pywfe

Austen Stone

Nov 10, 2023

CONTENTS:

CHAPTER

ONE

PYWFE - A PYTHON PACKAGE FOR THE WAVE FINITE ELEMENT METHOD

This package implements the Wave Finite Element Method (WFEM) in Python to analyse guided waves in 1 dimension. Initially Written to analyse mechanical waves in fluid filled pipes meshed in COMSOL.

Currently only works for infinite waveguides.

The *pywfe.Model* class provides a high level api to calculate the free and forced response in the waveguide.

1.1 pyWFE Package

pywfe. A Python implementation of the WFE method

1.2 pywfe.Model Class

1.2.1 Introduction

Brief description of the class and its purpose.

1.2.2 Constructor

`Model.__init__`(*K*, *M*, *dof*, *null=None*, *nullf=None*, *axis=0*, *logging_level=20*, *solver='transfer_matrix'*)

Initialise a Model object.

Parameters

K

[np.ndarray] Stiffness matrix $\mathbf{K}$ of shape (N, N).

M

[np.ndarray] Mass matrix $\mathbf{M}$ of shape (N, N).

dof

[dict] A dictionary containing the following keys:

- 'coord' : array-like, shape (n_{dim}, N) Coordinates of the degrees of freedom, where n_{dim} is the number of spatial dimensions and N is the total number of degrees of freedom in the initial total mesh.
- 'node' : array-like, shape $(N,)$ Node number that the degree of freedom sits on.
- 'fieldvar' : array-like, shape $(N,)$ Field variable for the degree of freedom (e.g., pressure, displacement in x, displacement in y).
- 'index' : array-like, shape $(N,)$ Index of the degree of freedom, used to keep track of the degrees of freedom when sorted.

null
[ndarray, optional] Null space constraint matrix (for boundary conditions) of shape (N, N). The default is None.

nullf
[ndarray, optional] Force null space constraint matrix (for boundary conditions) of shape (N, N). The default is None.

axis
[int, optional] The waveguide axis. Moves *dof['coord'][axis]* to *dof['coord'][0]*. The default is 0.

logging_level
[int, optional] Logging level. The default is 20.

solver
[str, optional] The form of the eigenvalue to use. The default is "transfer_matrix". Options are currently 'transfer_matrix' or 'polynomial'.

Returns

None.

1.2.3 Attributes

`Model.K`
Sorted stiffness matrix

`Model.M`
Sorted mass matrix

`Model.dof`
Sorted dof dictionary

`Model.node`
dictionary of node information

`Model.K_sub`
dictionary containing substructured stiffness matrices `'LL'`, `'LR`, `'RL'`, `'RR'`, `'LI'`, `'IL'`, `'RI'`, `'IR'`, `'II'`

`Model.M_sub`
dictionary containing substructured mass matrices.

`Model.eigensolution`
The eigensolution at a given frequency. Gives values and vectors corresponding to propagation constants and mode shapes

Model.**force**

The force vector corresponding to forces at each dof

1.2.4 Methods

class pywfe.**Model**(*K*, *M*, *dof*, *null=None*, *nullf=None*, *axis=0*, *logging_level=20*, *solver='transfer_matrix'*)

The main high level api in the pywfe package.

Methods

dispersion_relation(frequency_array[, ...])	Calculate frequency-wavenumber relation
displacements(x_r[, f, dofs])	gets the displacements for all degrees of freedom at specified x and f.
dofs_to_indices(dofs)	Generates indices for selected dofs
excited_amplitudes([f])	Find the excited amplitudes subject to a given force and frequency.
forces(x_r[, f, dofs])	Gets the total force on each degree of freedom.
form_dsm(f)	Forms the DSM of the model at a given frequency
frequency_sweep(f_arr[, x_r, quantities, ...])	Solves various quantities over specified frequency and response range.
generate_eigensolution(f)	Generates the sorted eigensolution at a given frequency.
left_dofs()	get the dofs on the left face of the segment
modal_displacements(x_r[, f, dofs])	Calculate the modal displacements at a given distance and frequency.
modal_forces(x_r[, f, dofs])	Generates the modal forces at given distance and frequency
phase_velocity(frequency_array[, direction, ...])	gets the phase velocity curves for a given frequency array
propagated_amplitudes(x_r[, f])	Calculate the propagated and superimposed amplitudes for a given distance and frequency.
save(folder[, source])	Save the model to a folder
see()	Creates interactive matplotlib widget to visualise mesh and inspect degrees of freedom.
select_dofs([fieldvar])	select the model degrees of freedom that correspond to specified field variable.
selection_index(dof)	Get the dof indices for a given selection.
transfer_function(f_arr, x_r[, dofs, derivative])	Gets the displacement over frequency at specified distance and dofs.
wavenumbers([f, direction, imag_threshold])	Calculates the wavenumbers of the system at a given frequency

