Print to PDF

Tutorial - Static and Modal Analyzes

Contents

This is the second part of a basic tutorial on how to use ROSS (rotordynamics open-source software), a Python library for rotordynamic analysis. In this tutorial, you will learn how to run several rotordynamic analyzes with your **rotor model**.

To get results, we always have to use one of the `.run_` methods available for a rotor object. These methods will return objects that store the analysis results and that also have plot methods available. These methods will use the plotly library to make graphs common to a rotordynamic analysis.

Rotor model

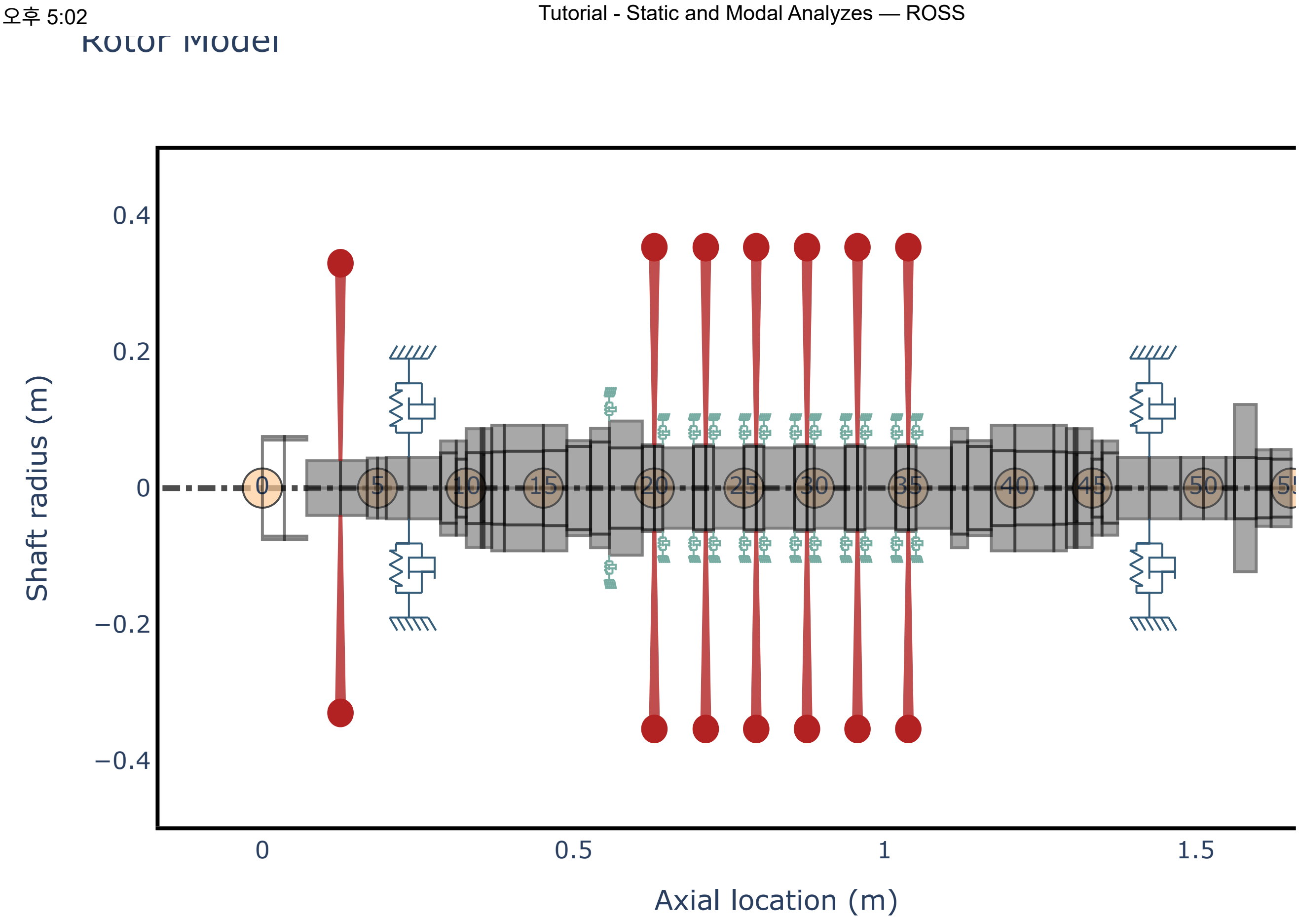
First, let's recover the rotor model built in the previous tutorial.

```
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"
```

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

stable



Rotor Analyses

In the last tutorial we have learnt how to create a rotor model with `Rotor` class. Now, we'll use the same class to run the simulation. There're some methods, most of them with the prefix `run_` you can use to run the rotordynamics analyses.

