UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Dinâmica de Rotores

Aluno:

```
Cristofer Antoni Souza Costa
```

Professor:

```
Aldemir Cavallini Jr.
```

Sumário

Introdução

O objetivo desse trabalho consiste em caracterizar o comportamento dinâmico de um sistema formado por um conjunto rotativo submetido a torção e flexão. O conjunto rotativo é composto por um eixo e disco (desbalanceado), suportado por um par de mancais. Deseja-se prever os seguintes aspectos ainda em etapa de projeto:

```
- Coportamento dinâmico;
- Velocidades críticas quanto ao comportamento vibracional lateral.
- Possíveis instabilidades.
- Resposta à aplicação de massa de desbalanço.
- Resposta a forças assíncronas.
```

Importando bibliotecas e efetuando configurações

In [1]:
```
'''
Importando módulos standard
'''
import os
from collections import namedtuple
'''
Bibliotecas de terceira parte
'''
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy.optimize import fsolve
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import cv2
'''
Importando a bilbioteca criada para a realização das análises dinâmicas
no contexto da disciplina
'''
import rotor_analysis as rd

Q_ = rd.Q_
```

Características do Conjunto Rotativo

Os elementos básicos que compõe o conjunto rotativo ora em estudo são: o disco; o eixo; os mancais; e os elementos de selagem. Visto que em aplicações reais, o desblancemento não pode ser completamente eliminado, massas de desbalanceamento também estão presente nessa modelagem.

Procedimento de modelagem:

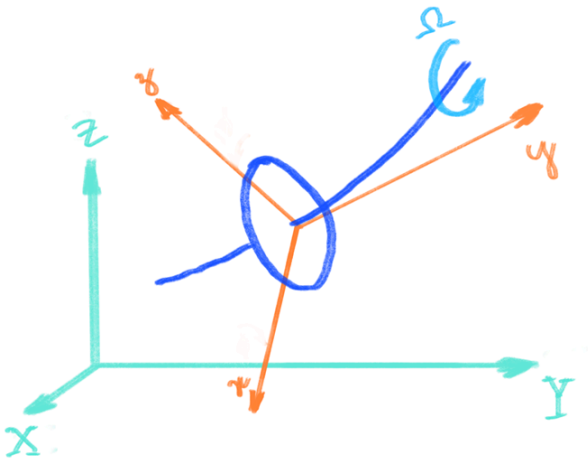
1. Determinar expressões para a energia cinética do disco, do eixo e das massas de dos balanços.

2. Determinar expressões para a energia de deformação, com intuito de caracterizar o eixo.

3. Forças serão aplicadas para caracterizar mancais e selagens para determinar o trabalho virtual.

Sistema de Referência

Buscando compreender os fenômenos relacionados com os movimentos descritos pelo conjunto rotativo, o sistema será representado matematicamente com o auxílio de dois sistemas de referência, um inercial $R_0(X-Y-Z)$ e outro móvel $R(x-y-z)$, conforme abaixo ilustrado.



O objetivo da utilização de sistemas móveis de referência na cinemática é facilitar a representação de movimentos complexos, subdividindo-os em vários movimentos mais simples que se somam para compor o movimento absoluto. Toda a representação matemática é baseada em cursores considerando uma origem predefinida, conforme descrito a seguir.

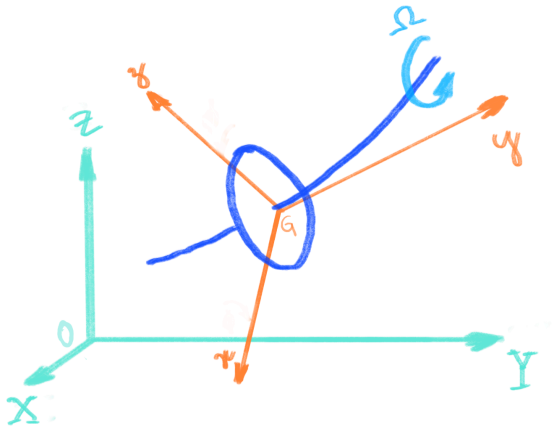
- **Sistema inercial**, composto pela origem O e cursores $\hat{\mathbf{X}}$, $\hat{\mathbf{Y}}$, $\hat{\mathbf{Z}}$, na forma

$$R_0 : \{O, \hat{\mathbf{X}}, \hat{\mathbf{Y}}, \hat{\mathbf{Z}}\}.$$

  Sua origem coincide com o centro geométrico dos mancais, e nesta base os vetores serão representados a partir dos cursores $\hat{\mathbf{X}}$, $\hat{\mathbf{Y}}$, $\hat{\mathbf{Z}}$. Por exemplo, o vetor posição pode ser expresso como

$$\mathbf{r}_{R_0} = X_0\hat{\mathbf{X}} + Y_0\hat{\mathbf{Y}} + Z_0\hat{\mathbf{Z}}.$$

  As grandezas escalares X_0, Y_0, e Z_0 indicam a amplitude deste vetor nas respectivas direções.



- **Sitema móvel**, composto pela origem G e cursores $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$, $\hat{\mathbf{z}}$, na forma

$$R : \{G, \hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}}\}.$$

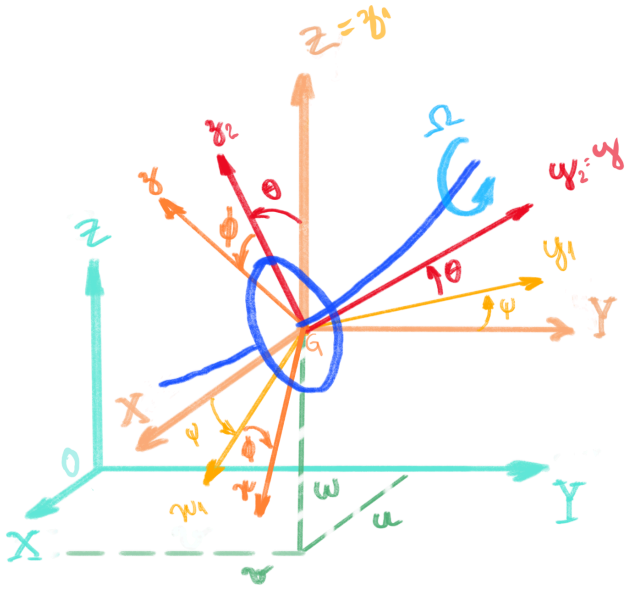
Sua origem coincide com o centro de massa do disco, e nesta base os vetores serão representados a partir de seus cursores, por exemplo, o vetor posição pode ser expresso como

$$\mathbf{r}_R = x\hat{\mathbf{x}} + y\hat{\mathbf{y}} + z\hat{\mathbf{z}}.$$

E, de forma análoga ao utilizado no sistema inercial, as grandezas escalares x, y e z indicam a amplitude deste vetor nas respectivas direções.

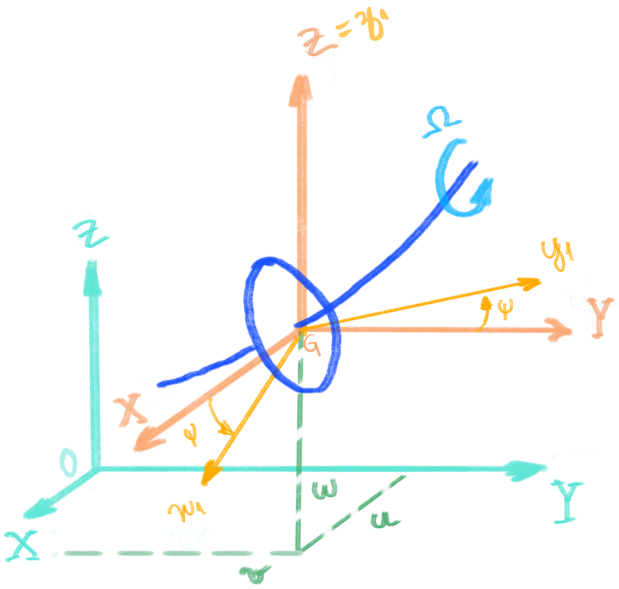
Transformação de Coordenadas

O fato de o sistema de coordenadas móvel girar, implica que os cursores do sistemas móvel $R(x-y-z)$ e o sistema inercial de referência $R_0(X-Y-Z)$ deixem de ser paralelos e passem a guardar uma relação entre os cursores do sistema inercial e do sistema móvel dependente dos angulos $\psi = \psi(t)$, $\theta = \theta(t)$ e $\phi = \phi(t)$ por exemplo, conforme figura abaixo. Esta relação entre sistemas é dada pela matriz de transformação de coordenadas, a qual leva a representação do vetor de uma base para outra base.



Para coincidir a orientação do sistema de referência móvel com o sistema inercial aplica-se um grupo de rotações tridimensionais a partir da origem. Essas rotações consecutivas não são comutativas, ou seja, a ordem que são aplicadas faz diferença no resultado final.

Assim sendo, podemos inicialmente aplicar uma rotação de angulo ψ em torno do eixo Z, de sentido positivo conforme a regra da mão direitra.



Dessa forma o vetor velocidade angular da base $R_1 : \{G, \hat{\mathbf{x}}_\mathbf{1}, \hat{\mathbf{y}}_\mathbf{1}, \hat{\mathbf{z}}_\mathbf{1}\}$ em relação à base inercial R_0 pode ser escrita como

$$\omega_{R_1/R_0} = \left\{ \begin{array}{c} 0 \\ 0 \\ \dot{\psi}(t) \end{array} \right\}_{R_1} \quad (1)$$

$$= \left\{ \begin{array}{c} 0 \\ 0 \\ \dot{\psi}(t) \end{array} \right\}_{R_0} \quad (2)$$

ou como

$$\omega_{R_1/R_0} = \dot{\psi}(t)\hat{\mathbf{z}}_\mathbf{1} = \dot{\psi}(t)\hat{\mathbf{Z}}.$$

Projetando-se os cursores da base móvel sobre a base inercial, chega-se a seguinte relação entre eles, escrita de forma matricial

$$\left\{ \begin{array}{c} x_1 \\ y_1 \\ z_1 \end{array} \right\}_{R_1} \quad (3) =$$

onde

$${}_{R_1}T_{R_0} = \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

é a matriz de transformação de coordenadas que, quando aplicada a um vetor representado no sistema R_0, gera um representação no sistema de referência intermediário $R_1 : \{G, \hat{\mathbf{x}}_\mathbf{1}, \hat{\mathbf{y}}_\mathbf{1}, \hat{\mathbf{z}}_\mathbf{1}\}$.

O código a seguir ilustra a transformação de um vetor representado na base R_0 para uma base R_1 rotacionada em 45°.

In [2]:
```
# Giro em torno de Z
psi = np.pi / 4
```

```
# Matriz de Transformação de Coordenadas R0_R1
transformation_R0_R1 = np.array(([np.cos(psi) , np.sin(psi), 0],
                                 [-np.sin(psi), np.cos(psi), 0],
                                 [0           , 0          , 1]))

# Verificando a implementação, definindo um vetor em R0
vector_R0 = np.array(([0, 1, 0]))

# Aplicando a matriz de transformação
vector_R1 = transformation_R0_R1 @ vector_R0
vector_R1
```

Out[2]:
```
array([0.70710678, 0.70710678, 0.        ])
```

Assim sendo, qualquer vetor descrito no sistema R_0 ou R_1 pode ser reescrito em outro sistema R_1 ou R_0 simplesmente quando os mesmos são multiplicados pela matriz de transformação de coordenadas ${}_{R_1}T_{R_0}$ ou ${}_{R_1}T_{R_0}^{-1}$.

