

Tutorial - Stochastic ROSS

Contents

- Random Sampling
- Classes Name
- ST_Material
- ST_ShaftElement
- ST_DiskElement
- ST_BearingElement
- ST_Rotor
- Running the simulation
- Obtaining results
- Plotting results

This is a basic tutorial on how to use STOCHASTIC ROSS - a ROSS' module for stochastic rotordynamics analysis. Before starting this tutorial, be sure you're already familiar with ROSS library.

If you've already used ROSS, you've noticed the graphs present deterministic results, considering a set of parameters. In other words, the model always produce the same output from a given starting condition or initial state. In STOCHASTIC ROSS, the concept is different, and we'll work it stochastic processes.

A stochastic process is defined as a indexed collection of random variables defined on a common probability space (Ω, \mathcal{F}, P) where Ω is a sample space, \mathcal{F} is a σ -algebra, and P is a probability measure. The index is often assumed to be time.

This new module allows you to work with random variables applied to the ROSS' functions. Basically, any element or material can be receive a parameter considered random. Moreover, some methods are also able to receive a random variable (random force, random unbalance...). It means that a parameter, once assumed deterministic (int or float in python language), now follows a distribution (list or array), like uniform distribution, normal distribution, etc.



As consequence, plots do not display deterministic results anymore. Instead, plots shows the expectation $E(X_t(t))$ (or mean) for a stochastic process and intervals of confidence (user choice).

Where:

- X_t is the stochastic process;
- t is the index time

```
import ross as rs
import ross.stochastic as srs
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"
```

Random Sampling

Arrays of random numbers can be creating using `numpy.random` package.

`numpy.random` has a large set of distributions that cover most of our needs to run STOCHASTIC ROSS. In this [LINK](#) you can find a list of numpy random numbers generators.

When using STOCHASTIC ROSS, **all the random variables must have the same size.**


Classes Name

It's important to highlight that in STOCHASTIC ROSS, the classes name are the same than ROSS, but with a "**ST_**" prefix to differ.

ST_Material

There is a class called ST_Material to hold material's properties, where:

`ST_Material` allows you to create a material with random properties. It creates an object containing a generator with random instances of `rs.Material`.

The instantiation is the same than `rs.Material` class. It has the same  `stable` assumptions. The only difference is that you are able to select some parameters to consider as random and instantiate it as a list.

The parameters which can be passed as random are:

- `rho` - Density
- `E` - Young's modulus
- `G_s` - Shear modulus
- `Poisson` - Poisson ratio

```
name : str
    Material name.
rho : float, list, pint.Quantity
    Density (kg/m**3).
    Input a list to make it random.
E : float, list, pint.Quantity
    Young's modulus (N/m**2).
    Input a list to make it random.
G_s : float, list
    Shear modulus (N/m**2).
    Input a list to make it random.
Poisson : float, list
    Poisson ratio (dimensionless).
    Input a list to make it random.
color : str
    Can be used on plots.
```

Note that, to instantiate a `ST_Material` class, you only need to give 2 out of the following parameters: `E`, `G_s`, `Poisson`.

Let's consider that the Young's Modulus is a random variable the follows a uniform distribution from $208e9$ to $211e9 \text{ N/m}^2$.

```
var_size = 5
E = np.random.uniform(208e9, 211e9, var_size)
rand_mat = srs.ST_Material(name="Steel", rho=7810, E=E, G_s=81.2e9)

# Random values for Young's Modulus
print(rand_mat["E"])
```

```
[2.09870978e+11 2.09622532e+11 2.08564528e+11 2.08840575e+11
 2.08383344e+11]
```

You can return the random Materials created using the following command: `iter()` It returns a generator with the random objects. It consumes less computational memory and runs loops faster.

 `stable`

```
list(iter(rand_mat))
```

```
[Material(name="Steel", rho=7.81000e+03, G_s=8.12000e+10, E=2.09871e+11, color='#52
Material(name="Steel", rho=7.81000e+03, G_s=8.12000e+10, E=2.09623e+11, color='#52
Material(name="Steel", rho=7.81000e+03, G_s=8.12000e+10, E=2.08565e+11, color='#52
Material(name="Steel", rho=7.81000e+03, G_s=8.12000e+10, E=2.08841e+11, color='#52
Material(name="Steel", rho=7.81000e+03, G_s=8.12000e+10, E=2.08383e+11, color='#52
```

You can pass one or all parameters as random (but remember the rule of given only 2 out of `E`, `G_s`, `Poisson`).

Let's see another example considering all parameters as random.

```
var_size = 5
E = np.random.uniform(208e9, 211e9, var_size)
rho = np.random.uniform(7780, 7850, var_size)
G_s = np.random.uniform(79.8e9, 81.5e9, var_size)
rand_mat = srs.ST_Material(name="Steel", rho=rho, E=E, G_s=G_s)

list(iter(rand_mat))
```

```
[Material(name="Steel", rho=7.79785e+03, G_s=8.00129e+10, E=2.10438e+11, color='#52
Material(name="Steel", rho=7.80866e+03, G_s=8.10989e+10, E=2.10230e+11, color='#52
Material(name="Steel", rho=7.84218e+03, G_s=8.02146e+10, E=2.09094e+11, color='#52
Material(name="Steel", rho=7.81669e+03, G_s=8.06195e+10, E=2.10574e+11, color='#52
Material(name="Steel", rho=7.78422e+03, G_s=8.13542e+10, E=2.08495e+11, color='#52
```

ST_ShaftElement

`ST_ShaftElement` allows you to create random shaft element. It creates an object containing a generator with random instances of `ShaftElement`.

The instantiation is the same than `rs.ShaftElement` class. It has the same parameters and the same beam model and assumptions. The only difference is that you are able to select some parameters to consider as random and instantiate it as a list.

