Tutorial - Time and Frequency Analyzes

Contents

This is the third part of a basic tutorial on how to use ROSS (rotordynamics open-source software). In this tutorial, you will learn how to run time and frequency 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.

We can also use units when plotting results. For example, for a unbalance response plot we have the `amplitude_units` argument and we can choose between any length unit available in pint such as ‘meter’, ‘inch’, etc.

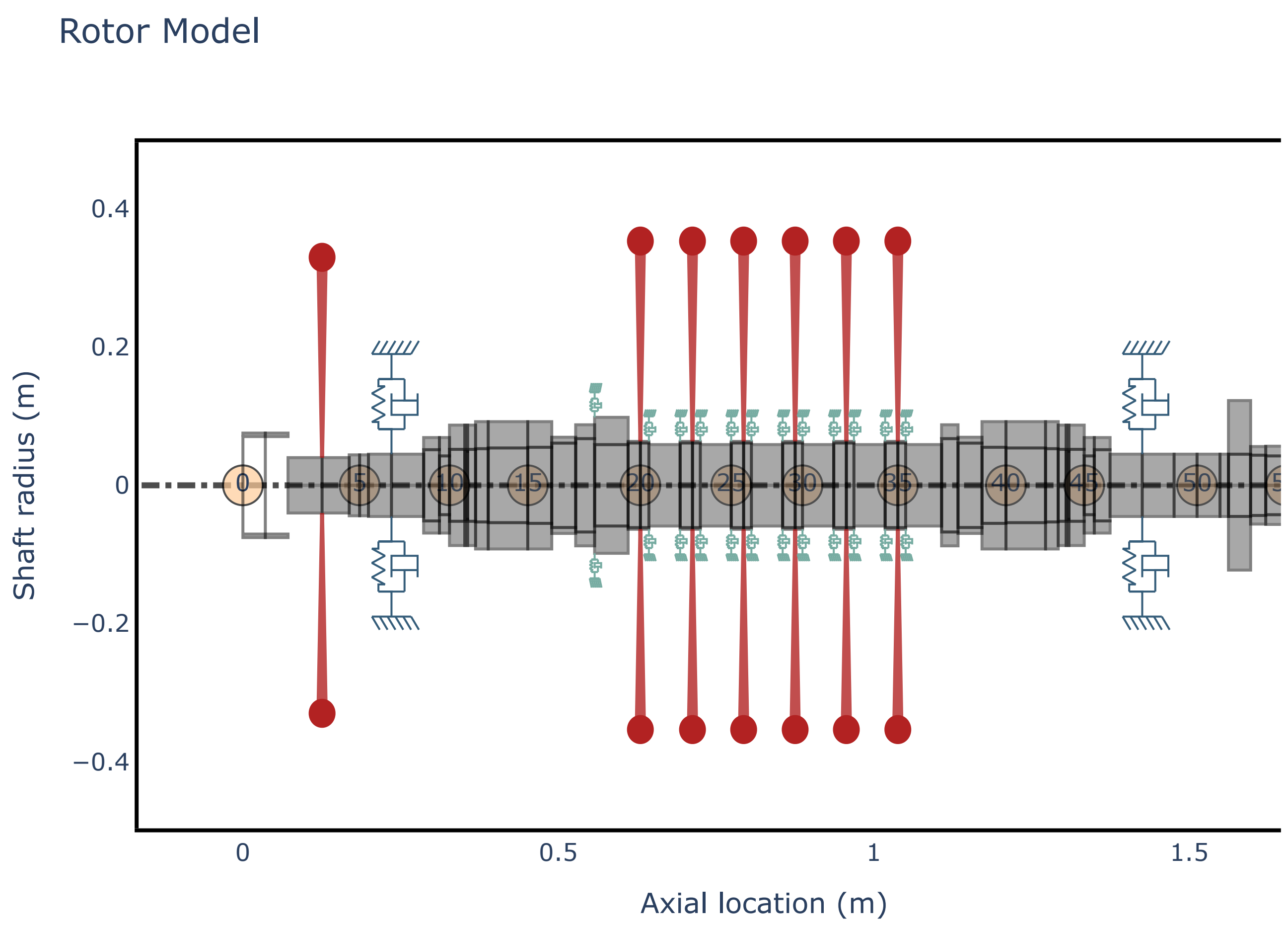
Rotor model

Again, let’s recover the rotor model built in the previous tutorial.

stable

```
import ross as rs
from ross.units import Q_
from ross.probe import Probe
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()
rotor3.plot_rotor(nodes=5)
```



Rotor Analyses

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_freq_response().plot()`)

stable

ROSS offers the following analyses:

- Frequency response
- Unbalance response
- Time response
- Undamped Critical Speed Map

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 Frequency Response

ROSS' method to calculate the Frequency Response Function is `run_freq_response()`. This method returns the magnitude and phase in the frequency domain. The response is calculated for each node from the rotor model.

