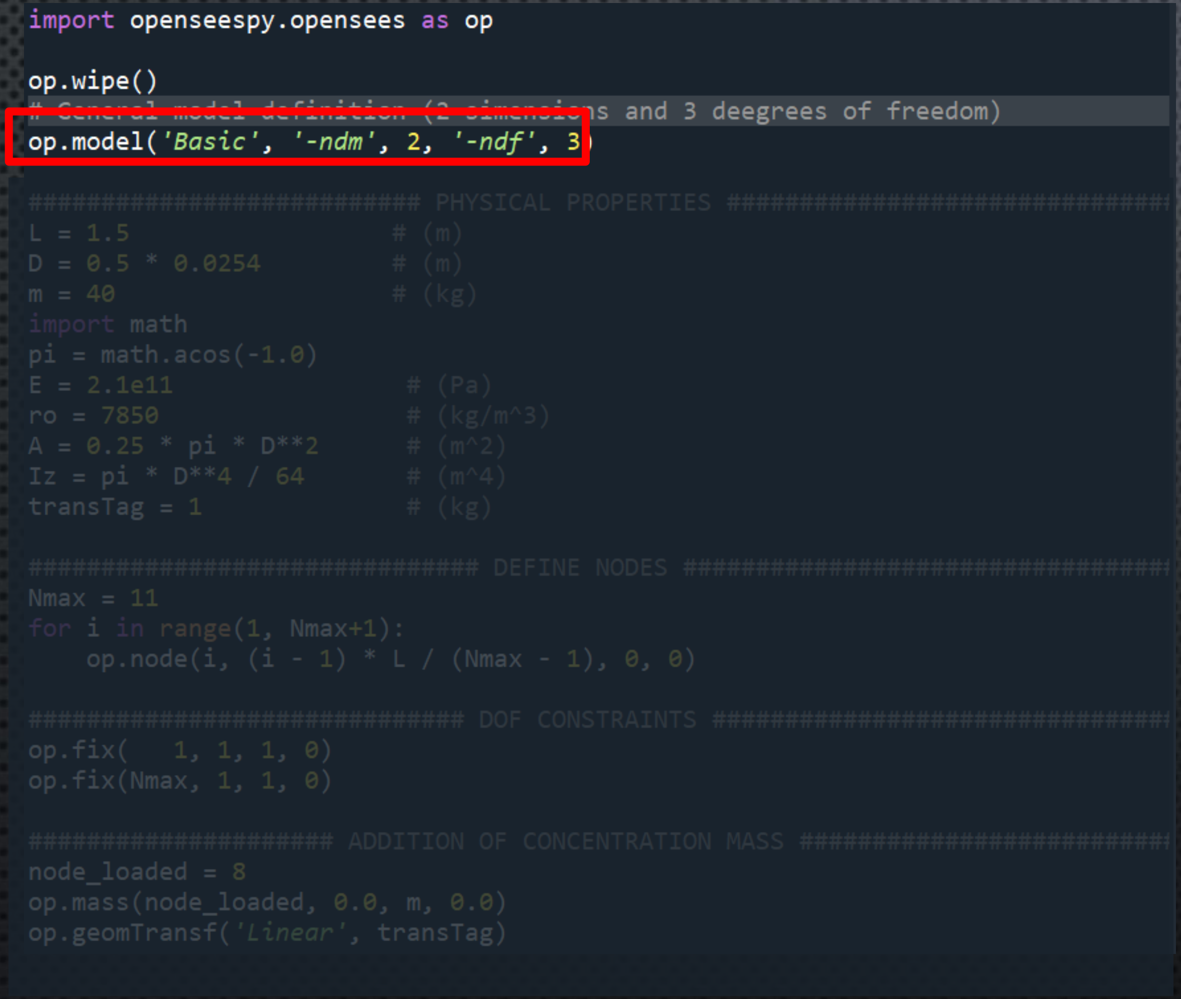


```
import openseespy.opensees as op

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:

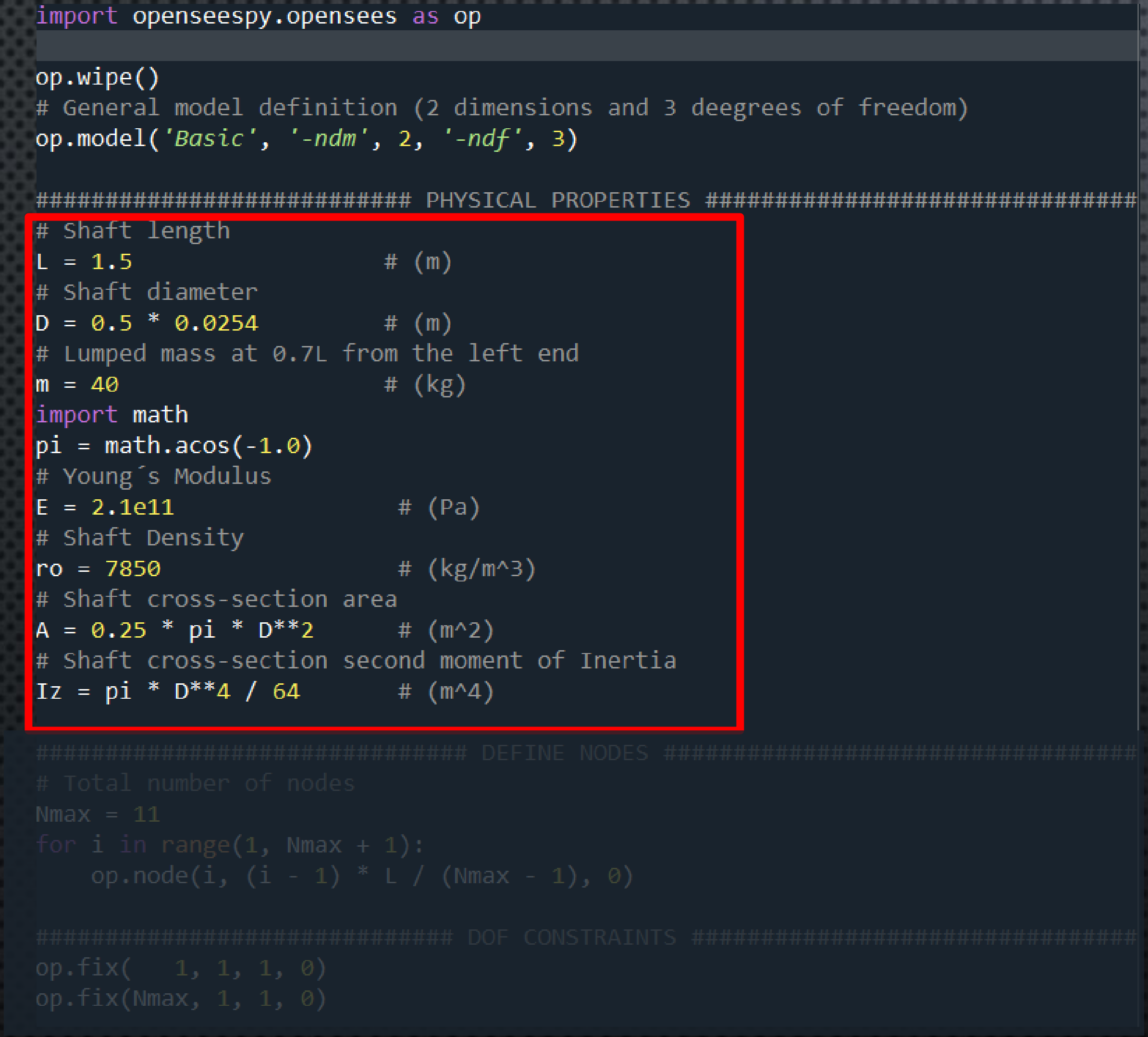
model('basic', '-ndm', ndm, '-ndf', ndf)

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

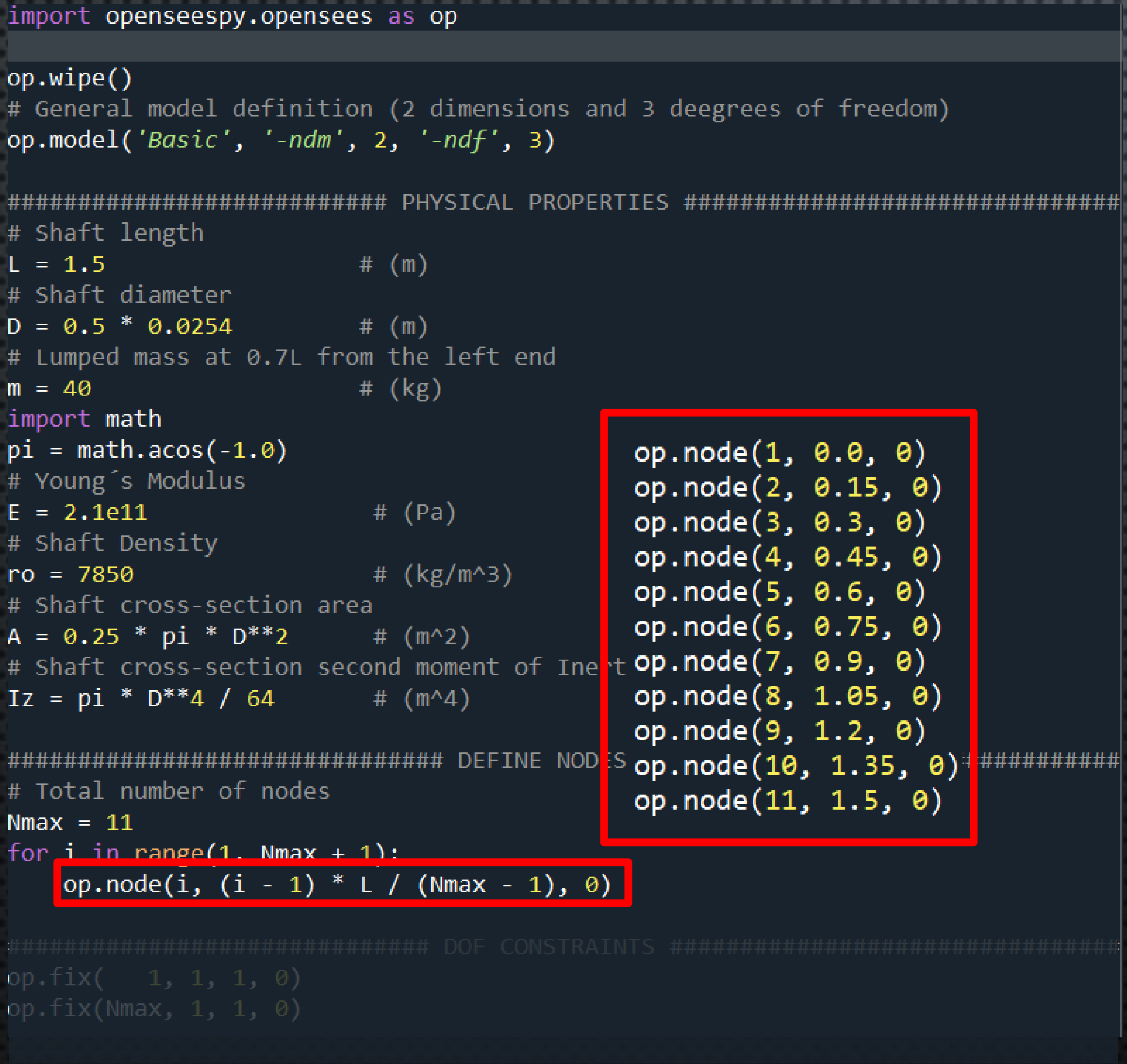


```
import openseespy.opensees as op

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

########################### PHYSICAL PROPERTIES ################################
# 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 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



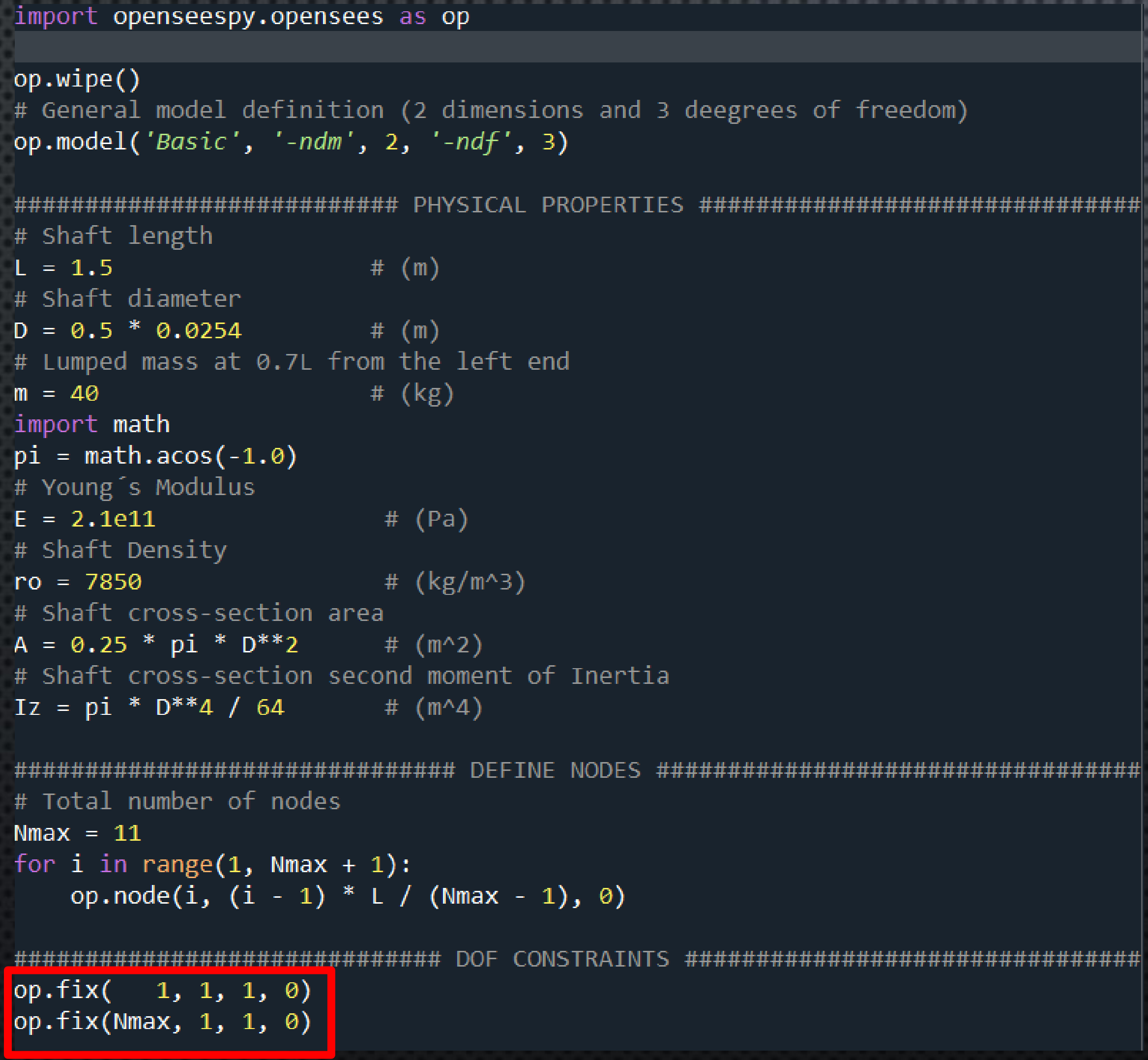
```python
import openseespy.opensees as op

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

########################### PHYSICAL PROPERTIES ################################
# 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 Ine
Iz = pi * D**4 / 64       # (m^4)

################################ DEFINE NOD                          ##########
# 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)
```

```python
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)
```

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.