The parameters which can be passed as random are:

- `L` - Length
- `idl` - Inner diameter of the element at the left position
- `odl` - Outer diameter of the element at the left position
- `idr` - Inner diameter of the element at the right position
- `odr` - Outer diameter of the element at the right position.

 stable

- `material` - Shaft material

The selected parameters must be appended to `is_random` list as string.

You can return the random shaft element created using the following command: `iter()`.

```
L : float, pint.Quantity, list
    Element length.
    Input a list to make it random.
idl : float, pint.Quantity, list
    Inner diameter of the element at the left position.
    Input a list to make it random.
odl : float, pint.Quantity, list
    Outer diameter of the element at the left position.
    Input a list to make it random.
idr : float, pint.Quantity, list, optional
    Inner diameter of the element at the right position
    Default is equal to idl value (cylindrical element)
    Input a list to make it random.
odr : float, pint.Quantity, list, optional
    Outer diameter of the element at the right position.
    Default is equal to odl value (cylindrical element)
    Input a list to make it random.
material : ross.material, list of ross.material
    Shaft material.
    Input a list to make it random.
n : int, optional
    Element number (coincident with it's first node).
    If not given, it will be set when the rotor is assembled
    according to the element's position in the list supplied to
shear_effects : bool, optional
    Determine if shear effects are taken into account.
    Default is True.
rotary_inertia : bool, optional
    Determine if rotary_inertia effects are taken into account.
    Default is True.
gyroscopic : bool, optional
    Determine if gyroscopic effects are taken into account.
    Default is True.
shear_method_calc : str, optional
    Determines which shear calculation method the user will adopt.
    Default is 'cowper'
is_random : list
    List of the object attributes to become random.
    Possibilities:
    ["L", "idl", "odl", "idr", "odr", "material"]
```

Cylindrical shaft element with random outer diameter

 [stable](#)

If you want to create a cylindrical element with random outer diameter, making sure both `odl` and `odr` are the same, input only `odl` parameter.

The same logic is applied to inner diameter.

```
# Creating a cylindrical shaft element with random outer diameter and material.
var_size = 5
L = 0.25
i_d = 0.0
o_d = np.random.uniform(0.04, 0.06, var_size)
is_random = ["odl", "material"]

r_s0 = srs.ST_ShaftElement(
    L=L,
    idl=i_d,
    odl=o_d,
    material=rand_mat,
    shear_effects=True,
    rotary_inertia=True,
    gyroscopic=True,
    is_random=is_random,
)
list(iter(r_s0))
```

```
[ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.057122, odr=0.057122, material='Stee
ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.041971, odr=0.041971, material='Stee
ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.041351, odr=0.041351, material='Stee
ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.044317, odr=0.044317, material='Stee
ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.056775, odr=0.056775, material='Stee
```

Conical shaft element with random outer diameter

If you want to create a conical element with random outer diameter, input lists for `odl` and `odr` parameters.

```
# Creating a conical shaft element with random outer diameter and material.
var_size = 5
L = 0.25
idl = 0.0
idr = 0.0
odl = np.random.uniform(0.04, 0.06, var_size)
odr = np.random.uniform(0.06, 0.07, var_size)
is_random = ["odl", "odr", "material"]

r_s1 = srs.ST_ShaftElement(
    L=L,
    idl=idl,
    odl=odl,
    idr=idr,
    odr=odr,
    material=rand_mat,
    shear_effects=True,
    rotary_inertia=True,
    gyroscopic=True,
    is_random=is_random,
)
list(iter(r_s1))
```

```
[ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.054486, odr=0.060876, material='Steel',
ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.047353, odr=0.065821, material='Steel',
ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.050009, odr=0.061408, material='Steel',
ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.051292, odr=0.067379, material='Steel',
ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.049958, odr=0.06732, material='Steel',
```

Creating a list of shaft elements

Let's see 2 examples of building rotor shafts:

- a shaft with 5 shaft elements considered random

```
shaft_elements = [
    ST_ShaftElement,
    ST_ShaftElement,
    ST_ShaftElement,
    ST_ShaftElement,
    ST_ShaftElement,
]
```

- a shaft with 5 elements, being only the 3rd element considered as random. So we want;

 [stable](#)

```
shaft_elements = [
    ShaftElement,
    ShaftElement,
    ST_ShaftElement,
    ShaftElement,
    ShaftElement,
]
```

First we create the deterministic shaft elements.

```
##### EXAMPLE 1 #####

# Creating 5 random shaft elements
from ross.materials import steel
L = 0.25
N = 5          # Number of elements
l_list = [L for _ in range(N)]
shaft_elements = [
    srs.ST_ShaftElement(
        L=L,
        idl=0.0,
        odl=np.random.uniform(0.04, 0.06, var_size),
        material=steel,
        shear_effects=True,
        rotary_inertia=True,
        gyroscopic=True,
        is_random=["odl"],
    )
    for l in l_list
]

# printing
for i in range(N):
    print("Element", i)
    print(list(iter(shaft_elements[i])))
```

```
Element 0
[ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.05952, odr=0.05952, material='Steel'
Element 1
[ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.059202, odr=0.059202, material='Stee
Element 2
[ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.045626, odr=0.045626, material='Stee
Element 3
[ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.047565, odr=0.047565, material='Stee
Element 4
[ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.046868, odr=0.046868, material='Stee
```


EXAMPLE 2

```
# Creating shaft elements
from ross.materials import steel
L = 0.25
i_d = 0.0
o_d = 0.05
N = 4          # Number of elements
l_list = [L for _ in range(N)]
shaft_elements = [
    rs.ShaftElement(
        L=L,
        idl=i_d,
        odl=o_d,
        material=steel,
        shear_effects=True,
        rotary_inertia=True,
        gyroscopic=True,
    )
    for l in l_list
]
shaft_elements

# Adding the random shaft element instance to the list
shaft_elements.insert(2, r_s0)
shaft_elements
```

```
[ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.05, odr=0.05, material='Steel', n=No
ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.05, odr=0.05, material='Steel', n=No
<ross.stochastic.st_shaft_element.ST_ShaftElement at 0x7f1bc73664a0>,
ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.05, odr=0.05, material='Steel', n=No
ShaftElement(L=0.25, idl=0.0, idr=0.0, odl=0.05, odr=0.05, material='Steel', n=No
```

ST_DiskElement

This class represents a random Disk element.