When plotting the results, you can choose to plot:

- **amplitude vs frequency**: `plot_magnitude()`
- **phase vs frequency**: `plot_phase()`
- **polar plot of amplitude vs phase**: `plot_polar_bode()`
- **all**: `plot()`

1.1.1 Clustering points

The number of solution points is an important parameter to determine the computational cost of the simulation. Besides the classical method, using `numpy.linspace`, which creates an evenly spaced array over a specified interval, ROSS offers an automatic method to create an `speed_range` array.

The method `clustering_points` generates an automatic array to run frequency response analyses. The frequency points are calculated based on the damped natural frequencies and their respective damping ratios. The greater the damping ratio, the more spread the points are. If the damping ratio, for a given critical speed, is smaller than 0.005, it stable e 0.005 (for this method only).

The main goal of this feature is getting a more accurate amplitude value for the respective critical frequencies and nearby frequency points.

1.1.2 Running frequency response

To run the this analysis, use the command `run_freq_response()`. You can give a specific `speed_range` or let the program run with the default options. In this case, no arguments are needed to input.

First, let's run an example with a "user-defined" `speed_range`. Setting an array to `speed_range` will disable all the frequency spacing parameters.

```
samples = 61
speed_range = np.linspace(315, 1150, samples) # rads/s
results1 = rotor3.run_freq_response(speed_range=speed_range)
```

```
results1.speed_range.size
```

```
61
```

Now, let's run an example using *clustering points* array.

```
# results1_2 = rotor2.run_freq_response(cluster_points=True, num_points=5, num_mode
```

```
# results1_2.speed_range.size
```

In the next section we'll check the difference between both results.

1.1.3 Plotting results - Bode Plot

We can plot the frequency response selecting the input and output degree of freedom.

- Input is the degree of freedom to be excited;
- Output is the degree of freedom to be observed.

stable

Each shaft node has 4 local degrees of freedom (dof) $[x, y, \alpha, \beta]$, and each degree of freedom has it own index:

- x -> index 0
- y -> index 1
- α -> index 2
- β -> index 3

To select a DoF to input and a DoF to the output, we have to use the following correlation:

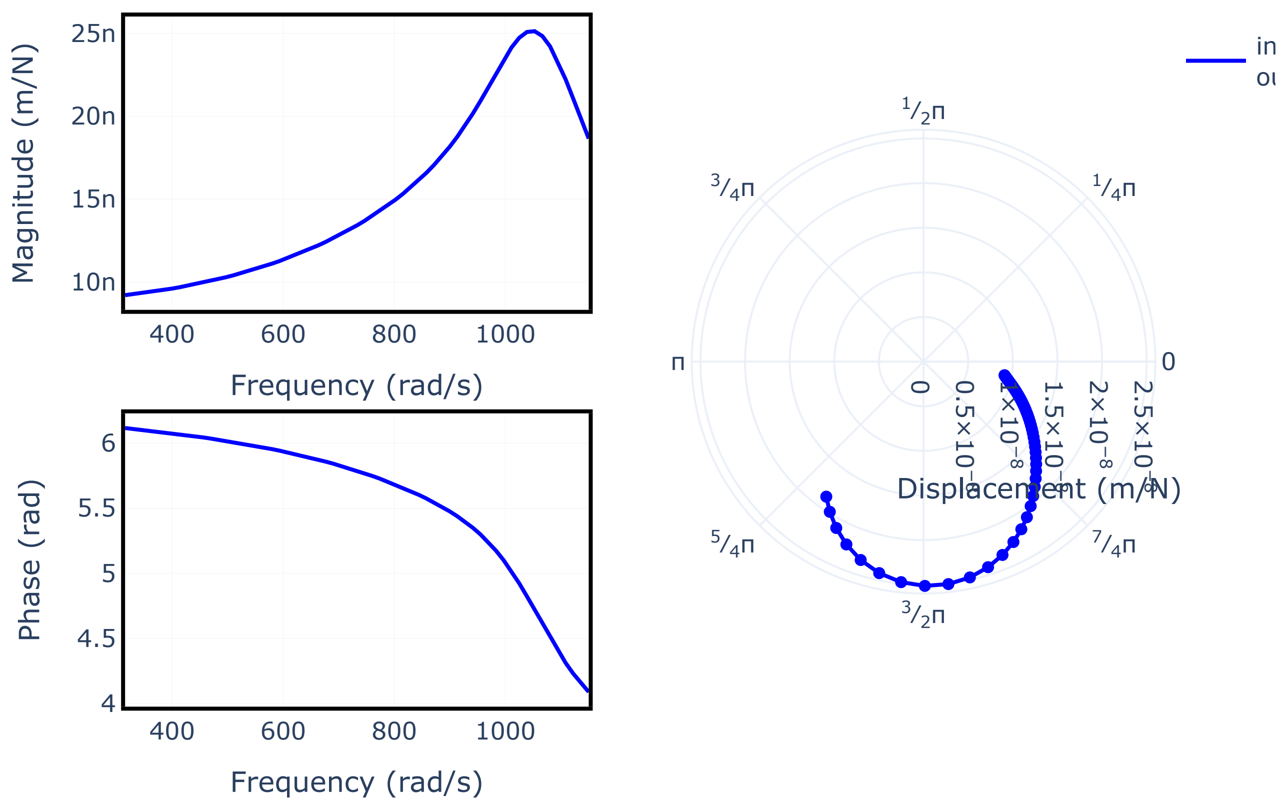
$$global\_dof = node\_number \cdot dof\_per\_node + dof\_index$$

