TRANSIENT ANALYSIS (FREQUENCY RESPONSE FUNCTION)

OpenSeesPy tutorial 5

TUTORIAL 5. INTRODUCTION

In this session, we've moved from the previous transient analysis to explore frequenca response function analysis. Our goal remains he same: to facilitate a comprehensive understanding of OpenSeesPy's fundamental framework.

While we begin with straightforward models to solidify your grasp of structural engineering basics, rest assured that this series also presents opportunities to tackle more intricate analyses in upcoming tutorials.

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

Length (L): 1.5 m

Diameter (D): 0.5 inches

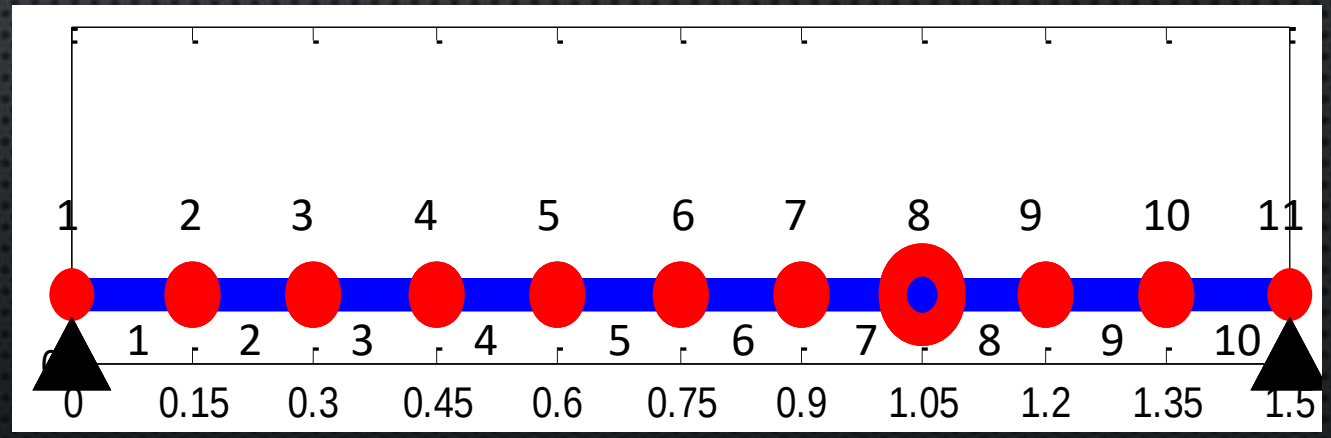
Mass (m): 40 kg

Density (rho): 7850 kg/m³

Young's Modulus (E): 2.1 GPa

Simply supported shaft.

Mass excess at 0.7L from left.



WIPE COMMAND

```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 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The wipe command ensures that **all openseespy variables** and previously defined commands are no longer active in the current model.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 ##############################
# 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 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 AND MATERIAL PARAMETERS

```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 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 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 cruscial 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

```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 Inert
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)
```

```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

```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 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)
```

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 e
        op.node(i, (i - 1) * L / (Nmax - 1), 0)
    freq_Ang.append(eigenvalue
```

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 trasnformation 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 (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)
```

- 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.

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)

########################### 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)
```

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.

DYNAMIC LOAD DEFINITION

```
############################ TRANSIENT ANALYSIS ################################
########################## DYNAMIC LOAD DEFINITION ##############################
# time step
dt = 0.001                          # (s)
# final time
tEnd = 30.0                         # (s)
# number of steps
nSteps = int(tEnd / dt)
# arrray containing all time
t = np.linspace(0, tEnd, nSteps) # (s)
# minimum angular frequency to consider
omegaMin = freq_Ang[0] / 3          # (rad/s)
# maximun angular frequency to consider
omegaMax = 3 * freq_Ang[2]          # (rad/s)
#number of frequencies to consider
omegaN = 1000
# list of Dynamic load frequency to consider
Omega = np.linspace(omegaMin, omegaMax, omegaN)
# list containing the maximun displament of selected node per algunar frequency
node_Disp_Omega = []
INDEX = 0
```

```
for omega in Omega:

    # load 1
    f1 = 0.05 * 1e-2 * m * omega**2 * np.sin(omega * t)                  # (N)
    # load 2
    f2 = 0.05 * 1e-2 * m * omega**2 * np.sin(0.5 * omega * t - pi / 3)   # (N)
    # Resultant load
    f = f1 + f2                                                          # (N)
    np.savetxt('time.txt', t)
    np.savetxt('Ampl.txt', f)
    # time series Tag
    tsTag = 1
    op.timeSeries('Path', tsTag, '-dt', dt, '-filePath', 'Ampl.txt', '-fileTim
```

In our previous tutorial (focused on free vibration), we explored the behavior of a system by defining initial displacement and observing it as it returned to equilibrium. In this session, we dive into forced vibration, where the dynamic loads are essential throughout the entire analysis. This slide introduces the concept of dynamic loads by defining two simple sinusoidal loads within distinct frequencies and phases.