Destaca-se aqui que as matrizes de rotação são conceituadas em Álgebra Linear como um caso especial das transformações lineares. São matrizes ortogonais e, logo, são quadradas e sua inversa é igual a sua transposta,

$$ {}_{R_1}T_{R_0}^{-1} = {}_{R_1}T_{R_0}^{T} = {}_{R_0}T_{R_1}, $$

e seu determinante é igual a um,

$$ \det\left({}_{R_1}T_{R_0}\right) = 1. $$

Assim sendo, temos

$$ \mathbf{r}_{R_1} = {}_{R_1}T_{R_0} \cdot \mathbf{r}_{R_0} $$
$$ \mathbf{r}_{R_0} = {}_{R_1}T_{R_0}^{T} \cdot \mathbf{r}_{R_1} $$

ou seja, a transformação linear representada pela matriz ${}_{R_1}T_{R_0}^{T}$ atuaria na transformação de coordenadas do sistema sistema móvel R_1 para o sistema inercial R_0.

Por exemplo, considerando-se um ângulo $\psi = \pi/4$, se tivermos o vetor

$$ \mathbf{a} = \left\{ \begin{array}{c} 0 \\ 1 \\ 0 \end{array} \right\}_{R_0} \quad (4) $$

representado na base R_0, aplicando-se a matriz de transformação ${}_{R_1}T_{R_0}$ neste vetor, podemos obteremos sua representação na base $R_1 : \{O, \hat{\mathbf{x}}_1, \hat{\mathbf{y}}_1, \hat{\mathbf{z}}_1\}$, na forma

$$ \mathbf{a} = \left\{ \begin{array}{c} \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \\ 0 \end{array} \right\}_{R_1} \quad (5) \quad . $$

O código a seguir ilustra as propriedades da matriz de transformação linear:

In [3]:
```
# o módulo do vector_R0 é igual ao vector_R1?
np.linalg.norm(vector_R0) == np.linalg.norm(vector_R1) == 1.0
```

Out[3]:
```
np.True_
```

In [4]:
```
# E voltando de R1 para R0, temos:
transformation_R0_R1.transpose() @ vector_R1
```

Out[4]:
```
array([4.26642159e-17, 1.00000000e+00, 0.00000000e+00])
```

In [5]:
```
# Determinante da transformação linear
np.linalg.det(transformation_R0_R1)
```

Out[5]: np.float64(1.0)

Aplicando rotações sucessivas, prosseguimos com o eixo X, temos a seguinte matriz de transformação:

$$\left\{ \begin{array}{c} x_2 \\ y_2 \\ z_2 \end{array} \right\}_{R_2} \quad (6) =$$

onde

$$_{R_2}T_{R_1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix}.$$

Dessa forma o vetor velocidade angular da base $R_2 : \{G, \hat{\mathbf{x}}_\mathbf{2}, \hat{\mathbf{y}}_\mathbf{2}, \hat{\mathbf{z}}_\mathbf{2}\}$ em relação à base R_1 pode ser escrita como

$$\omega_{R_2/R_1} = \left\{ \begin{array}{c} \dot{\theta}(t) \\ 0 \\ 0 \end{array} \right\}_{R_2} \quad (7)$$

$$= \left\{ \begin{array}{c} \dot{\theta}(t) \\ 0 \\ 0 \end{array} \right\}_{R_1} \quad (8)$$

ou como

$$\omega_{R_2/R_1} = \dot{\theta}(t)\hat{\mathbf{x}}_\mathbf{2} = \dot{\theta}(t)\hat{\mathbf{x}}_\mathbf{1}.$$

In [6]:
```
# Giro em torno de x1
theta = np.pi / 4

transformation_R1_R2 = np.array(([1, 0, 0],
                                 [0, np.cos(theta), np.sin(theta)],
                                 [0, -np.sin(theta), np.cos(theta)]))

# Verificando a implementação, considerando
# que a rotação em Z foi efeturada com o ângulo psi = 0.

# Aplicando a matriz de transformação
vector_R2 = transformation_R1_R2 @ vector_R0
vector_R2
```

Out[6]: array([0.   , 0.70710678, -0.70710678])

Para rotações positivas no eixo Y temos a seguinte matriz de transformação:

$$\left\{ \begin{array}{c} x \\ y \\ z \end{array} \right\}_{R} \quad (9) =$$

onde

$$_{R}T_{R_2} = \begin{bmatrix} \cos\phi & 0 & -\sin\phi \\ 0 & 1 & 0 \\ \sin\phi & 0 & \cos\phi \end{bmatrix}.$$

Dessa forma o vetor velocidade angular da base $R : \{G, \hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}}\}$ em relação à base R_2 pode ser escrita como

$$\omega_{R/R_2} = \left\{ \begin{array}{c} 0 \\ \dot{\phi}(t) \\ 0 \end{array} \right\}_{R_2} \quad (10)$$

$$= \begin{Bmatrix} 0 \\ \dot{\phi}(t) \\ 0 \end{Bmatrix}_{R_1} \quad (11)$$

ou como

$$\omega_{R/R_2} = \dot{\phi}(t)\hat{\mathbf{y}} = \dot{\phi}(t)\hat{\mathbf{y}}_{\mathbf{2}}.$$

In [7]:
```
## Giro em torno de Y
phi = np.pi / 4

transformation_R2_R = np.array(([np.cos(phi), 0,-np.sin(phi)],
                                [0, 1, 0],
                                [np.sin(phi), 0, np.cos(phi)]))

# Verificando a implementação, considerando
# que as rotações em Z e x1 foram efeturadas com
# os ângulos psi = theta = 0.

# Aplicando a matriz de transformação
vector_R = transformation_R2_R @ vector_R0
vector_R
```

Out[7]:
```
array([0., 1., 0.])
```

In [8]:
```
# Aplicando a matriz de transformação
vector_R = transformation_R2_R @ np.array([1,0,0])
vector_R
```

Out[8]:
```
array([0.70710678, 0.        , 0.70710678])
```

Vetor velocidade angular

Em seguida, procedemos computando a velocidade angular instantânea do frame R em relação ao frame inercial R_0 a partir da composição das três rotações, temos

$$\omega_{R/R_0} = \omega_{R_1/R_0} + \omega_{R_2/R_1} + \omega_{R/R_2},$$

ou seja,

$$\omega_{R/R_0} = \dot{\psi}(t)\hat{\mathbf{Z}} + \dot{\theta}(t)\hat{\mathbf{x}}_{\mathbf{1}} + \dot{\phi}(t)\hat{\mathbf{y}}.$$

Ajustando as bases por meio das matrizes de transformação para poder efetuar a soma vetorial temos que

$$\omega_{R/R_0} = \dot{\psi}(t)_R T_{R_2 R_2} T_{R_1} \hat{\mathbf{z}}_{\mathbf{1}} + \dot{\theta}(t)_R T_{R_2} \hat{\mathbf{x}}_{\mathbf{2}} + \dot{\phi}(t)\hat{\mathbf{y}}$$

que pode ser reescrito de forma matricial como representado no sistema de coordenadas R

$$\omega^R_{R/R_0} = \begin{bmatrix} \cos\phi & 0 & -\sin\phi \\ 0 & 1 & 0 \\ \sin\phi & 0 & \cos\phi \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix} \begin{Bmatrix} 0 \\ 0 \\ \dot{\psi}(t) \end{Bmatrix}_{R_1} \quad (12) \quad +$$

$$\begin{bmatrix} \cos\phi & 0 & -\sin\phi \\ 0 & 1 & 0 \\ \sin\phi & 0 & \cos\phi \end{bmatrix} \begin{Bmatrix} \dot{\theta}(t) \\ 0 \\ 0 \end{Bmatrix}_{R_2} \quad (13) \quad +$$

$$\begin{Bmatrix} 0 \\ \dot{\phi}(t) \\ 0 \end{Bmatrix}_{R} \quad (14)$$

resultando em

$$\omega^{R}_{R/R_0} = \begin{Bmatrix} -\dot{\psi}(t)\cos\theta\sin\phi + \dot{\theta}(t)\cos\phi \\ \dot{\phi}(t) + \dot{\psi}(t)\sin\theta \\ \dot{\psi}(t)\cos\theta\cos\phi + \dot{\theta}(t)\sin\phi \end{Bmatrix}_{R} \quad (15) \ .$$

Análise com dados numéricos

Tendo como base os dados numéricos do segundo capítulo do livro texto da disciplina, "Rotordynamics Prediction in Engineering, Second Edition" de Michel Lalanne and Guy Ferraris, foram replicadas as análises apresentadas no livro e expandidas.

Para a modelagem do sistema dinâmico foi criada uma bliblioteca em Python denominada `rotor_analysis` composta por três módulos, nomeados como `utilities.py`, `rotordynamics.py` e `results.py`, utilizando-se o paradigma de programação orientado a objeto (OOP - Object Oriented Paradigm). A biblioteca pint foi utilizada para manipular grandezas físicas de forma consistente bem como tornar a leitura do código mais autoexplicativa.

O módulo `utilities.py` fornece classes para modelar materiais e objetos geométricos como cilindros e discos. Ele calcula as diversas propriedades físicas, como volume, massa, área de superfície e momentos de inércia.

Classes:

```
Material: Representa um material com um nome e densidade.
Cylinder: Representa um cilindro oco e calcula suas propriedades.
Collection: Uma classe para manusear uma coleção de objetos (discos).
```

Já o módulo `rotordynamics.py` fornece classes para modelar um rotor girando com uma massa desbalanceada.

Classes:

```
Disc: Representa um disco, herdando de Cylinder.
Shaft: Representa um eixo, herdando de Cylinder.
Rotor: Representa um conjunto rotativo com um eixo e discos.
```

Diagrama de Campbell

O referido diagrama foi obtido modelando-se o o rotor com base na formulação apresentada no livro texto. As frequências naturais foram resolvidas por meio da solução do polinômio característico utilizando-se o módulo `scipy.optimize.fsolve` e a velocidades critícas foram obtidas por meio do método de numérico de Newton-Raphson implementado em `scipy.optimize.newton`, conforme segue:

In [9]:
```python
# Defining Material instances and properties
steel = rd.Material(name='Steel',
                    density=Q_(7800, 'kg/m^3'),
                    young_modulus=Q_(2e11,"Pa"))
print(steel)
```

```
Material(Steel)
Density: 7800 kg/m³
Young's Modulus: 200000000000.0 Pa
```

In [10]:
```python
# Shaft
L = Q_(0.4, 'm')

shaft = rd.Shaft(outer_radius=Q_(0.01, 'm'),
                 inner_radius=Q_(0.0, 'm'),
                 length=L,
                 material=steel)
print(shaft, shaft.material, sep="\n")
```

```
Shaft(outer=0.01 m, inner=0.0 m)
Material: Steel
Outer Radius: 0.01 m
Inner Radius: 0.0 m
Length: 0.4 m
Density: 7800 kg/m³
Volume: 1.2566×10⁻⁴ m³
Mass: 0.9802 kg
Cross-sectional Area: 3.1416×10⁻⁴ m²
Surface Area: 2.5761×10⁻² m²
Area Moment of Inertia, x: 7.8540×10⁻⁹ m⁴
Area Moment of Inertia, y: 7.8540×10⁻⁹ m⁴
Polar Moment of Inertia, z: 1.5708×10⁻⁸ m⁴
Mass Moment of Inertia, x: 1.3094×10⁻² kg·m²
Mass Moment of Inertia, y: 4.9009×10⁻⁵ kg·m²
Mass Moment of Inertia, z: 1.3094×10⁻² kg·m²
Material(Steel)
Density: 7800 kg/m³
Young's Modulus: 200000000000.0 Pa
```

In [11]:

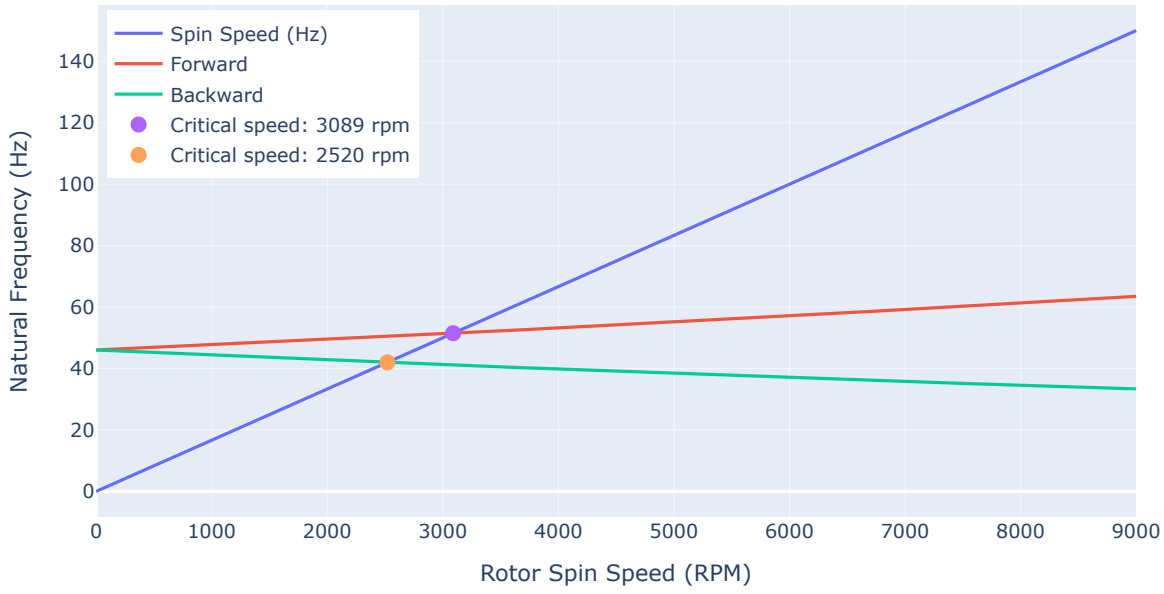
```python
# Disc
disc = rd.Disc(outer_radius=Q_(0.150, 'm'),
               inner_radius=Q_(0.010, 'm'),
               length=Q_(0.030, 'm'),
               material=steel,
               coordinate=L/3)

print(disc, disc.material, sep="\n")
```

```
Disc(outer=0.15 m, inner=0.01 m)
Material: Steel
Outer Radius: 0.15 m
Inner Radius: 0.01 m
Length: 0.03 m
Density: 7800 kg/m³
Volume: 2.1112×10⁻³ m³
Mass: 16.4670 kg
Cross-sectional Area: 7.0372×10⁻² m²
Surface Area: 1.7090×10⁻¹ m²
Area Moment of Inertia, x: 3.9760×10⁻⁴ m⁴
Area Moment of Inertia, y: 3.9760×10⁻⁴ m⁴
Polar Moment of Inertia, z: 7.9520×10⁻⁴ m⁴
Mass Moment of Inertia, x: 9.4273×10⁻² kg·m²
Mass Moment of Inertia, y: 1.8608×10⁻¹ kg·m²
Mass Moment of Inertia, z: 9.4273×10⁻² kg·m²
Material(Steel)
Density: 7800 kg/m³
Young's Modulus: 200000000000.0 Pa
```

In [12]:

```python
# Creating a Rotor instance
rotor_0 = rd.Rotor(
    shaft,
    disc
)
print(f"Natural Frequency at 0 rpm: {rotor_0.omega_0[1]:0.3f} Hz.")
data_0, fig_0 = rotor_0.plot_Campbell(return_data=True)
fig_0.show()
```

```
Natural Frequency at 0 rpm: 46.024 Hz.
```



Adicionando força axial ao rotor

No primeiro *approach* de modelagem aplicando-se ao conjunto rotativo uma força axial, foi aplicada a força de 10000 N tracionando o rotor descrito no livro texto. Esta força foi caracterizada como positiva neste estudo.

In [13]:
```
F0 = 1e4

# Create a Rotor instance with axial force
rotor_1 = rd.Rotor(
    shaft,
    disc,
    axial_force=Q_(+F0, 'N')
)
print(f"Natural Frequency at 0 rpm: {rotor_1.omega_0[1]:0.3f} Hz.")
data_1, _ = rotor_1.plot_Campbell(return_data=True)
```

```
Natural Frequency at 0 rpm: 48.341 Hz.
```

Já no segundo *approach* foi aplicada a força de 10000 N comprimindo o mesmo.

In [14]:
```
# Create a Rotor instance
rotor_2 = rd.Rotor(
    shaft,
    disc,
    axial_force=Q_(-F0, 'N')
)
print(f"Natural Frequency at 0 rpm: {rotor_2.omega_0[1]:0.3f} Hz.")
data_2, _ = rotor_2.plot_Campbell(return_data=True)
```

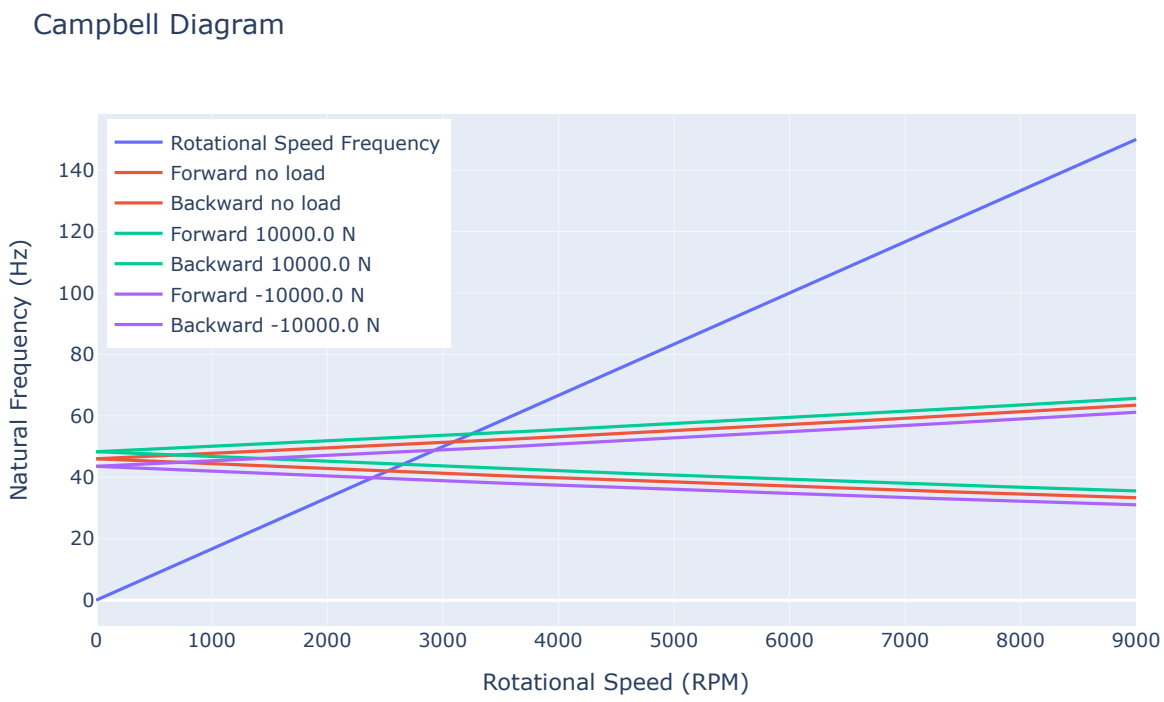
```
Natural Frequency at 0 rpm: 43.584 Hz.
```

Comparando os resultados

Como pode ser visto abaixo, analisando as três modelagens (sem força axial, força positiva e força negativa), observa-se a alteração das frequências naturais -- mesmo sem rotação. Com a aplicação da força de tração há um enrijecimento do sistema, elevando a frequência natural do mesmo. Já, o efeito oposto é observado quando a força negativa é aplicada.

O código abaixo gera uma representação gráfica do efeito dessas forças (positiva e negativa).

In [15]:
```
rd.campbell_diagram_axial_forces(
    data_0,
    data_1,
    data_2,
    F0
)
```




```
IntSlider(value=33, description='Sample')
Output()
```

Expandido o código: representação no espaço de estados e o cálculo das frequëncias naturais

Após a obtenção dos primeiros resultados, calculando-se as frequências naturais por meio do polinômio característico, foi implementado no código a obtenção das frequências naturais por meio do cálculo dos autovalores com auxílio da biblioteca `scipy.optimize.eig`.

Nesse sentido, foram calculados os autovalores da matrix `A` proveniente da formulação do problema utilizando-se a representação no espaço de estados.

In [16]:
```
'''
Computing the eigenvalues troughout matrix A (state-space representation matrix)
'''
eigenvalues, _ = np.linalg.eig(rotor_0.A(8640))

'''
    Processing the eigenvalues
    At this step the goal is to extract only positive eigenvalues and sort them by
    value to help to identify backward and forward related eigenvalues.
'''
roots = set()
for i in eigenvalues:
    roots.add(abs(i.imag / (2 * np.pi)))
roots = list(roots)
```

```
roots.sort(reverse=True)

# Data computed by means of Lalanne strategie
data = {
    "Forward" : data_0['Forward'],
    "Backward" : data_0['Backward']
}

# Create a DataFrame
df = pd.DataFrame(data,
                  index=data_0['Speed'])

# Display the DataFrame
print("Results solving the characteristc equation:")
print(df.loc[8640],'\n')
print("Results solving the state-space representation matrix:")
print('Forward: ', roots[0], '\tBackward: ', roots[1])
```

```
Results solving the characteristc equation:
Forward     62.703153
Backward    33.781547
Name: 8640.0, dtype: float64

Results solving the state-space representation matrix:
Forward:  62.70315311038258     Backward:  33.78154665241561
```

Resposta ao desbalanço

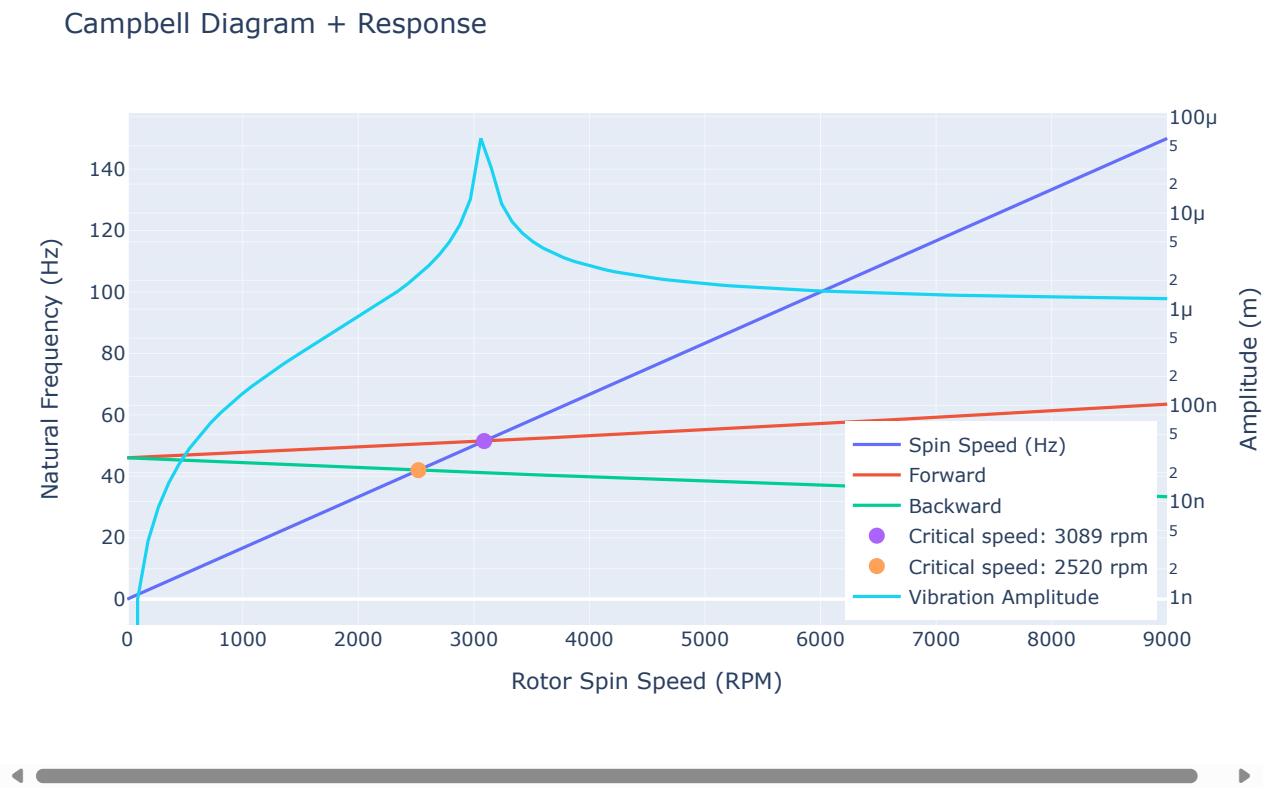
Nesta seção o equacionamento realizado considerou a aplicação de uma massa de desbalanço excitando o sitema conforme proposto no livro texto. Foram calculadas as amplitudes de vibração em função da velocidade de rotação.