For example: node 26, local dof y:

$$DoF = 26 \cdot 4 + 1 = 105$$

```
plot = results1.plot(inp=105, out=105)
plot

# converting the first plot yaxis to log scale
# plot = results1.plot(inp=105, out=105)
# plot.update_yaxes(type="log", row=1, col=1)
# plot
```

stable



```
# plot1_2 = results1_2.plot(inp=105, out=105)
# plot1_2

# # converting the first plot yaxis to log scale
# plot1_2 = results1_2.plot(inp=105, out=105)
# plot1_2.update_yaxes(type="log", row=1, col=1)
# plot1_2
```

1.2 Unbalance Response

ROSS' method to simulate the reponse to an unbalance is `run_unbalance_response()`. This method returns the unbalanced response in the frequency domain for a given magnitide and phase of the unbalance, the node where it's applied and a frequency range.

ROSS takes the magnitude and phase and converts to a complex force array applied to the given node:

stable

$$force = \begin{pmatrix} F \cdot e^{j\delta} \\ -jF \cdot e^{j\delta} \\ 0 \\ 0 \end{pmatrix}$$

where:

- F is the unbalance magnitude;
- δ is the unbalance phase;
- j is the complex number notation;

When plotting the results, you can choose to plot the:

- Bode plot options for a single degree of freedom:
  - amplitude vs frequency: `plot_magnitude()`
  - phase vs frequency: `plot_phase()`
  - polar plot of amplitude vs phase: `plot_polar_bode()`
  - all: `plot()`
- Deflected shape plot options:
  - deflected shape 2d: `plot_deflected_shape_2d()`
  - deflected shape 3d: `plot_deflected_shape_3d()`
  - bending moment: `plot_bending_moment()`
  - all: `plot_deflected_shape()`

`run_unbalance_response()` is also able to work with clustering points (*see section 7.4.1*).

1.2.1 Running unbalance response

To run the Unbalance Response, use the command `.unbalance_response()`

In this following example, we can obtain the response for a given unbalance and its respective phase in a selected node. Notice that it's possible to add multiple unbalances instantiating node, magnitude and phase as lists.

The method returns the force response array (complex values), the displacement magnitude (absolute value of the forced response) and the phase of the forced response.

stable

Let's run an example with 2 unbalances in phase, trying to excite the first and the third natural vibration mode.

```
Unbalance1: node = 29
            magnitude = 0.003
            phase = 0
Unbalance2: node = 33
            magnitude = 0.002
            phase = 0
```

```
n1 = 29
m1 = 0.003
p1 = 0

n2 = 33
m2 = 0.002
p2 = 0

frequency_range=np.linspace(315, 1150, 101)
results2 = rotor3.run_unbalance_response([n1, n2], [m1, m2], [p1, p2], frequency_ra
```