RESTRAINTS



```
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 ###############################
# 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:

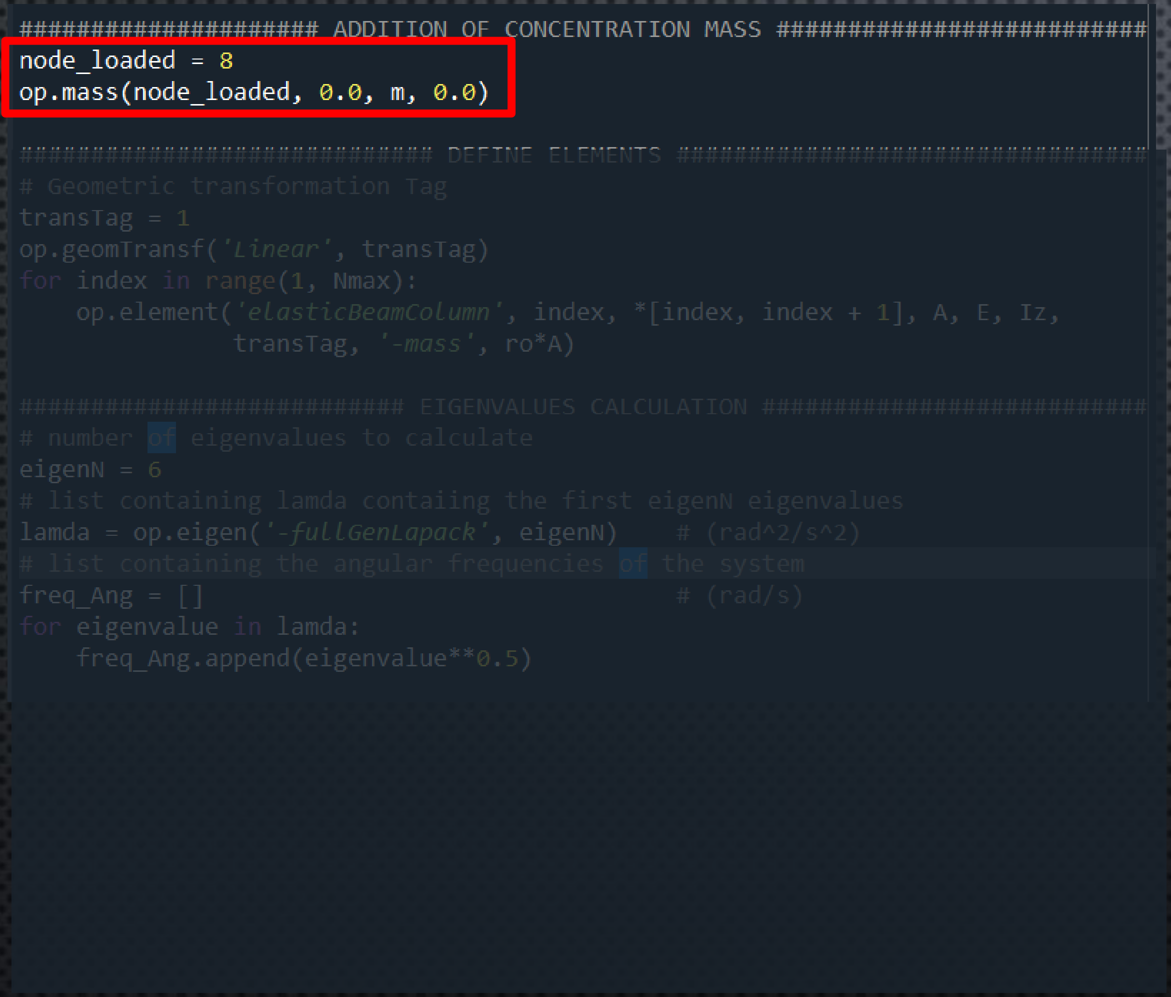
fix(nodeTag, *constrValues)

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 containg 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)
```

The definition of mass 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.

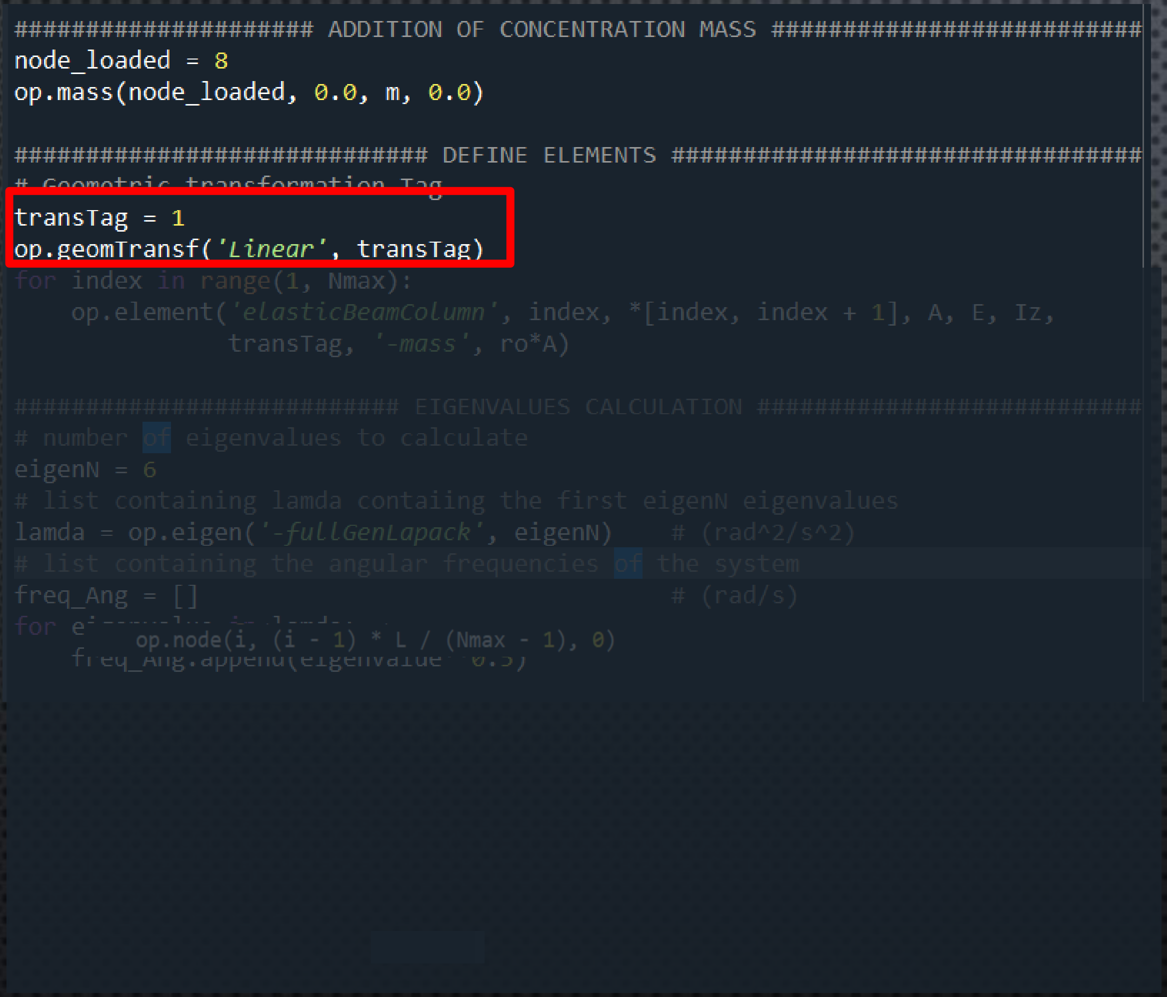
The syntax for the mass command is as follows:

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)
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)
```

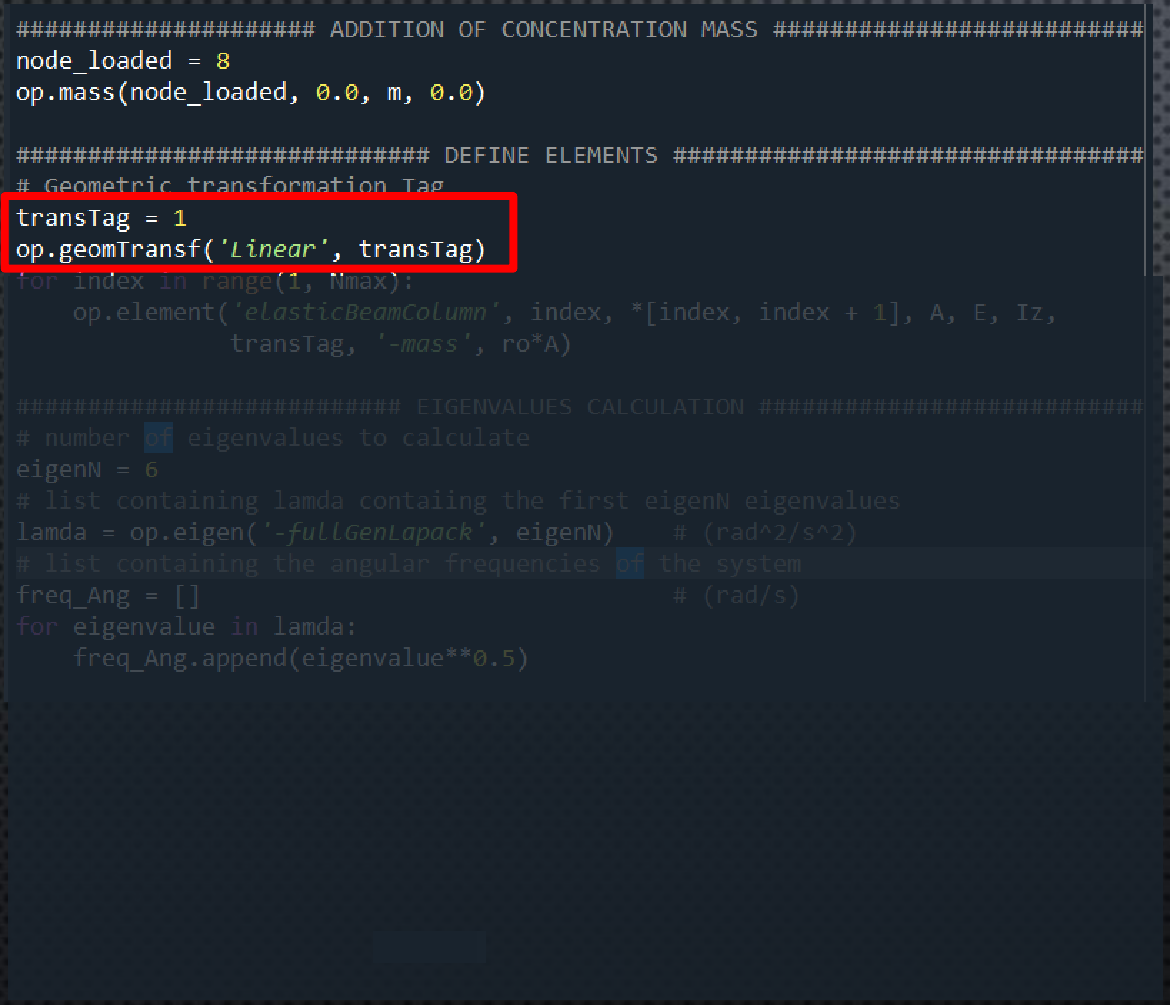
Since we plan to use an elastic beam element to model the shaft, a geometric transformation is necessary to convert the local coordinate system to the global coordinate system. The syntax for this command is as follows:

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)
```

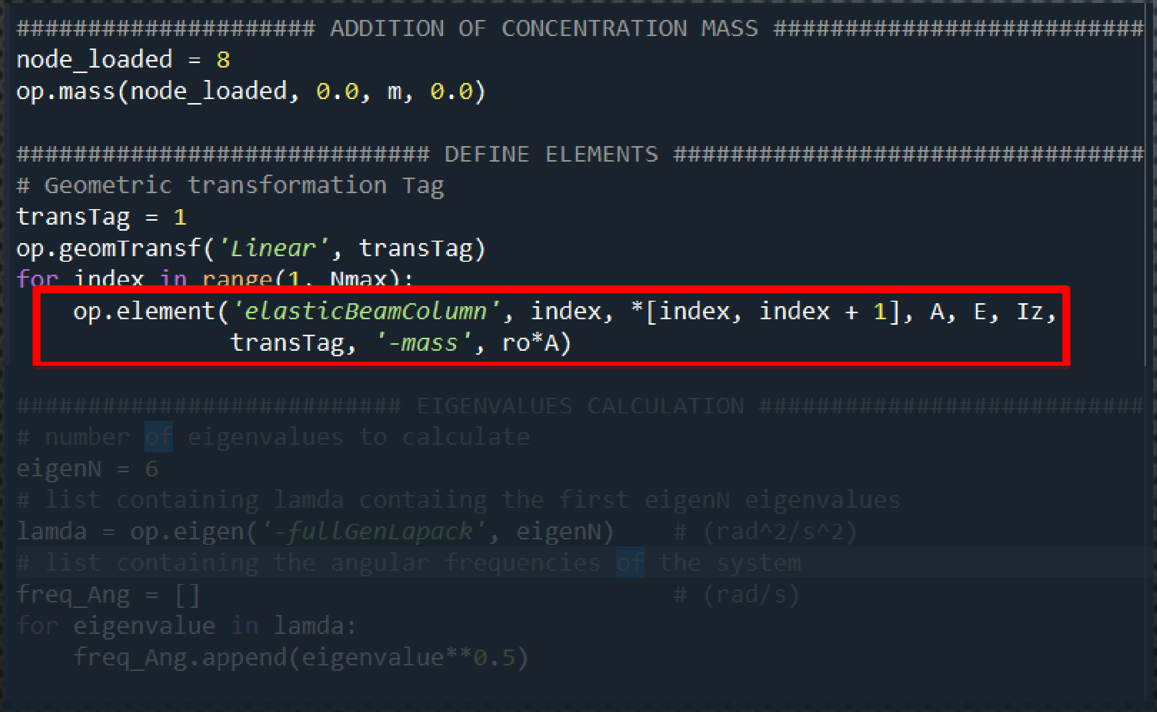
In this example, a linear transformation was used. When using the geoTransf command, the general format resembles the following:

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)
```

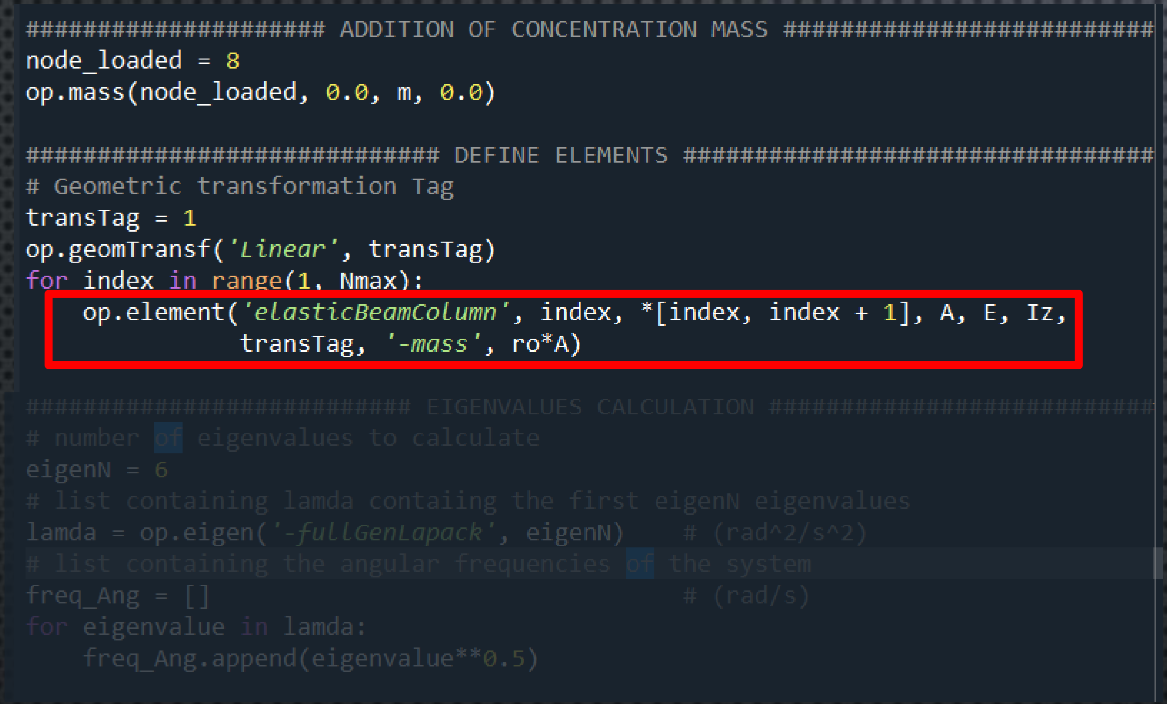
The preceding steps were essential to define the element. In this example, we have defined elastic BeamColumn elements. However, it's worth noting that the list of elements can vary and include zero-length, truss, joint, link elements, and more.

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

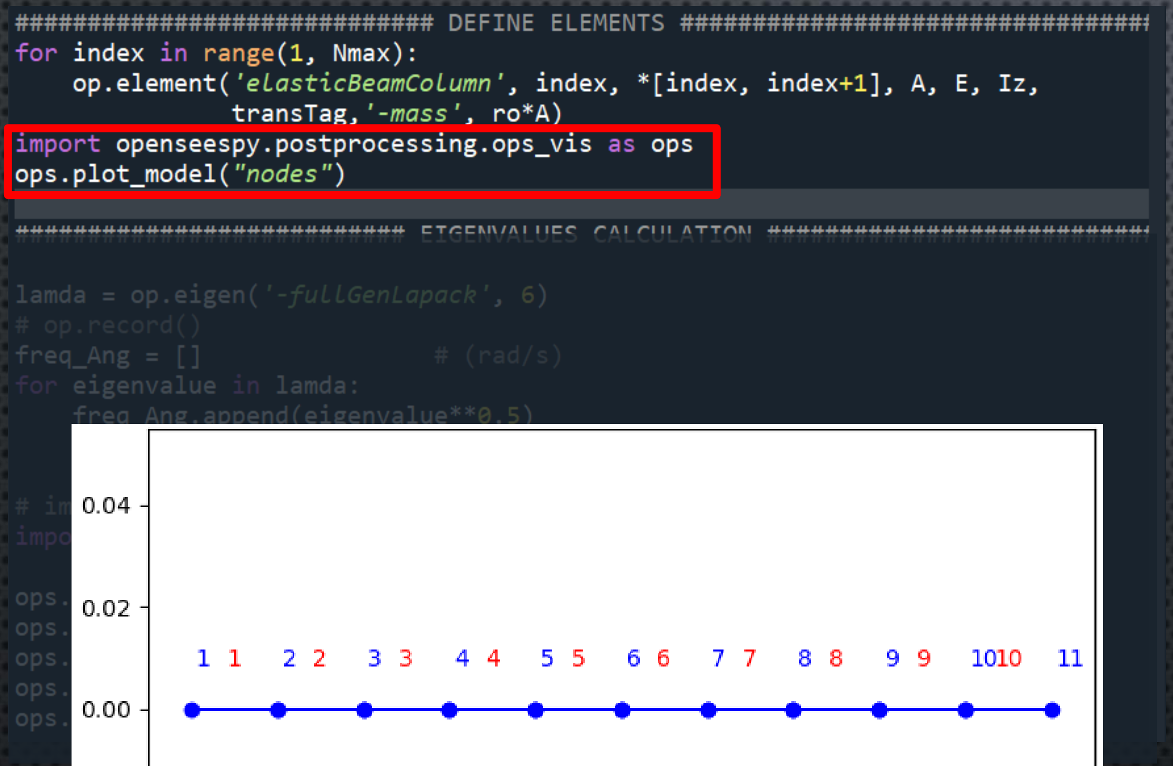
ELEMENT DEFINITION



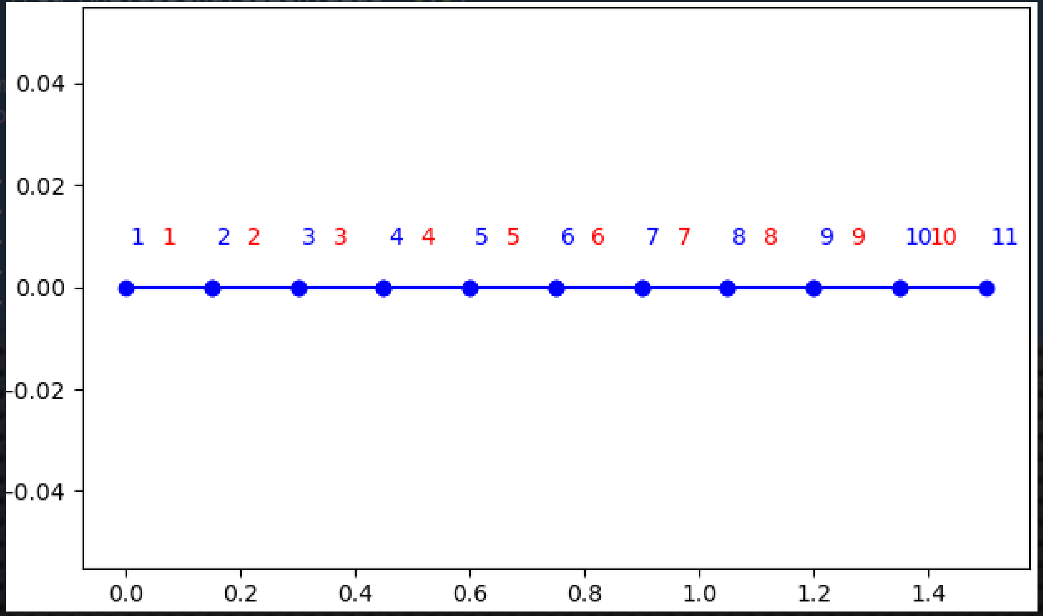
```
###################### ADDITION OF CONCENTRATION MASS ###########################
node_loaded = 8
op.mass(node_loaded, 0.0, m, 0.0)

############################# 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 containg 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)
```

- 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



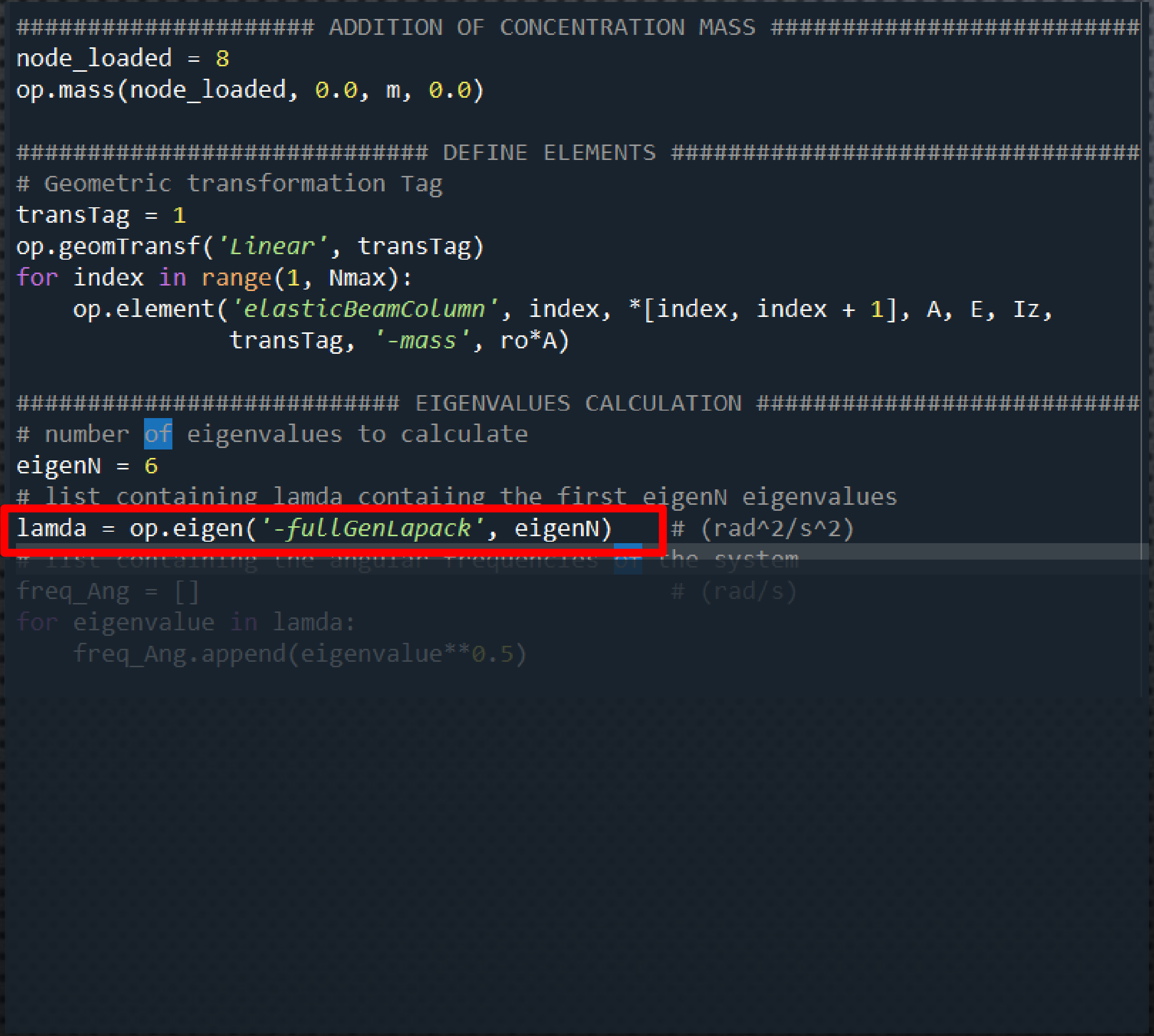
```
############################## DEFINE ELEMENTS ##################################
for index in range(1, Nmax):
    op.element('elasticBeamColumn', index, *[index, index+1], A, E, Iz,
               transTag,'-mass', ro*A)
import openseespy.postprocessing.ops_vis as ops
ops.plot_model("nodes")

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

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


# im
impo

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



```python
##################### 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 containg 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)
```