In [17]:
```
# Defining some parameters
rotor = rd.Rotor(shaft, disc)
speed_range = np.linspace(0, 9000, 101)
Unbalance = namedtuple(
    'Unbalance',
    ['mass', 'radius']
)
'''
 Reference:
     Rotordynamics Prediction in Engineering, Second Edition, by Michel
     Lalanne and Guy Ferraris, Chapter 2: Monorotors, Subsection 2.1.6, page 17.
'''
unbal = Unbalance(
    mass=Q_(1e-4, 'kg'),
    radius=Q_(0.15, 'm')
)
```

In [18]:
```
def A1_unbal(speed, *args):
    '''Function to compute the displacemente aplitude A1 of the unbalanced rotor at
    some speed.

    Args:
        speed (float): Rotational speed in RPM.
        args: unbalance named tuple with mass and radius as properties

    Reference:
        Rotordynamics Prediction in Engineering, Second Edition, by Michel
        Lalanne and Guy Ferraris equation
    '''
    unbalance = args[0]
    k1, k2 = rotor.stiffness
    m = rotor.mass
    spin = speed / 60 * 2 * np.pi
    a = rotor.a
    div = (k1 - m * spin**2) * (k2 - m * spin**2) - a**2 * spin**4
    return (k2 - (m + a) * spin**2) * unbalance.mass.m * unbalance.radius.m * (spin)**2 * rotor.f(rotor.disc
```

In [19]:
```
# Computing the values
values = []
for speed in speed_range:
    values.append(abs(A1_unbal(speed, unbal)))
```

In [20]:
```
# Generating the Standard Campbell Diagram
campbell_fig = rotor.plot_Campbell()

# Call the function to add the secondary y-axis
rd.add_secondary_yaxis(campbell_fig, values)

# Show the updatedvfigure
campbell_fig.show()
```



A seguir é calculado o valor da amplitude de vibração quando a rotação tende a infinto.

In [21]:
```
'''
 Reference:
     Rotordynamics Prediction in Engineering, Second Edition, by Michel
     Lalanne and Guy Ferraris equation 2.134
'''
unbal.mass.m * rotor.discs[0].outer_radius.m * rotor.f(rotor.discs[0].coordinate.m) \
/ (rotor.a - rotor.mass)
```

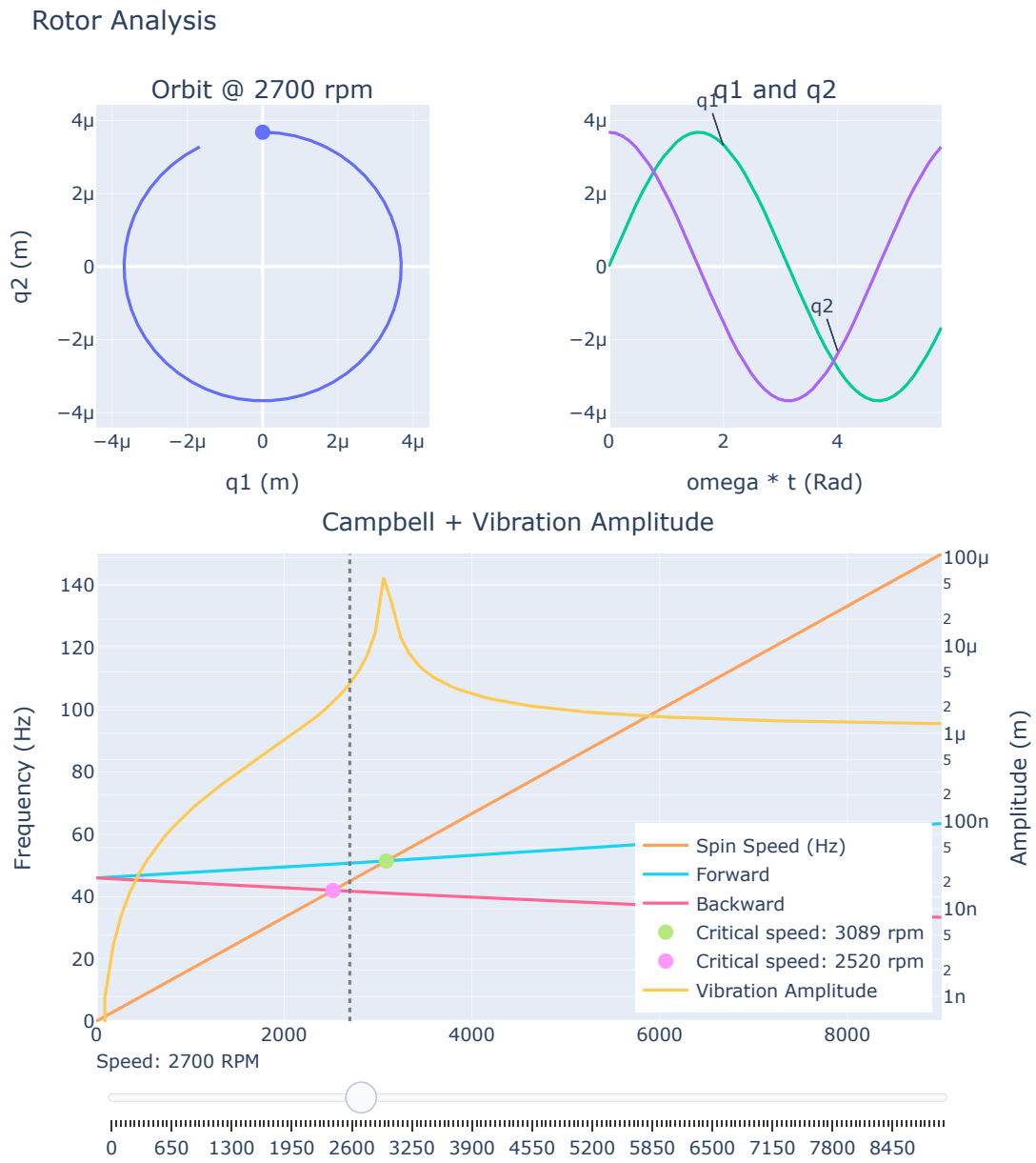
Out[21]: np.float64(-1.1371292754494908e-06)

Órbita

Após o cômputo da amplitude da resposta ao desbalanço, plota-se os deslocamentos $q1$ e $q2$ formando-se assim a orbita descrita pelo centro do eixo. No caso abaixo temos a órbita simulando o encoder posicionado em q2 (no topo do eixo), e o desbalanço posicionado na chaveta.

Combinando os gráficos anteriormente gerados, temos:

In [22]:
```
orbit_plot = rd.interactive_orbit_campbell(campbell_fig,
                                           A1_unbal,
                                           A1_unbal,
                                           unbal,
                                           initial_speed=2700,
                                           max_amplitude=2e-05)
orbit_plot.show()
```



Resposta à aplicação de uma força assíncrona

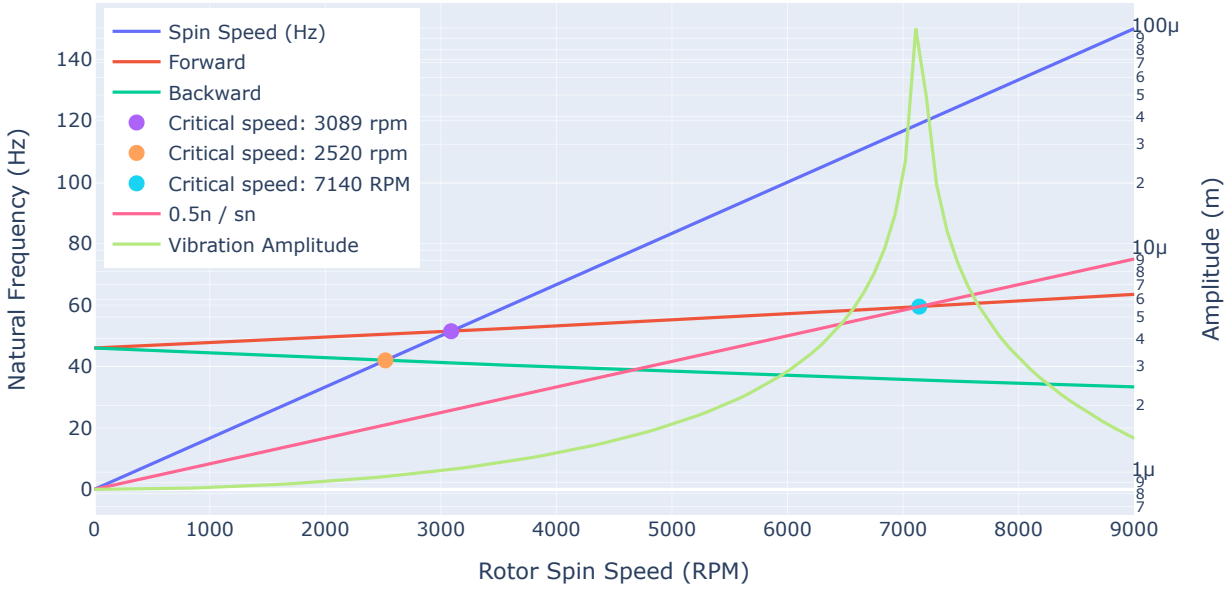
In [23]:
```
def A1_async(speed, *args):
    '''Compute the displacemente aplitude of the rotor with an asynchronous
    excitation.

    Args:
        speed (function): Rotational speed in RPM.
        F0 (float): Asynchronous force magnitude
        s (float): A multiplier thats desynchronize the excitation

    Remarks:
```

```
        len(args) shall be equals to 2
    '''
    F0, s = args
    k1, k2 = rotor.stiffness
    m = rotor.mass
    spin = speed / 60 * 2 * np.pi
    a = rotor.a
    div = s**2 * (s**2 * m**2 - a**2) * spin**4 - m * (k1 + k2) * s**2 * spin**2 + k1 * k2
    return (k2 - (m * s**2 + a * s) * spin**2) * F0 / div
```

In [24]:
```
values = []
s = 0.5
for speed in speed_range:
    values.append(abs(A1_async(speed, 1, s)))
```

