Print to PDF

Tutorial - Modeling

Contents

This is a basic tutorial on how to use ROSS (rotordynamics open-source software), a Python library for rotordynamic analysis. Most of this code follows object-oriented paradigm, which is represented in this UML DIAGRAM.

Before starting the tutorial, it is worth noting some of ROSS' design characteristics.

First, we can divide the use of ROSS in two steps:

- Building the model;
- Calculating the results.

We can build a model by instantiating elements such as beams (shaft), disks and bearings. These elements are all defined in classes with names such as `ShaftElement`, `BearingElement` and so on.

After instantiating some elements, we can then use these to build a rotor.

This tutorial is about building your **rotor model**. First, you will learn how to create and assign **materials**, how to instantiate the **elements** which composes the rotor and how to convert **units** in ROSS with pint library. This means that every time we call a function, we can use pint.Quantity as an argument for values that have units. If we give a float to the function ROSS will consider SI units as default.

In the following topics, we will discuss the most relevant classes for a quic stable) use ROSS.

```
from pathlib import Path
import ross as rs
import numpy as np

# uncomment the lines below if you are having problems with plots not showing
# import plotly.io as pio
# pio.renderers.default = "notebook"
```

Section 1: Material Class

There is a class called Material to hold material's properties. Materials are applied to shaft and disk elements.

1.1 Creating a material

To instantiate a Material class, you only need to give 2 out of the following parameters: `E` (Young's Moduluds), `G_s` (Shear Modulus) , `Poisson` (Poisson Coefficient), and the material density `rho`.

```
# from E and G_s
steel = rs.Material(name="Steel", rho=7810, E=211e9, G_s=81.2e9)
# from E and Poisson
steel2 = rs.Material(name="Steel", rho=7810, E=211e9, Poisson=0.3)
# from G_s and Poisson
steel3 = rs.Material(name="Steel", rho=7810, G_s=81.2e9, Poisson=0.3)

print(steel)

# returning attributes
print("="*36)
print(f"Young's Modulus: {steel.E}")
print(f"Shear Modulus:    {steel.G_s}")
```

```
Steel
-----------------------------------
Density         (kg/m**3): 7810.0
Young`s modulus (N/m**2):  2.11e+11
Shear modulus   (N/m**2):  8.12e+10
Poisson coefficient     :  0.29926108
====================================
Young's Modulus: 211000000000.0
Shear Modulus:    81200000000.0
```

Note: Adding 3 arguments to the Material class raises an error.

1.2 Saving materials

To save an already instantiated Material object, you need to use the following method.

```
steel.save_material()
```

1.3 Available materials

Saved Materials are stored in a **.toml file**, which can be read as .txt. The file is placed on ROSS root file with name `available_materials.toml`.

It's possible to access the Material data from the file. With the file opened, you can:

- modify the properties directly;
- create new materials;

It's important to **keep the file structure** to ensure the correct functioning of the class.

```
[Materials.Steel]
name = "Steel"
rho = 7810
E = 211000000000.0
Poisson = 0.2992610837438423
G_s = 81200000000.0
color = "#525252"
```

Do not change the dictionary keys and the order they're built.

To check what materials are available, use the command:

```
rs.Material.available_materials()
```

```
['Steel', 'AISI4140', 'A216WCB']
```

1.4 Loading materials

After checking the available materials, you should use the `Material.use_material('name')` method with the **name of the material** as a parameter.

```
steel5 = rs.Material.load_material('Steel')
```

Section 2: ShaftElement Class

`ShaftElement` allows you to create cylindrical and conical shaft elements. It means you can set differents outer and inner diameters for each element node.

There are some ways in which you can choose the parameters to model this element:

- Euler–Bernoulli beam Theory (`rotary_inertia=False, shear_effects=False`)
- Timoshenko beam Theory (`rotary_inertia=True, shear_effects=True` - used as default)

The matrices (mass, stiffness, damping and gyroscopic) will be defined considering the following local coordinate vector:

$[x_0, y_0, \alpha_0, \beta_0, x_1, y_1, \alpha_1, \beta_1]^T$ Where α_0 and α_1 are the bending on the yz plane β_0 and β_1 are the bending on the xz plane.

This element represents the rotor's shaft, all the other elements are correlated with this one.

2.1 Creating shaft elements

The next examples present different ways of how to create a ShaftElement object, from a single element to a list of several shaft elements with different properties.

When creating shaft elements, you don't necessarily need to input a specific node. If `n=None`, the `Rotor` class will assign a value to the element when building a rotor model (*see section 6*).

You can also pass the same `n` value to several shaft elements in the same rotor model.

2.1.1 Cylindrical shaft element

As it's been seen, a shaft element has 4 parameters for diameters. To simplify that, when creating a cylindrical element, you only need to give 2 of them: `idl` and `odl`. So the other 2 (`idr` and `odr`) get the same values.

stable

Note: you can give all the 4 parameters, as long they match each other.

```
# Cylindrical shaft element
L = 0.25
i_d = 0
o_d = 0.05
cy_elem = rs.ShaftElement(
    L=L,
    idl=i_d,
    odl=o_d,
    material=steel,
    shear_effects=True,
    rotary_inertia=True,
    gyroscopic=True,
)

print(cy_elem)
```

```
Element Number:             None
Element Lenght   (m):       0.25
Left Int. Diam.  (m):        0.0
Left Out. Diam.  (m):       0.05
Right Int. Diam. (m):        0.0
Right Out. Diam. (m):       0.05
-----------------------------------
Steel
-----------------------------------
Density         (kg/m**3): 7810.0
Young`s modulus (N/m**2):  2.11e+11
Shear modulus   (N/m**2):  8.12e+10
Poisson coefficient     :  0.29926108
```

2.1.2 Conical shaft element

To create a conical shaft elements, you must give all the 4 diamater parameters, and `idl != idr` and/or `odl != odr`.

```
# Conical shaft element
L = 0.25
idl = 0
idr = 0
odl = 0.05
odr = 0.07
co_elem = rs.ShaftElement(
    L=L,
    idl=idl,
    idr=idr,
    odl=odl,
    odr=odr,
    material=steel,
    shear_effects=True,
    rotary_inertia=True,
    gyroscopic=True,
)
print(co_elem)
```

```
Element Number:             None
Element Lenght   (m):       0.25
Left Int. Diam.  (m):        0.0
Left Out. Diam.  (m):       0.05
Right Int. Diam. (m):        0.0
Right Out. Diam. (m):       0.07
-----------------------------------
Steel
-----------------------------------
Density         (kg/m**3): 7810.0
Young`s modulus (N/m**2):  2.11e+11
Shear modulus   (N/m**2):  8.12e+10
Poisson coefficient     :  0.29926108
```

Returning element matrices

Use one of this methods to return the matrices:

- `.M()`: returns the mass matrix
- `.K(frequency)`: returns the stiffness matrix
- `.C(frequency)`: returns the damping matrix
- `.G()`: returns de gyroscopic matrix

```
# Mass matrix
# cy_elem.M()

# Stiffness matrix
# frequency = 0
# cy_elem.K(frequency)

# Damping matrix
# frequency = 0
# cy_elem.C(frequency)

# Gyroscopic matrix
# cy_elem.G()
```

2.1.3 List of elements - identical properties

Now we leanrt how to create elements, let's automate the process of creating multiple elements with identical properties.

In this example, we want 6 shaft elements with identical properties. This process can be done using a `for` loop or a list comprehension.

```
# Creating a list of shaft elements
L = 0.25
i_d = 0
o_d = 0.05
N = 6
shaft_elements = [
    rs.ShaftElement(
        L=L,
        idl=i_d,
        odl=o_d,
        material=steel,
        shear_effects=True,
        rotary_inertia=True,
        gyroscopic=True,
    )
    for _ in range(N)
]
shaft_elements
```

```
[ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.05,  odr=0.05, material='Steel', n=Nor
 ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.05,  odr=0.05, material='Steel', n=Nor
 ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.05,  odr=0.05, material='Steel', n=Nor
 ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.05,  odr=0.05, material='Steel', n=Nor
 ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.05,  odr=0.05, material='Steel', n=Nor
 ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.05,  odr=0.05, materia              r
```

stable

2.1.4 List of elements - different properties

Now we leanrt how to create elements, let's automate the process of creating multiple elements with identical properties.

In this example, we want 6 shaft elements which properties may not be the same. This process can be done using a `for` loop or a list comprehension, coupled with Python's `zip()` method.