`ST_DiskElement` allows you to create random disk element. It creates an object containing a generator with random instances of `rs.DiskElement`.

The instantiation is the same than `DiskElement` class. It has the same parameters and assumptions. The only difference is that you are able to select some parameters to consider as random and instantiate it as a list.

The parameters which can be passed as random are:

- `m` - mass

 **stable**

- `Id` - Diametral moment of inertia.
- `Ip` - Polar moment of inertia

The selected parameters must be appended to `is_random` list as string.

You can return the random disk element created using the following command: `iter()`.

```
n: int
    Node in which the disk will be inserted.
m : float, list
    Mass of the disk element.
    Input a list to make it random.
Id : float, list
    Diametral moment of inertia.
    Input a list to make it random.
Ip : float, list
    Polar moment of inertia
    Input a list to make it random.
tag : str, optional
    A tag to name the element
    Default is None
color : str, optional
    A color to be used when the element is represented.
    Default is '#b2182b' (Cardinal).
is_random : list
    List of the object attributes to become random.
    Possibilities:
        ["m", "Id", "Ip"]
```

All the values are following the S.I. convention for the units.

```
m = np.random.uniform(32.0, 33.0, var_size)
Id = np.random.uniform(0.17, 0.18, var_size)
Ip = np.random.uniform(0.32, 0.33, var_size)
is_random = ["m", "Id", "Ip"]

disk0 = srs.ST_DiskElement(n=2, m=m, Id=Id, Ip=Ip, is_random=is_random)
list(iter(disk0))
```

```
[DiskElement(Id=0.17634, Ip=0.32465, m=32.159, color='Firebrick', n=2, scale_factor=
DiskElement(Id=0.17615, Ip=0.32739, m=32.818, color='Firebrick', n=2, scale_factor=
DiskElement(Id=0.17734, Ip=0.32889, m=32.431, color='Firebrick', n=2, scale_factor=
DiskElement(Id=0.17144, Ip=0.32967, m=32.443, color='Firebrick', n=2, scale_factor=
DiskElement(Id=0.1735, Ip=0.32955, m=32.414, color='Firebrick', n=2, scale_factor=
```

From geometry DiskElement instantiation

Besides the instantiation previously explained, there is a way to instantiate a `ST_DiskElement` with only geometrical parameters (for cylindrical disks) and the disk's material, as we can see in the following code.

Use the classmethod `ST_DiskElement.from_geometry`.

```
n: int
    Node in which the disk will be inserted.
material: ross.Material, list of ross.Material
    Disk material.
    Input a list to make it random.
width: float, list
    The disk width.
    Input a list to make it random.
i_d: float, list
    Inner diameter.
    Input a list to make it random.
o_d: float, list
    Outer diameter.
    Input a list to make it random.
tag : str, optional
    A tag to name the element
    Default is None
is_random : list
    List of the object attributes to become random.
    Possibilities:
        ["material", "width", "i_d", "o_d"]

i_d = np.random.uniform(0.05, 0.06, var_size)
o_d = np.random.uniform(0.35, 0.39, var_size)
disk1 = srs.ST_DiskElement.from_geometry(n=3,
                                         material=steel,
                                         width=0.07,
                                         i_d=i_d,
                                         o_d=o_d,
                                         is_random=["i_d", "o_d"],
                                         )

list(iter(disk1))
```

```
[DiskElement(Id=0.58094, Ip=1.1125, m=60.476, color='Firebrick', n=3, scale_factor=
DiskElement(Id=0.56419, Ip=1.0796, m=59.754, color='Firebrick', n=3, scale_factor=
DiskElement(Id=0.63604, Ip=1.2201, m=63.587, color='Firebrick', n=3, scale_factor=
DiskElement(Id=0.51379, Ip=0.98106, m=56.97, color='Firebrick', n=3, scale_factor=
DiskElement(Id=0.47054, Ip=0.89673, m=54.295, color='Firebrick', n=3, scale_factor=
```

ST_BearingElement

This class represents a random bearing element.

`ST_BearingElement` allows you to create random disk element. It creates an object containing a generator with random instances of `rs.BearingElement`.

The instantiation is the same than `BearingElement` class. It has the same parameters and assumptions. The only difference is that you are able to select some parameters to consider as random and instantiate it as a list.

If you're considering constant coefficients, use an 1-D array to make it random. If you're considering varying coefficients to the frequency, use a 2-D array to make it random

The parameters which can be passed as random are:

- `kxx` - Direct stiffness in the x direction.
- `cxx` - Direct damping in the x direction.
- `kyy` - Direct stiffness in the y direction.
- `cyy` - Direct damping in the y direction.
- `kxy` - Cross coupled stiffness in the x direction.
- `cxy` - Cross coupled damping in the x direction.
- `kyx` - Cross coupled stiffness in the y direction.
- `cyx` - Cross coupled damping in the y direction.

The selected parameters must be appended to `is_random` list as string.