1.2.2 Plotting results - Bode Plot

To display the bode plot, use the command `.plot(probe)`

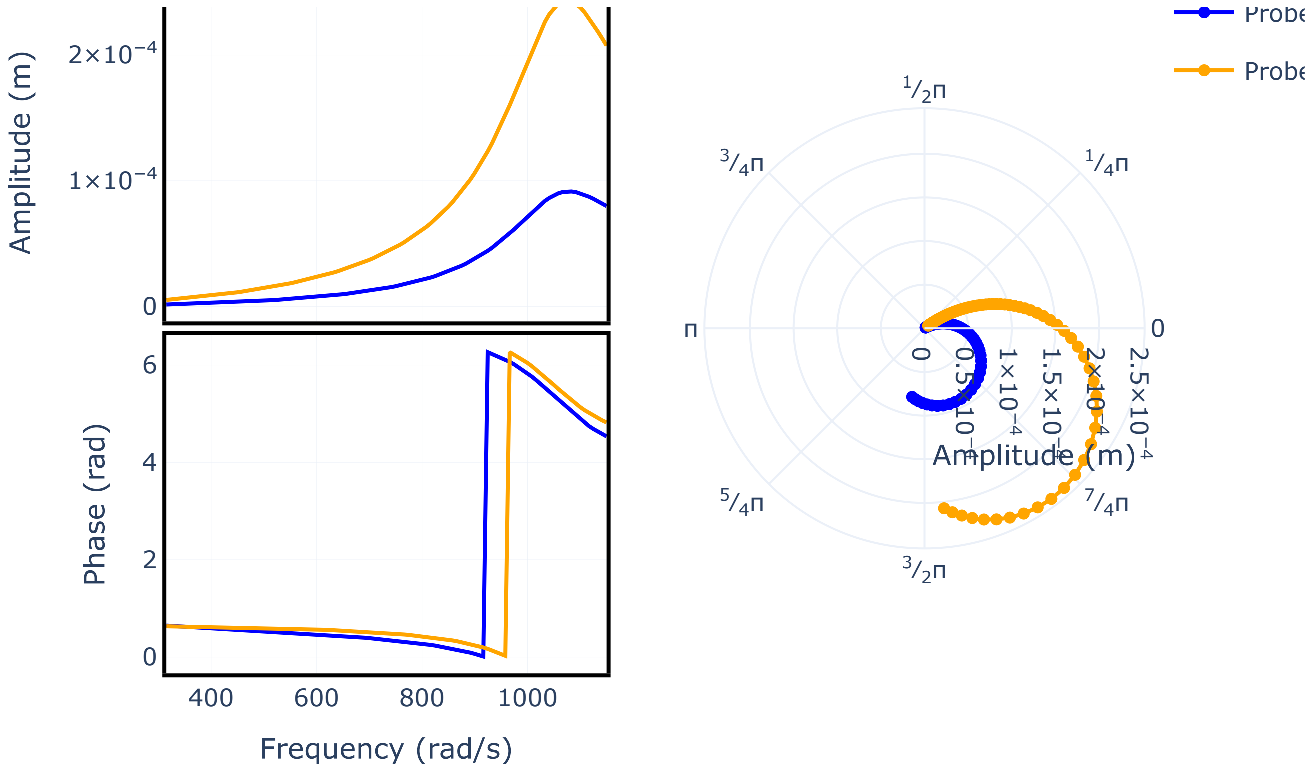
Where `probe` is a list of Probe objects that allows you to choose not only the node where to observe the response, but also the orientation.

Probe orientation equals 0° refers to `+X` direction (DoFX), and probe orientation equals 90° (or $\frac{\pi}{2} rad$) refers to `+Y` direction (DoFY).

You can insert multiple probes at once.

```
# probe = Probe(probe_node, probe_orientation)
probe1 = Probe(15, Q_(45, "deg")) # node 15, orientation 45°
probe2 = Probe(35, Q_(45, "deg")) # node 35, orientation 45°

results2.plot(probe=[probe1, probe2])

# converting the first plot yaxis to log scale
# plot2 = results2.plot(probe=[probe1, probe2], probe_units="rad")
# plot2.update_yaxes(type="log", row=1, col=1)
# plot2
```