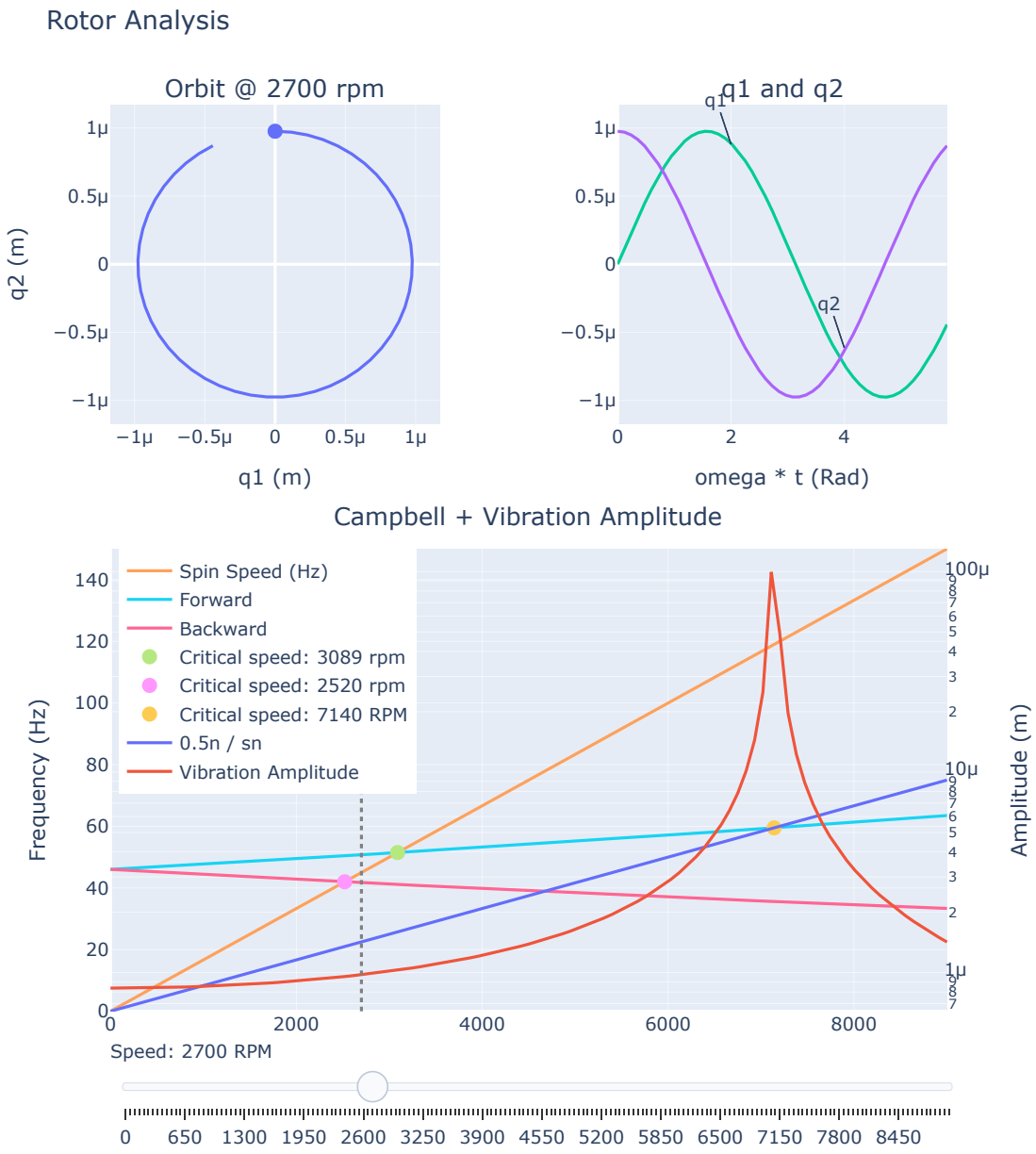
In [25]:
```
# Generating the Standard Campbell Diagram
campbell_fig = rotor.plot_Campbell()
data = campbell_fig.to_dict()

new_critical_speed = np.sqrt(rotor.stiffness[0]/(s*(rotor.mass*s - rotor.a))) * 60 / (2 * np.pi)

point_critical_speed = go.Scatter(
    x=[new_critical_speed],
    y=[new_critical_speed / 120],
    mode="markers",
    marker={"size": 10},
    hovertext=f"{new_critical_speed:.0f} RPM",
    name=f"Critical speed: {new_critical_speed:.0f} RPM"
)
trace05x = go.Scatter(
    x=data['data'][0]['x'],
    y=data['data'][0]['x'] * s / 60,  # Converting RPM to Hertz and then applying the muktplier s
    mode="lines",
    name="0.5n / sn",
)

campbell_fig.add_trace(point_critical_speed)
campbell_fig.add_trace(trace05x)

# Call the function to add the secondary y-axis
rd.add_secondary_yaxis(campbell_fig, values)

#adjusting the legend
campbell_fig.update_layout(
    legend=dict(yanchor="top", y=0.99, xanchor="left", x=0.01),
)
# Show the updatedvfigure
campbell_fig.show()
```

Campbell Diagram + Response



Órbita

In [26]:
```
orbit_plot = rd.interactive_orbit_campbell_async(
    campbell_fig,
    A1_async,
    A1_async,
    1,
    0.5,
    initial_speed=2700
)
orbit_plot.show()
```



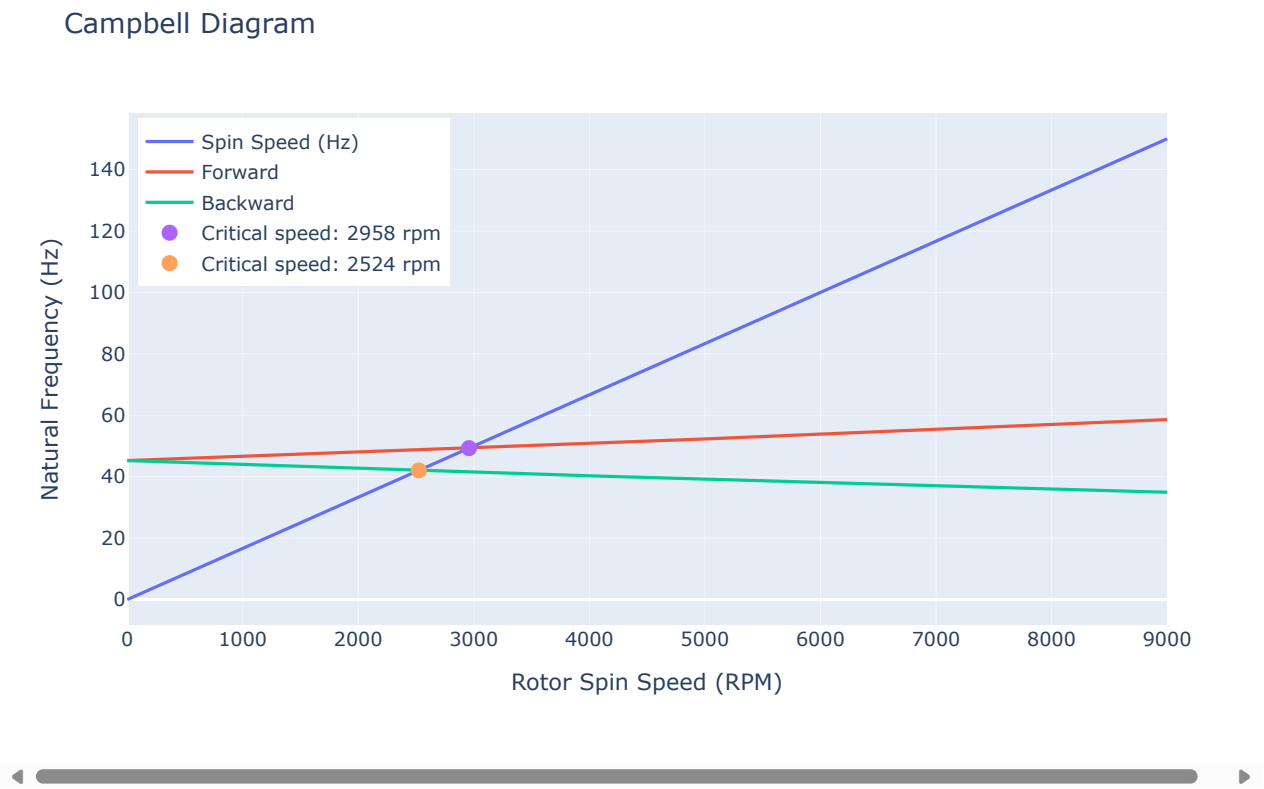
Expandindo o código para considerar dois discos

In [27]:
```
# Disc
L_h = Q_(0.030 / 2, 'm')
disc0 = rd.Disc(outer_radius=Q_(0.150, 'm'),
                inner_radius=Q_(0.010, 'm'),
                length=L_h,
                material=steel,
                coordinate=L/3)

disc1 = rd.Disc(outer_radius=Q_(0.150, 'm'),
                inner_radius=Q_(0.010, 'm'),
                length=L_h,
                material=steel,
                coordinate=L/3 + L_h)

# print(disc, disc.material, sep="\n")
```

In [28]:
```
# Creating a Rotor isntance
rotor1 = rd.Rotor(shaft, disc0, disc1, max_speed = Q_(9000, 'rpm'))
```

```
print(f"Natural Frequency at 0 rpm: {rotor1.omega_0[0]:0.3f} Hz.")
rotor1.plot_Campbell()
```

```
Natural Frequency at 0 rpm: 45.255 Hz.
```



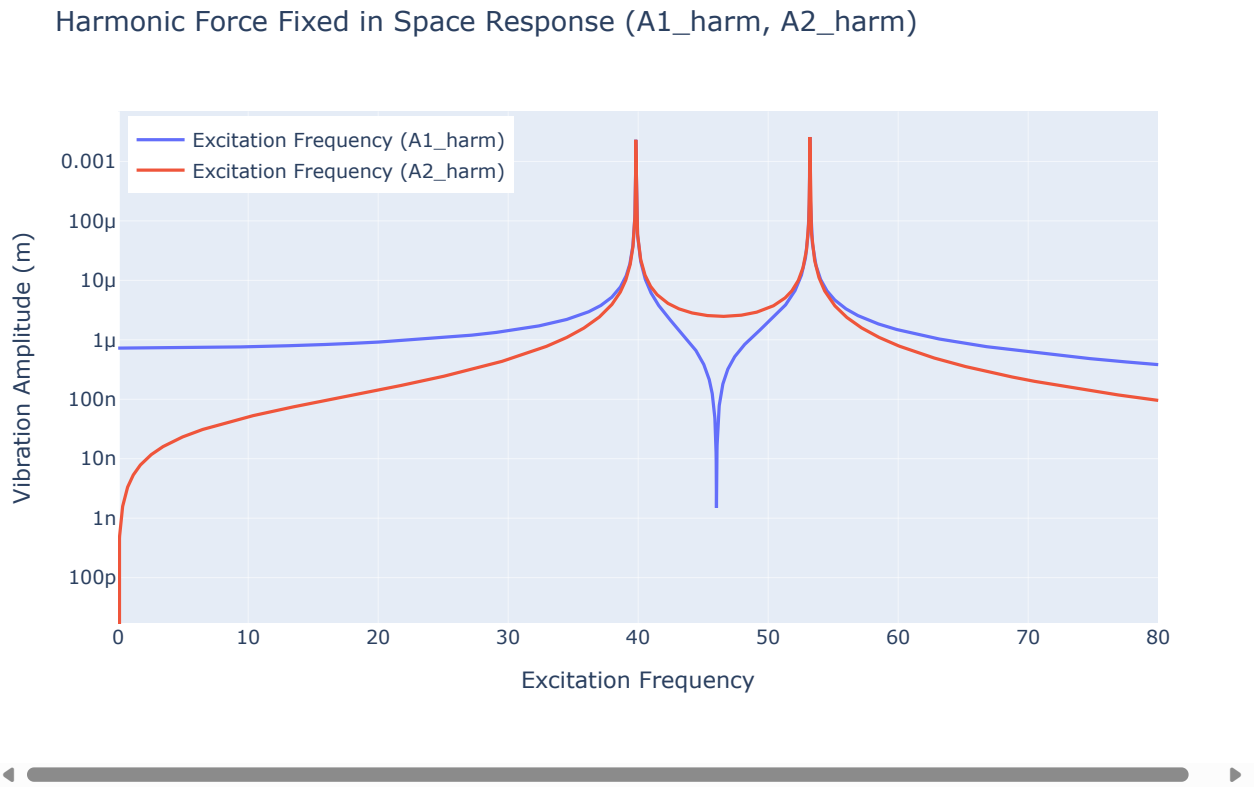
Resposta à excitação hamônica fixa no espaço