You can return the random disk element created using the following command: `iter()`.

```

n: int
    Node which the bearing will be located in
kxx: float, 1-D array, 2-D array
    Direct stiffness in the x direction.
cxx: float, 1-D array, 2-D array
    Direct damping in the x direction.
kyy: float, 1-D array, 2-D array, optional
    Direct stiffness in the y direction.
    (defaults to kxx)
kxy: float, 1-D array, 2-D array, optional
    Cross coupled stiffness in the x direction.
    (defaults to 0)
kyx: float, 1-D array, 2-D array, optional
    Cross coupled stiffness in the y direction.
    (defaults to 0)
cyy: float, 1-D array, 2-D array, optional
    Direct damping in the y direction.
    (defaults to cxx)
cxy: float, 1-D array, 2-D array, optional
    Cross coupled damping in the x direction.
    (defaults to 0)
cyx: float, 1-D array, 2-D array, optional
    Cross coupled damping in the y direction.
    (defaults to 0)
frequency: array, optional
    Array with the frequencies (rad/s).
tag: str, optional
    A tag to name the element
    Default is None.
n_link: int, optional
    Node to which the bearing will connect. If None the bearing is
    connected to ground.
    Default is None.
scale_factor: float, optional
    The scale factor is used to scale the bearing drawing.
    Default is 1.
is_random : list
    List of the object attributes to become stochastic.
    Possibilities:
    ["kxx", "kxy", "kyx", "kyy", "cxx", "cxy", "cyx", "cyy"]


```

Bearing with random constant values for each coefficient:

```
# Building bearing elements and matching their coefficients.
var_size = 5
kxx = np.random.uniform(1e6, 2e6, var_size)
cxx = np.random.uniform(1e3, 2e3, var_size)
brg0 = srs.ST_BearingElement(n=0,
                              kxx=kxx,
                              cxx=cxx,
                              is_random=["kxx", "cxx"],
                              )
# set kxx and cxx again, if you want different coefficients for the next bearing
# it will get new random values.
# kxx = np.random.uniform(1e6, 2e6, var_size)
# cxx = np.random.uniform(1e6, 2e6, var_size)
brg1 = srs.ST_BearingElement(n=5,
                              kxx=kxx,
                              cxx=cxx,
                              is_random=["kxx", "cxx"],
                              )

list(iter(brg0))
```

```
[BearingElement(n=0, n_link=None,
  kxx=[1728863.4131801394], kxy=[0],
  kyx=[0], kyy=[1728863.4131801394],
  cxx=[1899.8252151986867], cxy=[0],
  cyx=[0], cyy=[1899.8252151986867],
  frequency=None, tag=None),
BearingElement(n=0, n_link=None,
  kxx=[1333325.5903531597], kxy=[0],
  kyx=[0], kyy=[1333325.5903531597],
  cxx=[1081.9618331789402], cxy=[0],
  cyx=[0], cyy=[1081.9618331789402],
  frequency=None, tag=None),
BearingElement(n=0, n_link=None,
  kxx=[1216836.7111646465], kxy=[0],
  kyx=[0], kyy=[1216836.7111646465],
  cxx=[1964.7711232304057], cxy=[0],
  cyx=[0], cyy=[1964.7711232304057],
  frequency=None, tag=None),
BearingElement(n=0, n_link=None,
  kxx=[1415232.3814475657], kxy=[0],
  kyx=[0], kyy=[1415232.3814475657],
  cxx=[1178.169733597559], cxy=[0],
  cyx=[0], cyy=[1178.169733597559],
  frequency=None, tag=None),
BearingElement(n=0, n_link=None,
  kxx=[1017272.8848591695], kxy=[0],
  kyx=[0], kyy=[1017272.8848591695],
  cxx=[1696.0658539914496], cxy=[0],
  cyx=[0], cyy=[1696.0658539914496],
  frequency=None, tag=None)]
```

The coefficients could be an array with different values for different rotations.  **stable** In this case you only have to give a parameter 'frequency' which is an array with the same size as the coefficients array.

To make it random, check the example below:

```
kxx = [np.random.uniform(1e6, 2e6, var_size),
        np.random.uniform(2.3e6, 3.3e6, var_size)]
cxx = [np.random.uniform(1e3, 2e3, var_size),
        np.random.uniform(2.1e3, 3.1e3, var_size)]
frequency = np.linspace(500, 800, len(kxx))
brg2 = srs.ST_BearingElement(n=1,
                             kxx=kxx,
                             cxx=cxx,
                             frequency=frequency,
                             is_random=["kxx", "cxx"],
                             )

list(iter(brg2))
```

```
[BearingElement(n=1, n_link=None,
  kxx=[1260527.92501607 2469613.38746566], kxy=[0, 0],
  kyx=[0, 0], kyy=[1260527.92501607 2469613.38746566],
  cxx=[1346.65672263 2424.22039485], cxy=[0, 0],
  cyx=[0, 0], cyy=[1346.65672263 2424.22039485],
  frequency=[500. 800.], tag=None),
BearingElement(n=1, n_link=None,
  kxx=[1614108.87276582 2711212.21947759], kxy=[0, 0],
  kyx=[0, 0], kyy=[1614108.87276582 2711212.21947759],
  cxx=[1248.87733963 2123.34218695], cxy=[0, 0],
  cyx=[0, 0], cyy=[1248.87733963 2123.34218695],
  frequency=[500. 800.], tag=None),
BearingElement(n=1, n_link=None,
  kxx=[1786280.07484952 2760164.64335148], kxy=[0, 0],
  kyx=[0, 0], kyy=[1786280.07484952 2760164.64335148],
  cxx=[1048.26691752 2981.73881762], cxy=[0, 0],
  cyx=[0, 0], cyy=[1048.26691752 2981.73881762],
  frequency=[500. 800.], tag=None),
BearingElement(n=1, n_link=None,
  kxx=[1558769.90415474 2846545.56728008], kxy=[0, 0],
  kyx=[0, 0], kyy=[1558769.90415474 2846545.56728008],
  cxx=[1961.22461187 2256.12094697], cxy=[0, 0],
  cyx=[0, 0], cyy=[1961.22461187 2256.12094697],
  frequency=[500. 800.], tag=None),
BearingElement(n=1, n_link=None,
  kxx=[1632005.09936429 2713570.273376 ], kxy=[0, 0],
  kyx=[0, 0], kyy=[1632005.09936429 2713570.273376 ],
  cxx=[1598.48476959 3048.93346808], cxy=[0, 0],
  cyx=[0, 0], cyy=[1598.48476959 3048.93346808],
  frequency=[500. 800.], tag=None)]
```