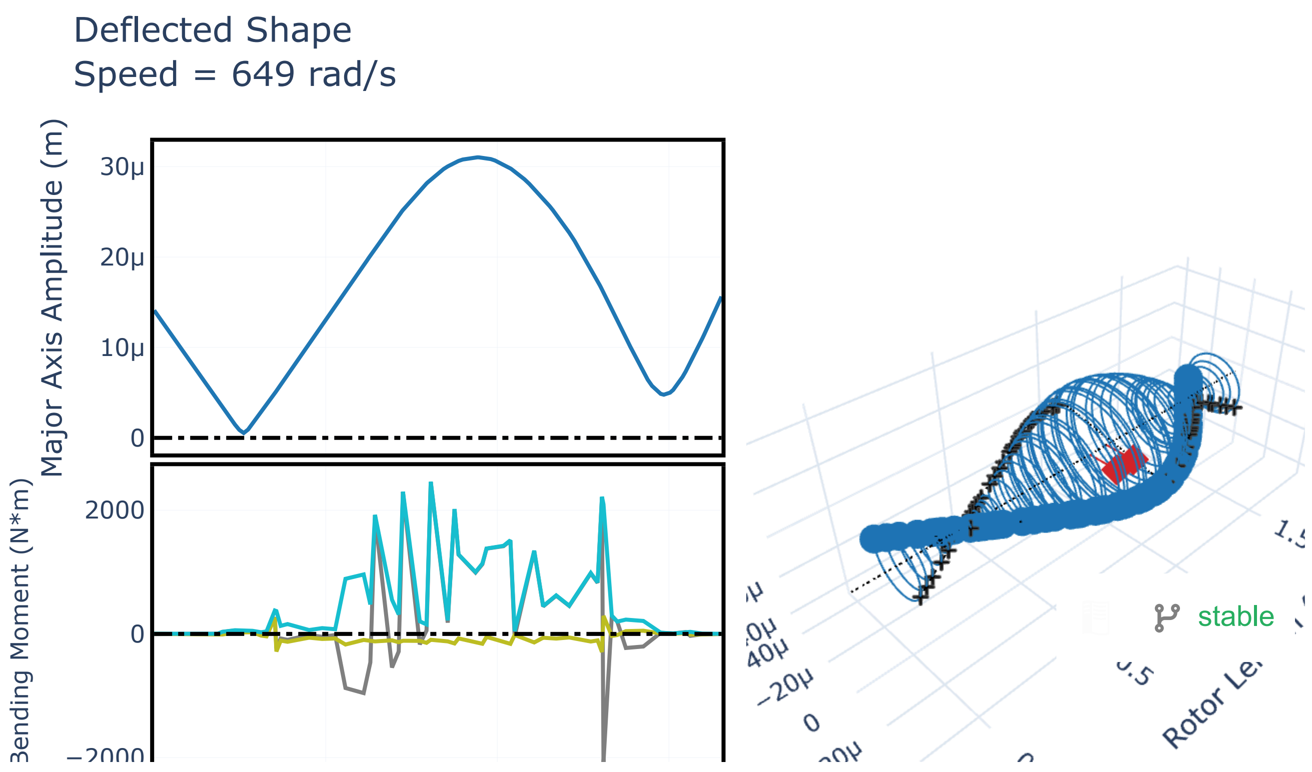
stable

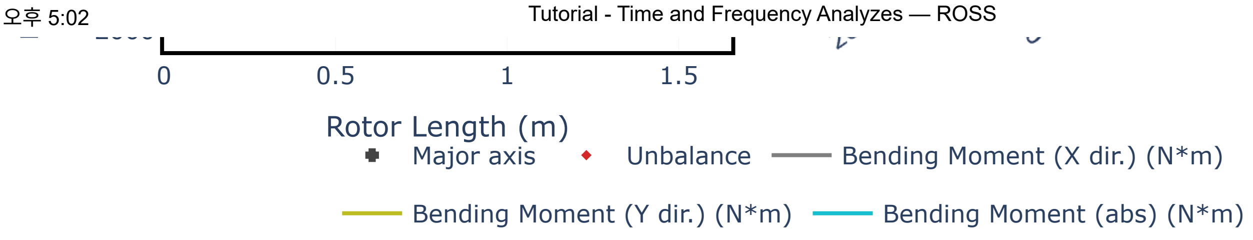


1.2.3 Plotting results - Deflected shape

To display the deflected shape configuration, use the command `.plot_deflected_shape()`

```
results2.plot_deflected_shape(speed=649)
```





1.3 Time Response

ROSS' method to calculate displacements due a force in time domain is `run_time_response()`. This function will take a rotor object and plot its time response given a force and a time array.

The force input must be a matrix $M \times N$, where:

- M is the size of the time array;
- N is the rotor's number of DoFs (you can access this value via attribute `.ndof`).

Each row from the matrix represents a node, and each column represents a time step.

Time Response allows you to plot the response for:

- a list of probes
- an orbit for a given node (2d plot)
- all the nodes orbits (3d plot)

1.3.1 Running time response

To run the Time Response, use the command `.run_time_response()`.

Building the force matrix is not trivial. We recommend creating a matrix of zeros using *numpy.zeros()* and then, adding terms to the matrix.

In this examples, let's create an harmonic force on node 26, in x and y directions (remember index notation from Frequency Response (section 7.4.2). We'll plot results from 0 to 10 seconds of simulation.

```
speed = 600.0
time_samples = 1001
node = 26
t = np.linspace(0, 16, time_samples)

F = np.zeros((time_samples, rotor3.ndof))

# component on direction x
F[:, 4 * node + 0] = 10 * np.cos(2 * t)
# component on direction y
F[:, 4 * node + 1] = 10 * np.sin(2 * t)

response3 = rotor3.run_time_response(speed, F, t)
```