solver

Description of solver.

K

Sorted stiffness matrix

M

Sorted mass matrix

`dof`

Sorted dof dictionary

`K_sub`

dictionary containing substructured stiffness matrices `'LL', 'LR, 'RL', 'RR', 'LI', 'IL', 'RI', 'IR', 'II'`

`M_sub`

dictionary containing substructured mass matrices.

`node`

dictionary of node information

`delta`

Waveguide segment length

`N`

Number of dofs on both left and right faces combined

`eigensolution`

The eigensolution at a given frequency. Gives values and vectors corresponding to propagation constants and mode shapes

`force`

The force vector corresponding to forces at each dof

`dofs_to_indices`(*dofs*)

Generates indices for selected dofs

Parameters

dofs
[str or list or dict] 'all' specifies all dofs. A list of integers is interpreted as the dof indices. A dof dictionary, created with *model.select_dofs()*

Returns

inds
[np.ndarray] array of dof indices.

`form_dsm`(*f*)

Forms the DSM of the model at a given frequency

Parameters

f
[float] frequency at which to form the DSM.

Returns

DSM
[ndarray] (ndof, ndof) sized array of type complex. The condensed DSM.

`generate_eigensolution`(*f*)

Generates the sorted eigensolution at a given frequency. If frequency is None or the presently calculated frequency, then reuse the previously calculated eigensolution.

Parameters

f
[float] The frequency at which to calculate the eigensolution.

Returns

eigensolution
[Eigensolution (namedtuple)]

The sorted eigensolution. The named tuple fields are:

- lambda_[plus]/[minus] : +/- going eigenvalues
- phi_[plus]/[minus] : +/- going right eigenvectors
- psi_[plus]/[minus] : +/- going left eigenvectors

`wavenumbers`(*f=None*, *direction='plus'*, *imag_threshold=None*)

Calculates the wavenumbers of the system at a given frequency

Parameters

f
[float, optional] Frequency at which to calculated wavenumbers. The default is None.

direction
[str, optional] Choose positive going or negative going waves. The default is "plus".

imag_threshold
[float, optional] Imaginary part of wavenumber above which will be set to np.nan. The default is None.

Returns

k
[ndarray] The array of wavenumbers at this frequency.

`dispersion_relation`(*frequency_array*, *direction='plus'*, *imag_threshold=None*)

Calculate frequency-wavenumber relation

Parameters

frequency_array
[ndarray] Frequencies to calculate.

direction
[str, optional] Choose positive going or negative going waves. The default is "plus".

imag_threshold
[float, optional] Imaginary part of wavenumber above which will be set to np.nan. The default is None.

Returns

wavenumbers
[ndarray] (nfreq, n_waves) sized array of type complex.

`phase_velocity`(*frequency_array*, *direction='plus'*, *imag_threshold=None*)

gets the phase velocity curves for a given frequency array

Parameters

frequency_array
[np.ndarray] DESCRIPTION.

direction
[str, optional] Direction of the waves. The default is 'plus'.

imag_threshold
[float, optional] Imaginary threshold above which set to np.nan. The default is None.

Returns

ndarray
phase velocity.

`excited_amplitudes`(*f=None*)

Find the excited amplitudes subject to a given force and frequency. If the solution has already been calculated for the same inputs, reuse the old solution.