ST_Rotor

This class will create several instances of `rs.Rotor` class. The number of rotors depends on the amount of random elements instantiated and their respect

 stable ¹

To use this class, you only have to give all the already instantiated elements in a list format, as it follows.

```
shaft_elements : list
    List with the shaft elements
disk_elements : list
    List with the disk elements
bearing_seal_elements : list
    List with the bearing elements
point_mass_elements: list
    List with the point mass elements
tag : str
    A tag for the rotor

rotor1 = srs.ST_Rotor(
    shaft_elements,
    [disk0, disk1],
    [brg0, brg1],
)
```

Running the simulation

After you verify that everything is fine with the rotor, you should run the simulation and obtain results. To do that you only need to use the one of the `.run_()` methods available.

For now, STOCHASTIC ROSS has only a few stochastic analysis as shown below.

Obtaining results

These are the following stochastic analysis you can do with the program:

- `.run_campbell()` - Campbell Diagram
- `.run_freq_response()` - Frequency response
- `.run_unbalance_response()` - Unbalance response
- `.run_time_response()` - Time response

Plotting results

As it has been spoken before, STOCHASTIC ROSS presents results, not deterministic as ROSS does, but in the form of expectation (mean values) and percentiles (or confidence intervals). When plotting these analysis, it will always display the expectation and you are able to choose which percentile to plot.

To return a plot, you need to enter the command `.plot()` right before the command the run an analysis: `.run_something().plot()`

`.plot()` methods have two main arguments:

```
percentile : list, optional
    Sequence of percentiles to compute, which must be between
    0 and 100 inclusive.
conf_interval : list, optional
    Sequence of confidence intervals to compute, which must be between
    0 and 100 inclusive.
```

Plot interaction

You can click on the legend label to ommit an object from the graph.

Campbell Diagram

This function will calculate the damped natural frequencies for a speed range.

```
speed_range : array
    Array with the desired range of frequencies.
frequencies : int, optional
    Number of frequencies that will be calculated.
    Default is 6.
frequency_type : str, optional
    Choose between displaying results related to the undamped natural
    frequencies ("wn") or damped natural frequencies ("wd").
    The default is "wd".
```

To run the Campbell Diagram, use the command `.run_campbell()`

To plot the figure, use `.run_campbell().plot()`

Notice that there're two plots. You can plot both or one of them:

- damped natural frequency vs frequency;
 - use `.run_campbell().plot_nat_freq()`
- log dec vs frequency
 - use `.run_campbell().plot_log_dec()`



stable

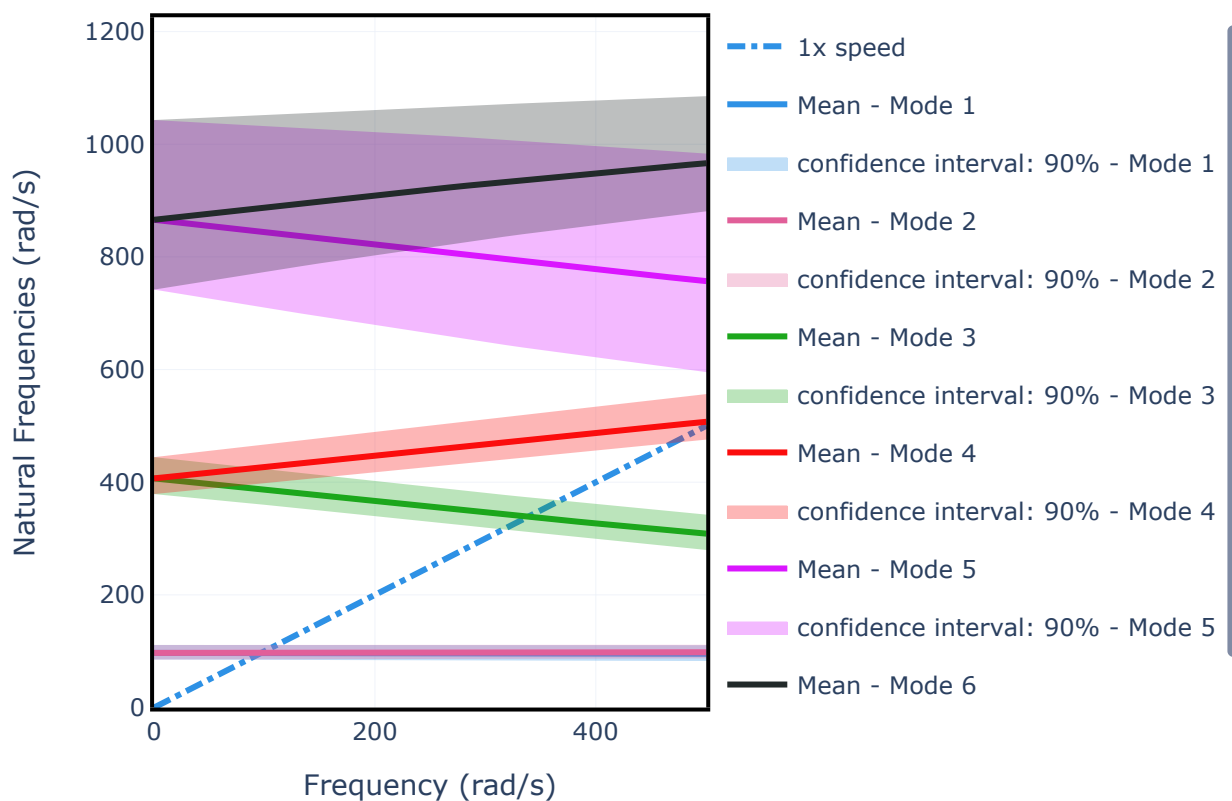
```

samples = 31
speed_range = np.linspace(0, 500, samples)

camp = rotor1.run_campbell(speed_range)

fig1 = camp.plot_nat_freq(conf_interval=[90])
fig1.show(renderer="notebook")

```



Frequency Response

```

speed_range : array
    Array with the desired range of frequencies.
inp : int
    Degree of freedom to be excited.
out : int
    Degree of freedom to be observed.
modes : list, optional
    Modes that will be used to calculate the frequency response
    (all modes will be used if a list is not given).