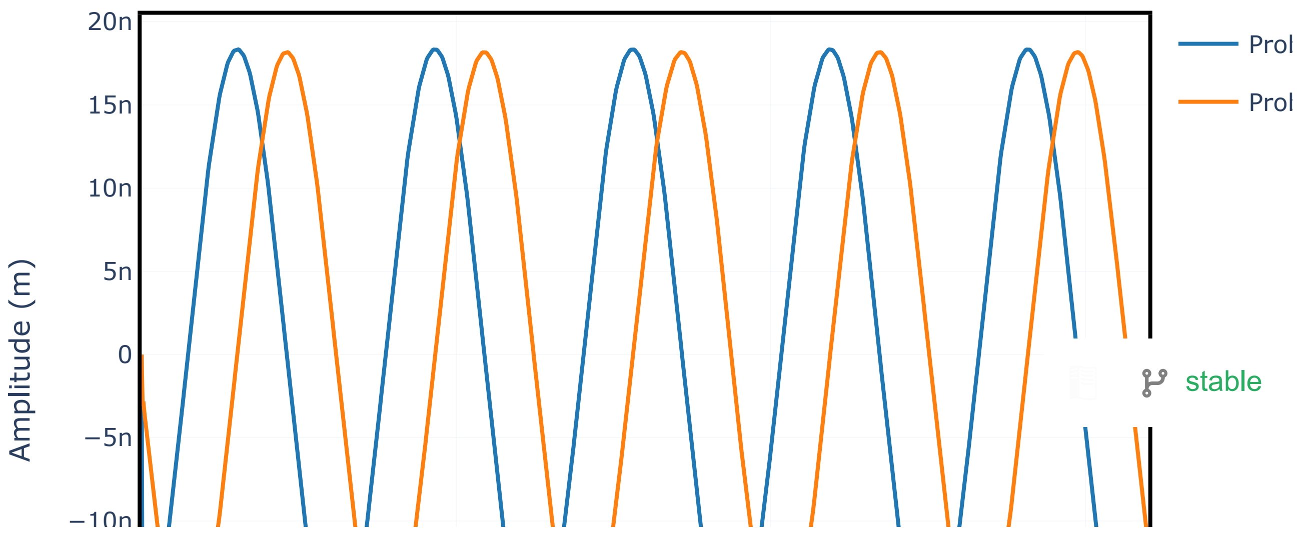
1.3.2 Plotting results

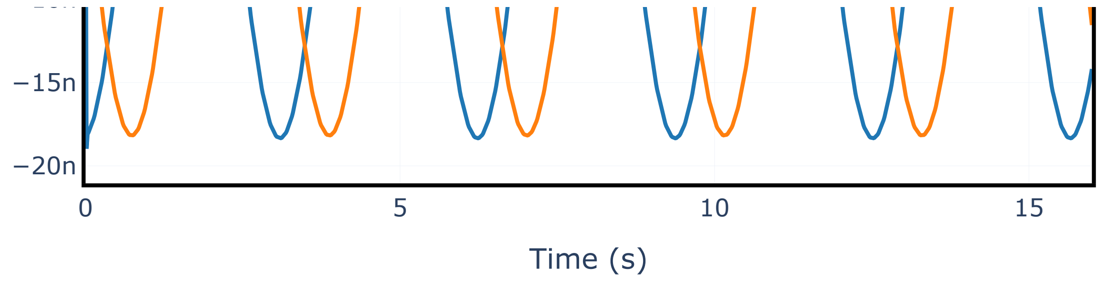
There 3 (three) different options to plot the time response:

- `.plot_1d()`: plot time response for given probes.
- `.plot_2d()`: plot orbit of a selected node of a rotor system.
- `.plot_3d()`: plot orbits for each node on the rotor system in a 3D view.