For Most of the methods, you can use the command `.plot()` to display a graphical visualization of the results (e.g `run_campbel().plot()`, `run_modal().plot_mode_3d(mode)`).

ROSS offers the following analyses:

- Static analysis
- Modal analysis
- Campbell Diagram

Plotly library

ROSS uses **Plotly** for plotting results. All the figures can be stored and manipulated following Plotly API.

The following sections presents the results and how to return the Plotly Figures.

1.1 Static Analysis

This method runs the static analysis for the rotor. It calculate the static deformation due the gravity effects (shaft and disks weight). It also returns the bending moment and shearing force on each node, and you can return a free-body-diagram representation for the rotor, with the self weight, disks weight and reaction forces on bearings displayed.

1.1.1 Running static analysis

To run the simulation, use the `.run_static()` method. You can define a variable to store the results.

Storing the results, it's possible to return the following arrays:

- `disk_forces_nodal`
- `disk_forces_tag`
- `bearing_forces_nodal`
- `bearing_forces_tag`
- `disp_y`
- `Vx`
- `Bm`

```
static = rotor3.run_static()
```

Returning forces

Disk forces

- `.disk_forces_nodal`: Returns a dictionaty expliciting the node where the disk is located and the force value.
- `.disk_forces_tag`: Returns a dictionaty expliciting the the disk tag is located and the force value.

stable

Bearing forces

- `.bearing_forces_nodal`: Returns a dictionaty expliciting the node where the bearing is located and the force value.
- `.bearing_forces_tag`: Returns a dictionaty expliciting the the bearing tag is located and the force value.

```
print("Disk forces - nodes")
print(rotor3.disk_forces_nodal)
print("")
print("Disk forces - tags")
print(rotor3.disk_forces_tag)

print("")
print("Bearing forces - nodes")
print(rotor3.bearing_forces_nodal)
print("")
print("Bearing forces - tags")
print(rotor3.bearing_forces_tag)
```

```
Disk forces - nodes
{'node_3': 148.2741036647235, 'node_20': 67.76283441291268, 'node_23': 67.9589641796

Disk forces - tags
{'Disk 0': 148.2741036647235, 'Disk 1': 67.76283441291268, 'Disk 2': 67.95896417966

Bearing forces - nodes
{'node_7': 1216.282756777383, 'node_48': 1204.6514712821229}

Bearing forces - tags
{'Bearing 0': 1216.282756777383, 'Bearing 1': 1204.6514712821229}
```

Other attributes from static analysis

- `.Vx`: Shearing force array
- `.Bm`: Bending moment array
- `.deformation` Displacement in Y direction

1.1.2 Plotting results

With results stored, you can use some methods to plot the results. Currently there're four plots you can retrieve from static analysis:

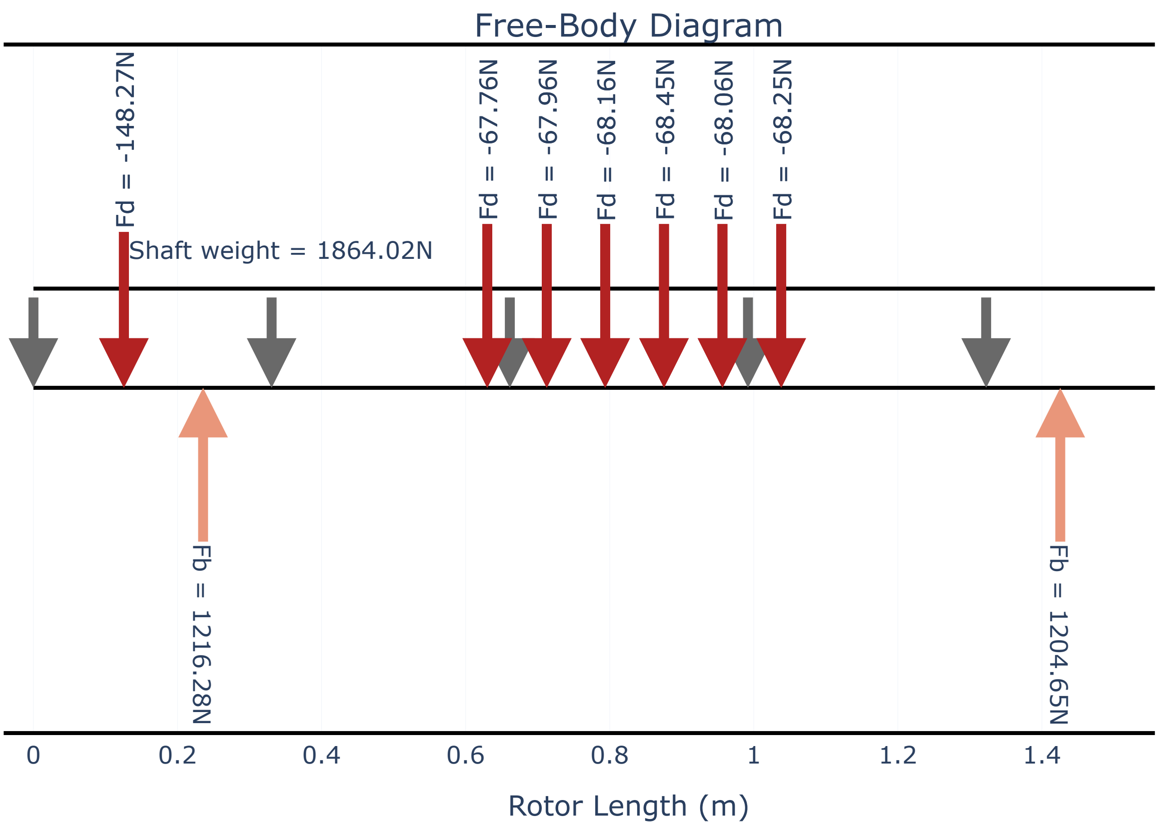
stable

- `.plot_free_body_diagram()`

- `.plot_deformation()`
- `.plot_shearing_force()`
- `.plot_bending_moment()`

Plotting free-body-diagram

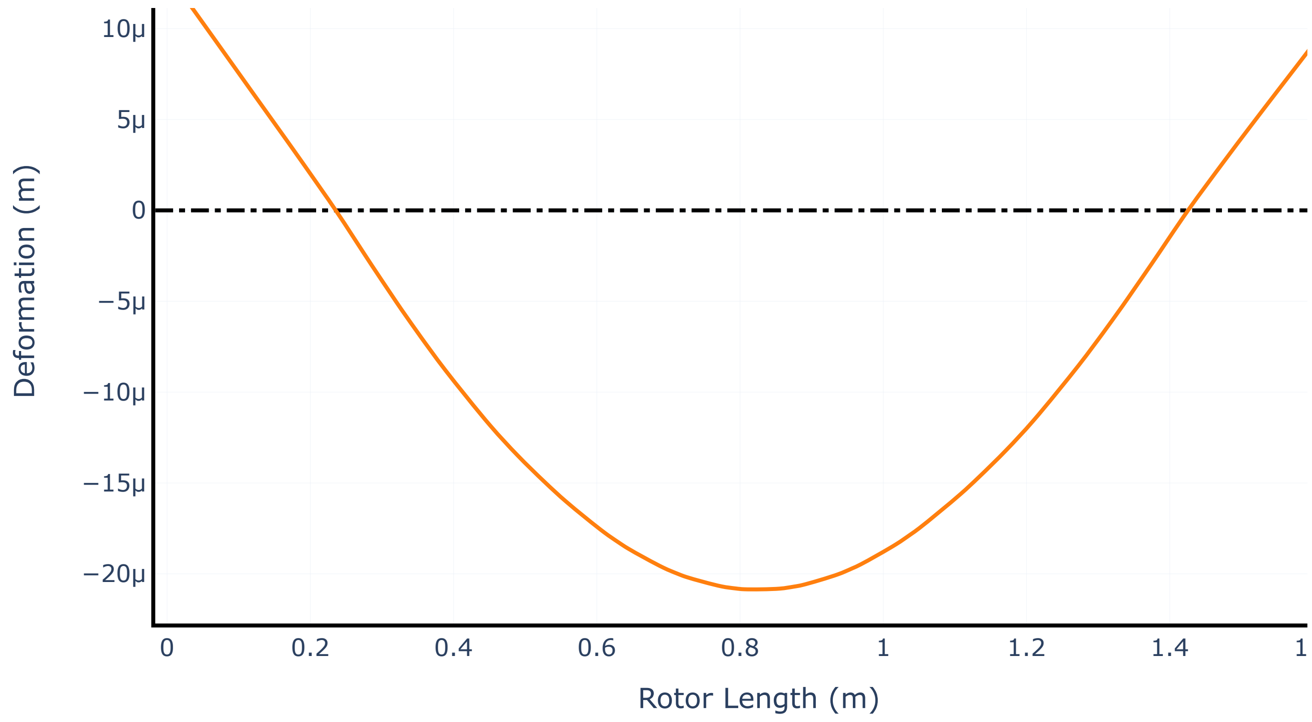
```
static.plot_free_body_diagram()
```



Plotting deformation

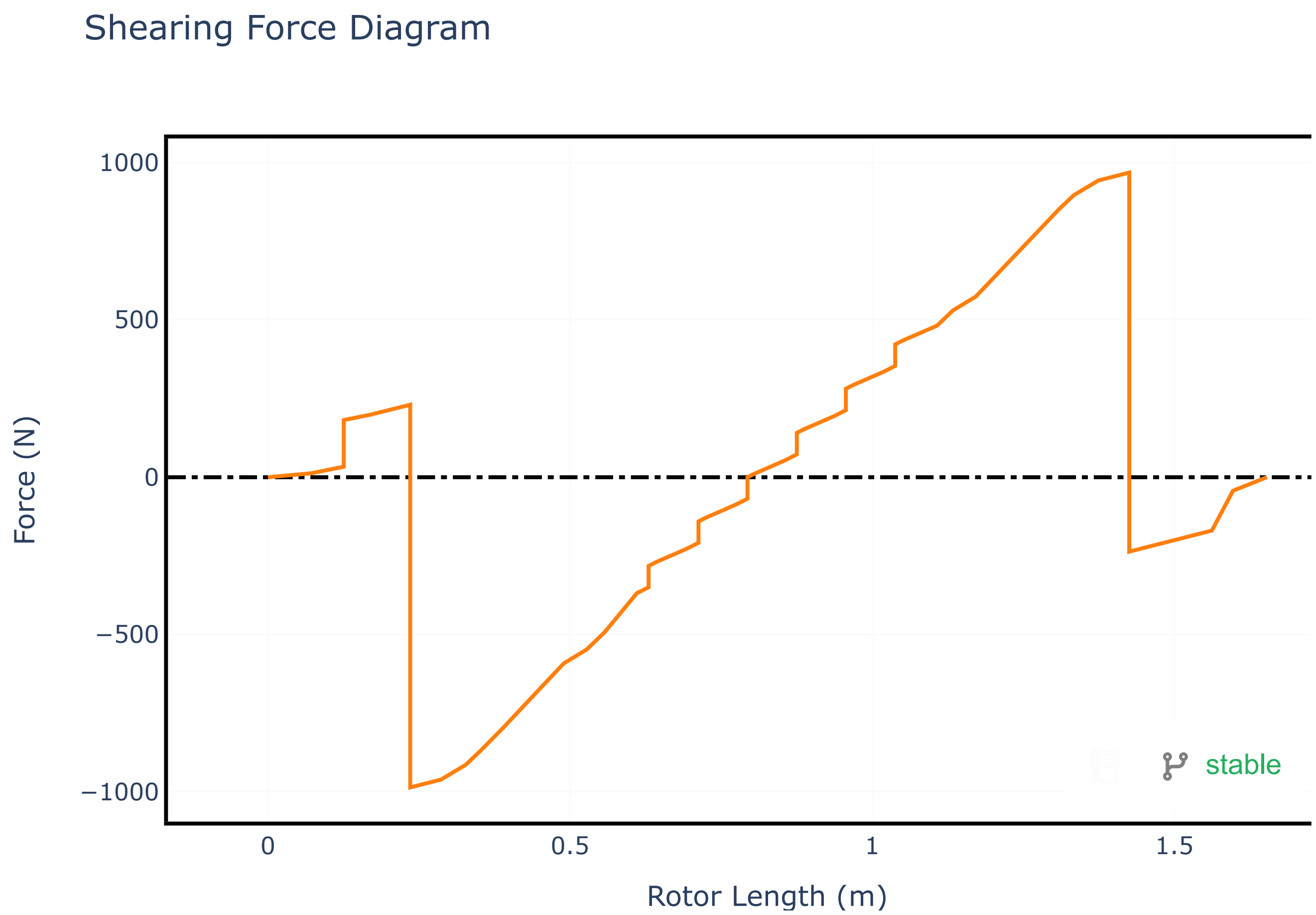
```
static.plot_deformation()
```

stable



Plotting shearing force diagram

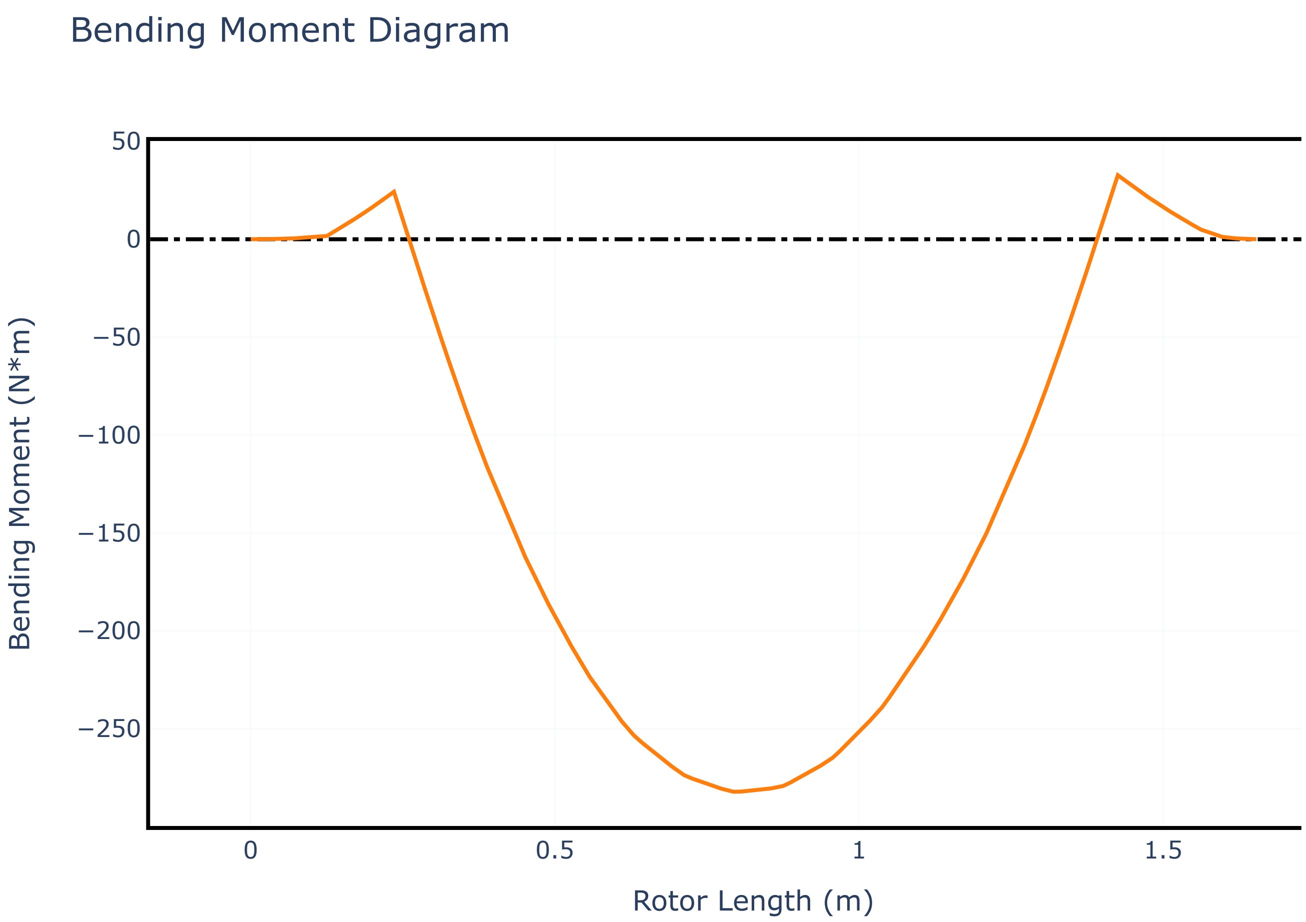
```
static.plot_shearing_force()
```



stable

Plotting bending moment diagram

```
static.plot_bending_moment()
```



1.2 Modal Analysis

ROSS performs the modal analysis through method `run_modal()`. This method calculates natural frequencies, damping ratios and mode shapes.

You must select a speed, which will be used as excitation frequency to calculate the system's eigenvalues and eigenvectors, and the number of eigenvalues and eigenvectors to be calculated is an optional argument (`num_modes`).