Parameters

f
[float, optional] Frequency. The default is None.

Returns

e_plus
[ndarray] Positive excited wave amplitudes.

e_minus
[ndarray] Negative excited wave amplitudes.

`propagated_amplitudes`(*x_r*, *f=None*)

Calculate the propagated and superimposed amplitudes for a given distance and frequency.

Parameters

x_r
[float] Axial response distance.

f
[float, optional] Frequency. The default is None.

Returns

b_plus, b_minus
[ndarray] Positive and negative amplitudes.

`modal_displacements`(*x_r*, *f=None*, *dofs='all'*)

Calculate the modal displacements at a given distance and frequency. Each column corresponds to a different wavemode, each row is a different degree of freedom.

Parameters

x_r
[float] Axial response distance.

f
[float, optional] Frequency. The default is None.

Returns

q_j_plus, q_j_minus
[ndarray] The modal displacements for positive and negative going waves.

`displacements`(*x_r*, *f=None*, *dofs='all'*)

gets the displacements for all degrees of freedom at specified x and f.

Parameters

x_r
[float] response distance (can be array like).

f
[float, optional] Frequency. The default is None.

Returns

ndarray
displacements for each degree of freedom.

`modal_forces`(*x_r*, *f=None*, *dofs='all'*)

Generates the modal forces at given distance and frequency

Parameters

x_r
[float] Response distance.

f
[float, optional] Frequency. The default is None.

Returns

np.ndarray
modal force array.

`forces`(*x_r*, *f=None*, *dofs='all'*)

Gets the total force on each degree of freedom.

Parameters

x_r
[float] Response distance.

f
[float, optional] Frequency. The default is None.

Returns

np.ndarray
forces.

`frequency_sweep`(*f_arr*, *x_r=0*, *quantities=['displacements']*, *mac=False*, *dofs='all'*)

Solves various quantities over specified frequency and response range. Includes Modal Assurance Critereon (MAC) sorting.

Parameters

f_arr
[np.ndarray] Array of frequencies.

x_r
[float or np.ndarray, optional] Response distance. The default is 0.

quantities
[list, optional] Quantities to solve for. The default is ['displacements'].

mac
[bool, optional] Whether to sort modal quantities according to MAC. The default is False.

dofs
[list, optional] Select specific degrees of freedom. The default is 'all'.

Returns

dict
Dictionary of output for specified quantities.

`transfer_function`(*f_arr*, *x_r*, *dofs='all'*, *derivative=0*)

Gets the displacement over frequency at specified distance and dofs.

Parameters

f_arr
[np.ndarray] Frequency array.

x_r
[float or np.ndarray] Response distance.

dofs
[list, optional] List of dofs to return. The default is "all".

Returns

ndarray
Displacements over frequency and distance.

`select_dofs`(*fieldvar=None*)

select the model degrees of freedom that correspond to specified field variable.

Parameters

fieldvar
[str or list, optional] The fieldvariable or list thereof to select for. The default is None.

Returns

dofs
[dict] Reduced dof dictionary.

`left_dofs`()

get the dofs on the left face of the segment

Returns

dofs
[dict] dof dictionary.

`selection_index`(*dof*)

Get the dof indices for a given selection.

Parameters

dof
[dict] dof dictionary.

Returns

np.ndarray
1D array of indices for selected dofs.

`see`()

Creates interactive matplotlib widget to visualise mesh and inspect degrees of freedom.

Returns

None.

`save`(*folder*, *source='local'*)

Save the model to a folder

Parameters

folder
[str] folder name.

source
[str, optional] Save to `'local'` or `'database'`. The default is 'local'.

Returns

None.

1.3 core

1.3.1 model_setup

This module contains the functions for setting up the WFE model.

This includes:

- Creating the relevant *dof* dict data
- Applying the boundary conditions
- Sorting M, K and dofs to left and right faces
- Creating *node* dict data

`pywfe.core.model_setup.`**`generate_dof_info`**(*dof: dict*, *axis=0*)

Generates the *dof* dictionary, including which face each dof is on. Also rolls sets the waveguide axis and created index array if none given.