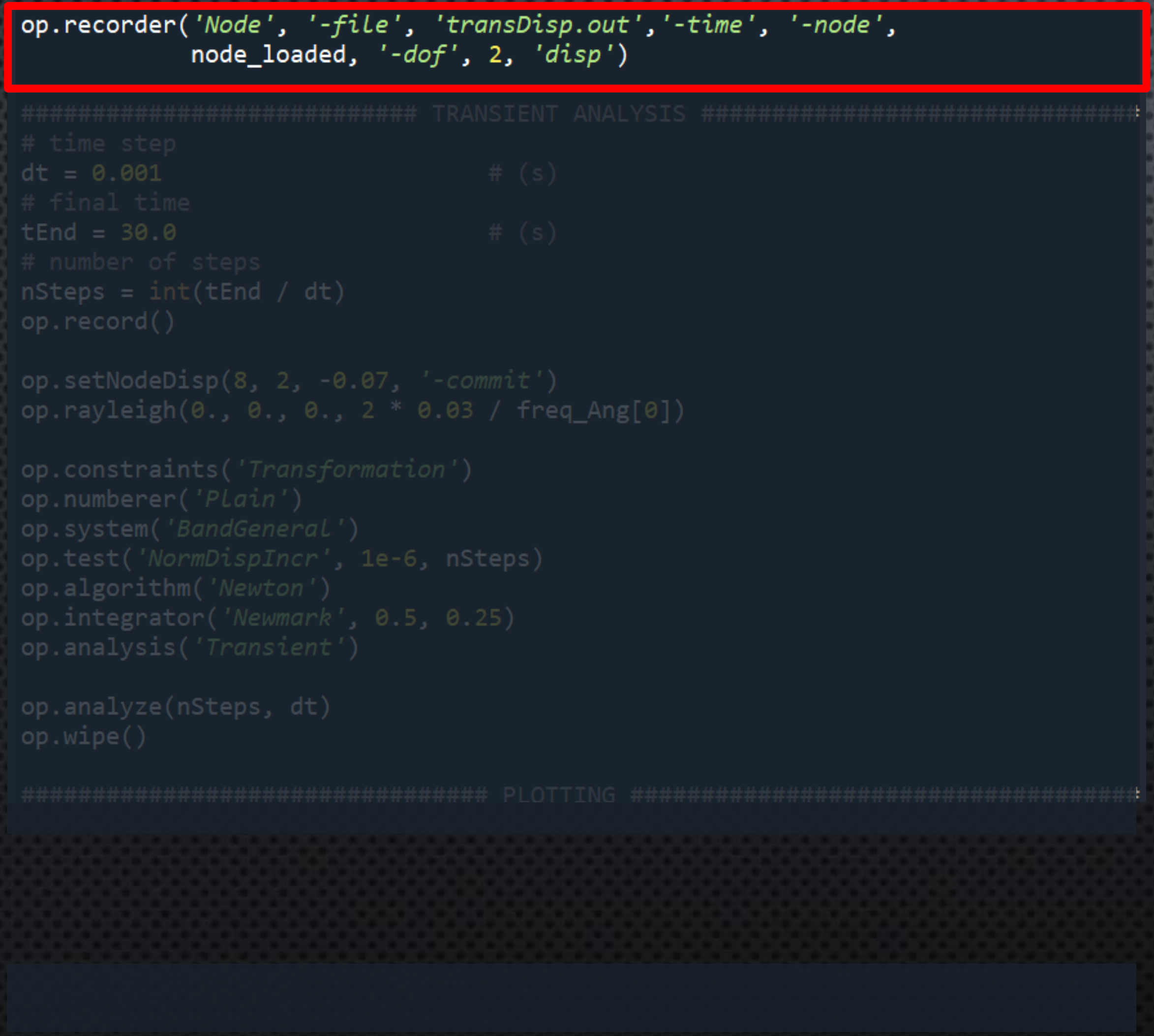
After the elements are properly defined, we proceed to calculate the eigenvalues using the eigen command. This command, in a general form, is represented as follow:

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.

RECORDER COMMAND



```
op.recorder('Node', '-file', 'transDisp.out','-time', '-node',
            node_loaded, '-dof', 2, 'disp')
############################ TRANSIENT ANALYSIS ###############################
# time step
dt = 0.001                         # (s)
# final time
tEnd = 30.0                        # (s)
# number of steps
nSteps = int(tEnd / dt)
op.record()

op.setNodeDisp(8, 2, -0.07, '-commit')
op.rayleigh(0., 0., 0., 2 * 0.03 / freq_Ang[0])

op.constraints('Transformation')
op.numberer('Plain')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, nSteps)
op.algorithm('Newton')
op.integrator('Newmark', 0.5, 0.25)
op.analysis('Transient')

op.analyze(nSteps, dt)
op.wipe()

################################# PLOTTING ####################################
```

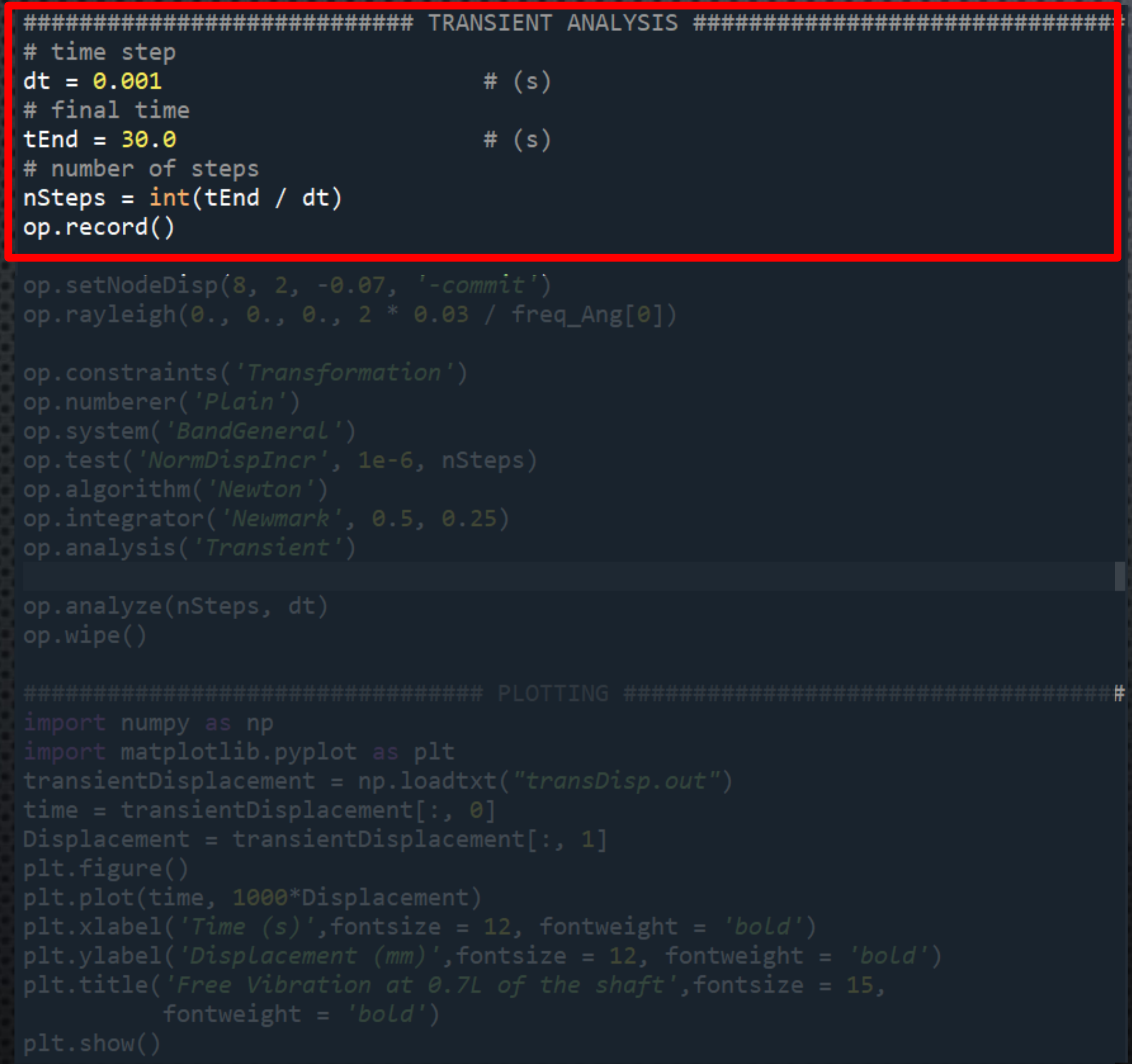
To record the displacement of the loaded node, you can use the recorder command. For the specific use on the example, the syntax is as follows:

recorder('Node', '-file', filename, '-time', '-node', *nodeTags=[], '-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.
- 'node' flags the recording for specific nodes defined by nodeTags.
- 'dof' is the specified dof to record.
- respType is a string indicating the required response.

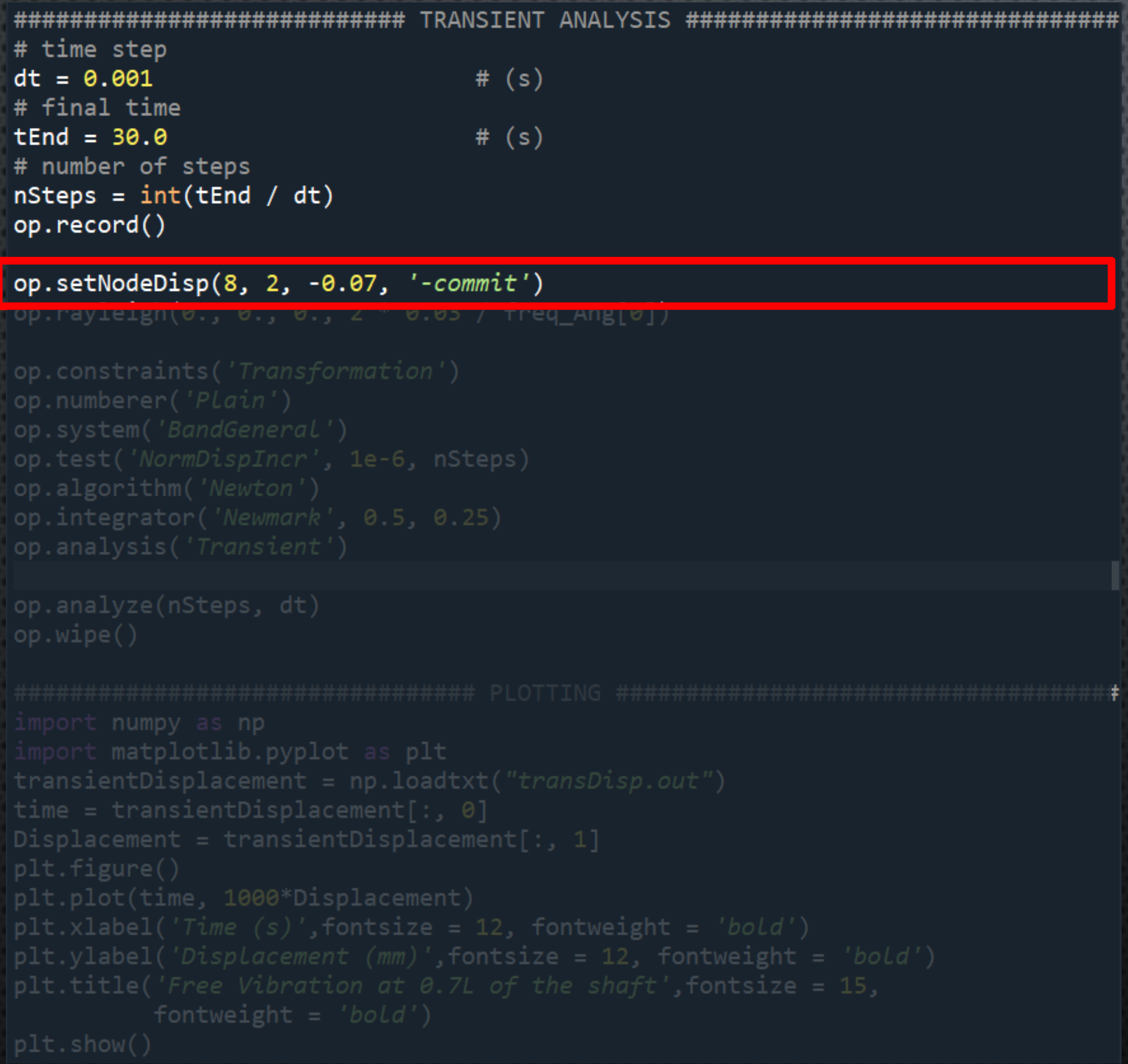
RECORD COMMAND



```
############################ TRANSIENT ANALYSIS ###############################
# time step
dt = 0.001                        # (s)
# final time
tEnd = 30.0                       # (s)
# number of steps
nSteps = int(tEnd / dt)
op.record()
op.setNodeDisp(8, 2, -0.07, '-commit')
op.rayleigh(0., 0., 0., 2 * 0.03 / freq_Ang[0])

op.constraints('Transformation')
op.numberer('Plain')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, nSteps)
op.algorithm('Newton')
op.integrator('Newmark', 0.5, 0.25)
op.analysis('Transient')


op.analyze(nSteps, dt)
op.wipe()

################################# PLOTTING ###################################
import numpy as np
import matplotlib.pyplot as plt
transientDisplacement = np.loadtxt("transDisp.out")
time = transientDisplacement[:, 0]
Displacement = transientDisplacement[:, 1]
plt.figure()
plt.plot(time, 1000*Displacement)
plt.xlabel('Time (s)',fontsize = 12, fontweight = 'bold')
plt.ylabel('Displacement (mm)',fontsize = 12, fontweight = 'bold')
plt.title('Free Vibration at 0.7L of the shaft',fontsize = 15,
          fontweight = 'bold')
plt.show()
```

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

SETNODEDISP COMMAND



```
############################ TRANSIENT ANALYSIS ###############################
# time step
dt = 0.001                      # (s)
# final time
tEnd = 30.0                     # (s)
# number of steps
nSteps = int(tEnd / dt)
op.record()

op.setNodeDisp(8, 2, -0.07, '-commit')
op.rayleigh(0., 0., 0., 2 * 0.05 / freq_Ang[0])

op.constraints('Transformation')
op.numberer('Plain')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, nSteps)
op.algorithm('Newton')
op.integrator('Newmark', 0.5, 0.25)
op.analysis('Transient')

op.analyze(nSteps, dt)
op.wipe()

################################# PLOTTING ###################################
import numpy as np
import matplotlib.pyplot as plt
transientDisplacement = np.loadtxt("transDisp.out")
time = transientDisplacement[:, 0]
Displacement = transientDisplacement[:, 1]
plt.figure()
plt.plot(time, 1000*Displacement)
plt.xlabel('Time (s)',fontsize = 12, fontweight = 'bold')
plt.ylabel('Displacement (mm)',fontsize = 12, fontweight = 'bold')
plt.title('Free Vibration at 0.7L of the shaft',fontsize = 15,
          fontweight = 'bold')
plt.show()
```