stable

After running the modal analysis, it's possible to return the following attributes:

- eigenvalues (evalues);
- eigenvectors (evectors);
- damped natural frequencies (wd);
- undamped natural frequencies (wn);
- damping ratio (damping_ratio);
- logarithmic decrement (log_dec).

1.2.1 Running modal analysis

To run the modal analysis, choose a speed to instantiate the method. For different speeds, change the the argument and run `run_modal()` once again.

Returning undamped natural frequencies

```
rotor_speed = 100.0 # rad/s
modal = rotor3.run_modal(rotor_speed)
print(f"Undamped natural frequencies:\n {modal.wn}")
```

```
Undamped natural frequencies:
 [ 322.46731503  340.20234838 1052.62052762 1060.87679238 2243.16590423
 2257.55966949]
```

Returning damped natural frequencies

```
# modal.wd
print(f"Damped natural frequencies:\n {modal.wd}")
```

```
Damped natural frequencies:
 [   0.           0.        1033.02623168 1043.25981267 2231.6100209
 2246.03661749]
```

Returning the damping ratio

```
# modal.damping_ratio
print(f"Damping ratio for each mode:\n {modal.damping_ratio}")
```

stable

```
Damping ratio for each mode:
 [1.         1.         0.19204959 0.18148375 0.10137382 0.10090769]
```

Returning logarithmic decrement

```
# modal.log_dec
print(f"Logarithmic decrement for each mode:\n {modal.log_dec}")
```

```
Logarithmic decrement for each mode:
 [       inf        inf 1.22957133 1.15955161 0.64024882 0.63727447]
```

1.2.2 Plotting results

Once `run_modal()` is completed, you can check for the rotor's mode shapes. You can plot each one of the modes calculated.