Parameters

dof
[dict] DESCRIPTION.

axis
[TYPE, optional] DESCRIPTION. The default is 0.

Returns

dof
[TYPE] DESCRIPTION.

`pywfe.core.model_setup.`**`apply_boundary_conditions`**(*K*, *M*, *dof*, *null*, *nullf*)

Applies boundary conditions according to null constraint matrices. Resorts and removes degrees of freedom as needed. (NOT FINISHED)

Parameters

K
[ndarray] (ndof, ndof) sized array of type float or complex.

M
[ndarray] (ndof, ndof) sized array of type float or complex.

dof
[dict] dof dictionary.

null
[ndarray] (ndof, ndof) sized array of type float.

nullf
[ndarray] (ndof, ndof) sized array of type float.

Returns

K
[ndarray] (ndof, ndof) sized array of type float or complex.

M
[ndarray] (ndof, ndof) sized array of type float or complex.

dof
[dict] dof dictionary.

pywfe.core.model_setup.**order_system_faces**(*K*, *M*, *dof*)

Parameters

K
[ndarray] (ndof, ndof) sized array of type float or complex.

M
[ndarray] (ndof, ndof) sized array of type float or complex.

dof
[dict] dof dictionary.

Returns

K
[ndarray] (ndof, ndof) sized array of type float or complex.

M
[ndarray] (ndof, ndof) sized array of type float or complex.

dof
[dict] dof dictionary.

pywfe.core.model_setup.**substructure_matrices**(*K*, *M*, *dof*)

Creates dictionaries for the submatrices of K and M

Parameters

K
[ndarray] (ndof, ndof) sized array of type float or complex.

M
[ndarray] (ndof, ndof) sized array of type float or complex.

dof
[dict] dof dictionary.

Returns

K_sub
[dict] dictionary of substructured stiffness matrices.

M_sub
[dict] dictionary of substructured mass matrices.

pywfe.core.model_setup.**create_node_dict**(*dof*)

Creates node dictionary for nodes on the left face of the model

Parameters

dof
[dict] dof dictionary.

Returns

node
[dict] node dictionary.

1.3.2 eigensolvers

This module contains different solvers for the WFE eigenproblem.

pywfe.core.eigensolvers.**transfer_matrix**(*DSM*)

Classical transfer matrix formulation of the WFE eigenproblem.

The transfer function is defined as

$$\mathbf{T} = \begin{bmatrix} -D_{LR}^{-1}D_{LL} & D_{LR}^{-1} \\ -D_{RL} + D_{RR}D_{LR}^{-1}D_{LL} & -D_{RR}D_{LR}^{-1} \end{bmatrix}$$

which leads to the eigenvalue problem

$$T\mathbf{\Phi} = \lambda\mathbf{\Phi}$$

The left eigenvectors can be found by considering $\mathbf{T}^T$

Parameters

DSM
[(N,N) ndarray (float or complex)] The dynamic stiffness matrix of the system. NxN array of type float or complex.

Returns

vals
[ndarray] 1-D array of length N type complex.

left_eigenvectors
[ndarray] NxN array of type float or complex. Column i is vector corresponding to vals[i]

right_eigenvectors
[ndarray] NxN array of type float or complex. Column i is vector corresponding to vals[i]

pywfe.core.eigensolvers.**polynomial**(*DSM*)

[unfinished] Polynomial form of the eigenproblem

Parameters

DSM
[(N,N) ndarray (float or complex)] The dynamic stiffness matristrucaxisym-x of the system. NxN array of type float or complex.