```



stable

We can put the frequency response of 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.

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

- $x \rightarrow$ index 0
- $y \rightarrow$ index 1
- $\alpha \rightarrow$ index 2
- $\beta \rightarrow$ index 3

Taking the rotor built as example, let's excite the node 3 (in the y direction) and observe the response on the node 2 (also in y direction):

$global_dof = dof_per_node * node_number + dof_index$

node 2, local dof y :

$$out = 4 * 2 + 1 = 9$$

node 3, local dof y :

$$inp = 4 * 3 + 1 = 13$$

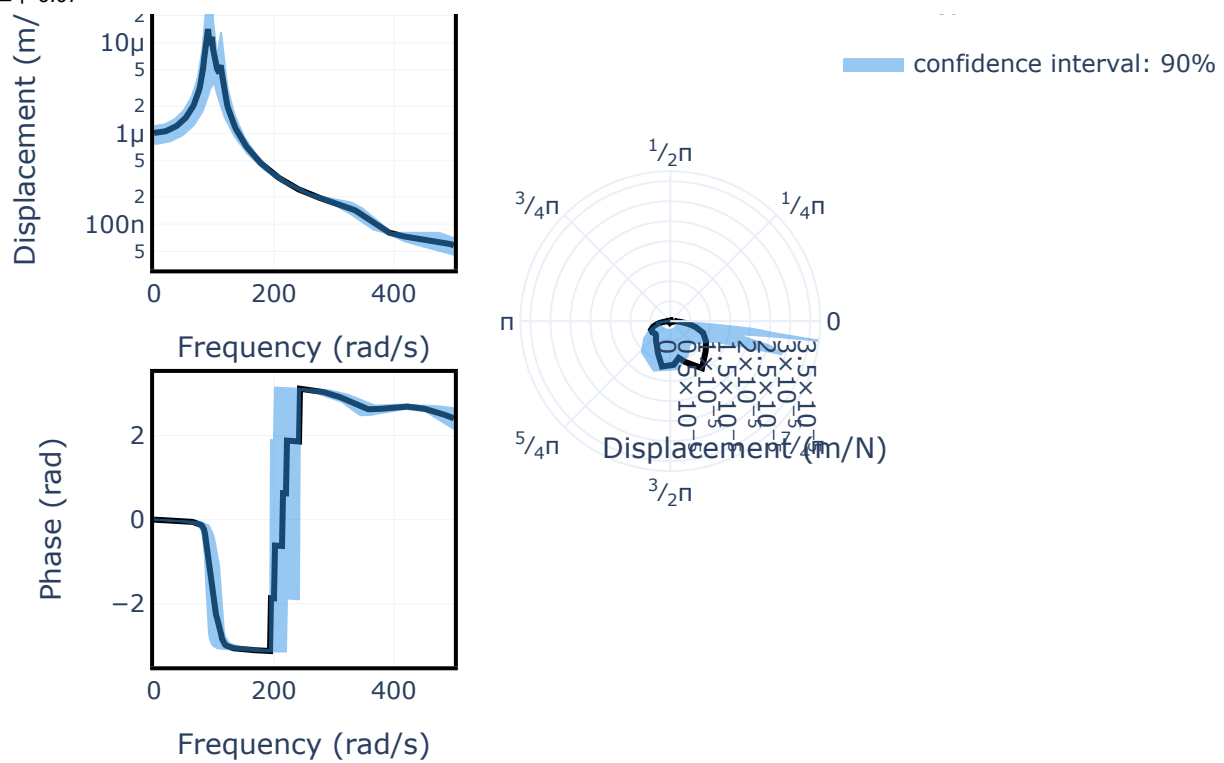
To run the Frequency Response, use the command `.run_freq_response()`

To plot the figure, use the command `run_freq_response().plot()`

```
speed_range = np.linspace(0, 500, 301)
inp = 13
out = 9
freqresp = rotor1.run_freq_response(inp, out, speed_range)
```

```
fig2 = freqresp.plot(conf_interval=[90], mag_kwargs=dict(yaxis=dict(type="log")))
fig2.show(renderer="notebook")
```





Unbalance Response

This method returns the unbalanced response for a mdof system given magnitude and phase of the unbalance, the node where it's applied and a frequency range.

```
node : list, int
    Node where the unbalance is applied.
magnitude : list, float
    Unbalance magnitude.
    If node is int, input a list to make it random.
    If node is list, input a list of lists to make it random.
phase : list, float
    Unbalance phase.
    If node is int, input a list to make it random.
    If node is list, input a list of lists to make it random.
frequency_range : list, float
    Array with the desired range of frequencies.
```

In this analysis, you can enter **magnitude** and **phase** as random variables.

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

To plot the figure, use the command `.run_unbalance_response().plot(probe)`

Where `probe` is a list of tuples that allows you to choose not only the `stable` , observe the response, but also the orientation.