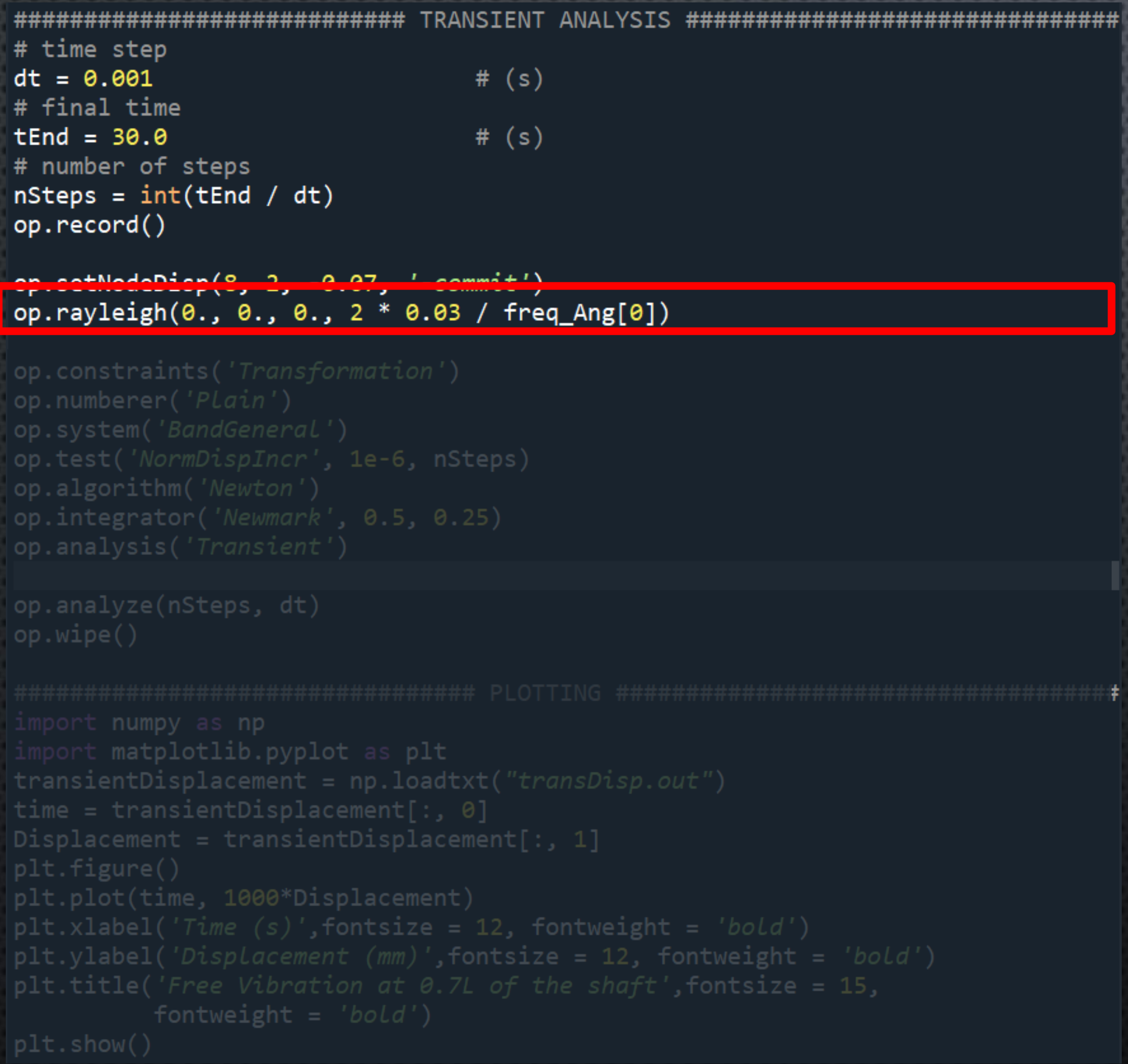
The setNodeDisp command is employed to specify and set a nodal displacement at a specified degree of freedom. The syntax is as follows:

setNodeDisp(nodeTag, dof, value, '-commit')

Where:

- nodeTag is a tag referring to the node where the displacement is set.
- dof is the DOF for which the displacement is being specified.
- Value is the displacement value to be applied.
- 'commit' is the commit nodal state (optional).

RAYLEIGH COMMAND



```
############################ TRANSIENT ANALYSIS ###############################
# time step
dt = 0.001                          # (s)
# final time
tEnd = 30.0                         # (s)
# number of steps
nSteps = int(tEnd / dt)
op.record()

op.setNodeDisp(8, 2, -0.07, '-commit')
op.rayleigh(0., 0., 0., 2 * 0.03 / freq_Ang[0])

op.constraints('Transformation')
op.numberer('Plain')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, nSteps)
op.algorithm('Newton')
op.integrator('Newmark', 0.5, 0.25)
op.analysis('Transient')

op.analyze(nSteps, dt)
op.wipe()

################################# PLOTTING ###################################
import numpy as np
import matplotlib.pyplot as plt
transientDisplacement = np.loadtxt("transDisp.out")
time = transientDisplacement[:, 0]
Displacement = transientDisplacement[:, 1]
plt.figure()
plt.plot(time, 1000*Displacement)
plt.xlabel('Time (s)',fontsize = 12, fontweight = 'bold')
plt.ylabel('Displacement (mm)',fontsize = 12, fontweight = 'bold')
plt.title('Free Vibration at 0.7L of the shaft',fontsize = 15,
          fontweight = 'bold')
plt.show()
```

This command is utilized to assign damping to all previously defined elements and nodes. Damping is defined as a combination of stiffness and mass-proportional damping matrices:

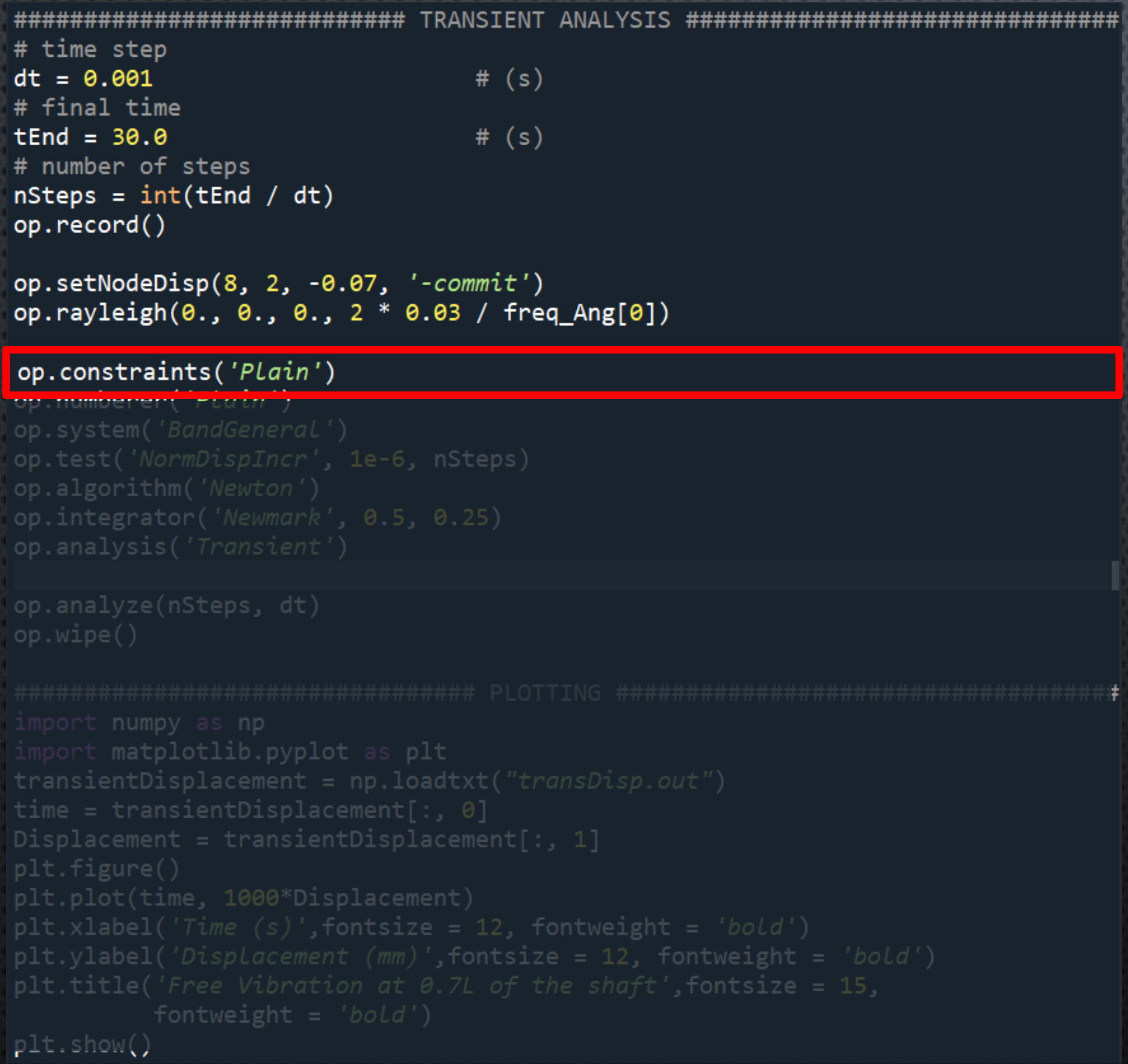
The syntax for the Rayleigh command is as follows:

rayleigh(alphaM, betaK, betaKinit, betaKcomm)

Where:

- alphaM is a factor applied to elements or nodes mass matrix.
- betaK is a factor applied to elements' current stiffness matrix.
- betaKinit is a factor applied to elements initial stiffness matrix.
- betaKcomm is a facto applied to elements committed stiffened matrix.