Returns

vals
[ndarray] 1-D array of length N type complex.

left_eigenvectors
[ndarray] NxN array of type float or complex. Column i is vector corresponding to vals[i]

right_eigenvectors
[ndarray] NxN array of type float or complex. Column i is vector corresponding to vals[i]

1.3.3 classify_modes

This module contains the functionality needed to sort eigensolutions of the WFE method into positive and negative going waves.

pywfe.core.classify_modes.**classify_wavemode**(*f*, *eigenvalue*, *eigenvector*, *threshold*)

Identify if a wavemode is positive going or negative going

Parameters

f
[float] frequency of eigensolution.

eigenvalue
[complex] Eigenvalue to be checked.

eigenvector
[nodarray, complex] Corresponding eigenvector.

threshold
[float] Threshold for classification. How close to unity does an eigenvalue have to be?

Returns

direction
[str] `'right'` or `'left'`.

pywfe.core.classify_modes.**sort_eigensolution**(*f*, *eigenvalues*, *right_eigenvectors*, *left_eigenvectors*)

Sort the eigensolution into positive and negative going waves

Parameters

f
[float] Frequency of eigensolution.

eigenvalues
[ndarray, complex] Eigenvalues solved at this frequency.

right_eigenvectors
[ndarray, complex] Right eigenvectors solved at this frequency.

left_eigenvectors
[TYPE] Left eigenvectors solved at this frequency..

Returns

named tuple
Eigensolution tuple.

1.3.4 forced_problem

This module contains the functionality needed to apply forces to a WFE model.

pywfe.core.forced_problem.**calculate_excited_amplitudes**(*eigensolution*, *force*)

Calculates the directly excited amplitudes subject to a given force and modal solution.

Parameters

eigensolution
[namedtuple] eigensolution.

force
[np.ndarray] force vector.

Returns

e_plus
[np.ndarray] directly excited modal amplitudes (positive).

e_minus
[np.ndarray] directly excited modal amplitudes (negative).

pywfe.core.forced_problem.**generate_reflection_matrices**(*eigensolution*, *A_right*, *B_right*, *A_left*, *B_left*)

Calculates the reflection matrices from boundary matrices.

Parameters

eigensolution
[TYPE] DESCRIPTION.

A_right
[np.ndarray] A matrix on the right boundary.

B_right
[np.ndarray] B matrix on the right boundary.

A_left
[np.ndarray] A natrix on the left boundary.

B_left
[np.ndarray] B matrix on the left boundary.

Returns

R_right
[np.ndarray] Right reflection matrix.

R_left
[np.ndarray] Left reflection matrix.

pywfe.core.forced_problem.**calculate_propagated_amplitudes**(*e_plus*, *e_minus*, *k_plus*, *L*, *R_right*, *R_left*, *x_r*, *x_e=0*)

Calculates the ampltiudes of waves after propagation to response point

Parameters

e_plus
[np.ndarray] positive directly excited amplitudes.

e_minus
[np.ndarray] negative directly excited amplitudes.

k_plus
[np.ndarray] wavenumber array.

L
[float] Length of waveguide.

R_right
[np.ndarray] Right reflection matrix.

R_left
[np.ndarray] Left reflection matrix.

x_r
[float, np.ndarray] Response distance.

x_e
[float,] Excitation distance. The default is 0.

Returns

b_plus
[np.ndarray] positive propagated amplitudes.

b_minus
[np.ndarray] negative propagated amplitudes.

pywfe.core.forced_problem.**calculate_modal_displacements**(*eigensolution*, *b_plus*, *b_minus*)

Calculates the displacement of each mode (last axis is modal)

Parameters

eigensolution
[namedtuple] eigensolution.

b_plus
[np.ndarray] positive propagated amplitudes.

b_minus
[np.ndarray] negative propagated amplitudes.

Returns

q_j_plus
[np.ndarray] positive going modal displacements.

q_j_minus
[np.ndarray] negative going modal displacements.

pywfe.core.forced_problem.**calculate_modal_forces**(*eigensolution*, *b_plus*, *b_minus*)

Calculates the internal forces of each mode (last axis is modal)

Parameters

eigensolution
[namedtuple] eigensolution.

b_plus
[np.ndarray] positive propagated amplitudes.

b_minus
[np.ndarray] negative propagated amplitudes.

Returns

f_j_plus
[np.ndarray] Positive going modal forces.

f_j_minus
[np.ndarray] Negative going modal forces.