In [29]:

```
def A1_harm(speed, *args):
    '''Compute the displacemente aplitude of the rotor with an harmonic
    excitation fixed in space

    Args:
        speed (function): Rotational speed in RPM.
        F0 (float): Asynchronous force magnitude
        f (float): Excitation frequency

    Remarks:
        len(args) shall be equals to 2.
    '''
    F0, f = args
    omega = 2 * np.pi * f
    k1, k2 = rotor.stiffness
    m = rotor.mass
    spin = speed / 60 * 2 * np.pi
    a = rotor.a
    div = (k1 - m * omega**2) * (k2 - m * omega**2) - a**2 * spin**2 * omega**2

    return F0 * rotor.f(L.m/3*2) * (k2 -  m * omega**2 ) / div


def A2_harm(speed, *args):
    '''Compute the displacemente aplitude of the rotor with an harmonic
    excitation fixed in space

    Args:
        speed (function): Rotational speed in RPM.
        F0 (float): Asynchronous force magnitude
        f (float): Excitation frequency
```

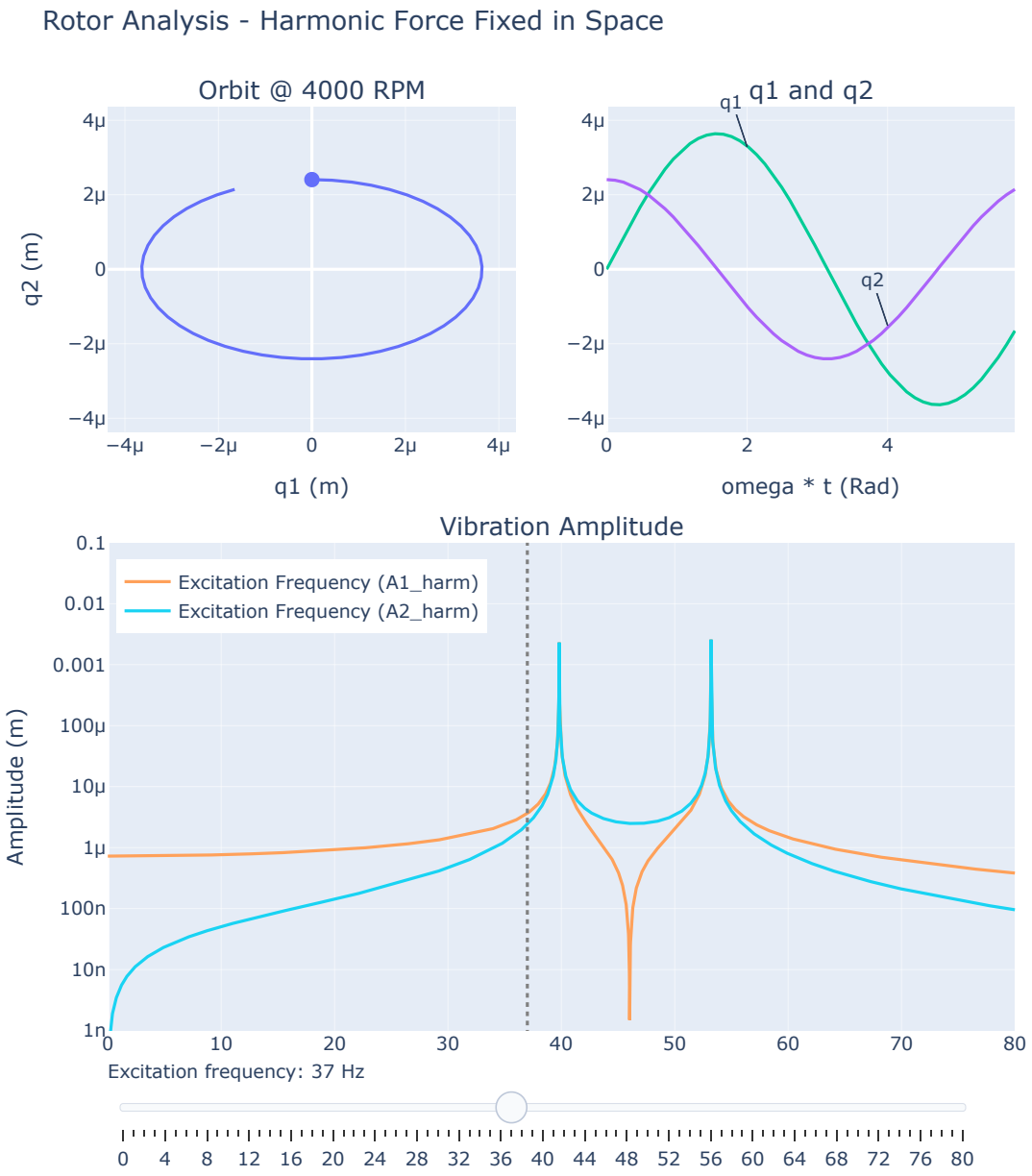
```
    Remarks:
        len(args) shall be equals to 2.
    '''
    F0, f = args
    omega = 2 * np.pi * f
    k1, k2 = rotor.stiffness
    m = rotor.mass
    spin = speed / 60 * 2 * np.pi
    a = rotor.a
    div = (k1 - m * omega**2) * (k2 - m * omega**2) - a**2 * spin**2 * omega**2
    return --a * spin * omega * F0 * rotor.f(L.m/3*2) / div
```

In [30]:
```
rd.plot_vibration_amplitude(A1_harm, A2_harm)
```



Órbita excitação hamônica fixa no espaço

In [31]:
```
response = rd.plot_vibration_amplitude(A1_harm, A2_harm)

rd.interactive_orbit_fixed_speed(
    response,
    A1_harm,
    A2_harm,
    4000,
    1,
    37
)
```



Salvando frames para fazer um video

In [32]:
```
rd.save_orbit_frames(
    response,
    A1_harm,
    A2_harm,
    4000,
    1,
    37,
    output_dir='frames'
)
```

```
Frames saved in directory: frames
```

In [33]:
```
# Create a video from the saved images
rd.create_video_from_frames('frames', 'fixed_in_space.mp4', 20, 'h264')
```

Ploting the orbit from the state space representation model

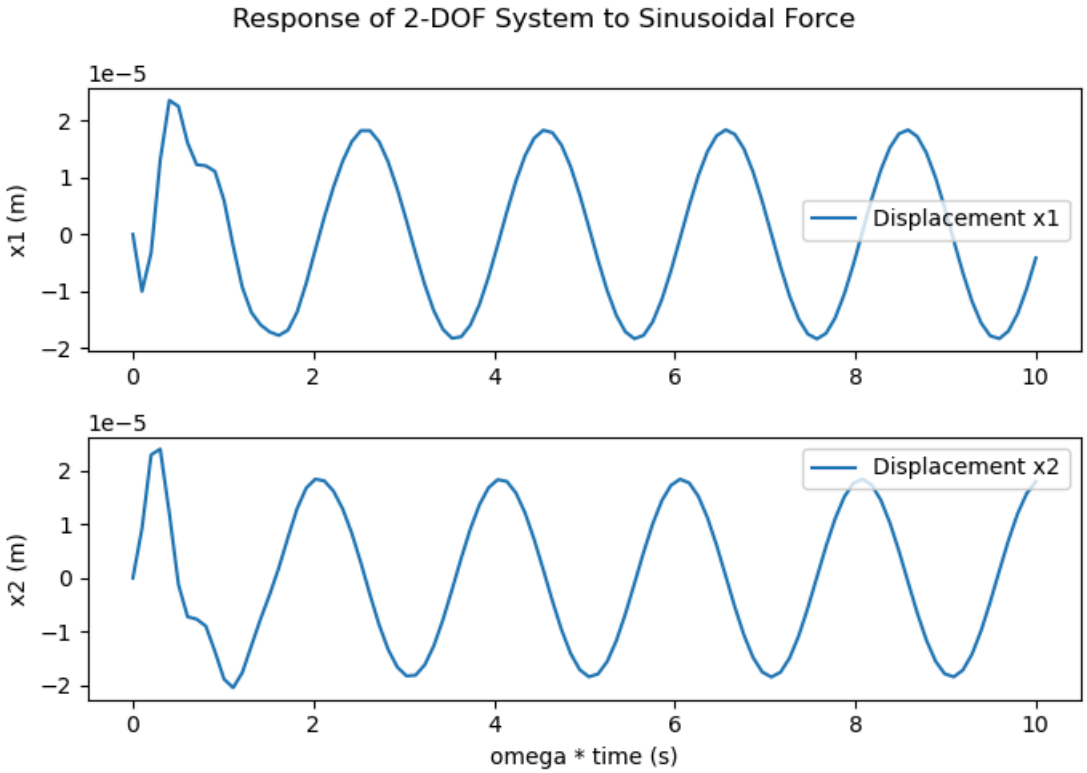
In [34]:
```
class Damping():
    def __init__(self, cxx:float=None, czz:float=None, beta:float=0):
```

```
        self.cxx = cxx
        self.czz = czz
        self.beta = beta
```

In [35]:

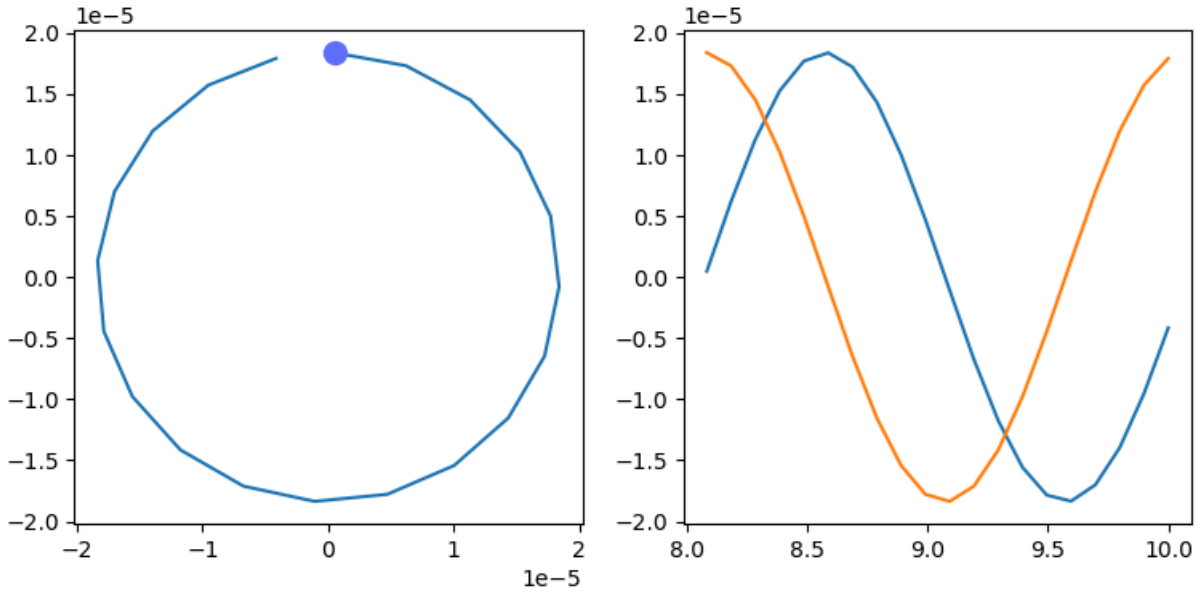
```
from scipy.integrate import odeint

# Defining some parameters
cxx = 50
czz = 50
damping = Damping(cxx, czz, 1)

# Creating a new rotor
rotor = rd.Rotor(
    shaft,
    disc,
    damping=damping
)

speed = 3000
spin = speed / 60 * 2 * np.pi

# Unbalance Force
F = unbal.mass.m * unbal.radius.m * (spin)**2 * rotor.f(rotor.discs[0].coordinate.m)

# Initial conditions
z0 = np.zeros(4)

amostras = 100
signal_length = 10

# Time vector
t = np.linspace(0, signal_length, amostras)

def forcing_function(t, omega):
    """
    Define the sinusoidal forcing function.

    Parameters
    ----------
    t : float
        Time variable.
    omega : float
        Force frequency in radians per second.

    Returns
    -------
    np.ndarray
        Array containing the sine and cosine components of the force.
    """
    return np.array([np.sin(omega * t), np.cos(omega * t)])

def state_space_model(z, t, A, B, F, omega):
    """
    State-space model of the system including the sinusoidal force.

    Parameters
    ----------
    z : np.ndarray
        State vector containing displacements and velocities.
    t : float
        Time variable.
    A : np.ndarray
        System matrix.
    B : np.ndarray
        Input matrix.
    F : float
        Magnitude of the unbalance force.
    omega : float
        Force frequency in radians per second.
```

```
    Returns
    -------
    np.ndarray
        Derivative of the state vector.
    """
    u = F * forcing_function(t, omega)
    dzdt = A @ z + B @ u
    return dzdt

# Solve the differential equation
solution = odeint(state_space_model, z0, t, args=(rotor.A(speed), rotor.B, F, spin))

# Extract displacements and velocities
x = solution[:, :2]
v = solution[:, 2:]

# Plot the results
plt.figure(figsize=[7, 5])
for i in range(2):
    plt.subplot(2, 1, i+1)
    plt.plot(t[:], x[:, i], label=f'Displacement x{i+1}')
    plt.ylabel(f'x{i+1} (m)')
    plt.legend()
    if i == 1:
        plt.xlabel('omega * time (s)')

plt.suptitle('Response of 2-DOF System to Sinusoidal Force')
plt.tight_layout()
plt.show()
```