CONSTRAINTS COMMAND



```
############################ TRANSIENT ANALYSIS ################################
# time step
dt = 0.001                         # (s)
# final time
tEnd = 30.0                        # (s)
# number of steps
nSteps = int(tEnd / dt)
op.record()

op.setNodeDisp(8, 2, -0.07, '-commit')
op.rayleigh(0., 0., 0., 2 * 0.03 / freq_Ang[0])

op.constraints('Plain')
op.numberer('Plain')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, nSteps)
op.algorithm('Newton')
op.integrator('Newmark', 0.5, 0.25)
op.analysis('Transient')

op.analyze(nSteps, dt)
op.wipe()

################################# PLOTTING ###################################
import numpy as np
import matplotlib.pyplot as plt
transientDisplacement = np.loadtxt("transDisp.out")
time = transientDisplacement[:, 0]
Displacement = transientDisplacement[:, 1]
plt.figure()
plt.plot(time, 1000*Displacement)
plt.xlabel('Time (s)',fontsize = 12, fontweight = 'bold')
plt.ylabel('Displacement (mm)',fontsize = 12, fontweight = 'bold')
plt.title('Free Vibration at 0.7L of the shaft',fontsize = 15,
          fontweight = 'bold')
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:

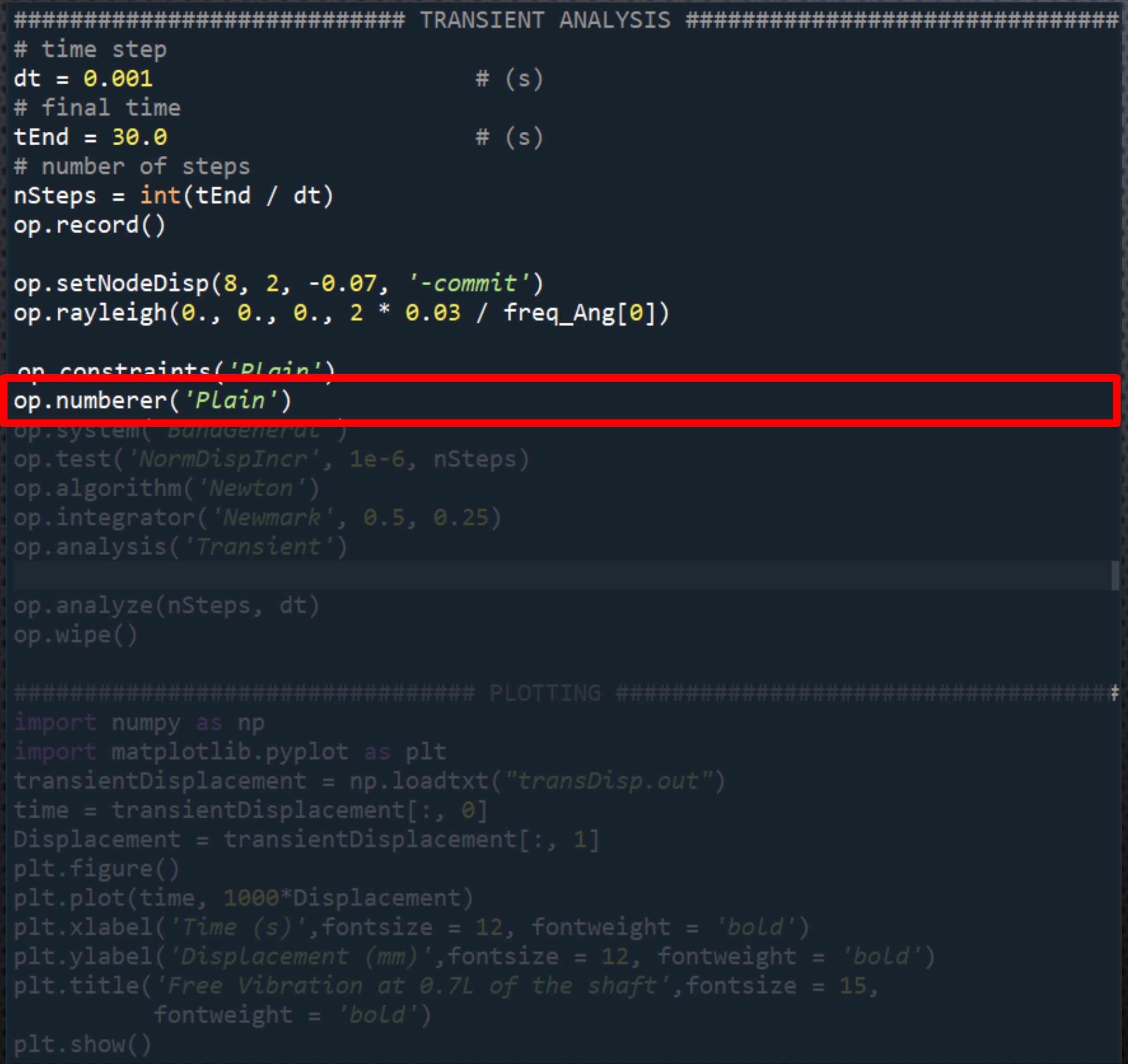
pattern(patternType, patternTag, *patternArgs)

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.

NUMBERER COMMAND



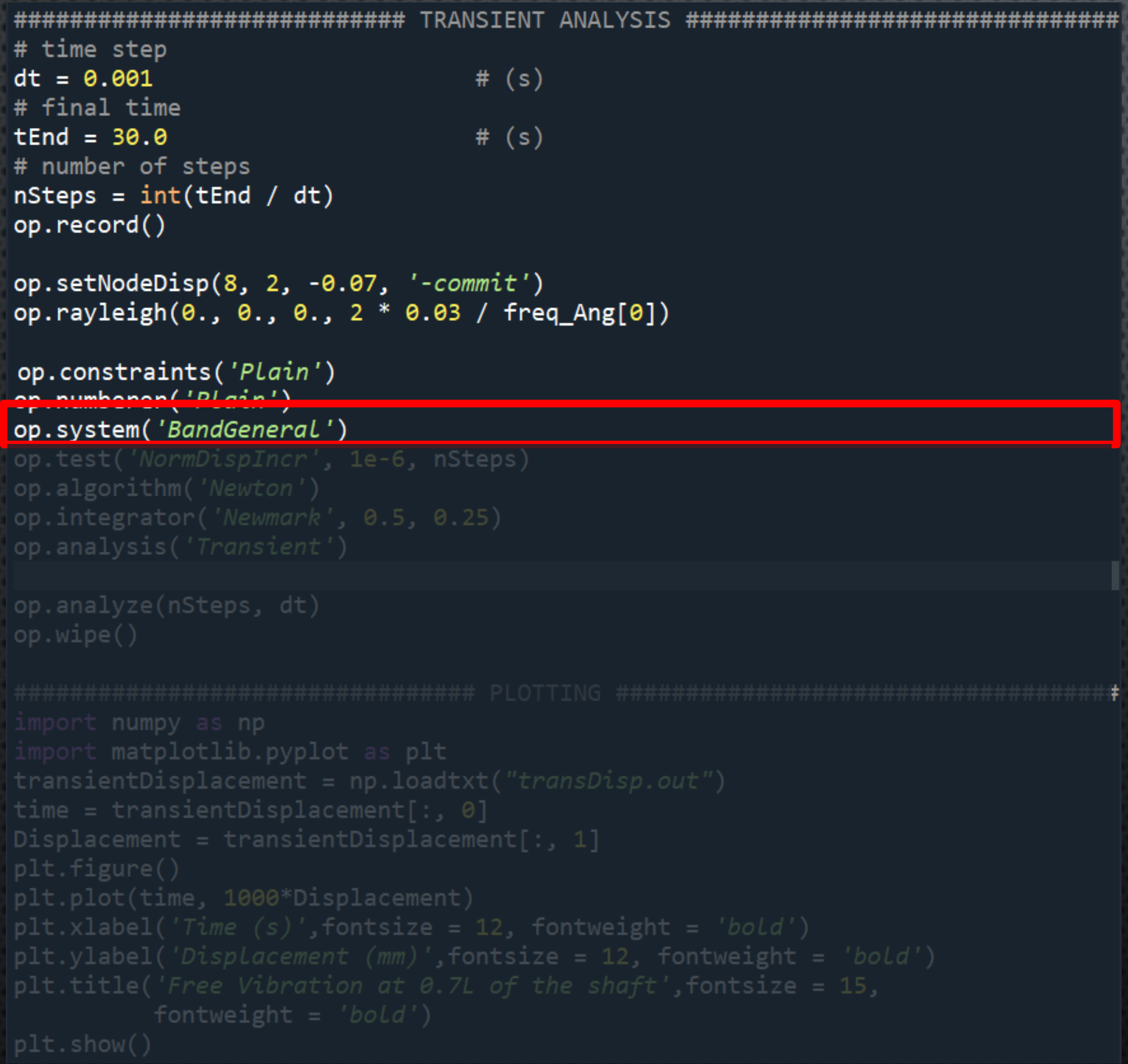
```
############################ TRANSIENT ANALYSIS ###############################
# time step
dt = 0.001                          # (s)
# final time
tEnd = 30.0                         # (s)
# number of steps
nSteps = int(tEnd / dt)
op.record()

op.setNodeDisp(8, 2, -0.07, '-commit')
op.rayleigh(0., 0., 0., 2 * 0.03 / freq_Ang[0])

op.constraints('Plain')
op.numberer('Plain')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, nSteps)
op.algorithm('Newton')
op.integrator('Newmark', 0.5, 0.25)
op.analysis('Transient')

op.analyze(nSteps, dt)
op.wipe()

################################# PLOTTING ####################################
import numpy as np
import matplotlib.pyplot as plt
transientDisplacement = np.loadtxt("transDisp.out")
time = transientDisplacement[:, 0]
Displacement = transientDisplacement[:, 1]
plt.figure()
plt.plot(time, 1000*Displacement)
plt.xlabel('Time (s)',fontsize = 12, fontweight = 'bold')
plt.ylabel('Displacement (mm)',fontsize = 12, fontweight = 'bold')
plt.title('Free Vibration at 0.7L of the shaft',fontsize = 15,
          fontweight = 'bold')
plt.show()
```

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

The numbered can be different types:

- Plain: assigns degrees of freedom sequentially without 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



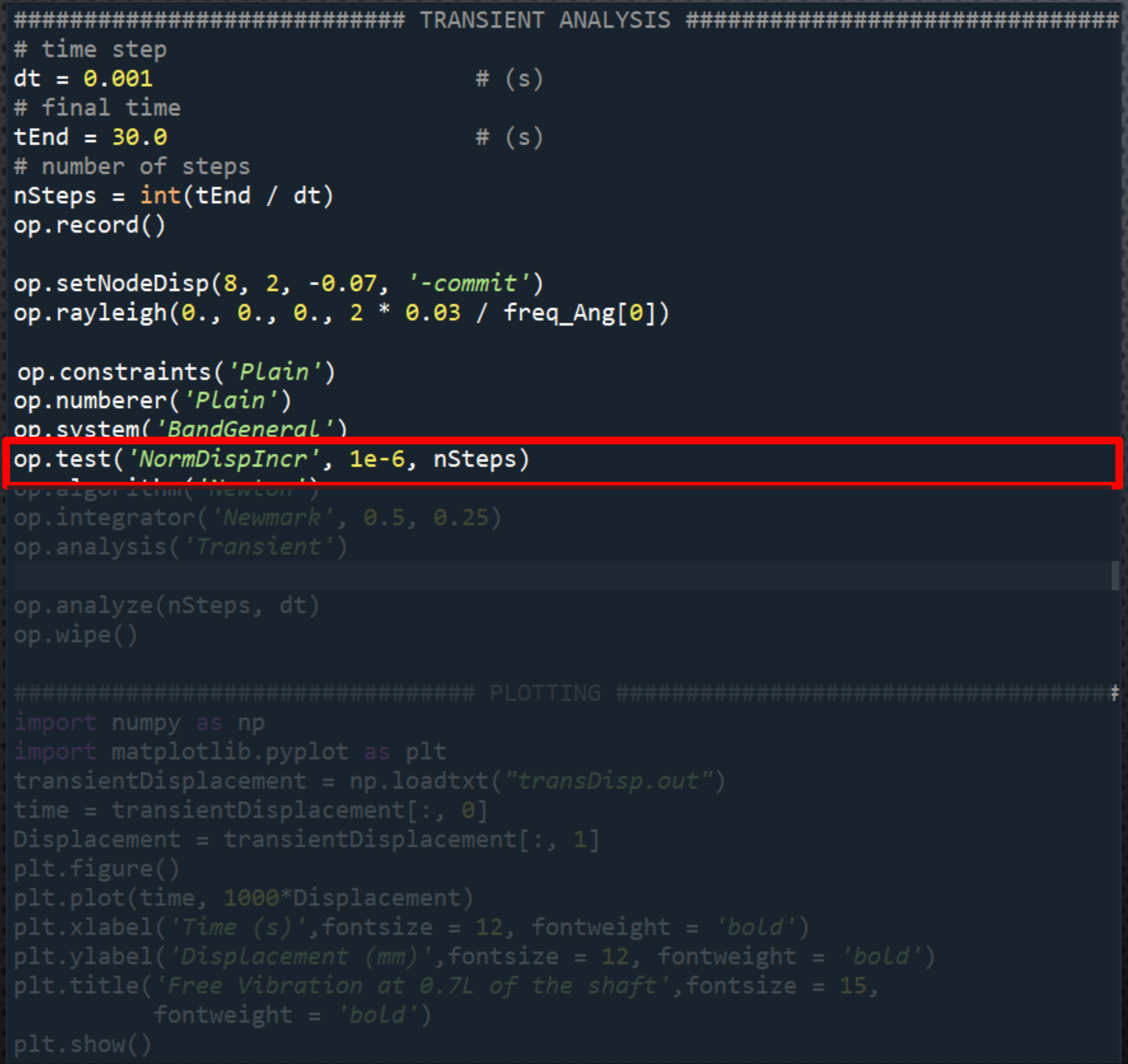
```
############################# TRANSIENT ANALYSIS ################################
# time step
dt = 0.001                          # (s)
# final time
tEnd = 30.0                         # (s)
# number of steps
nSteps = int(tEnd / dt)
op.record()

op.setNodeDisp(8, 2, -0.07, '-commit')
op.rayleigh(0., 0., 0., 2 * 0.03 / freq_Ang[0])

 op.constraints('Plain')
op.numberer('Plain')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, nSteps)
op.algorithm('Newton')
op.integrator('Newmark', 0.5, 0.25)
op.analysis('Transient')

op.analyze(nSteps, dt)
op.wipe()

################################# PLOTTING ####################################
import numpy as np
import matplotlib.pyplot as plt
transientDisplacement = np.loadtxt("transDisp.out")
time = transientDisplacement[:, 0]
Displacement = transientDisplacement[:, 1]
plt.figure()
plt.plot(time, 1000*Displacement)
plt.xlabel('Time (s)',fontsize = 12, fontweight = 'bold')
plt.ylabel('Displacement (mm)',fontsize = 12, fontweight = 'bold')
plt.title('Free Vibration at 0.7L of the shaft',fontsize = 15,
          fontweight = 'bold')
plt.show()
```

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 BandGeneral type was used. This command is employed to construct a linear system of equation object, making it suitable for a wide range of structural analysis scenarios.

TEST COMMAND



```
############################ TRANSIENT ANALYSIS ###############################
# time step
dt = 0.001                         # (s)
# final time
tEnd = 30.0                        # (s)
# number of steps
nSteps = int(tEnd / dt)
op.record()

op.setNodeDisp(8, 2, -0.07, '-commit')
op.rayleigh(0., 0., 0., 2 * 0.03 / freq_Ang[0])

 op.constraints('Plain')
op.numberer('Plain')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, nSteps)
op.algorithm('Newton')
op.integrator('Newmark', 0.5, 0.25)
op.analysis('Transient')

op.analyze(nSteps, dt)
op.wipe()

################################# PLOTTING ###################################
import numpy as np
import matplotlib.pyplot as plt
transientDisplacement = np.loadtxt("transDisp.out")
time = transientDisplacement[:, 0]
Displacement = transientDisplacement[:, 1]
plt.figure()
plt.plot(time, 1000*Displacement)
plt.xlabel('Time (s)',fontsize = 12, fontweight = 'bold')
plt.ylabel('Displacement (mm)',fontsize = 12, fontweight = 'bold')
plt.title('Free Vibration at 0.7L of the shaft',fontsize = 15,
          fontweight = 'bold')
plt.show()
```

The test command is used to control the convergence criteria in a structural analysis. The test can be different types to ensure accuracy and efficiency.

These include NormBalance, NormDispIncr, energyIncr, RelativeNormUnbalance, and more.

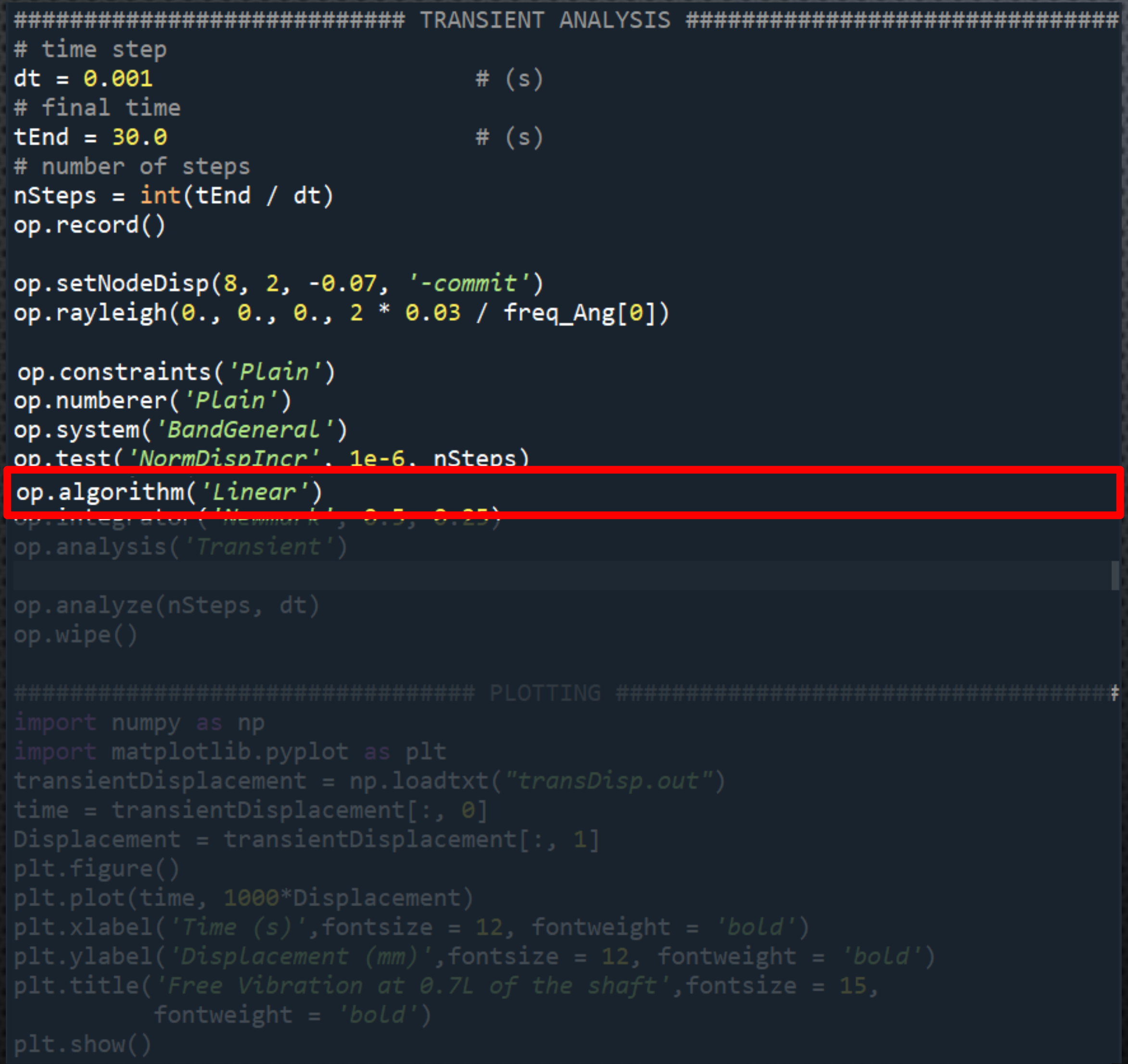
In the example being discussed, the NormDispIncr test type was used, with the syntax:

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