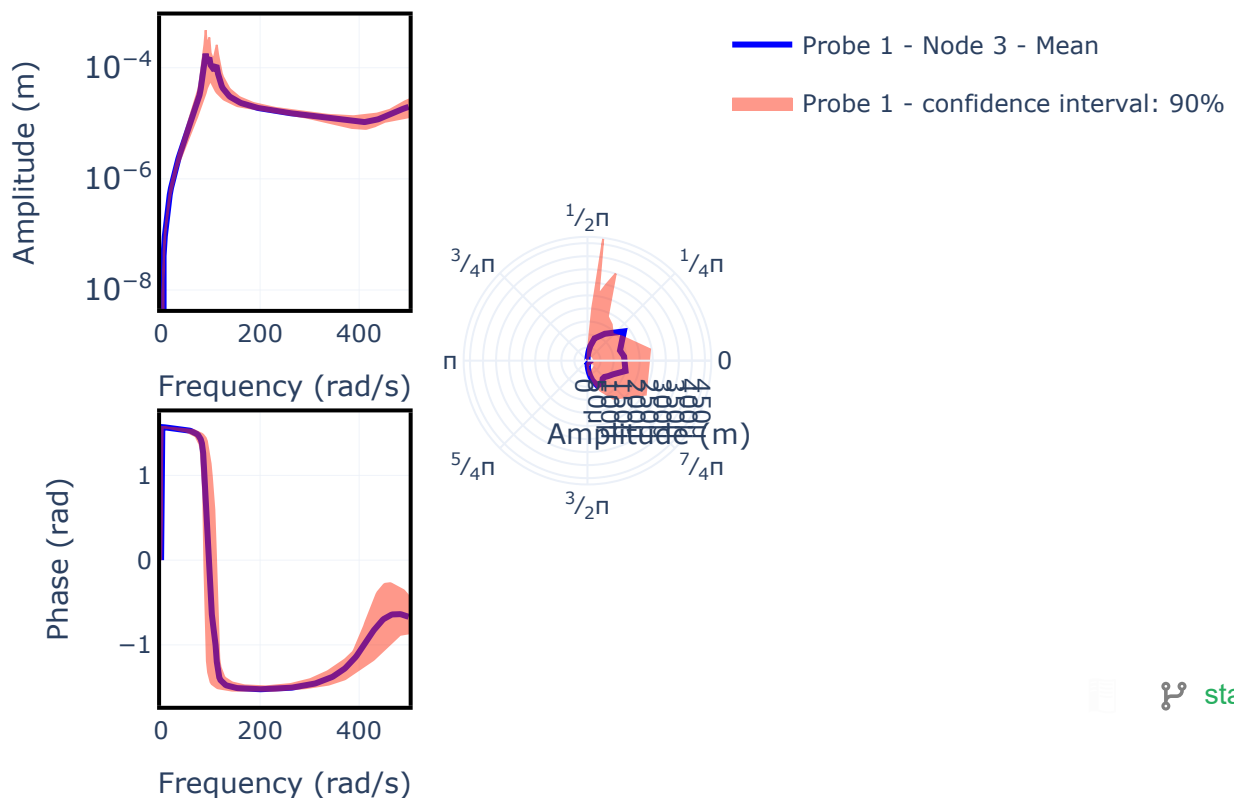
Probe orientation equals 0° refers to +X direction (DoFX), and probe orientation equals 90° (or $\frac{\pi}{2}rad$) refers to +Y direction (DoFY).

In this following example, we can obtain the response for a random unbalance(kg.m) with a uniform distribution and its respective phase in a selected node. Notice that it's possible to add multiple unbalances instantiating node, magnitude and phase as lists.

```
Unbalance: node = 3
           magnitude = np.random.uniform(0.001, 0.002, 10)
           phase = 0
```

```
freq_range = np.linspace(0, 500, 201)
n = 3
m = np.random.uniform(0.001, 0.002, 10)
p = 0.0
dof = 13
results = rotor1.run_unbalance_response(n, m, p, freq_range)
```

```
fig3 = results.plot(probe=[Probe(3, np.pi/2)], conf_interval=[90], mag_kwargs=dict(
fig3.show(renderer="notebook")
```



Time Response

This function will take a rotor object and plot its time response given a force and a time. The **force** and **ic** parameters can be passed as random.

This function takes the following parameters:

```
speed: float
    Rotor speed
force : 2-dimensional array, 3-dimensional array
    Force array (needs to have the same number of rows as time array).
    Each column corresponds to a dof and each row to a time step.
    Inputting a 3-dimensional array, the method considers the force as
    a random variable. The 3rd dimension must have the same size than
    ST_Rotor.rotor_list
time_range : 1-dimensional array
    Time array.
ic : 1-dimensional array, 2-dimensional array, optional
    The initial conditions on the state vector (zero by default).
    Inputting a 2-dimensional array, the method considers the
    initial condition as a random variable.
```

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

To plot the figure, use the command `.run_time_response().plot()`

In the following example, let's apply harmonic forces to the node 3 in both directions x and y . Also lets analyze the first 10 seconds from the response for a speed of 100.0 rad/s (~955.0 RPM).