DYNAMIC LOAD DEFINITION (CONTINUED)

```
########################## TRANSIENT ANALYSIS ##############################
######################### DYNAMIC LOAD DEFINITION ##########################
# time step
dt = 0.001                          # (s)
# final time
tEnd = 30.0                         # (s)
# number of steps
nSteps = int(tEnd / dt)
# arrray containing all time
t = np.linspace(0, tEnd, nSteps) # (s)
# minimum angular frequency to consider
omegaMin = freq_Ang[0] / 3          # (rad/s)
# maximun angular frequency to consider
omegaMax = 3 * freq_Ang[2]          # (rad/s)
#number of frequencies to consider
omegaN = 1000
# list of Dynamic load frequency to consider
Omega = np.linspace(omegaMin, omegaMax, omegaN)
# list containing the maximun displament of selected node per algunar frequency
node_Disp_Omega = []
INDEX = 0

for omega in Omega:

    # load 1
    f1 = 0.05 * 1e-2 * m * omega**2 * np.sin(omega * t)                 # (N)
    # load 2
    f2 = 0.05 * 1e-2 * m * omega**2 * np.sin(0.5 * omega * t - pi / 3)  # (N)
    # Resultant load
    f = f1 + f2                                                         # (N)
    np.savetxt('time.txt', t)
    np.savetxt('Ampl.txt', f)
    # time series Tag
    tsTag = 1
    op.timeSeries('Path', tsTag, '-dt', dt, '-filePath', 'Ampl.txt', '-fileTime
```

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.

CONSTRAINTS COMMAND

```
########################  ANALYSIS PARAMETERS ##############################
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.record()
op.analyze(nSteps, dt)

op.wipe()
################################### PLOTTING #####################################
# list containing the recorded displacement and time
dispList = np.loadtxt('transientAn.out')
# list containing hte saved time
time = np.loadtxt('time.txt')
# list containing the saved loads
load = np.loadtxt('Ampl.txt')
################################ DYNAMIC RESPONSE ################################
from matplotlib import pyplot as plt
plt.plot(dispList[:,0], dispList[:,1])
plt.xlabel('time (s)')
plt.ylabel('y Displacement (m)')
################################## DYNAMIC LOAD ##################################
plt.figure()
plt.plot(time, load)
plt.xlabel('time (s)')
plt.ylabel('load (N)')
```

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

```
##########################  ANALYSIS PARAMETERS ##############################
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.record()
op.analyze(nSteps, dt)

op.wipe()
################################### PLOTTING #####################################
# list containing the recorded displacement and time
dispList = np.loadtxt('transientAn.out')
# list containing hte saved time
time = np.loadtxt('time.txt')
# list containing the saved loads
load = np.loadtxt('Ampl.txt')
################################ DYNAMIC RESPONSE ################################
from matplotlib import pyplot as plt
plt.plot(dispList[:,0], dispList[:,1])
plt.xlabel('time (s)')
plt.ylabel('y Displacement (m)')
################################## DYNAMIC LOAD ##################################
plt.figure()
plt.plot(time, load)
plt.xlabel('time (s)')
plt.ylabel('load (N)')
```

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 PARAMETERS ##############################
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.record()
op.analyze(nSteps, dt)

op.wipe()
################################### PLOTTING #####################################
# list containing the recorded displacement and time
dispList = np.loadtxt('transientAn.out')
# list containing hte saved time
time = np.loadtxt('time.txt')
# list containing the saved loads
load = np.loadtxt('Ampl.txt')
################################ DYNAMIC RESPONSE ################################
from matplotlib import pyplot as plt
plt.plot(dispList[:,0], dispList[:,1])
plt.xlabel('time (s)')
plt.ylabel('y Displacement (m)')
################################## DYNAMIC LOAD ##################################
plt.figure()
plt.plot(time, load)
plt.xlabel('time (s)')
plt.ylabel('load (N)')
```

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 blems 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 PARAMETERS ###############################
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.record()
op.analyze(nSteps, dt)

op.wipe()
################################### PLOTTING ###########################################
# list containing the recorded displacement and time
dispList = np.loadtxt('transientAn.out')
# list containing hte saved time
time = np.loadtxt('time.txt')
# list containing the saved loads
load = np.loadtxt('Ampl.txt')
################################## DYNAMIC RESPONSE ##################################
from matplotlib import pyplot as plt
plt.plot(dispList[:,0], dispList[:,1])
plt.xlabel('time (s)')
plt.ylabel('y Displacement (m)')
################################## DYNAMIC LOAD ######################################
plt.figure()
plt.plot(time, load)
plt.xlabel('time (s)')
plt.ylabel('Load (N)')
```