We create lists for each property, where each term refers to a single element:

```
# OPTION No.1:
# Using zip() method
L =   [0.20, 0.20, 0.10, 0.10, 0.20, 0.20]
i_d = [0.01,    0,    0,    0,    0, 0.01]
o_d = [0.05, 0.05, 0.06, 0.06, 0.05, 0.05]
shaft_elements = [
    rs.ShaftElement(
        L=l,
        idl=idl,
        odl=odl,
        material=steel,
        shear_effects=True,
        rotary_inertia=True,
        gyroscopic=True,
    )
    for l, idl, odl in zip(L, i_d, o_d)
]
shaft_elements
```

```
[ShaftElement(L=0.2, idl=0.01, idr=0.01, odl=0.05,  odr=0.05, material='Steel', n=Nc
 ShaftElement(L=0.2, idl=0.0, idr=0.0, odl=0.05,  odr=0.05, material='Steel', n=None
 ShaftElement(L=0.1, idl=0.0, idr=0.0, odl=0.06,  odr=0.06, material='Steel', n=None
 ShaftElement(L=0.1, idl=0.0, idr=0.0, odl=0.06,  odr=0.06, material='Steel', n=None
 ShaftElement(L=0.2, idl=0.0, idr=0.0, odl=0.05,  odr=0.05, material='Steel', n=None
 ShaftElement(L=0.2, idl=0.01, idr=0.01, odl=0.05,  odr=0.05, material='Steel', n=Nc
```

```
# OPTION No.2:
# Using list index
L =   [0.20, 0.20, 0.10, 0.10, 0.20, 0.20]
i_d = [0.01,    0,    0,    0,    0, 0.01]
o_d = [0.05, 0.05, 0.06, 0.06, 0.05, 0.05]
N = len(L)
shaft_elements = [
    rs.ShaftElement(
        L=L[i],
        idl=i_d[i],
        odl=o_d[i],
        material=steel,
        shear_effects=True,
        rotary_inertia=True,
        gyroscopic=True,
    )
    for i in range(N)
]
shaft_elements
```

```
[ShaftElement(L=0.2, idl=0.01, idr=0.01, odl=0.05,  odr=0.05, material='Steel', n=Nc
 ShaftElement(L=0.2, idl=0.0, idr=0.0, odl=0.05,  odr=0.05, material='Steel', n=None
 ShaftElement(L=0.1, idl=0.0, idr=0.0, odl=0.06,  odr=0.06, material='Steel', n=None
 ShaftElement(L=0.1, idl=0.0, idr=0.0, odl=0.06,  odr=0.06, material='Steel', n=None
 ShaftElement(L=0.2, idl=0.0, idr=0.0, odl=0.05,  odr=0.05, material='Steel', n=None
 ShaftElement(L=0.2, idl=0.01, idr=0.01, odl=0.05,  odr=0.05, material='Steel', n=Nc
```

2.2 Creating shaft elements via Excel

There is an option for creating a list of shaft elements via an Excel file. The classmethod `.from_table()` reads an Excel file created and converts it to a list of shaft elements.

A header with the names of the columns is required. These names should match the names expected by the routine (usually the names of the parameters, but also similar ones). The program will read every row bellow the header until they end or it reaches a NaN, which means if the code reaches to an empty line, it stops iterating.

An example of Excel content can be found at ROSS GitHub repository at *ross/tests/data/shaft_si.xls*, spreadsheet "Model".

You can load it using the following code.

```
shaft_file = Path("shaft_si.xls")
shaft = rs.ShaftElement.from_table(
    file=shaft_file, sheet_type="Model", sheet_name="Model"
)
```

Section 3: DiskElement Class

The class `DiskElement` allows you to create disk elements, representing rotor equipments which can be considered only to add mass and inertia to the system, disregarding the stiffness.

ROSS offers 3 (three) ways to create a disk element:

1. Inputing mass and inertia data
2. Inputing geometrical and material data
3. From Excel table

3.1 Creating disk elements from inertia properties

If you have access to the mass and inertia properties of a equipment, you can input the data directly to the element.

Disk elements are useful to represent equipments which mass and inertia are significant, but the stiffness can be neglected.

3.1.1 Creating a single disk element

This example below shows how to instantiate a disk element according to the mass and inertia properties:

```
disk = rs.DiskElement(
    n=0,
    m=32.58,
    Ip=0.178,
    Id=0.329,
    tag="Disk"
)
disk
```

```
DiskElement(Id=0.329, Ip=0.178, m=32.58, color='Firebrick', n=0, scale_factor=1.0, t
```

3.1.2 Creating a list of disk element

stable

This example below shows how to create a list of disk element according to the mass and inertia properties. The logic is the same applied to shaft elements.

```
# OPTION No.1:
# Using zip() method
n_list = [2, 4]
m_list = [32.6, 35.8]
Id_list = [0.17808928, 0.17808928]
Ip_list = [0.32956362, 0.38372842]
disk_elements = [
    rs.DiskElement(
        n=n,
        m=m,
        Id=Id,
        Ip=Ip,
    )
    for n, m, Id, Ip in zip(n_list, m_list, Id_list, Ip_list)
]
disk_elements
```

```
[DiskElement(Id=0.17809, Ip=0.32956, m=32.6, color='Firebrick', n=2, scale_factor=1.
 DiskElement(Id=0.17809, Ip=0.38373, m=35.8, color='Firebrick', n=4, scale_factor=1.
```

```
# OPTION No.2:
# Using list index
n_list = [2, 4]
m_list = [32.6, 35.8]
Id_list = [0.17808928, 0.17808928]
Ip_list = [0.32956362, 0.38372842]
N = len(n_list)
disk_elements = [
    rs.DiskElement(
        n=n_list[i],
        m=m_list[i],
        Id=Id_list[i],
        Ip=Ip_list[i],
    )
    for i in range(N)
]
disk_elements
```

```
[DiskElement(Id=0.17809, Ip=0.32956, m=32.6, color='Firebrick', n=2, scale_factor=1.
 DiskElement(Id=0.17809, Ip=0.38373, m=35.8, color='Firebrick', n=4, scale_factor=1.
```

3.2 Creating disk elements from geometrical properties

stable

Besides the instantiation previously explained, there is a way to instantiate a DiskElement with only geometrical parameters (an approximation for cylindrical disks) and the disk's material,

as we can see in the following code. In this case, there's a class method (`rs.DiskElement.from_geometry()`) which you can use.

ROSS will take geometrical parameters (outer and inner diameters, and width) and convert them into mass and inertia data. Once again, considering the disk as a cylinder.

3.2.1 Creating a single disk element

This example below shows how to instantiate a disk element according to the geometrical and material properties:

```
disk1 = rs.DiskElement.from_geometry(
    n=4,
    material=steel,
    width=0.07,
    i_d=0.05,
    o_d=0.28
)
print(disk1)

print("="*76)
print(f"Disk mass:              {disk1.m}")
print(f"Disk polar inertia:     {disk1.Ip}")
print(f"Disk diametral inertia: {disk1.Id}")
```

```
Tag:                     None
Node:                    4
Mass           (kg):     32.59
Diam. inertia  (kg*m**2): 0.17809
Polar. inertia (kg*m**2): 0.32956
============================================================================
Disk mass:              32.58972765304033
Disk polar inertia:     0.32956362089137037
Disk diametral inertia: 0.17808928257067666
```

3.2.2 Creating a list of disk element

This example below shows how to create a list of disk element according to the geometrical and material properties. The logic is the same applied to shaft elements.

```
# OPTION No.1:
# Using zip() method
n_list = [2, 4]
width_list = [0.7, 0.7]
i_d_list = [0.05, 0.05]
o_d_list = [0.15, 0.18]
disk_elements = [
    rs.DiskElement.from_geometry(
        n=n,
        material=steel,
        width=width,
        i_d=i_d,
        o_d=o_d,
    )
    for n, width, i_d, o_d in zip(n_list, width_list, i_d_list, o_d_list)
]
disk_elements
```

```
[DiskElement(Id=3.6408, Ip=0.26836, m=85.875, color='Firebrick', n=2, scale_factor=1
 DiskElement(Id=5.5224, Ip=0.56007, m=128.38, color='Firebrick', n=4, scale_factor=1
```

```
# OPTION No.2:
# Using list index
n_list = [2, 4]
width_list = [0.7, 0.7]
i_d_list = [0.05, 0.05]
o_d_list = [0.15, 0.18]
N = len(n_list)
disk_elements = [
    rs.DiskElement.from_geometry(
        n=n_list[i],
        material=steel,
        width=width_list[i],
        i_d=i_d_list[i],
        o_d=o_d_list[i],
    )
    for i in range(N)
]
disk_elements
```

```
[DiskElement(Id=3.6408, Ip=0.26836, m=85.875, color='Firebrick', n=2, scale_factor=1
 DiskElement(Id=5.5224, Ip=0.56007, m=128.38, color='Firebrick', n=4, scale_factor=1
```

3.3 Creating disk elements via Excel

stable

The third option for creating disk elements is via an Excel file. The classmethod `.from_table()` reads an Excel file created and converts it to a list of disk elements. This

method accepts **only mass and inertia** inputs.

A header with the names of the columns is required. These names should match the names expected by the routine (usually the names of the parameters, but also similar ones). The program will read every row bellow the header until they end or it reaches a NaN, which means if the code reaches to an empty line, it stops iterating.

You can take advantage of the excel file used to assemble shaft elements, to assemble disk elements, just add a new spreadsheet to your Excel file and specify the correct `sheet_name`.

An example of Excel content can be found at diretory *ross/tests/data/shaft_si.xls*, spreadsheet "More".

```
file_path = Path("shaft_si.xls")
list_of_disks = rs.DiskElement.from_table(file=file_path, sheet_name="More")
list_of_disks
```

```
[DiskElement(Id=0.0, Ip=0.0, m=15.12, color='Firebrick', n=3, scale_factor=1, tag=Nc
 DiskElement(Id=0.025, Ip=0.047, m=6.91, color='Firebrick', n=20, scale_factor=1, ta
 DiskElement(Id=0.025, Ip=0.047, m=6.93, color='Firebrick', n=23, scale_factor=1, ta
 DiskElement(Id=0.025, Ip=0.048, m=6.95, color='Firebrick', n=26, scale_factor=1, ta
 DiskElement(Id=0.025, Ip=0.048, m=6.98, color='Firebrick', n=29, scale_factor=1, ta
 DiskElement(Id=0.025, Ip=0.048, m=6.94, color='Firebrick', n=32, scale_factor=1, ta
 DiskElement(Id=0.025, Ip=0.048, m=6.96, color='Firebrick', n=35, scale_factor=1, ta
```

Section 4: Bearing and Seal Classes

ROSS has a serie of classe to represent element that adds stiffness and / or damping to a rotor system. They're suitable to represent mainly bearings, supports and seals. Each one aims to represent some types of bearing and seal.