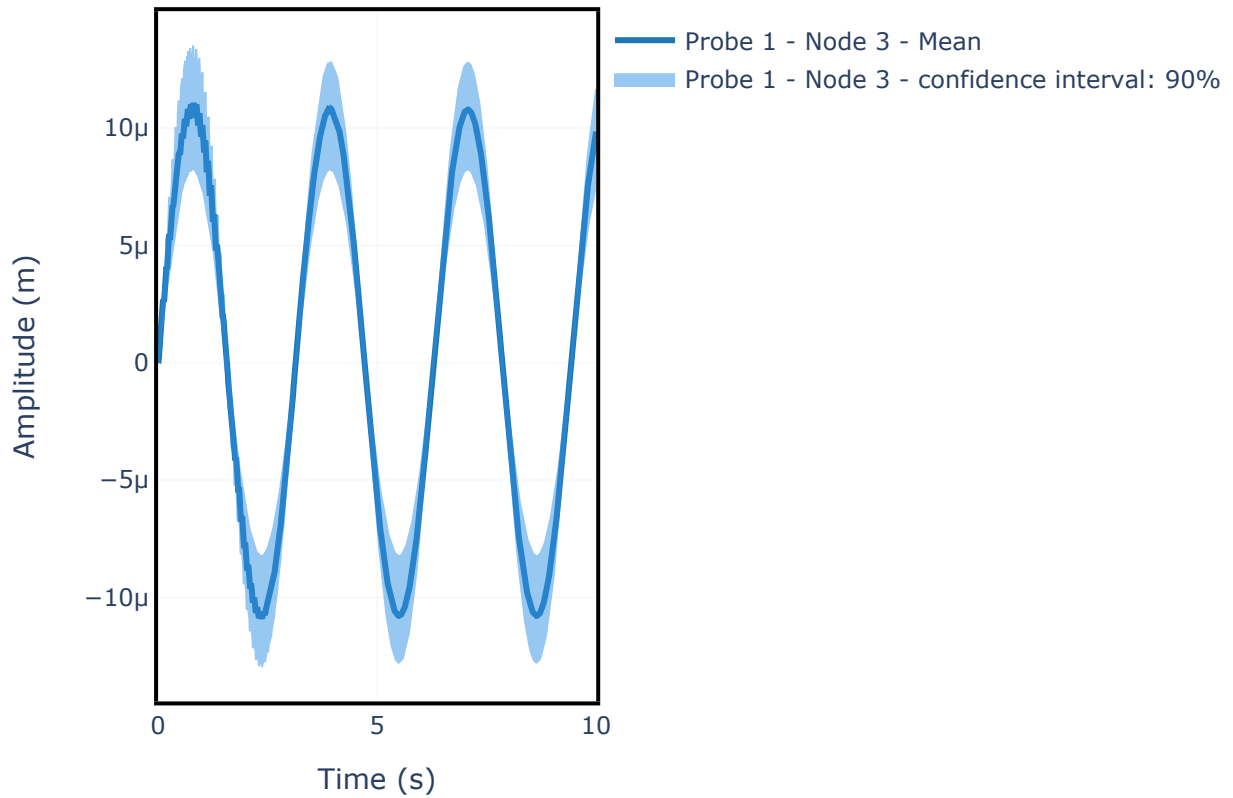
```
size = 1000
ndof = rotor1.ndof
node = 3 # node where the force is applied
dof = 9
speed = 100.0

t = np.linspace(0, 10, size)
F = np.zeros((size, ndof))
F[:, 4 * node] = 10 * np.cos(2 * t)
F[:, 4 * node + 1] = 10 * np.sin(2 * t)
results = rotor1.run_time_response(speed, F, t)
```

Plotting Time Response 1D, 2D and 3D

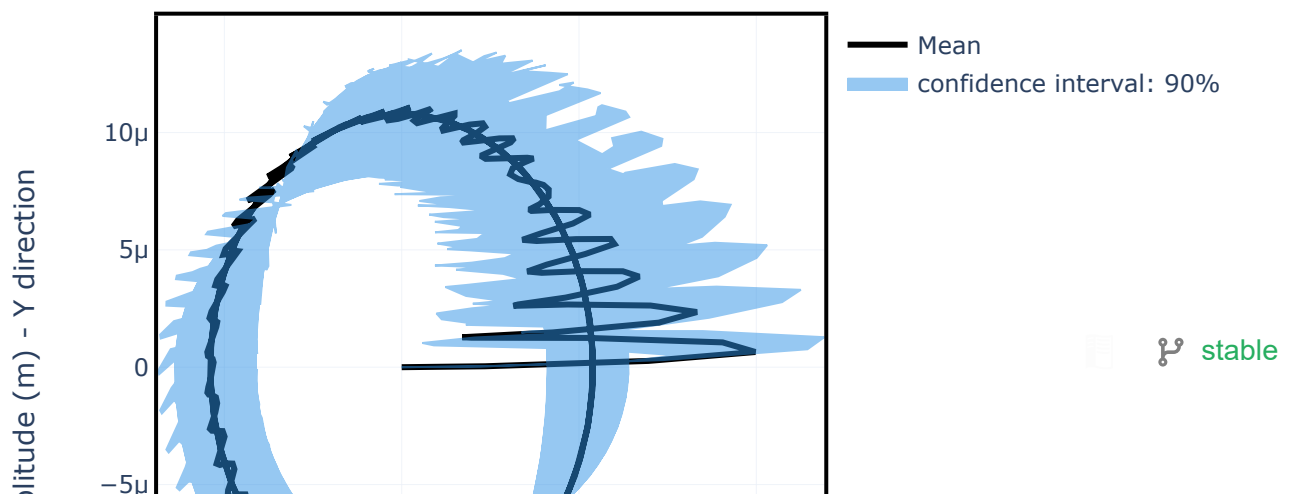
 [stable](#)

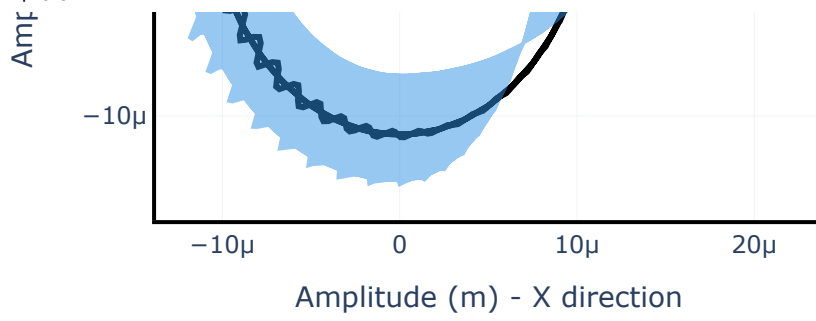
```
fig4 = results.plot_1d(probe=[Probe(3, np.pi / 2)], conf_interval=[90])
fig4.show(renderer="notebook")
```



```
fig5 = results.plot_2d(node=node, conf_interval=[90])
fig5.show(renderer="notebook")
```

Response for node 3





```
fig6 = results.plot_3d(conf_interval=[90])  
fig6.show(renderer="notebook")
```

— Mean