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 PARAMETERS ##############################
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.record()
op.analyze(nSteps, dt)

op.wipe()
################################### PLOTTING #######################################
# list containing the recorded displacement and time
dispList = np.loadtxt('transientAn.out')
# list containing hte saved time
time = np.loadtxt('time.txt')
# list containing the saved loads
load = np.loadtxt('Ampl.txt')
################################## DYNAMIC RESPONSE ################################
from matplotlib import pyplot as plt
plt.plot(dispList[:,0], dispList[:,1])
plt.xlabel('time (s)')
plt.ylabel('y Displacement (m)')
################################## DYNAMIC LOAD ####################################
plt.figure()
plt.plot(time, load)
plt.xlabel('time (s)')
plt.ylabel('Load (N)')
```

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

```
############################  ANALYSIS PARAMETERS ###############################
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.record()
op.analyze(nSteps, dt)

op.wipe()
################################### PLOTTING #########################################
# list containing the recorded displacement and time
dispList = np.loadtxt('transientAn.out')
# list containing hte saved time
time = np.loadtxt('time.txt')
# list containing the saved loads
load = np.loadtxt('Ampl.txt')
################################## DYNAMIC RESPONSE ##################################
from matplotlib import pyplot as plt
plt.plot(dispList[:,0], dispList[:,1])
plt.xlabel('time (s)')
plt.ylabel('y Displacement (m)')
################################### DYNAMIC LOAD #####################################
plt.figure()
plt.plot(time, load)
plt.xlabel('time (s)')
plt.ylabel('load (N)')
```

The integrator command is a critical component used to create an integrator object, which holds a pivotal role in determining the interpretation of terms within the system of equations Ax=B.
This command is employed to determine:

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

The integrator offers a diverse array of options to customize the analysis to specific requirements. These option include: CentralDifference, Newmark, Hilber-Hughes-Taylor, Explicit Difference, and more.
For static analysis, alternative options are available to suit your specific needs.

ANALYSIS COMMAND

```
##########################  ANALYSIS PARAMETERS ##############################
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.record()
op.analyze(nSteps, dt)

op.wipe()
################################### PLOTTING #####################################
# list containing the recorded displacement and time
dispList = np.loadtxt('transientAn.out')
# list containing hte saved time
time = np.loadtxt('time.txt')
# list containing the saved loads
load = np.loadtxt('Ampl.txt')
################################ DYNAMIC RESPONSE ################################
from matplotlib import pyplot as plt
plt.plot(dispList[:,0], dispList[:,1])
plt.xlabel('time (s)')
plt.ylabel('y Displacement (m)')
################################## DYNAMIC LOAD ##################################
plt.figure()
plt.plot(time, load)
plt.xlabel('time (s)')
plt.ylabel('load (N)')
```

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

```
############################  ANALYSIS PARAMETERS ###############################
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.record()
op.analyze(nSteps, dt)

op.wipe()
################################### PLOTTING ###################################
# list containing the recorded displacement and time
dispList = np.loadtxt('transientAn.out')
# list containing hte saved time
time = np.loadtxt('time.txt')
# list containing the saved loads
load = np.loadtxt('Ampl.txt')
################################ DYNAMIC RESPONSE ##############################
from matplotlib import pyplot as plt
plt.plot(dispList[:,0], dispList[:,1])
plt.xlabel('time (s)')
plt.ylabel('y Displacement (m)')
################################## DYNAMIC LOAD ################################
plt.figure()
plt.plot(time, load)
plt.xlabel('time (s)')
plt.ylabel('load (N)')
```

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.

RESTARTING ANALYSIS

```
############################ RESTART OF ANALYSIS ################################
    # list containing the recorded displacement and time
    dispList = np.loadtxt('transientAn.out')
    node_Disp_Omega.append(1000*max(dispList[int(nSteps / 1.2): -2, 1]))

    op.model('Basic', '-ndm', 2, '-ndf', 3)