All the class will return four stiffness coefficients (k_{xx}, k_{xy}, k_{yx}, k_{yy}) and four damping coefficients (c_{xx}, c_{xy}, c_{yx}, c_{yy}), which will be used to assemble the stiffness and damping matrices.

The main difference between these classes are the arguments the user must input to create the element.

Available bearing classes and class methods:

stable

- 1. `BearingElement`: represents a general (journal) bearing element.

- 2. `SealElement`: represents a general seal element.
- 3. `BallBearingElement`: A bearing element for ball bearings
- 4. `RollerBearingElement`: A bearing element for roller bearings.
- 5. `MagneticBearingElement`: A bearing element for magnetic bearings.
  - 5.1. `param_to_coef`: A bearing element for magnetic bearings from electromagnetic parameters

The classes from item 2 to 5 inherits from `BearingElement` class. It means, you can use the same methods and commands, set up to `BearingElement`, in the other classes.

4.1 BearingElement Class

This class will create a bearing element. Bearings are elements that only add stiffness and damping properties to the rotor system. These parameters are defined by 8 dynamics coefficients (4 stiffness coefficients and 4 damping coefficients).

Parameters can be a constant value or speed dependent. For speed dependent parameters, each argument should be passed as an array and the correspondent speed values should also be passed as an array. Values for each parameter will be interpolated for the speed.

Bearing elements are single node elements and linked to “ground”, but it’s possible to create a new node with `n_link` argument to introduce a link with other elements. Useful to add bearings in series or co-axial rotors.

4.1.1 Bearing with constant coefficients

Bearings can have a constant value for each coefficient. In this case, it’s **not necessary** to give a value to `frequency` argument.

The next example shows how to instantiate a **single bearing with constant coefficients**:

```
stfx = 1e6
stfy = 0.8e6
bearing1 = rs.BearingElement(n=0, kxx=stfx, kyy=stfy, cxx=1e3)
print(bearing1)

print("="*55)
print(f"Kxx coefficient: {bearing1.kxx}")
```

stable

```
BearingElement(n=0, n_link=None,
 kxx=[1000000.0], kxy=[0],
 kyx=[0], kyy=[800000.0],
 cxx=[1000.0], cxy=[0],
 cyx=[0], cyy=[1000.0],
 mxx=[0], mxy=[0],
 myx=[0], myy=[0],
 frequency=None, tag=None)
=======================================================
Kxx coefficient: [1000000.0]
```

4.1.2 Bearing with varying coefficients

The coefficients could be an array with different values for different rotation speeds, in that case you only have to give a parameter 'frequency' which is a array with the same size as the coefficients array.

The next example shows how to instantiate a **single bearing with speed dependent parameters**:

```
bearing2 = rs.BearingElement(
    n=0,
    kxx=np.array([0.5e6, 1.0e6, 2.5e6]),
    kyy=np.array([1.5e6, 2.0e6, 3.5e6]),
    cxx=np.array([0.5e3, 1.0e3, 1.5e3]),
    frequency=np.array([0, 1000, 2000]),
)

print(bearing2)
print("="*79)
print(f"Kxx coefficient: {bearing2.kxx}")
```

```
BearingElement(n=0, n_link=None,
 kxx=[ 500000. 1000000. 2500000.], kxy=[0, 0, 0],
 kyx=[0, 0, 0], kyy=[1500000. 2000000. 3500000.],
 cxx=[ 500. 1000. 1500.], cxy=[0, 0, 0],
 cyx=[0, 0, 0], cyy=[ 500. 1000. 1500.],
 mxx=[0, 0, 0], mxy=[0, 0, 0],
 myx=[0, 0, 0], myy=[0, 0, 0],
 frequency=[   0. 1000. 2000.], tag=None)
===============================================================================
Kxx coefficient: [ 500000. 1000000. 2500000.]
```

If the size of coefficient and frequency arrays do not match, an `ValueError` is raised

The next example shows the instantiate of a **bearing with odd parameters**:

```
bearing_odd = rs.BearingElement( # odd dimensions
    n=0,
    kxx=np.array([0.5e6, 1.0e6, 2.5e6]),
    kyy=np.array([1.5e6, 2.0e6, 3.5e6]),
    cxx=np.array([0.5e3, 1.0e3, 1.5e3]),
    frequency=np.array([0, 1000, 2000, 3000])
)
```

4.1.3 Inserting bearing elements in series

Bearing and seal elements are 1-node element, which means the element attaches to a given node from the rotor shaft and it's connect to the "ground". However, there's an option to couple multiple elements in series, using the `n_link` argument. This is very useful to simulate structures which support the machine, for example.

`n_link` opens a new node to the rotor system, or it can be associated to another rotor node (useful in co-axial rotor models). Then, the new BearingElement node, is set equal to the `n_link` from the previous element.

```
stfx = 1e6
stfy = 0.8e6
bearing3 = rs.BearingElement(n=0, kxx=stfx, kyy=stfy, cxx=1e3, n_link=1, tag="journa
bearing4 = rs.BearingElement(n=1, kxx=1e7, kyy=1e9, cxx=10, tag="support")
print(bearing3)
print(bearing4)
```

```
BearingElement(n=0, n_link=1,
 kxx=[1000000.0], kxy=[0],
 kyx=[0], kyy=[800000.0],
 cxx=[1000.0], cxy=[0],
 cyx=[0], cyy=[1000.0],
 mxx=[0], mxy=[0],
 myx=[0], myy=[0],
 frequency=None, tag='journal_bearing')
BearingElement(n=1, n_link=None,
 kxx=[10000000.0], kxy=[0],
 kyx=[0], kyy=[1000000000.0],
 cxx=[10], cxy=[0],
 cyx=[0], cyy=[10],
 mxx=[0], mxy=[0],
 myx=[0], myy=[0],
 frequency=None, tag='support')
```

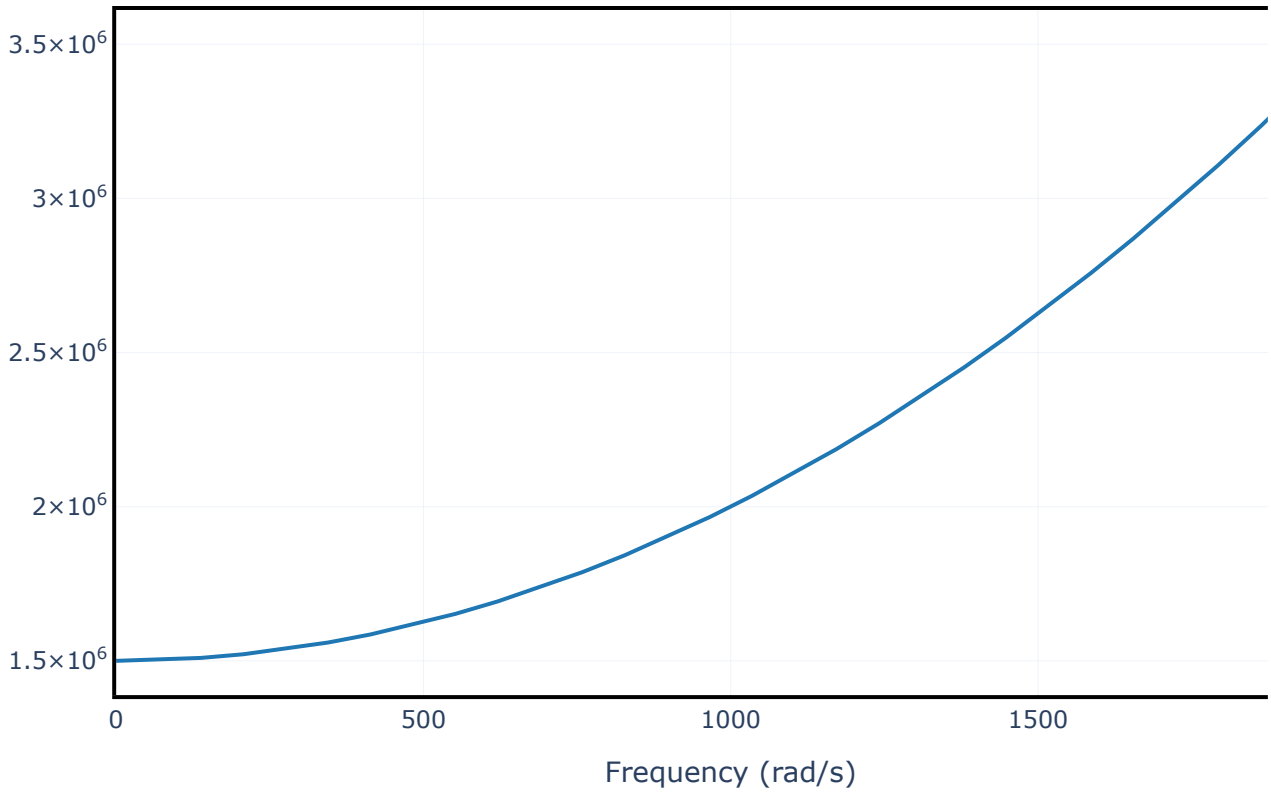
stable

4.1.4 Visualizing coefficients graphically

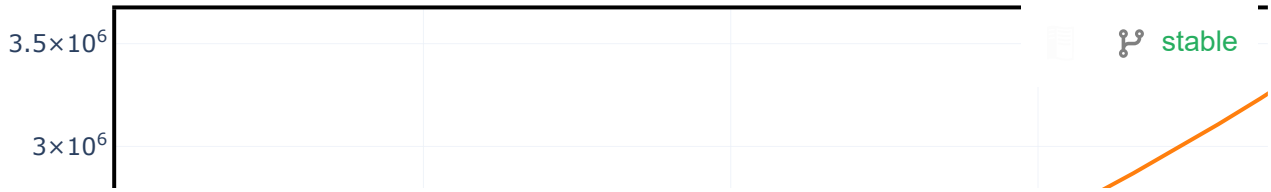
If you want to visualize how the coefficients varies with speed, you can select a specific coefficient and use the `.plot()` method.