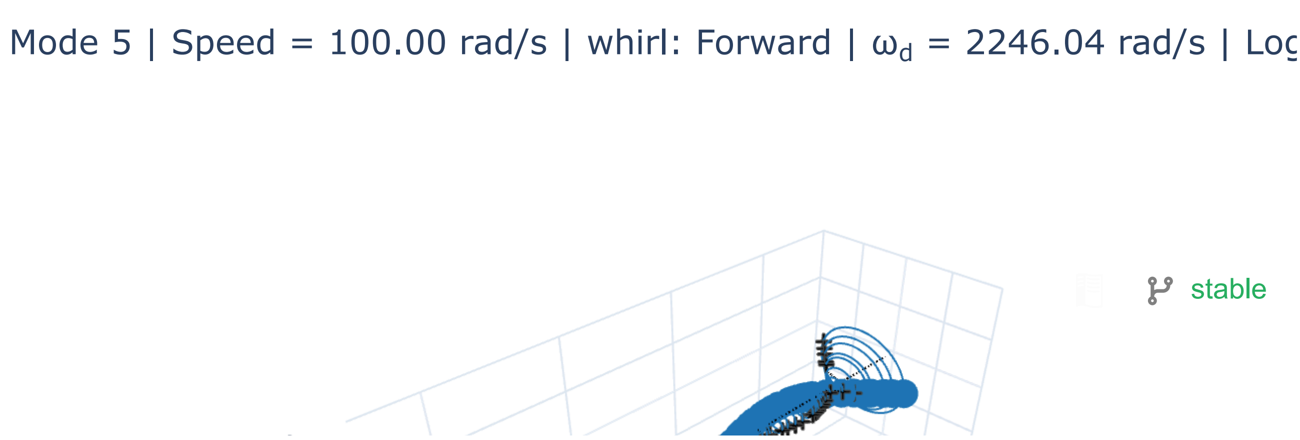
Besides, there're two options for visualization:

- `plot_mode_2d` - plotting 2D view
- `plot_mode_3d` - plotting 3D view

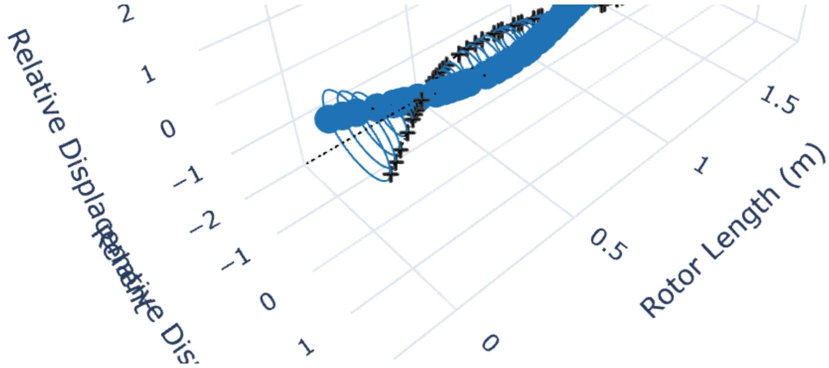
Plotting 3D view

Use the command `.plot_mode_3d(mode)`.

```
mode = 5
modal.plot_mode_3d(mode)
```