1.4 utils

1.4.1 io_utils

This module contains the functionality needed to save and load pywfe.Model objects

1.4.2 comsol_loader

This module contains the functionality needed to convert COMSOL data extracted from MATLAB LiveLink into a pywfe.Model class.

`pywfe.utils.comsol_loader.`**`load_comsol`**(*folder*, *axis=0*, *logging_level=20*, *solver='transfer_matrix'*)

Parameters

folder
[string] path to the folder containing the COMSOL LiveLink data.

axis
[int, optional] Waveguide axis. The default is 0.

logging_level
[int, optional] Logging level. The default is 20 (info).

Returns

model
[pywfe.model class] a pywfe model.

`pywfe.utils.comsol_loader.`**`comsol_i2j`**(*filename*, *skiprows=0*)

Converts complex 'j' imaginary unit from COMSOL to python 'j'

Parameters

filename
[string] filename to convert.

skiprows
[int, optional] see numpy loadtxt. The default is 1.

Returns

None.

1.4.3 frequency_sweep

This module contains the fucntion for calculating various quantities over an array of frequencies for a pywfe.Model object.

pywfe.utils.frequency_sweep.**frequency_sweep**(*model*, *f_arr*, *quantities*, *x_r=0*, *mac=False*, *dofs='all'*)

Perform a sweep over frequency array, extracting specified quatities at each step. Modal assurance criterion can be used to track modes through frequency by modeshape similarity

Parameters

model
[pywfe.Model] The model to perform the sweep with.

f_arr
[np.ndarray float] frequency array.

quantities
[list of str type] a list of strings specifying the quantities to be calculated. These are: - phi_plus: the (positive going) eigenvectors - excited_amplitudes: see *pywfe.Model.excited_amplitudes* - propagated_amplitudes: see *pywfe.Model.propagated_amplitudes* - modal_displacements: see *pywfe.Model.modal_displacements* - wavenumbers: see *pywfe.Model.wavenumbers* - displacements: see *pywfe.Model.displacements* - forces: see *pywfe.Model.forces*

x_r
[float, np.ndarray, optional] response distance(s). The default is 0.

mac
[bool, optional] Use the modal assurance criterion to sort waves. The default is False.

dofs
[dofs, optional] The selected degrees of freedom. See *pywfe.Model.dofs_to_inds*. The default is 'all'.

Returns

output
[dict] Dictionary of outputs for specified quantities.

1.4.4 modal_assurance

This module contains functions for sorting frequency sweept data by mode index

pywfe.utils.modal_assurance.**mac_matrix**(*modes_prev*, *modes_next*)

Compute the Modal Assurance Criterion (MAC) matrix.

Created on Tue Aug 22 11:24:28 2023

@author: Austen

1.5 Examples

Here you can find examples that demonstrate how to use the *pywfe* package.

1.5.1 Analytical Beam Example

In this example, we'll go through the process of setting up a model of an Euler-Bernoulli beam using the `pywfe` package.

Introduction

An Euler-Bernoulli beam can be described with a finite element approximation giving the mass and stiffness matrices:

$$\mathbf{M} = \frac{\rho A l}{420}\begin{bmatrix} 156 & 22l & 54 & -13l \\ 22l & 4l^2 & 13l & -3l^2 \\ 54 & 13l & 156 & -22l \\ -13l & -3l^2 & -22l & 4l^2 \end{bmatrix} \quad \mathbf{K} = \frac{EI}{l^3}\begin{bmatrix} 12 & 6l & -12 & 6l \\ 6l & 4l^2 & -6l & 2l^2 \\ -12 & -6l & 12 & -6l \\ 6l & 2l^2 & -6l & 4l^2 \end{bmatrix}$$

For a beam segment of length l, cross-sectional area A made from a material with Young's modulus and density E, ρ, and second moment of area I. These matrices relate the displacement/rotation vector $[w_1, \theta_1, w_2, \theta_2]^T$ with the force/moment vector $[F_1, M_1, f_2, F_2]^T$ by

$$\begin{bmatrix} w_1 \\ \theta_1 \\ w_2 \\ \theta_2 \end{bmatrix} (\mathbf{K} - \omega^2 \mathbf{M}) = \begin{bmatrix} F_1 \\ M_1 \\ F_2 \\ M_2 \end{bmatrix}$$