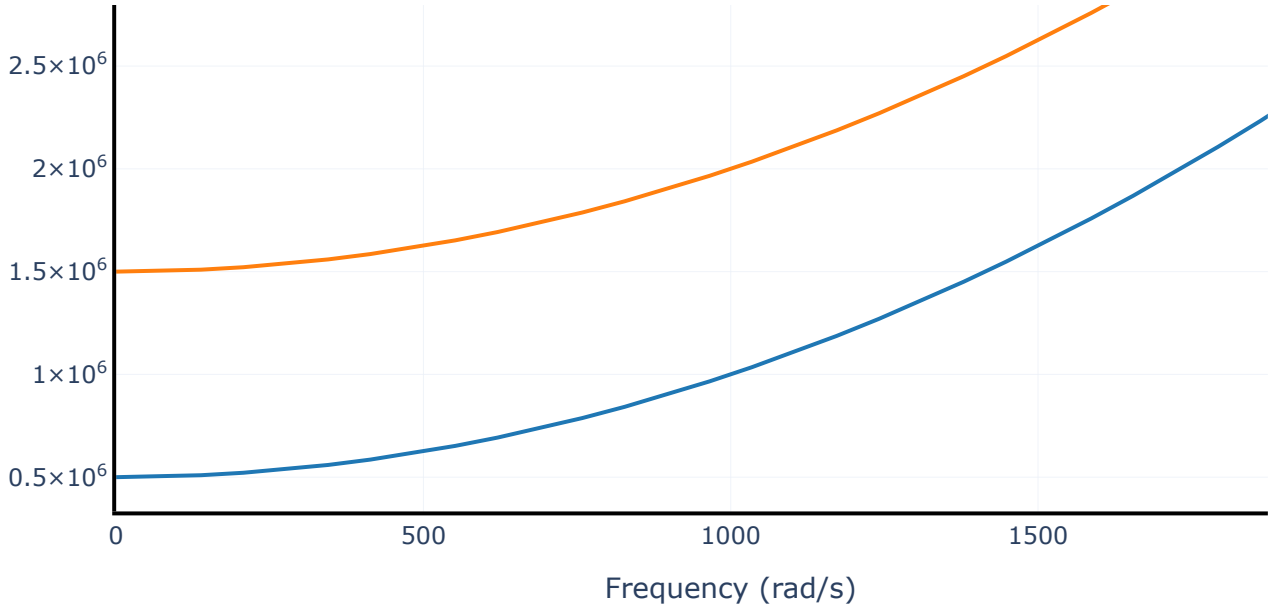
Let's return to the example done in **4.1.2** and check how k_{yy} and c_{yy} varies. You can check for all the 8 dynamic coefficients as you like.

```
bearing2.plot('kyy')
```

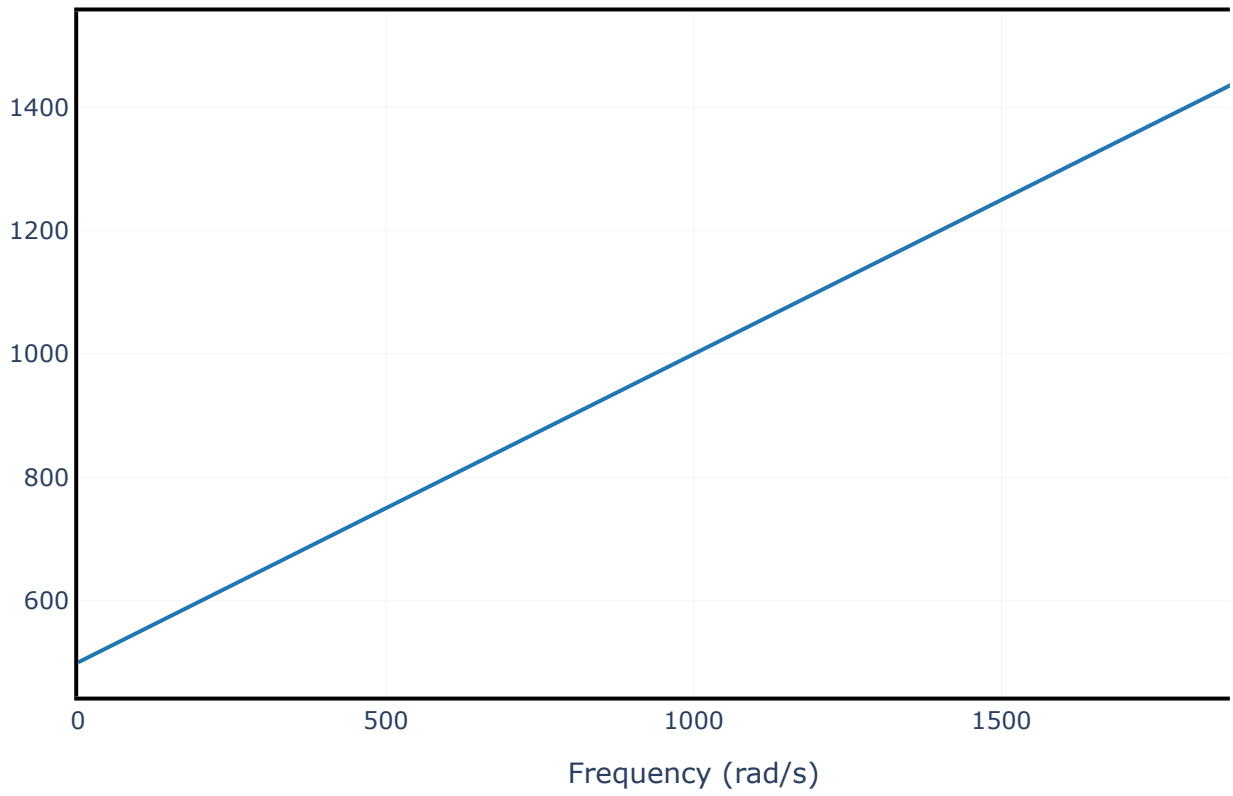



```
bearing2.plot(['kxx', 'kyy'])
```






```
bearing2.plot('cyy')
```



stable

4.2 SealElement Class

`SealElement` class method have the exactly same arguments than `BearingElement`. The differences are found in some considerations when assembbling a full rotor model. For example, a SealElement won't generate reaction forces in a static analysis. So, even they are very similar when built, they have different roles in the model.

Let's see an example:

```
stfx = 1e6
stfy = 0.8e6
seal = rs.SealElement(n=0, kxx=stfx, kyy=stfy, cxx=1e3, cyy=0.8e3)
seal
```

```
SealElement(n=0, n_link=None,
 kxx=[1000000.0], kxy=[0],
 kyx=[0], kyy=[800000.0],
 cxx=[1000.0], cxy=[0],
 cyx=[0], cyy=[800.0],
 mxx=[0], mxy=[0],
 myx=[0], myy=[0],
 frequency=None, tag=None)
```

4.3 BallBearingElement Class

This class will create a bearing element based on some geometric and constructive parameters of ball bearings. The main difference is that cross-coupling stiffness and damping are not modeled in this case.

Let's see an example:

```
n = 0
n_balls= 8
d_balls = 0.03
fs = 500.0
alpha = np.pi / 6
tag = "ballbearing"
ballbearing = rs.BallBearingElement(
    n=n,
    n_balls=n_balls,
    d_balls=d_balls,
    fs=fs,
    alpha=alpha,
    tag=tag,
)
ballbearing
```

```
BallBearingElement(n=0, n_link=None,
 kxx=[46416883.847697675], kxy=[0.0],
 kyx=[0.0], kyy=[100906269.23412538],
 cxx=[580.211048096221], cxy=[0.0],
 cyx=[0.0], cyy=[1261.3283654265672],
 mxx=[0], mxy=[0],
 myx=[0], myy=[0],
 frequency=None, tag='ballbearing')
```

4.4 RollerBearingElement Class

This class will create a bearing element based on some geometric and constructive parameters of roller bearings. The main difference is that cross-coupling stiffness and damping are not modeled in this case.