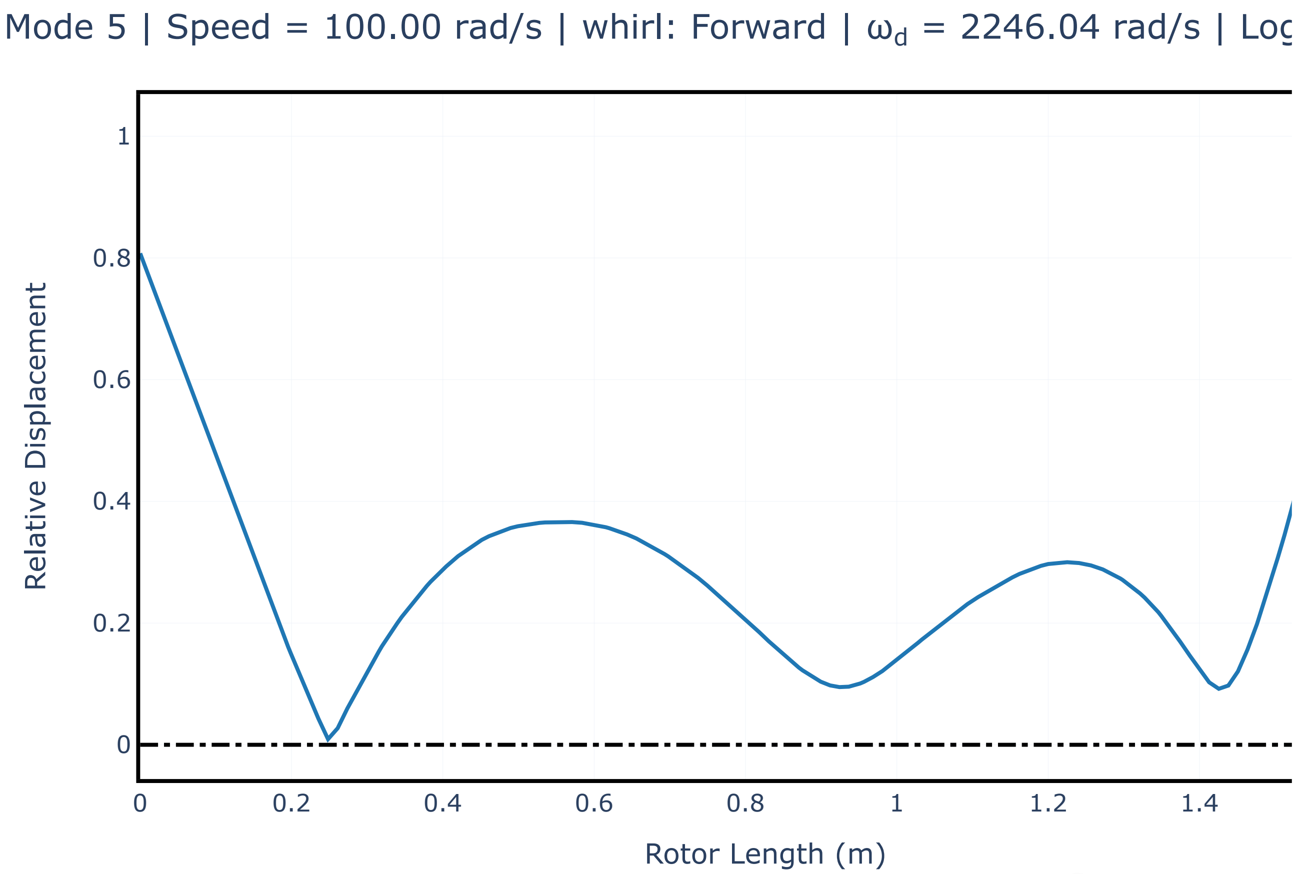
stable



Plotting 2D view

Use the command `.plot_mode_2d(mode)`.

```
modal.plot_mode_2d(mode)
```



stable

1.3 Campbell Diagram

Also called "whirl speed map" in rotordynamics, ROSS calculate and plots the modes' damped eigenvalues and the logarithmic decrement as a function of rotor speed.

To run the Campbell Diagram, use the command `.run_campbell()`. The user must input an array of speeds, which will be iterated to calculate each point on the graph.

This method returns the damped natural frequencies, logarithmic decrement and the whirl values (values indicating the whirl direction: backward or forward).

1.3.1 Running campbell diagram

In this example the whirl speed map is calculated for a speed range from 0 to 1000 rad/s (~9550 RPM).

Storing the results, it's possible to return the following arrays:

- `wd`
- `log_dec`
- `whirl_values`

Each value in these arrays is calculated for each speed value in `speed_range`

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

campbell = rotor3.run_campbell(speed_range)
```

```
c:\users\vinic\onedrive\desktop\digital_twin\github\ross\ross\rotor_assembly.py:117

Extrapolating bearing coefficients. Be careful when post-processing the results.
```

```
# results for each frequency
frequency_index = 0
print(campbell.wd[:, frequency_index])
```

stable

```
[   0.            0.            0.            0.            0.
    0.            0.            0.            0.          587.93352155
  800.71637223  947.50369431 1010.24341263 1009.42909453 1008.74707768
 1008.20525922 1007.80837754 1007.5580103  1007.45256546 1007.48784109
 1007.65818165 1007.95784535 1008.38073528 1008.92053742 1009.57067638
 1010.32433485 1011.17479229 1012.11554892 1013.14033565 1014.24319497
 1015.4185505 ]
```