```
############################ TRANSIENT ANALYSIS ################################
# time step
dt = 0.001                          # (s)
# final time
tEnd = 30.0                         # (s)
# number of steps
nSteps = int(tEnd / dt)
op.record()

op.setNodeDisp(8, 2, -0.07, '-commit')
op.rayleigh(0., 0., 0., 2 * 0.03 / freq_Ang[0])

op.constraints('Plain')
op.numberer('Plain')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, nSteps)
op.algorithm('Linear')
op.integrator('Newmark', 0.5, 0.25)
op.analysis('Transient')

op.analyze(nSteps, dt)
op.wipe()

################################## PLOTTING ###################################
import numpy as np
import matplotlib.pyplot as plt
transientDisplacement = np.loadtxt("transDisp.out")
time = transientDisplacement[:, 0]
Displacement = transientDisplacement[:, 1]
plt.figure()
plt.plot(time, 1000*Displacement)
plt.xlabel('Time (s)',fontsize = 12, fontweight = 'bold')
plt.ylabel('Displacement (mm)',fontsize = 12, fontweight = 'bold')
plt.title('Free Vibration at 0.7L of the shaft',fontsize = 15,
          fontweight = 'bold')
plt.show()
```

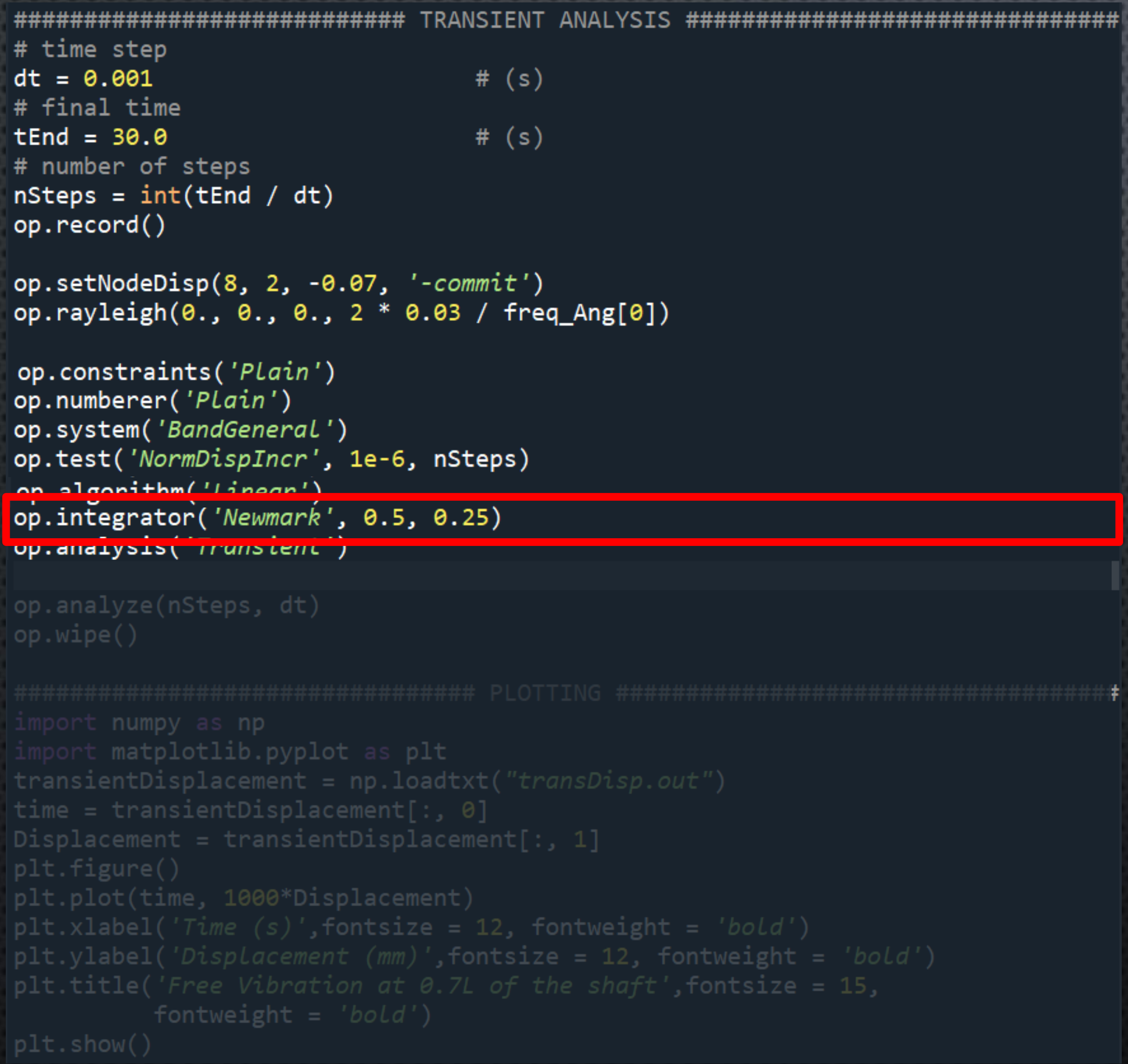
The algorithm command serves purpose of constructing a solution object, which plays a crusial tole in determining the sequence of steps undertaken to solve the non-linear equations in an analysis.

This command provides a wide array of options, allowing you to customize the analyisis to meet specific requirements. Some of the most common include:

- Linear
- Newton
- NewtonLineSearch
- ModifiedNewton
- KrylovNewton

Notice that there are more options to adjust according to your model and analysis.

INTEGRATOR COMMAND



```
############################ TRANSIENT ANALYSIS ################################
# time step
dt = 0.001                      # (s)
# final time
tEnd = 30.0                     # (s)
# number of steps
nSteps = int(tEnd / dt)
op.record()

op.setNodeDisp(8, 2, -0.07, '-commit')
op.rayleigh(0., 0., 0., 2 * 0.03 / freq_Ang[0])

 op.constraints('Plain')
op.numberer('Plain')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, nSteps)
op.algorithm('Linear')
op.integrator('Newmark', 0.5, 0.25)
op.analysis('Transient')

op.analyze(nSteps, dt)
op.wipe()

################################# PLOTTING ###################################
import numpy as np
import matplotlib.pyplot as plt
transientDisplacement = np.loadtxt("transDisp.out")
time = transientDisplacement[:, 0]
Displacement = transientDisplacement[:, 1]
plt.figure()
plt.plot(time, 1000*Displacement)
plt.xlabel('Time (s)',fontsize = 12, fontweight = 'bold')
plt.ylabel('Displacement (mm)',fontsize = 12, fontweight = 'bold')
plt.title('Free Vibration at 0.7L of the shaft',fontsize = 15,
          fontweight = 'bold')
plt.show()
```

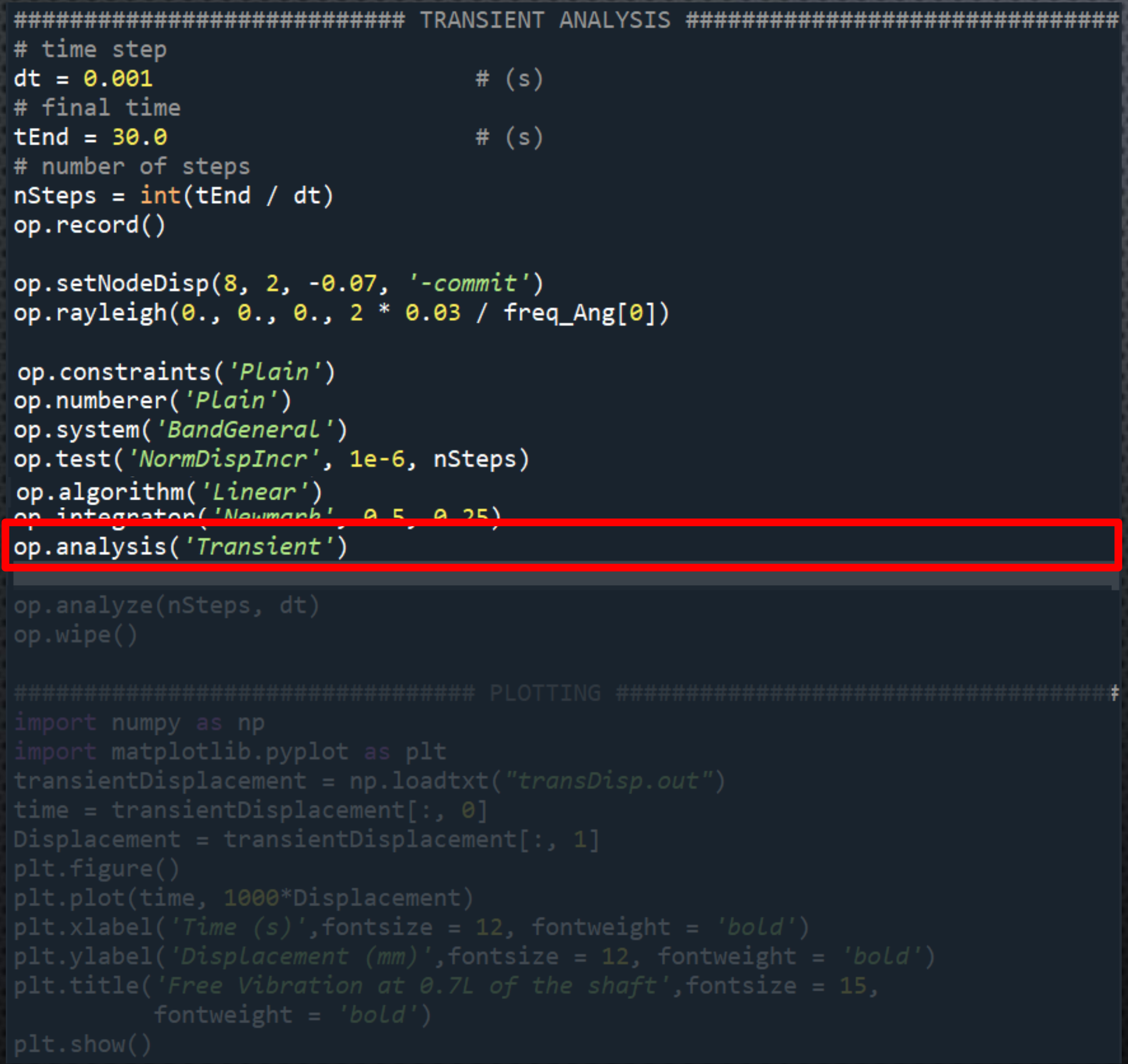
The integrator command is used to determine the sequence of steps taken to solve the nonlinear equations during structural analysis.

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

- Central Difference
- Newmark Method
- Hilber-Hughes-Taylor Method
- Generalized Alpha Method
- TRBDF2
- Explicit Difference
- PFEM integrator

For static analyses, there are more options available.

ANALYSIS COMMAND



```
############################ TRANSIENT ANALYSIS ###############################
# time step
dt = 0.001                          # (s)
# final time
tEnd = 30.0                         # (s)
# number of steps
nSteps = int(tEnd / dt)
op.record()

op.setNodeDisp(8, 2, -0.07, '-commit')
op.rayleigh(0., 0., 0., 2 * 0.03 / freq_Ang[0])

 op.constraints('Plain')
op.numberer('Plain')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, nSteps)
op.algorithm('Linear')
op.integrator('Newmark', 0.5, 0.25)
op.analysis('Transient')

op.analyze(nSteps, dt)
op.wipe()