The FE model only has two nodes with two degrees of freedom each. The analytical formulation of an infinite beam has well known solutions. The dispersion relation for transverse waves is

$$k = \sqrt{\frac{\omega}{a}}$$

The transfer mobility is subject to a transverse point force at $x = 0$ is

$$v(x, \omega) = -\frac{\omega}{4EIk^3}\left(ie^{-kx} - e^{-ikx}\right)$$

Creating pywfe Model of Beam

To begin with we define the system parameters

```
import numpy as np
import pywfe
import matplotlib.pyplot as plots

E = 2.1e11  # young mod
rho = 7850  # density
h = 0.1  # bean cross section side length length
A = h**2  # beam cross sectional area
I = h**4 / 12  # second moment of area

a = np.sqrt(E*I/(rho*A))  # factor in dispersion relation
```

and define the known solutions for the analytical dispersion relation and transfer mobility

```
def euler_wavenumber(f):
    # wavenumber of euler bernoulli beam
    return np.sqrt(2*np.pi*f/a)


def transfer_velocity(f, x):
    # transfer velocity for beam x > 0
    k = euler_wavenumber(f)
    omega = 2*np.pi*f

    return -omega/(4*E*I*k**3) * (1j*np.exp(-k*x) - np.exp(-1j*k*x))
```

For the FE discretisation, the beam length must be significantly shorter than the minimum wavelength. We define maximum frequency and find the maximum wavenumber analytically to set the beam length for WFE modelling.

```
f_max = 1e3  # maximum frequency
lambda_min = 2*np.pi/euler_wavenumber(f_max)  # mimimum wavelength
l_max = lambda_min / 10  # unit cell length max - 10 unit cells per wavelength

l = np.round(l_max, decimals=1)  # rounded unit cell length chosen
```

Now the mass and stiffness matrices can be defined

```
# stiffness matrix
K = E*I/(l**3) * np.array([

    [12,    6*l,    -12,   6*l],
    [6*l, 4*l**2, -6*l, 2*l**2],
    [-12,   -6*l,    12,  -6*l],
    [6*l, 2*l**2, -6*l, 4*l**2]

])

# mass matrix
M = rho*A*l/420 * np.array([
```

(continues on next page)

(continued from previous page)

```
    [156,   22*l,      54,    -13*l],
    [22*l,  4*l**2,  13*l,  -3*l**2],
    [54,    13*l,     156,    -22*l],
    [-13*l, -3*l**2,  -22*l, 4*l**2]

])
```

These, along with the 'mesh' information are all that are needed to create the *pywfe.Model* object. The mesh information is given with a dictionary with three keys *node*, *fieldvar* and *coord*. These specify the node number, field variable, and coordinates in 1-3D of each degree of freedom in the model. The beam has 4 degrees of freedom, ordered as in the displacement vectors. Thus we define the *dof* dictionary

```
dof = {'node': [0, 0, 1, 1],
    'fieldvar': ['w', 'phi']*2,
    'coord': [
                [0, 0, l, l],
                [0, 0, 0, 0]
]
}
```

which describes the two nodes, the field quantities *w*, *phi* (repeated on each node), and the coordinates of each degree of freedom. The coordinates are given in *x* and *y* with two lists for demonstrative purposes. Only the first is required for this 1D model.