Let's see an example:

```
n = 0
n_rollers= 8
l_rollers = 0.03
fs = 500.0
alpha = np.pi / 6
tag = "rollerbearing"
rollerbearing = rs.RollerBearingElement(
    n=n,
    n_rollers=n_rollers,
    l_rollers=l_rollers,
    fs=fs,
    alpha=alpha,
    tag=tag
)
rollerbearing
```

```
RollerBearingElement(n=0, n_link=None,
 kxx=[272821927.4006065], kxy=[0.0],
 kyx=[0.0], kyy=[556779443.6747072],
 cxx=[3410.2740925075814], cxy=[0.0],
 cyx=[0.0], cyy=[6959.74304593384],
 mxx=[0], mxy=[0],
 myx=[0], myy=[0],
 frequency=None, tag='rollerbearing')
```

4.5 MagneticBearingElement Class

stable

This class creates a magnetic bearing element. You can input electromagnetic parameters and PID gains. ROSS converts it to stiffness and damping coefficients. To do ir, use the class

`MagneticBearingElement()`

See the following reference for the electromagnetic parameters g0, i0, ag, nw, alpha: Book: Magnetic Bearings. Theory, Design, and Application to Rotating Machinery Authors: Gerhard Schweitzer and Eric H. Maslen Page: 84-95

From: “Magnetic Bearings. Theory, Design, and Application to Rotating Machinery” Authors: Gerhard Schweitzer and Eric H. Maslen Page: 354

Let’s see an example:

```
n = 0
g0 = 1e-3
i0 = 1.0
ag = 1e-4
nw = 200
alpha = 0.392
kp_pid = 1.0
kd_pid = 1.0
k_amp = 1.0
k_sense = 1.0
tag = "magneticbearing"
mbearing = rs.MagneticBearingElement(
    n=n,g0=g0,i0=i0,ag=ag,nw=nw,alpha=alpha, kp_pid=kp_pid,kd_pid=kd_pid, k_amp=k_am
)
mbearing
```

```
MagneticBearingElement(n=0, n_link=None,
 kxx=[-4640.623377181318], kxy=[0.0],
 kyx=[0.0], kyy=[-4640.623377181318],
 cxx=[4.645268645827145], cxy=[0.0],
 cyx=[0.0], cyy=[4.645268645827145],
 mxx=[0], mxy=[0],
 myx=[0], myy=[0],
 frequency=None, tag='magneticbearing')
```

4.6 Creating bearing elements via Excel

There’s an option for creating bearing elements via an Excel file. The classmethod `.from_table()` reads an Excel file created and converts it to a `BearingElement` instance. Differently from creating shaft or disk elements, this method creates only a single bearing element. To create a list of bearing elements, the user should open several spreadsheets in the Excel file and run a list comprehension loop appending each elemnet to

A header with the names of the columns is required. These names should match the names expected by the routine (usually the names of the parameters, but also similar ones). The program will read every row bellow the header until they end or it reaches a NaN, which means if the code reaches to an empty line, it stops iterating.

```
n : int
    The node in which the bearing will be located in the rotor.
file: str
    Path to the file containing the bearing parameters.
sheet_name: int or str, optional
    Position of the sheet in the file (starting from 0) or its name. If none is pass
    assumed to be the first sheet in the file.
```

An example of Excel content can be found at diretory *ross/tests/data/bearing_seal_si.xls*, spreadsheet "XLUserKCM".

```
# single bearing element
file_path = Path("bearing_seal_si.xls")
bearing = rs.BearingElement.from_table(n=0, file=file_path)
bearing
```

```
BearingElement(n=0, n_link=None,
 kxx=[1.37981e+07 2.99519e+07 5.35657e+07 8.51442e+07 1.20733e+08 1.59519e+08
 1.97885e+08 2.35240e+08 2.71250e+08], kxy=[0 0 0 0 0 0 0 0 0],
 kyx=[0 0 0 0 0 0 0 0 0], kyy=[1.37981e+07 2.99519e+07 5.35657e+07 8.51442e+07 1.207
 1.97885e+08 2.35240e+08 2.71250e+08],
 cxx=[102506 127450 144989 153563 155122 150835 145086 141871 140702], cxy=[0 0 0 0
 cyx=[0 0 0 0 0 0 0 0 0], cyy=[102506 127450 144989 153563 155122 150835 145086 1418
 mxx=[0, 0, 0, 0, 0, 0, 0, 0, 0], mxy=[0, 0, 0, 0, 0, 0, 0, 0, 0],
 myx=[0, 0, 0, 0, 0, 0, 0, 0, 0], myy=[0, 0, 0, 0, 0, 0, 0, 0, 0],
 frequency=[ 314.15926536  418.87902048  523.5987756   628.31853072  733.03828584
  837.75804096  942.47779608 1047.1975512  1151.91730632], tag=None)
```

As `.from_table()` creates only a single bearing, let's see an example how to create multiple elements without typing the same command line multiple times.

- First, in the EXCEL file, create multiple spreadsheets. Each one must hold the bearing coefficients and frequency data.
- Then, create a list holding the node numbers for each bearing (respecting the order of the spreadsheets from the EXCEL file).
- Finally, create a loop which iterates over the the nodes list and the sprea

```
# list of bearing elements

# nodes = list with the bearing elements nodes number
# file_path = Path("bearing_seal_si.xls")
# bearings = [rs.BearingElement.from_table(n, file_path, sheet_name=i) for i, n in e
```

Section 5: PointMass Class

The `PointMass` class creates a point mass element. This element can be used to link other elements in the analysis. The mass provided can be different on the x and y direction (e.g. different support inertia for x and y directions).

`PointMass` also keeps the mass, stiffness, damping and gyroscopic matrices sizes consistence. When adding 2 bearing elements in series, it opens a new node with new degrees of freedom (DoF) (*see section 4.1.3*) and expands the stiffness and damping matrices. For this reason, it's necessary to add mass values to those DoF to match the matrices sizes.

If you input the argument `m`, the code automatically replicate the mass value for both directions "x" and "y".

Let's see an example of creating point masses:

```
# inputting m
p0 = rs.PointMass(n=0, m=2)
p0.M() # returns de mass matrices for the element
```

```
array([[2., 0.],
       [0., 2.]])
```

```
# inputting mx and my
p1 = rs.PointMass(n=0, mx=2, my=3)
p1.M()
```

```
array([[2., 0.],
       [0., 3.]])
```

Section 6: Rotor Class

`Rotor` is the main class from ROSS. It takes as argument lists with all elements and assembles the mass, gyroscopic, damping and stiffness global matrices for the system. The object created has several methods that can be used to evaluate the dynamics of the model (they all start with the prefix `.run_`).

To use this class, you must input all the already instantiated elements in a list format.

If the shaft elements are not numbered, the class set a number for each one, according to the element's position in the list supplied to the rotor constructor.

To assemble the matrices, the `Rotor` class takes the local DoF's index from each element (element method `.dof_mapping()`) and calculate the global index

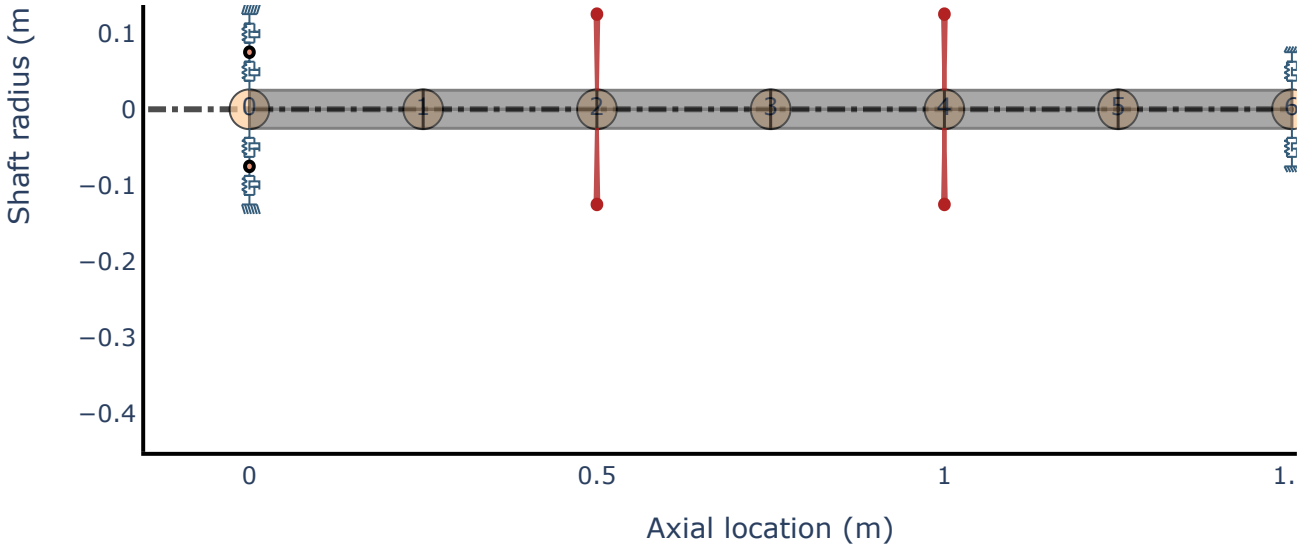
6.1 Creating a rotor model

Let's create a simple rotor model with $1.5m$ length with 6 identical shaft elements, 2 disks, 2 bearings in the shaft ends and a support linked to the first bearing. First, we create the elements, then we input them to the `Rotor` class.

```
n = 6

shaft_elem = [
    rs.ShaftElement(
        L=0.25,
        idl=0.0,
        odl=0.05,
        material=steel,
        shear_effects=True,
        rotary_inertia=True,
        gyroscopic=True,
    )
    for _ in range(n)
]

disk0 = rs.DiskElement.from_geometry(
    n=2, material=steel, width=0.07, i_d=0.05, o_d=0.28
)
disk1 = rs.DiskElement.from_geometry(
    n=4, material=steel, width=0.07, i_d=0.05, o_d=0.28
)
disks = [disk0, disk1]

stfx = 1e6
stfy = 0.8e6
bearing0 = rs.BearingElement(0, kxx=stfx, kyy=stfy, cxx=0, n_link=7)
bearing1 = rs.BearingElement(6, kxx=stfx, kyy=stfy, cxx=0)
bearing2 = rs.BearingElement(7, kxx=stfx, kyy=stfy, cxx=0)

bearings = [bearing0, bearing1, bearing2]

pm0 = rs.PointMass(n=7, m=30)
pointmass = [pm0]

rotor1 = rs.Rotor(shaft_elem, disks, bearings, pointmass)

print("Rotor total mass = ", np.round(rotor1.m, 2))
print("Rotor center of gravity =", np.round(rotor1.CG, 2))

# plotting the rotor model
rotor1.plot_rotor()
```

```
Rotor total mass =  88.18
Rotor center of gravity = 0.75
```