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

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

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

############################## DEFINE ELEMENTS ##################################
    op.geomTransf('Linear', transTag)
    for index_ele in range(1, Nmax):
            op.element('elasticBeamColumn', index_ele,
                       *[index_ele, index_ele + 1], A, E, Iz, transTag,'-mass',
    INDEX += 1
    print(f'Step Number: {INDEX}/{omegaN} done')
```

```
################################ PLOTTING ###################################
from matplotlib import pyplot as plt
plt.plot(Omega,node_Disp_Omega)
plt.xlabel('Angular frequency (rad/s)')
plt.ylabel('max. Amplitude (mm)')
```

In this tutorial, our primary objective is to assess the maximum amplitudes of the response signal across a broad range of dynamic load frequencies. To achieve this, we've adopted a strategy of restarting the analysis after evaluating each frequency point.

It's essential to note that during each restart, the only parameter that undergoes change in the analysis is the load frequency value. All other conditions and parameters remain constant. This approach ensures the consistency of conditions and aids in the validation of our analysis results.

PLOTTING

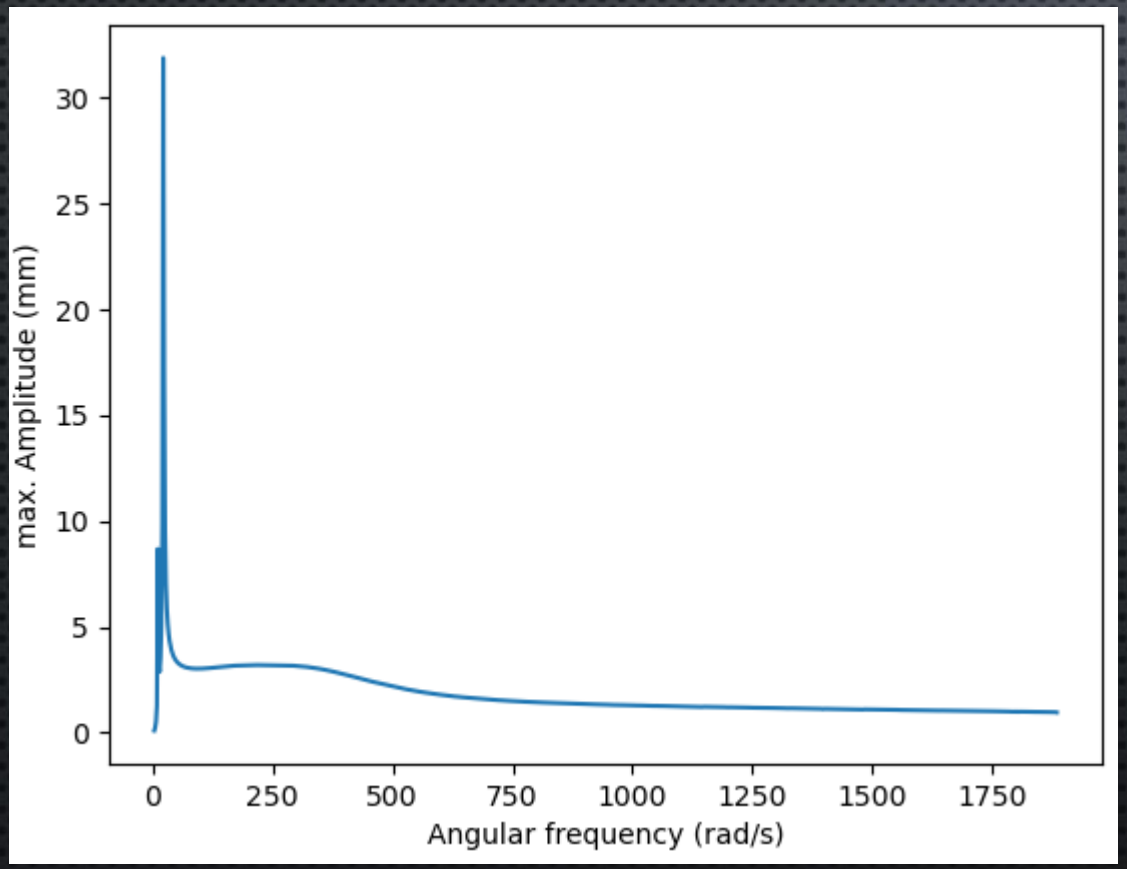
```
############################ RESTART OF ANALYSIS ##############################
    # list containing the recorded displacement and time
    dispList = np.loadtxt('transientAn.out')
    node_Disp_Omega.append(1000*max(dispList[int(nSteps / 1.2): -2, 1]))

    op.model('Basic', '-ndm', 2, '-ndf', 3)