The pywfe.Model object can now be created

```
beam_model = pywfe.Model(K, M, dof)
```

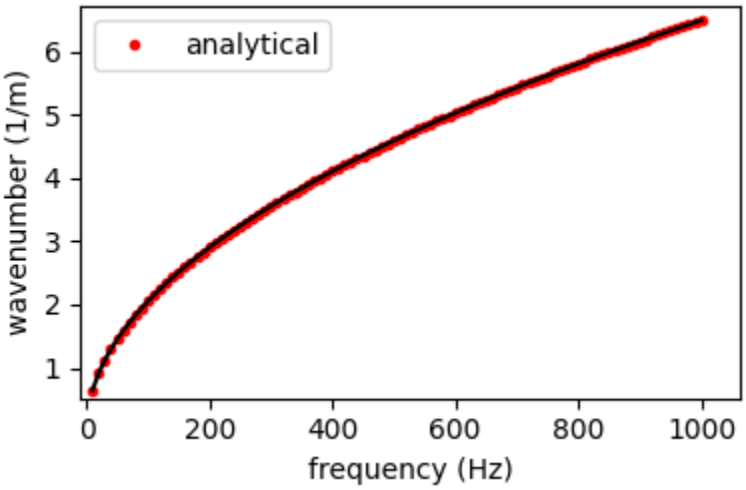
At this point, you might want to check the model with *pywfe.Model.see()*, which creates an interactive matplotlib view of the nodes in the mesh. In this case however there is only one node to look at.

Usage

Free Waves

Firstly let's check the dispersion relation with the analytical solution

```
#create frequency array
f_arr = np.linspace(10, f_max, 100)

# calculate the wfe wavenumbers
k_wfe = beam_model.dispersion_relation(f_arr)

plt.plot(f_arr, euler_wavenumber(f_arr), '.', color='red', label='analytical')
plt.plot(f_arr, k_wfe, color='black')

plt.legend(loc='best')
plt.xlabel("frequency (Hz)")
plt.ylabel("wavenumber (1/m)")
```

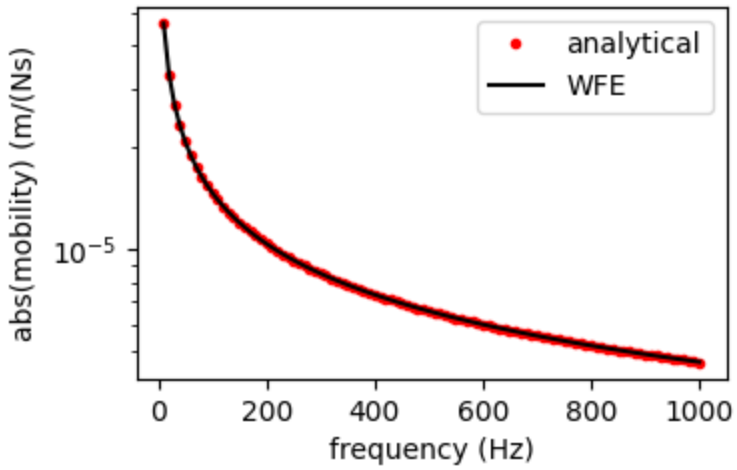


Forcing

Forces can be added to degrees of freedom by changing elements of the *Model.force* array. We compare the mobility in the WFE model with the known solution

```
beam_model.force[0] = 1

x_r = 0

w = beam_model.transfer_function(f_arr, x_r=x_r, dofs=[0], derivative=1)

plt.semilogy(f_arr, abs(transfer_velocity(f_arr, x_r)), '.', color='red', label=
↪'analytical')
plt.semilogy(f_arr, abs(w), color='black', label='WFE')

plt.legend(loc='best')
plt.xlabel("frequency (Hz)")
plt.ylabel("abs(mobility) (m/(Ns)")
```

The *transfer_function* method calculates the response over all frequencies at the response distance *x_r*. The response distance can also be a list or array, in which case a higher dimensional array will be returned. The *dofs* keyword argument specifies for which degrees of freedom the output should be returned. In this case we want the same dof as the one we're forcing. The *derivative* keyword argument applies n derivatives in the frequency domain, i.e a multiplication of the displacement by $i\omega$. So the output of the method call is the transverse velocity at x=0 for a transverse unit point force. This is the mobility of the beam and is compared with the analytical solution.



See *pywfe.Model.transfer_function()* for more information

More Functionality

For more functionality see *pywfe.Model*

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

C

D

E

F

G

K

L

M

N

O

P

S

T

W