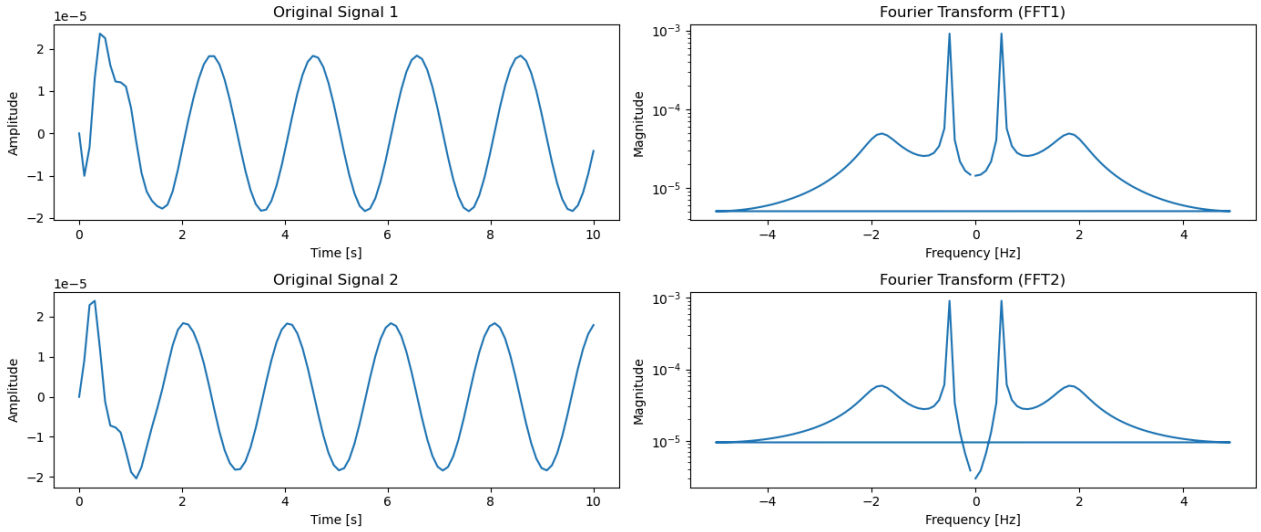
In [36]:
```
# Create the figure and axis objects
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(9, 4))
corte = int(amostras * 0.8)
# Plot orbit data
ax1.plot(x[corte:, 0], x[corte:, 1], label='Orbit')
ax1.plot(x[corte:, 0][0], x[corte:, 1][0], 'o', markersize=10, color='#636EFA')

# Plot sine data
ax2.plot(t[corte:], x[corte:, 0], label='q1')
ax2.plot(t[corte:],  x[corte:, 1], label='q2')
```

Out[36]: [<matplotlib.lines.Line2D at 0x1da01e0c650>]



In [37]:
```
# Compute the Fourier Transform using scipy.fft
rd.FFT(x[:, 0], x[:, 1], signal_length)
```

```
Samples:                100
Duration:               10 s
Sampling Frequency:     10.0 Hz
```



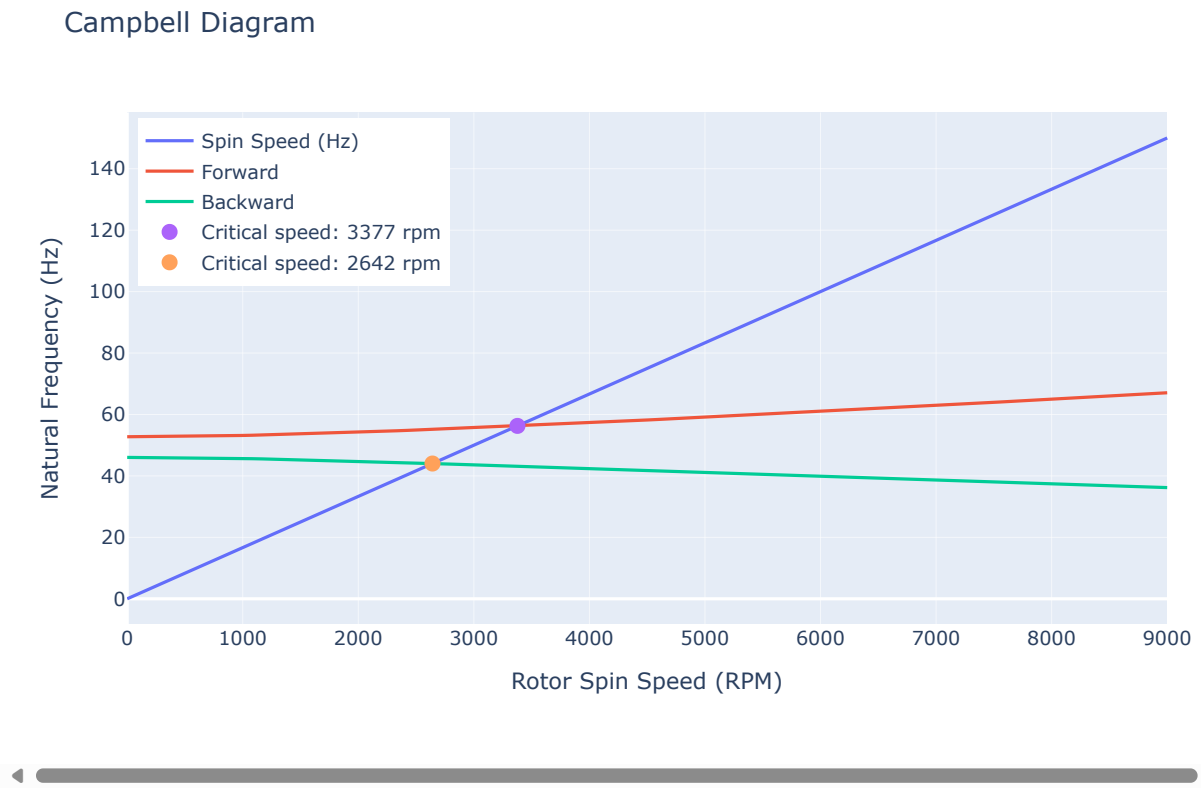
Rotor Assimétrico

In [38]:
```
import pint
```

In [39]:
```
class Spring():
    def __init__(self, coordinate:pint.Quantity, kxx:float=None, kzz:float=None):
        self.coordinate = coordinate
        if kxx:
            self.kxx = kxx
        else:
            self.kxx = Q_(0, "N/m")
        if kzz:
            self.kzz = kzz
        else:
            self.kzz = Q_(0, "N/m")
```

In [40]:
```
'''
 Reference:
     Rotordynamics Prediction in Engineering, Second Edition, by Michel
     Lalanne and Guy Ferraris, Chapter 2: Monorotors, Subsection 2.1.6, page 17.
'''
# Defining some parameters
L2 = 2 * L / 3
kzz = Q_(5e5, "N/m")
spring = Spring(L2, kzz=kzz)

rotor = rd.Rotor(shaft, disc, spring=spring)

speed_range = np.linspace(0, 9000, 101)
print('Frequências Naturais @ 0 RPM: ', round(rotor.omega_0[0], 2), ',', round(rotor.omega_0[1], 2))
rotor.plot_Campbell()
```

```
Frequências Naturais @ 0 RPM:  52.75 , 46.02
```