```python
# results for each frequency
frequency_index = 1
print(campbell.log_dec[:, frequency_index])
```

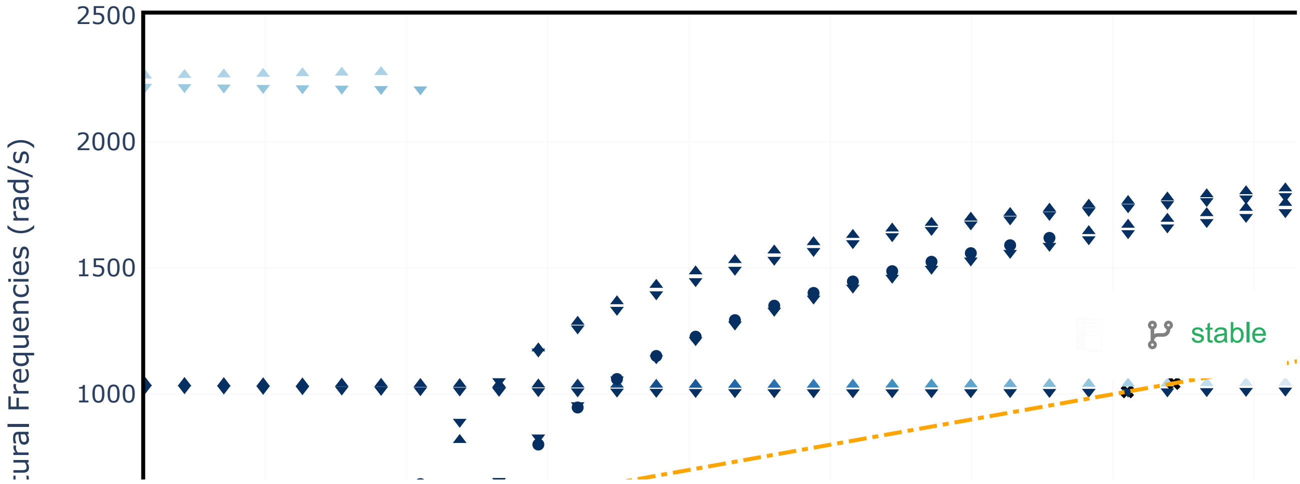
```
[           inf            inf            inf 2.51999437e+04
            inf            inf            inf 3.28326975e+01
 2.93693009e+01 1.61460643e+01 1.23356360e+01 1.03356380e+01
 9.62144988e-01 9.41828734e-01 9.20841287e-01 8.99291484e-01
 8.77283841e-01 8.54922606e-01 8.32302539e-01 8.09505400e-01
 7.86595988e-01 7.63619965e-01 7.40614114e-01 7.17608746e-01
 6.94631115e-01 6.71705921e-01 6.48853026e-01 6.26088003e-01
 6.03421978e-01 5.80862655e-01 5.58415461e-01]
```

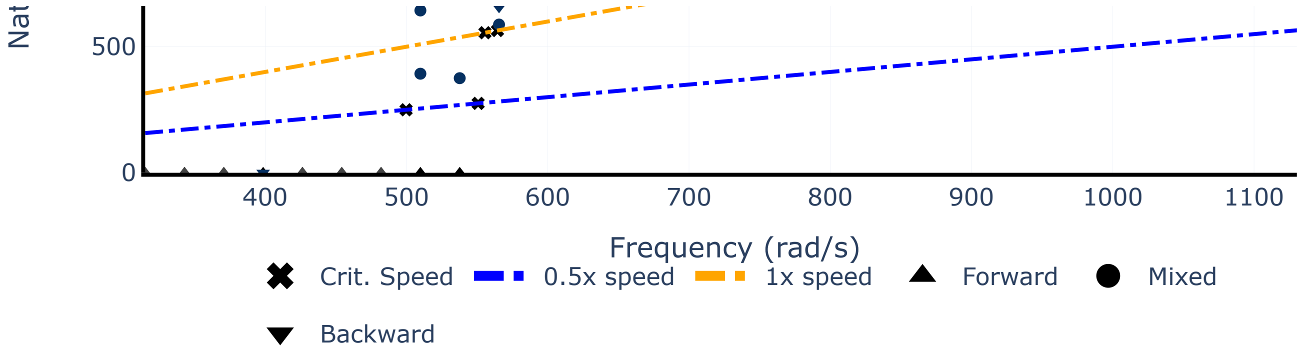
1.3.2 Plotting results

Now that the results are stored, use `.plot()` method to display the Campbell Diagram plot.

For the Campbell Diagram, you can plot more than one harmonic. As default, the plot display only the 1x speed option. Input a list with more harmonics to display it at the graph.

```python
campbell.plot(harmonics=[0.5, 1])
```




Nat
500
0
400
500
600
700
800
900
1000
1100
Frequency (rad/s)
Crit. Speed
0.5x speed
1x speed
Forward
Mixed
Backward

stable