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

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

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

############################## DEFINE ELEMENTS ################################
    op.geomTransf('Linear', transTag)
    for index_ele in range(1, Nmax):
            op.element('elasticBeamColumn', index_ele,
                       *[index_ele, index_ele + 1], A, E, Iz, transTag,'-mass',
    INDEX += 1
    print(f'Step Number: {INDEX}/{omegaN} done')

################################## PLOTTING ###################################
from matplotlib import pyplot as plt
plt.plot(Omega,node_Disp_Omega)
plt.xlabel('Angular frequency (rad/s)')
plt.ylabel('max. Amplitude (mm)')
```

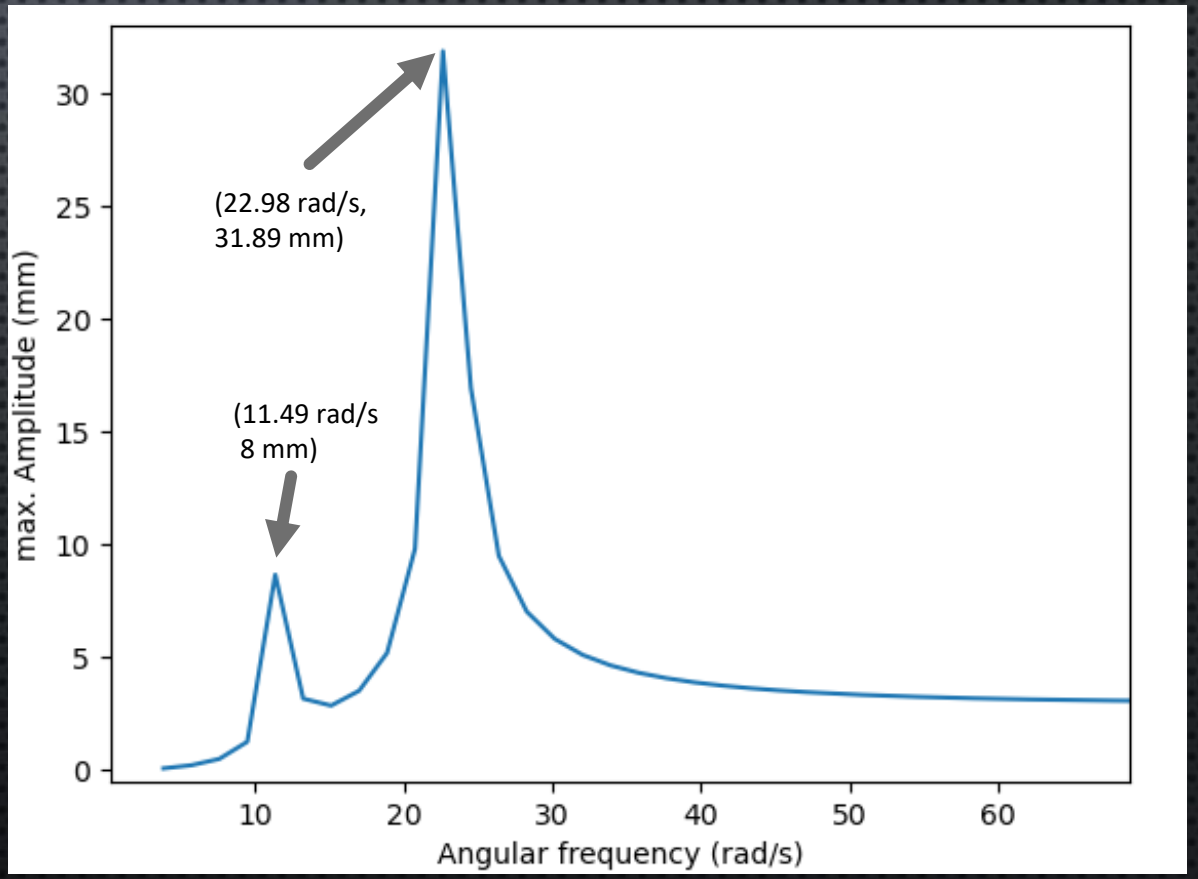
Once the analysis is completed, it was proceeded to visualize and the results.
In the upper part of the slide, we detail how to plot and visualize modes that were previously recorded during the analysis.

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

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

FREQUENCY RESPONSE FUNCTION





REFERENCES

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

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