Rotor Model



6.2 Creating a rotor from sections

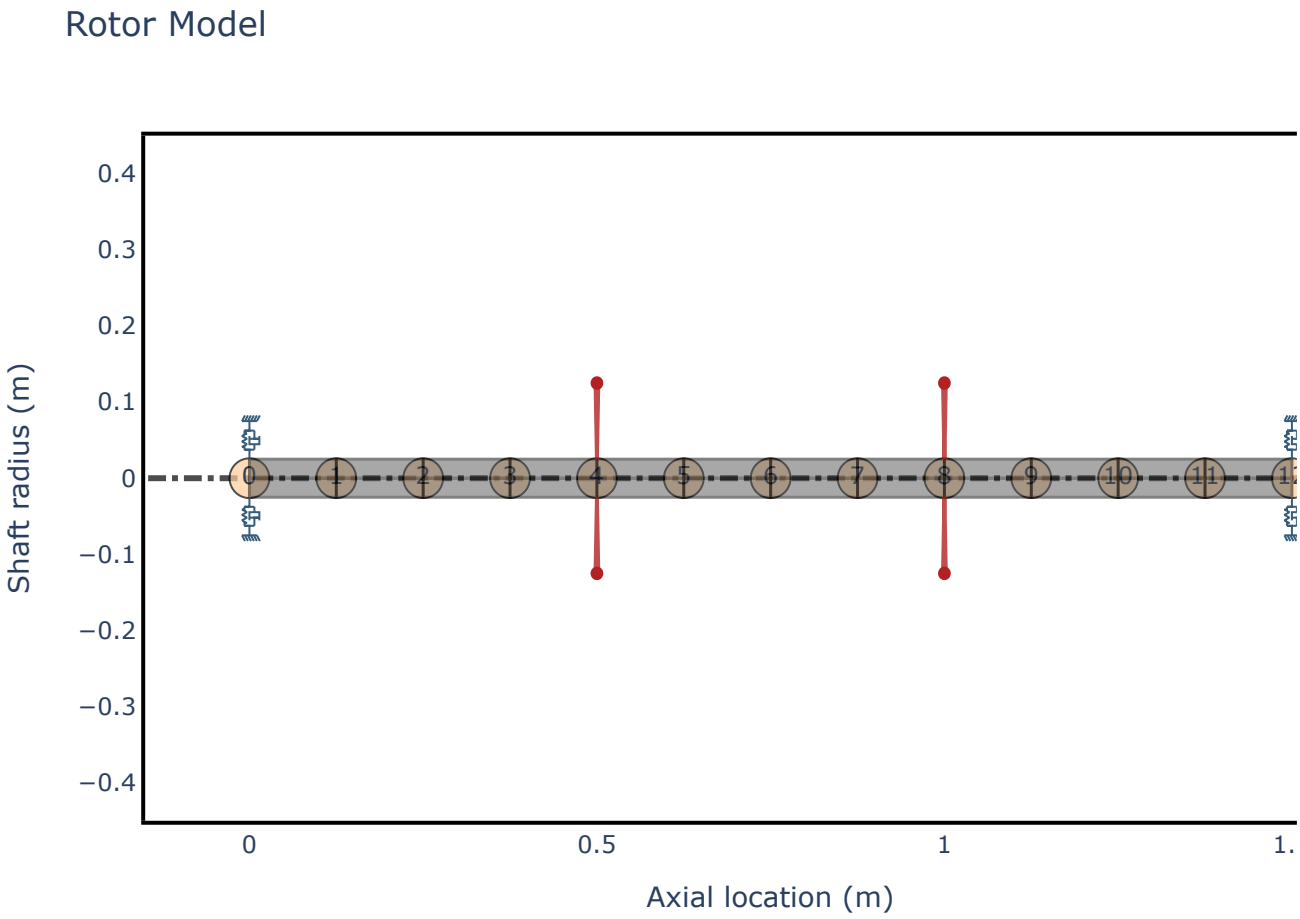
An alternative to build rotor models is dividing the rotor in sections. Each section gets the same number of shaft elements.

There's an important difference in this class method when placing disks and bearings. The argument `n` will refer, not to the element node, but to the section node. So if your model has 3 sections with 4 elements each, there're 4 section nodes and 13 element nodes.

Let's repeat the rotor model from the last example, but using `.from_section()` class method, without the support.

```
i_d = 0
o_d = 0.05

# inner diameter of each section
i_ds_data = [0, 0, 0]
# outer diameter of each section
o_ds_data = [0.05, 0.05, 0.05]
# length of each section
leng_data = [0.5, 0.5, 0.5]

material_data = [steel, steel, steel]
# material_data = steel
stfx = 1e6
stfy = 0.8e6

# n = 0 refers to the section 0, first node
bearing0 = rs.BearingElement(n=0, kxx=stfx, kyy=stfy, cxx=1e3)

# n = 3 refers to the section 2, last node
bearing1 = rs.BearingElement(n=3, kxx=stfx, kyy=stfy, cxx=1e3)
bearings = [bearing0, bearing1]

# n = 1 refers to the section 1, first node
disk0 = rs.DiskElement.from_geometry(
    n=1,
    material=steel,
    width=0.07,
    i_d=0.05,
    o_d=0.28
)

# n = 2 refers to the section 2, first node
disk1 = rs.DiskElement.from_geometry(
    n=2,
    material=steel,
    width=0.07,
    i_d=0.05,
    o_d=0.28
)
disks = [disk0,disk1]

rotor2 = rs.Rotor.from_section(
    brg_seal_data=bearings,
    disk_data=disks,
    idl_data=i_ds_data,
    leng_data=leng_data,
    odl_data=o_ds_data,
    nel_r=4,
    material_data=steel,
)

print("Rotor total mass = ", np.round(rotor2.m, 2))
print("Rotor center of gravity =", np.round(rotor2.CG, 2))
rotor2.plot_rotor()
```

```
Rotor total mass =  88.18
Rotor center of gravity = 0.75
```



6.3 Visualizing the rotor model

It is interesting to plot the rotor to check if the geometry checks with what you wanted to the model. Use the `.plot_rotor()` method to create a plot.

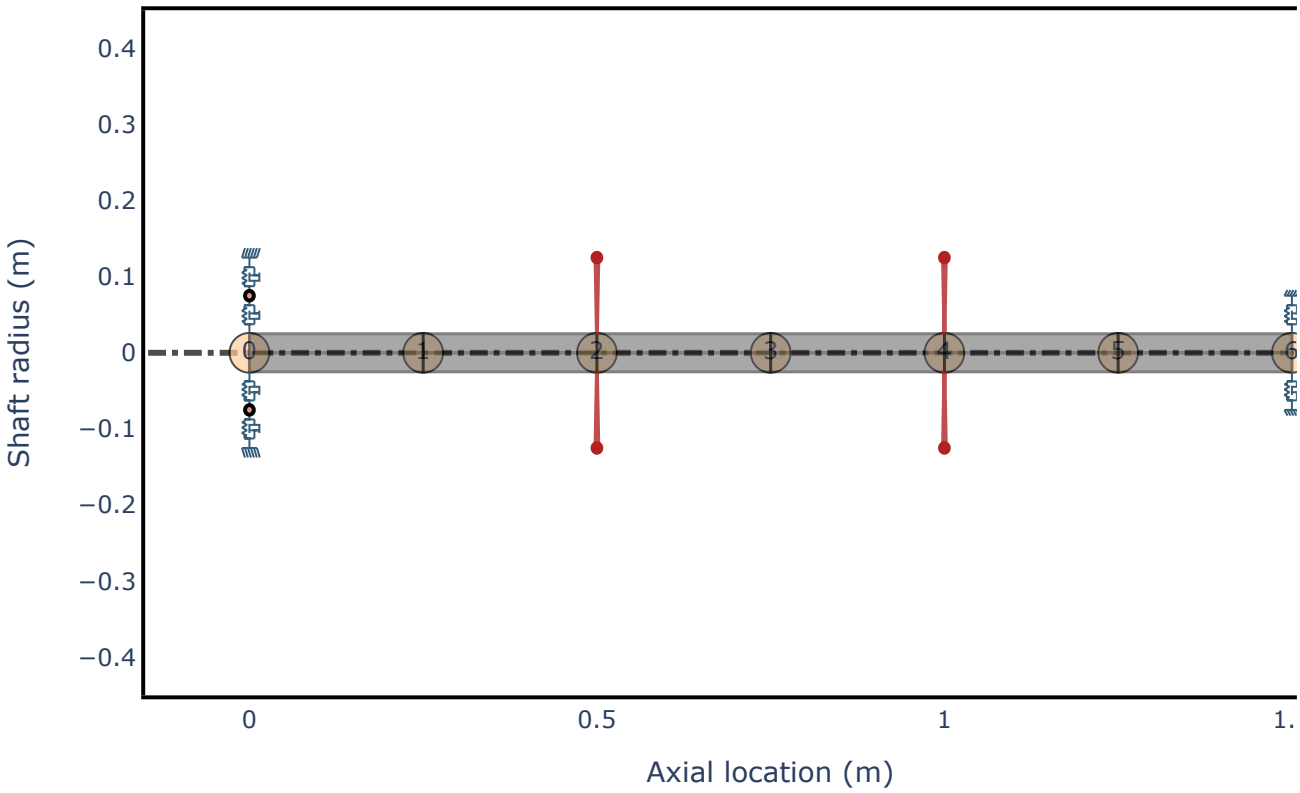
`nodes` argument is useful when your model has lots of nodes and the visualization of nodes label may be confusing. Set an increment to the plot nodes label