Ploting time response for list of probes

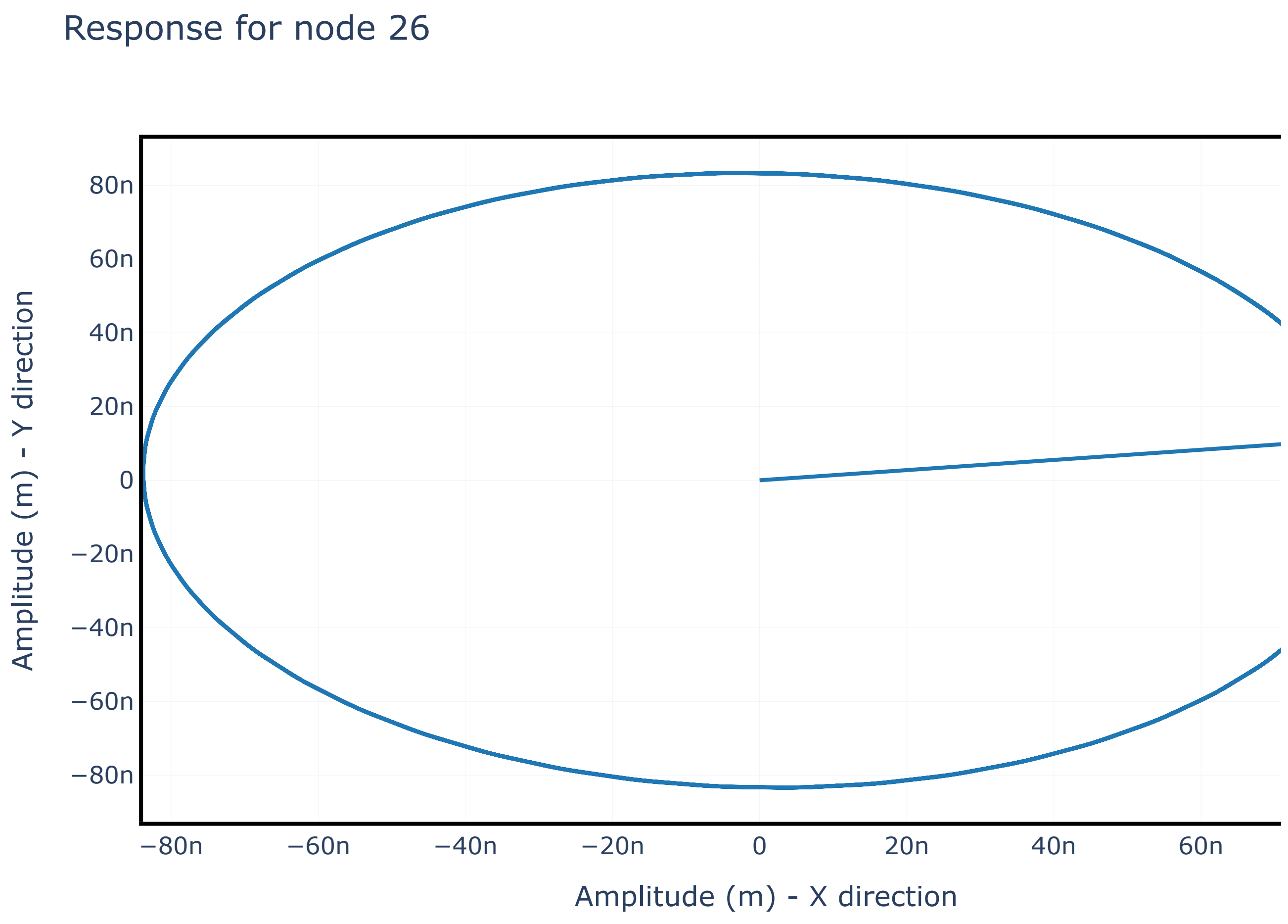
```
probe1 = Probe(3, 0)   # node 3, orientation 0° (X dir.)
probe2 = Probe(3, Q_(90, "deg"))  # node 3, orientation 90°(Y dir.)

response3.plot_1d(probe=[probe1, probe2])
```





Ploting orbit response for a single node

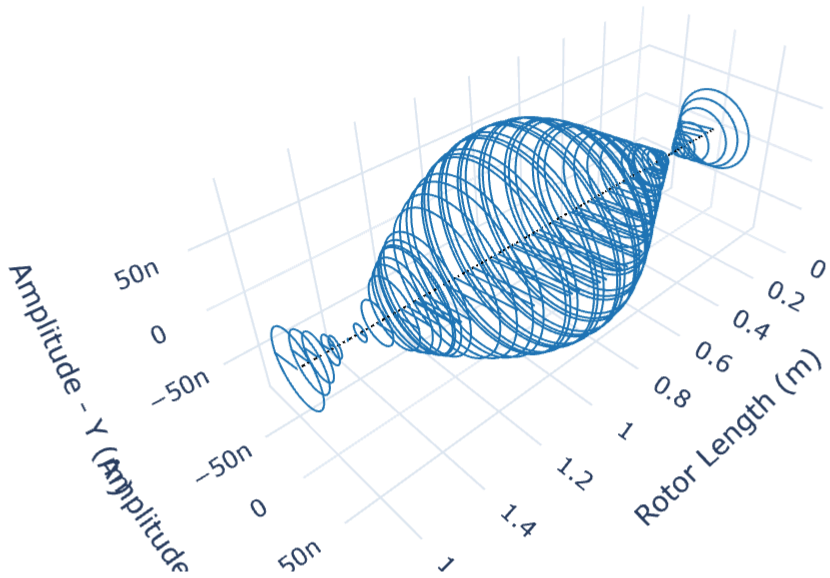
```
node = 26
response3.plot_2d(node=node)
```



Ploting orbit response for all nodes

stable

```
response3.plot_3d()
```



1.4 Undamped Critical Speed Map (UCS)

This method will plot the undamped critical speed map for a given range of stiffness values. If the range is not provided, the bearing stiffness at rated speed will be used to create a range.