################################# PLOTTING ##################################
import numpy as np
import matplotlib.pyplot as plt
transientDisplacement = np.loadtxt("transDisp.out")
time = transientDisplacement[:, 0]
Displacement = transientDisplacement[:, 1]
plt.figure()
plt.plot(time, 1000*Displacement)
plt.xlabel('Time (s)',fontsize = 12, fontweight = 'bold')
plt.ylabel('Displacement (mm)',fontsize = 12, fontweight = 'bold')
plt.title('Free Vibration at 0.7L of the shaft',fontsize = 15,
          fontweight = 'bold')
plt.show()
```

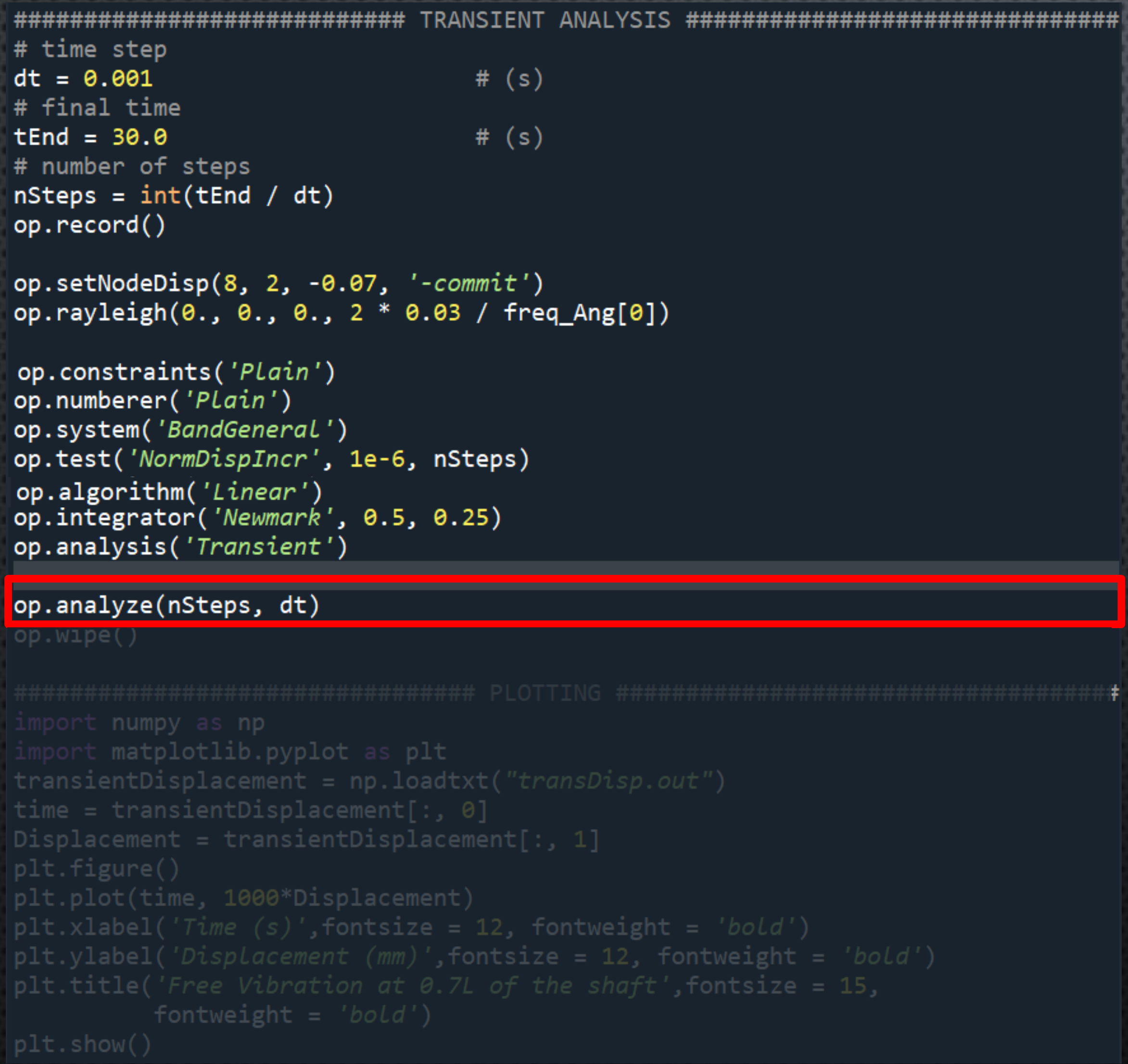
The analysis command is used to specify the type of analysis. It determines the nature of the simulation. The syntax is as follows:

analysis(analysisType)

The analysisType parameter can take on various values, including:

- ‘Static’ is used for static structural analysis, focusing on equilibrium under constant loads.
- ‘Transient’ employed in transient dynamic analysis, with time-dependent loads.
- ‘VariableTransient’ is useful for modeling changing parameter over time.
- ‘PFEM’ (Particle Finite Element Method) is used for fluid-structure interaction or particle-based simulations.

ANALYZE COMMAND



```
############################ TRANSIENT ANALYSIS ###############################
# time step
dt = 0.001                         # (s)
# final time
tEnd = 30.0                        # (s)
# number of steps
nSteps = int(tEnd / dt)
op.record()

op.setNodeDisp(8, 2, -0.07, '-commit')
op.rayleigh(0., 0., 0., 2 * 0.03 / freq_Ang[0])

 op.constraints('Plain')
op.numberer('Plain')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, nSteps)
op.algorithm('Linear')
op.integrator('Newmark', 0.5, 0.25)
op.analysis('Transient')

op.analyze(nSteps, dt)
op.wipe()

################################## PLOTTING ###################################
import numpy as np
import matplotlib.pyplot as plt
transientDisplacement = np.loadtxt("transDisp.out")
time = transientDisplacement[:, 0]
Displacement = transientDisplacement[:, 1]
plt.figure()
plt.plot(time, 1000*Displacement)
plt.xlabel('Time (s)',fontsize = 12, fontweight = 'bold')
plt.ylabel('Displacement (mm)',fontsize = 12, fontweight = 'bold')
plt.title('Free Vibration at 0.7L of the shaft',fontsize = 15,
          fontweight = 'bold')
plt.show()
```

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:

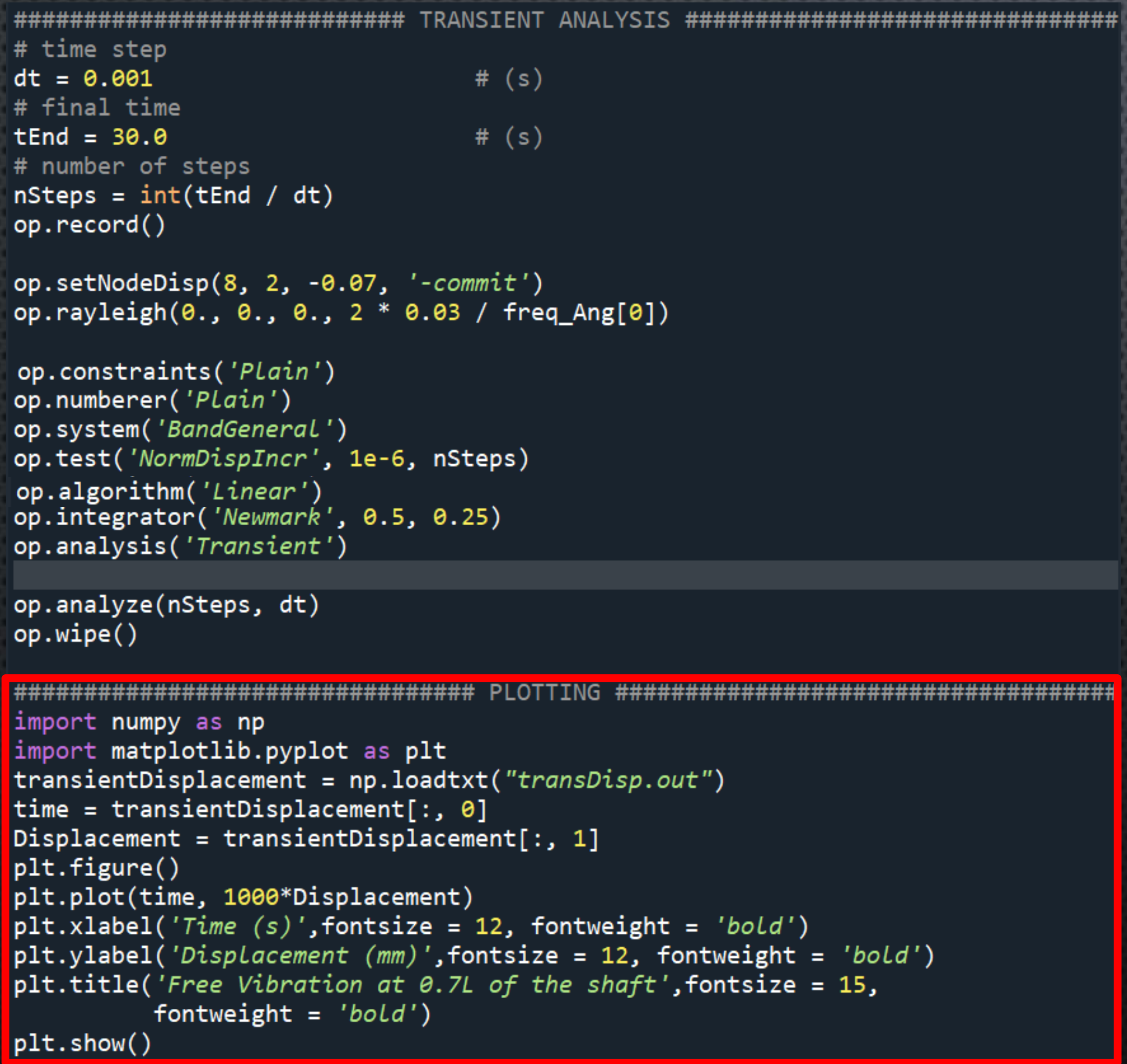
analyze(numIncr, dt, dtMin, dtMax, Jd)

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 this time dt parameter was implemented since it is a transient analysis.

VISUALIZING ANALYSIS RESULTS



```
############################ TRANSIENT ANALYSIS ################################
# time step
dt = 0.001                        # (s)
# final time
tEnd = 30.0                       # (s)
# number of steps
nSteps = int(tEnd / dt)
op.record()

op.setNodeDisp(8, 2, -0.07, '-commit')
op.rayleigh(0., 0., 0., 2 * 0.03 / freq_Ang[0])

op.constraints('Plain')
op.numberer('Plain')
op.system('BandGeneral')
op.test('NormDispIncr', 1e-6, nSteps)
op.algorithm('Linear')
op.integrator('Newmark', 0.5, 0.25)
op.analysis('Transient')

op.analyze(nSteps, dt)
op.wipe()

################################# PLOTTING ####################################
import numpy as np
import matplotlib.pyplot as plt
transientDisplacement = np.loadtxt("transDisp.out")
time = transientDisplacement[:, 0]
Displacement = transientDisplacement[:, 1]
plt.figure()
plt.plot(time, 1000*Displacement)
plt.xlabel('Time (s)',fontsize = 12, fontweight = 'bold')
plt.ylabel('Displacement (mm)',fontsize = 12, fontweight = 'bold')
plt.title('Free Vibration at 0.7L of the shaft',fontsize = 15,
          fontweight = 'bold')
plt.show()
```

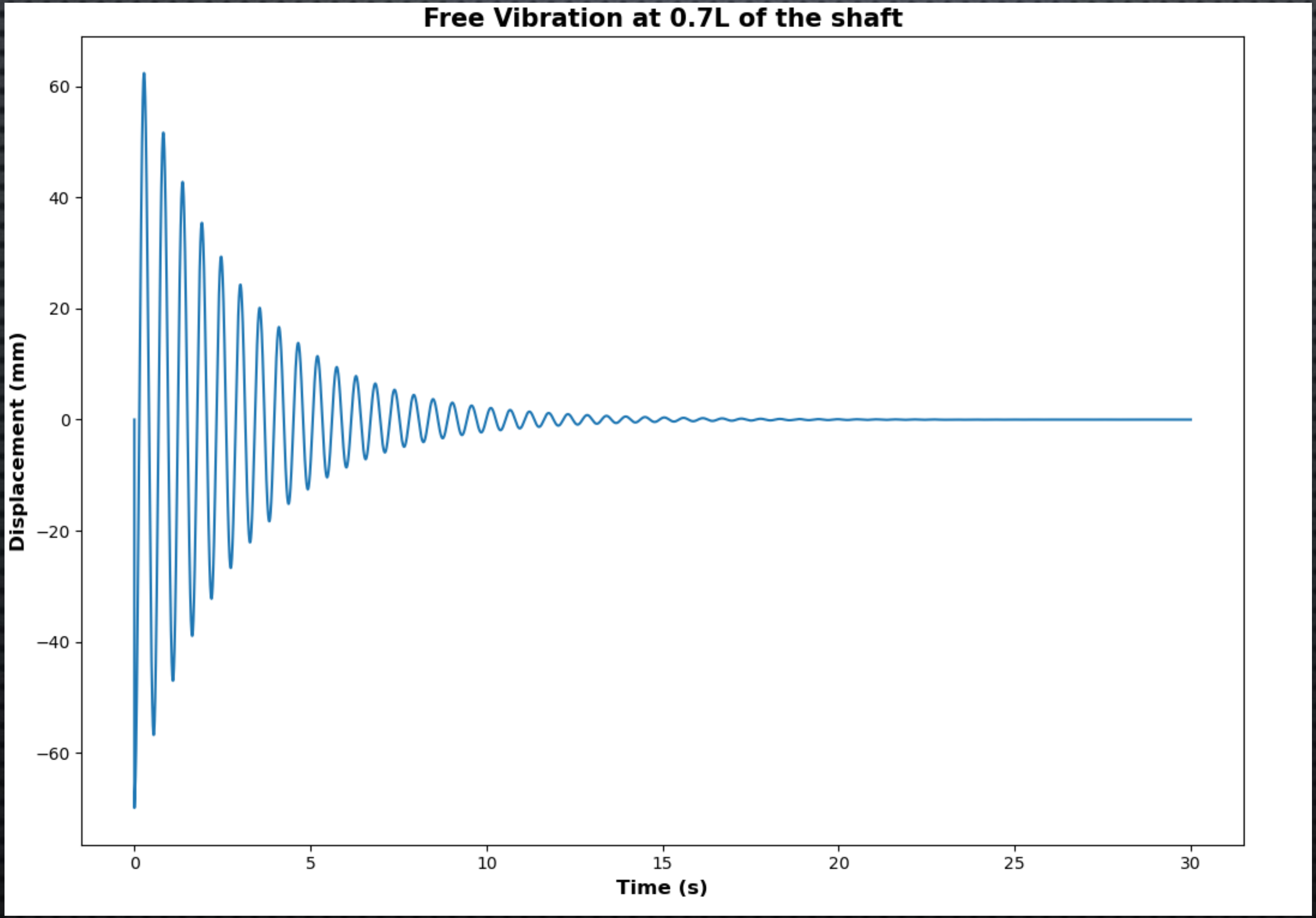
In this example, the results are visualized using the following steps:

Loading Data: The analysis results were saved in an output file (transDisp) using the record command.

Data visualization: to visualize the results, the matplotlib library was used as can be seen below.

Finally, The figure obtained can be seen in the next slide.

FREE VIBRATION OF LOADED NODE



REFERENCES

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

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