ROSS uses **PLOTLY** as main plotting library:

With the Plotly, you can hover the mouse icon over the shaft, disk and point mass elements to check some of their parameters.

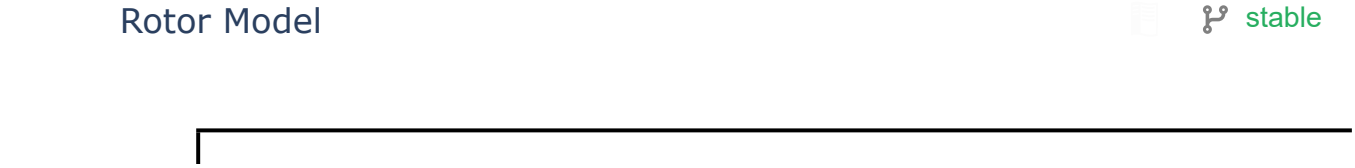
```
rotor1.plot_rotor()
```

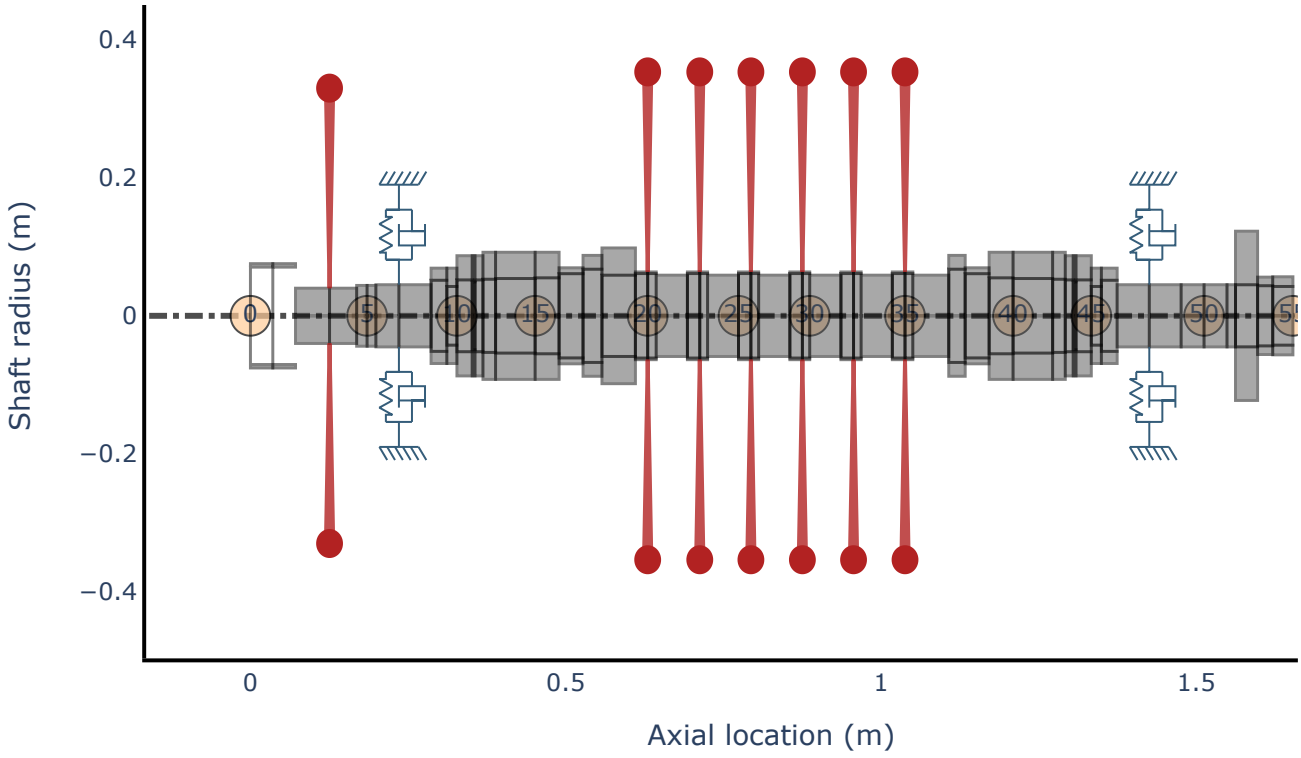
stable



Let's visualize another rotor example with **overlapping shaft elements**:

```
shaft_file = Path("shaft_si.xls")
shaft = rs.ShaftElement.from_table(
    file=shaft_file, sheet_type="Model", sheet_name="Model"
)

file_path = Path("shaft_si.xls")
list_of_disks = rs.DiskElement.from_table(file=file_path, sheet_name="More")

bearing1 = rs.BearingElement.from_table(n=7, file="bearing_seal_si.xls")
bearing2 = rs.BearingElement.from_table(n=48, file="bearing_seal_si.xls")

bearings = [bearing1, bearing2]

rotor3 = rs.Rotor(shaft, list_of_disks, bearings)


node_increment = 5
rotor3.plot_rotor(nodes=node_increment)
```





6.4 Saving a rotor model

You can save a rotor model using the method `.save()`. This method saves the each element type and the rotor object in different *.toml* files.

You just need to input a name and the diretory, where it will be saved. If you don't input a file_path, the rotor model is saved inside the "ross" folder.

To save the `rotor2` we can use:

```
rotor2.save('rotor2.toml')
```

6.5 Loading a rotor model

You can load a rotor model using the method `.load()`. This method loads a previously saved rotor model.

You just need to input the file path to the method.

Now, let's load the `rotor2` we saved before:

```
rotor2_1 = rs.Rotor.load('rotor2.toml')
rotor2_1 == rotor2
```

```
True
```

Section 7: ROSS Units System

ROSS uses an units system package called Pint.

`Pint` defines, operates and manipulates **physical quantities**: the product of a numerical value and a unit of measurement. It allows arithmetic operations between them and conversions from and to different units.

With `Pint`, it's possible to define units to every element type available in ROSS and manipulate the units when plotting graphs. ROSS takes the user-defined units and internally converts them to the International System (SI).

Important: It's not possible to manipulate units for attributes from any class. Attributes' values are always returned converted to SI. **Only plot methods** are able to manipulate the output unit.

7.1 Inserting units

Working with `Pint` requires a specific syntax to assign an unit to an argument.

First of all, it's necessary to import a function called `Q_` from `ross.units`. This function must be assigned to every variable that are desired to have units, followed by a *tuple* containing the magnitude and the unit (in string format).

The example below shows how to create a material using `Pint`, and how it is returned to the user.

```
from ross.units import Q_

rho = Q_(487.56237, "lb/foot**3")  # Imperial System
E = Q_(211.e9, "N/m**2")           # International System
G_s=Q_(81.2e9, "N/m**2")           # International System

steel4 = rs.Material(name="steel", rho=rho, E=E, G_s=G_s)
```

Note: Taking a closer look to the output values, the material density is converted to the SI and it's returned this way to the user.

```
print(steel4)
```

```
steel
---------------------------------
Density         (kg/m**3): 7810.0
Young`s modulus (N/m**2):  2.11e+11
Shear modulus   (N/m**2):  8.12e+10
Poisson coefficient     :  0.29926108
```

The same syntax applies to elements instantiation, if units are desired. Besides, notice the output is displayed in SI units.