Whether a synchronous analysis is desired, the method selects only the foward modes and the frequency of the first forward mode will be equal to the speed.

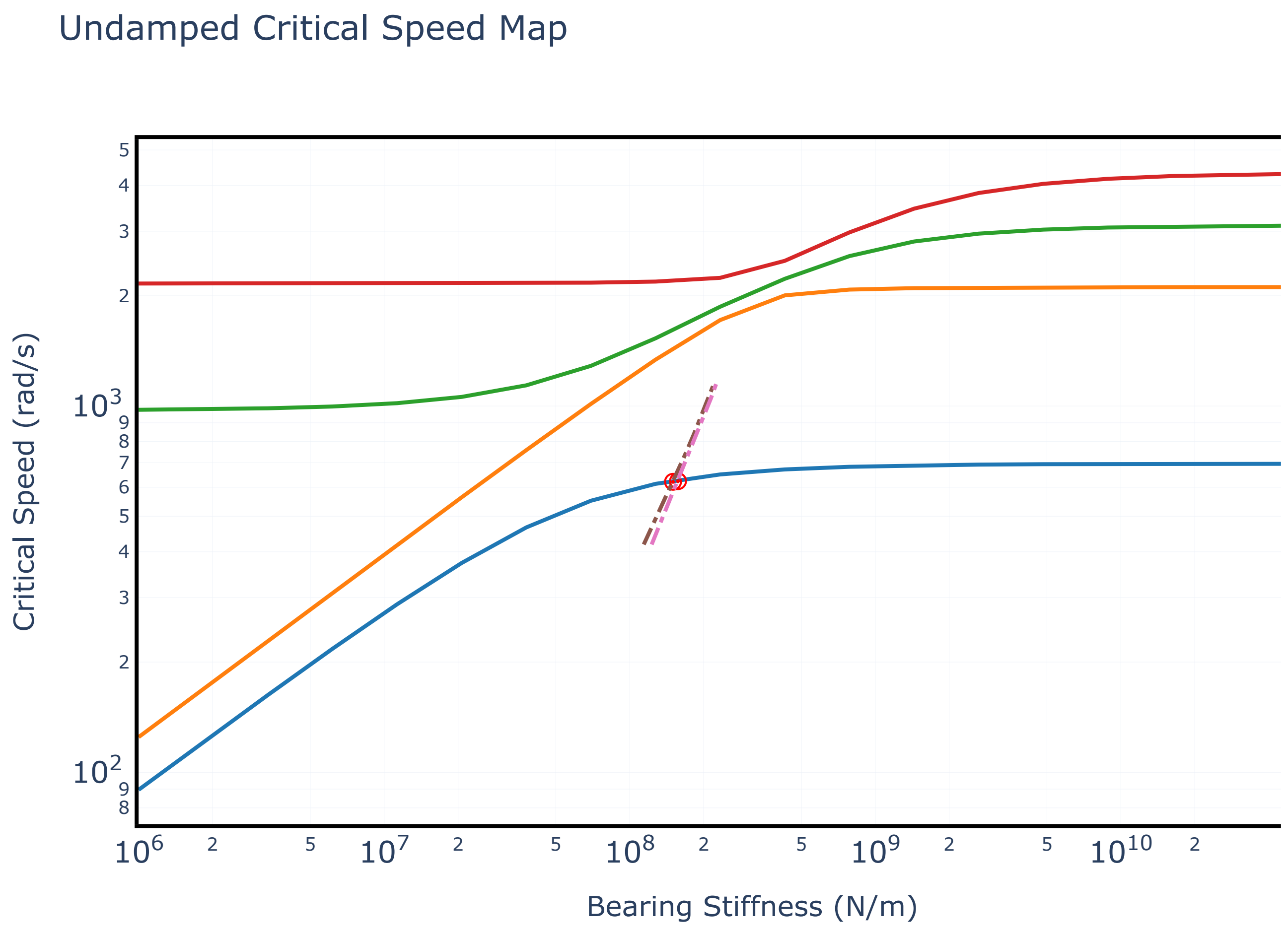
To run the UCS Map, use the command `.plot_ucs()`.

1.4.1 Running and plotting UCS Map

In this example the UCS Map is calculated for a stiffness range from **10E6** to **10E11** N/m. The other options are left to default.

stable

```
stiff_range = (6, 11)
ucs_results = rotor3.run_ucs(stiffness_range=stiff_range, num=20, num_modes=16)
ucs_fig = ucs_results.plot()
ucs_fig
```


Undamped Critical Speed Map
Critical Speed (rad/s)
Bearing Stiffness (N/m)
10^6
10^7
10^8
10^9
10^10
10^2
10^3