Shaft Element using Pint

```
L = Q_(10, "in")
i_d = Q_(0., "meter")
o_d = Q_(0.05, "meter")

elem_pint = rs.ShaftElement(L=L, idl=i_d, odl=o_d, material=steel)
print(elem_pint)
```

```
Element Number:             None
Element Lenght   (m):      0.254
Left Int. Diam.  (m):        0.0
Left Out. Diam.  (m):       0.05
Right Int. Diam. (m):        0.0
Right Out. Diam. (m):       0.05
---------------------------------
Steel
---------------------------------
Density         (kg/m**3): 7810.0
Young`s modulus (N/m**2):  2.11e+11
Shear modulus   (N/m**2):  8.12e+10
Poisson coefficient     :  0.29926108
```

Bearing Element using Pint

```
kxx = Q_(2.54e4, "N/in")
cxx = Q_(1e2, "N*s/m")

brg_pint = rs.BearingElement(n=0, kxx=kxx, cxx=cxx)
print(brg_pint)
```

```
BearingElement(n=0, n_link=None,
 kxx=[1000000.0], kxy=[0],
 kyx=[0], kyy=[1000000.0],
 cxx=[100.0], cxy=[0],
 cyx=[0], cyy=[100.0],
 mxx=[0], mxy=[0],
 myx=[0], myy=[0],
 frequency=None, tag=None)
```

7.2 Manipulating units for plotting

The plot methods presents arguments to change the units for each axis. This kind of manipulation does not affect the resulting data stored. It only converts the data on the graphs.

The arguments names follow a simple logic. It is the “axis name” underscore “units” (axisname_units). It should help the user to identify which axis to modify. For example:

- frequency_units:
  - “rad/s”, “RPM”, “Hz”...
- amplitude_units:
  - “m”, “mm”, “in”, “foot”...
- displacement_units:
  - “m”, “mm”, “in”, “foot”...
- rotor_length_units:
  - “m”, “mm”, “in”, “foot”...
- moment_units:
  - “N/m”, “lbf/foot”...

It’s not necessary to add units previously to each element or material to use `Pint` with plots. But keep in mind ROSS will considers results values in the SI units.

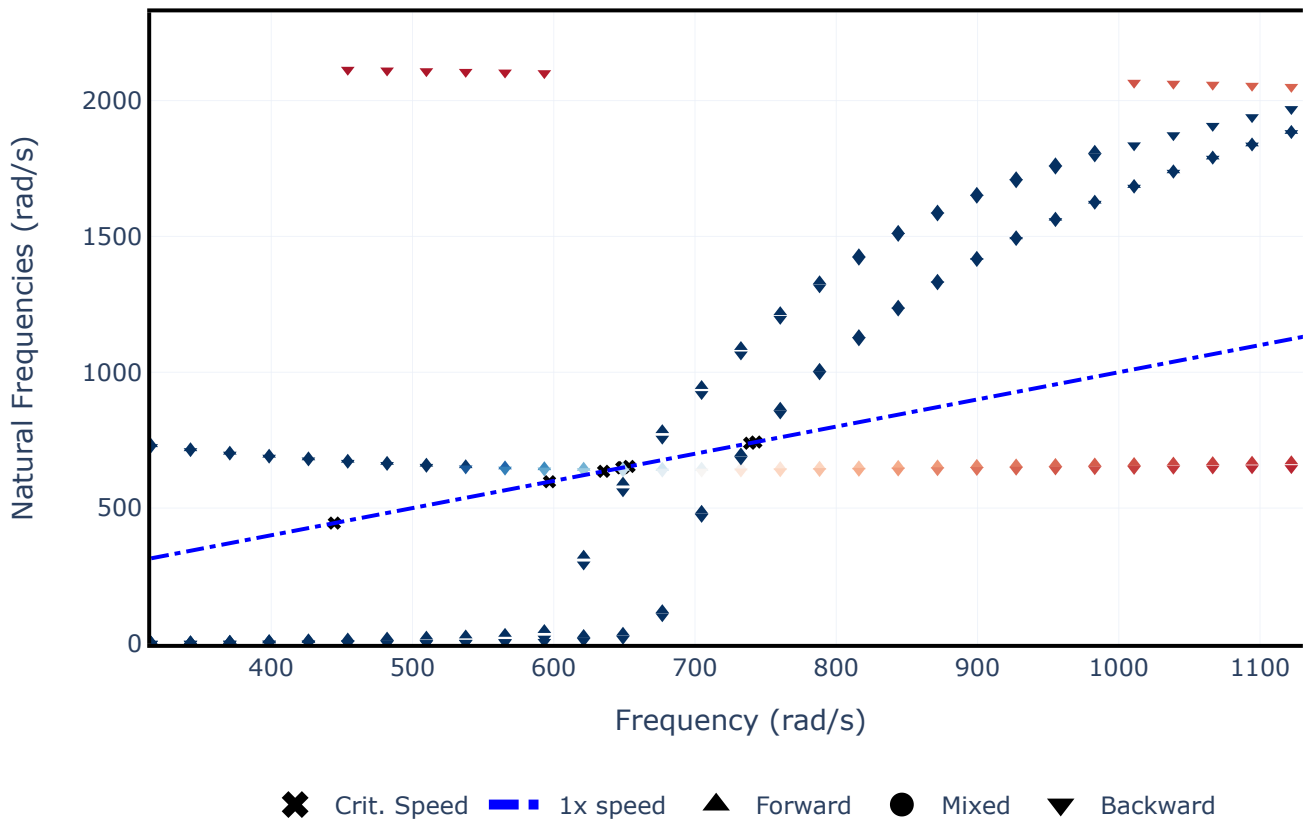
Note: If you input data using the Imperial System, for example, without using Pint, ROSS will consider it's in SI if you try to manipulate the units when plotting.

Let's run a simple example of manipulating units for plotting.

```
samples = 31
speed_range = np.linspace(315, 1150, samples)

campbell = rotor3.run_campbell(speed_range)
```

Plotting with default options will bring graphs with SI units. X and Y axes representing the frequencies are set to `rad/s`

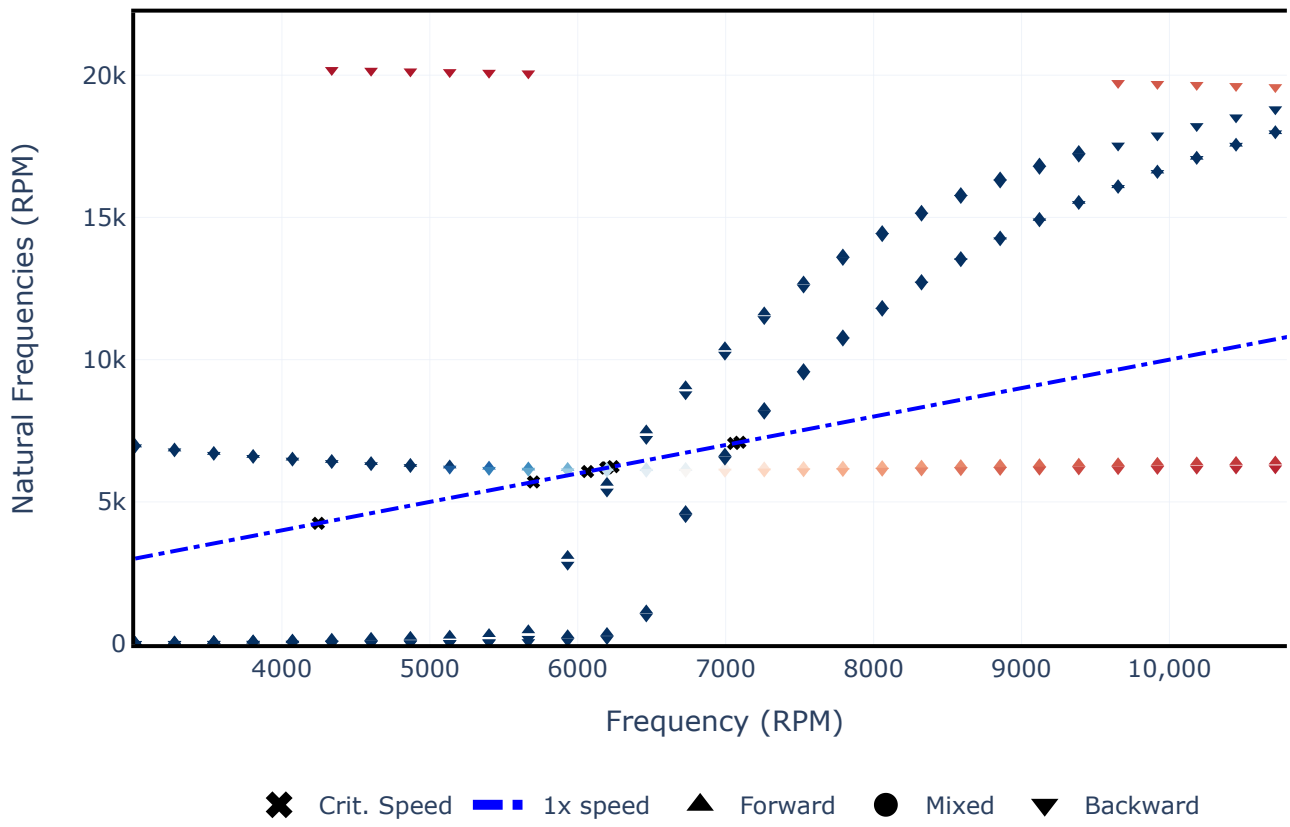
```
campbell.plot()
```



Now, let's change the units to `RPM`.

Just by adding `frequency_units="rpm"` to plot method, you'll change the pl


```
campbell.plot(frequency_units="RPM")
```


Natural Frequencies (RPM)
20k
15k
10k
5k
0
4000
5000
6000
7000
8000
9000
10,000
Frequency (RPM)
Crit. Speed
1x speed
Forward
Mixed
Backward