Resposta a solicitação

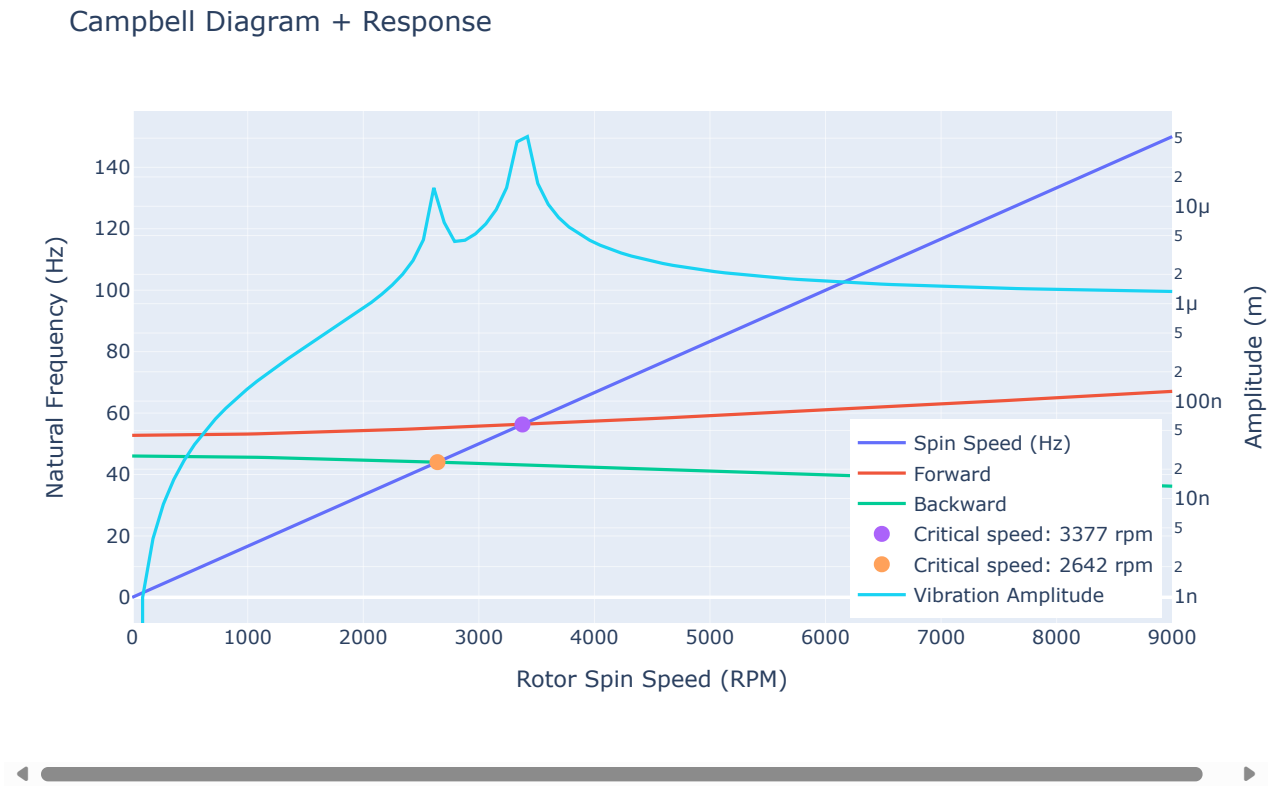
In [41]:
```
def A1_non_sym(speed, *args):
    '''Function to compute the displacemente aplitude A1 of the unbalanced rotor at
    some speed.

    Args:
        speed (float): Rotational speed in RPM.
        args: unbalance named tuple with mass and radius as properties

    Reference:
        Rotordynamics Prediction in Engineering, Second Edition, by Michel
        Lalanne and Guy Ferraris equation
    '''
    unbalance = args[0]
    k1, k2 = rotor.stiffness
    m = rotor.mass
    spin = speed / 60 * 2 * np.pi
    a = rotor.a
    div = (k1 - m * spin**2) * (k2 - m * spin**2) - a**2 * spin**4
```

```
    return (k2 - (m + a) * spin**2) * unbalance.mass.m * unbalance.radius.m * (spin)**2 * rotor.f(rotor.disc


def A2_non_sym(speed, *args):
    '''Function to compute the displacemente aplitude A2 of the unbalanced rotor at
    some speed.

    Args:
        speed (float): Rotational speed in RPM.
        args: unbalance named tuple with mass and radius as properties

    Reference:
        Rotordynamics Prediction in Engineering, Second Edition, by Michel
        Lalanne and Guy Ferraris equation
    '''
    unbalance = args[0]
    k1, k2 = rotor.stiffness
    m = rotor.mass
    spin = speed / 60 * 2 * np.pi
    a = rotor.a
    div = (k1 - m * spin**2) * (k2 - m * spin**2) - a**2 * spin**4
    return (k1 - (m + a) * spin**2) * unbalance.mass.m * unbalance.radius.m * (spin)**2 * rotor.f(rotor.disc
```

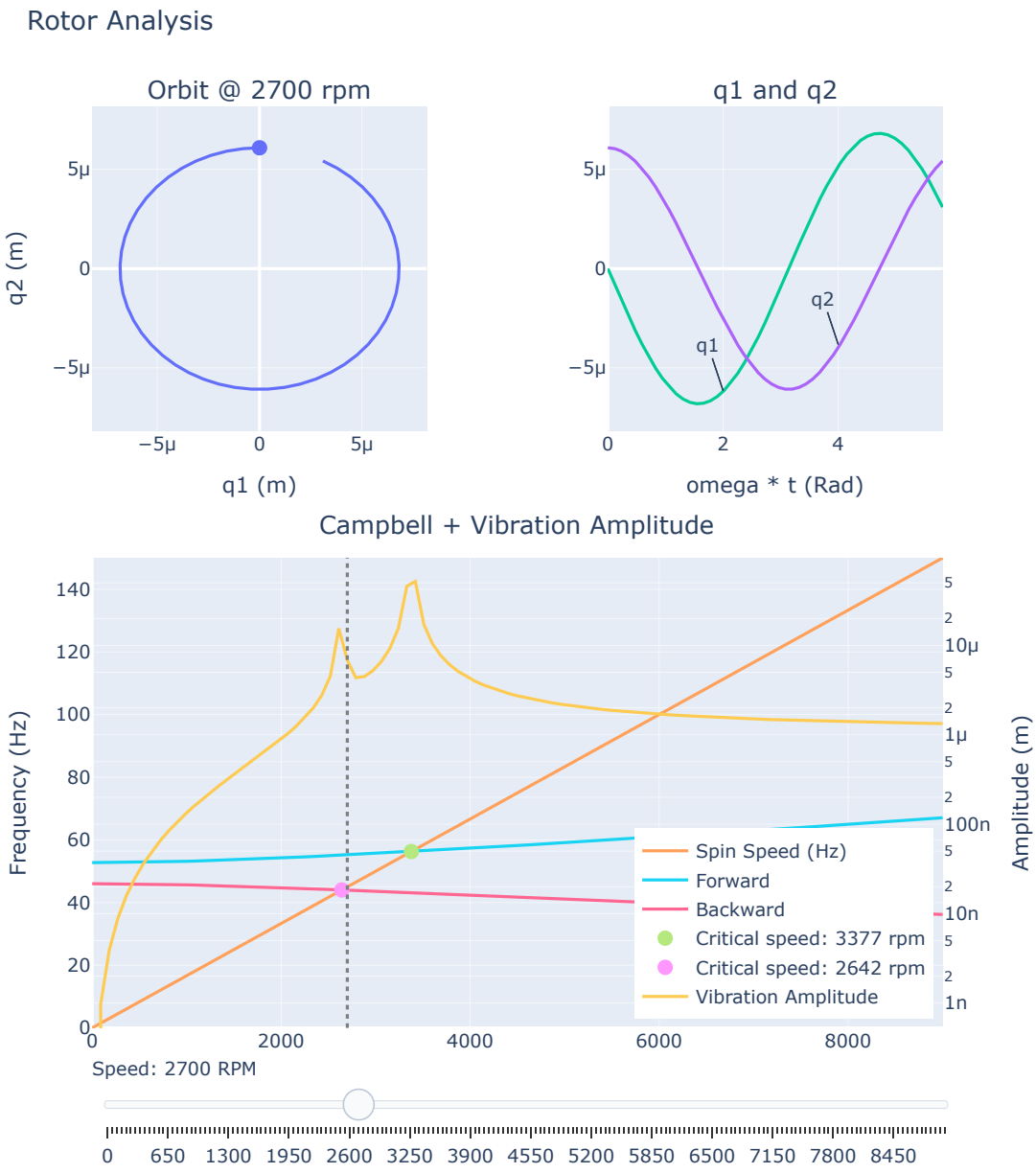
In [42]:
```
# Computing the values
values = []
values_A1 = []
values_A2 = []
for speed in speed_range:
    values_A1.append(abs(A1_non_sym(speed, unbal)))
    values_A2.append(abs(A2_non_sym(speed, unbal)))
    values.append(max(values_A1[-1], values_A2[-1]))
```

In [43]:
```
# Generating the Standard Campbell Diagram
campbell_fig = rotor.plot_Campbell()

# Call the function to add the secondary y-axis
rd.add_secondary_yaxis(campbell_fig, values_A1, title = "A1")
rd.add_secondary_yaxis(campbell_fig, values_A2, title = "A2")

# Show the updatedvfigure
campbell_fig.show()
```

In [44]:
```
# Generating the Standard Campbell Diagram
campbell_fig = rotor.plot_Campbell()

# Call the function to add the secondary y-axis
rd.add_secondary_yaxis(campbell_fig, values)

# Show the updatedvfigure
campbell_fig.show()
```



In [45]:
```
orbit_plot = rd.interactive_orbit_campbell(campbell_fig,
                                           A1_non_sym,
                                           A2_non_sym,
                                           unbal,
                                           initial_speed=2700)
orbit_plot.show()
```



Resposta a força assíncrona

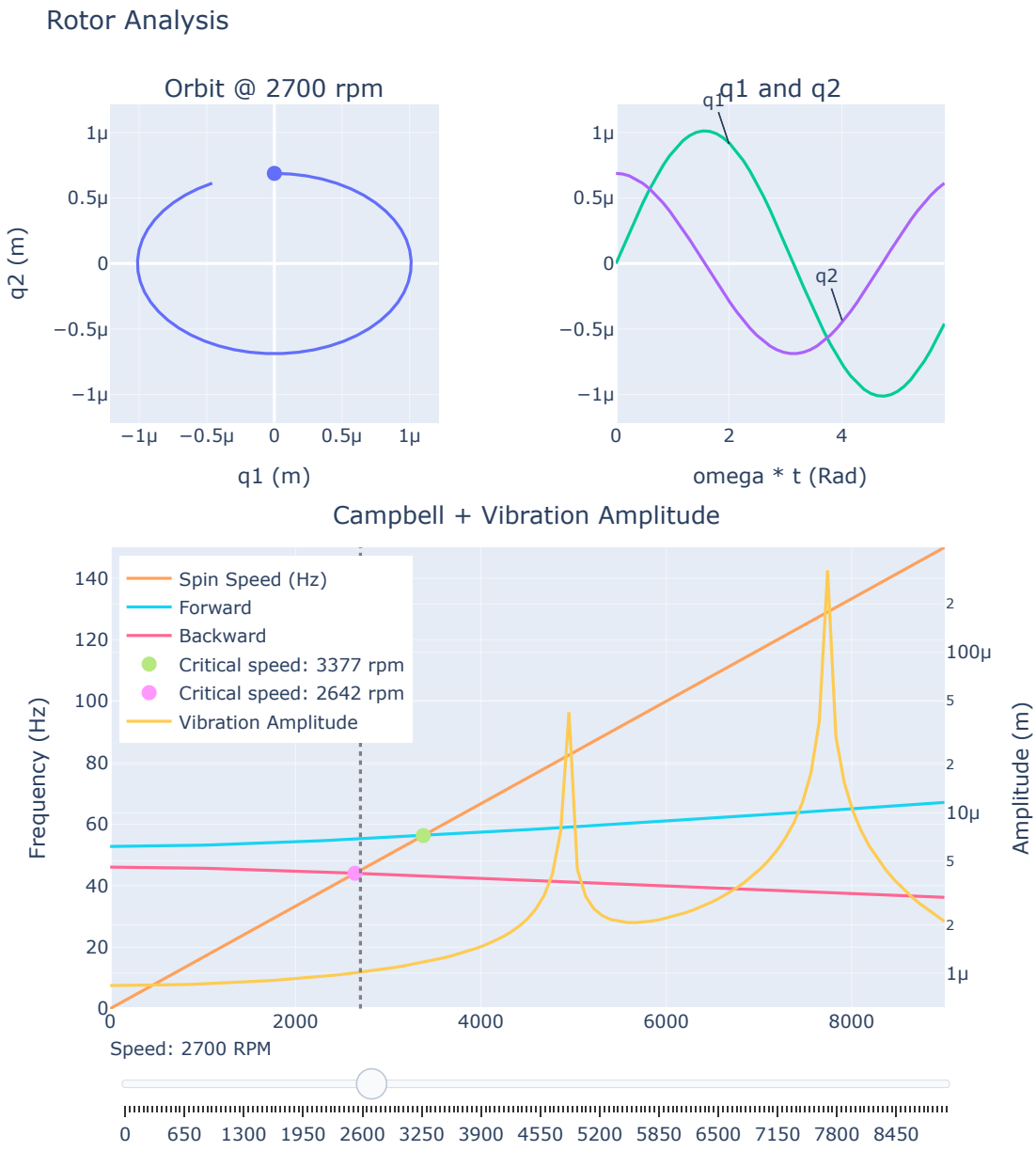
In [46]:

```python
def A2_async(speed, *args):
    '''Compute the displacemente aplitude of the rotor with an asynchronous
    excitation.

    Args:
        speed (function): Rotational speed in RPM.
        F0 (float): Asynchronous force magnitude
        s (float): A multiplier thats desynchronize the excitation

    Remarks:
        len(args) shall be equals to 2
    '''
    F0, s = args
    k1, k2 = rotor.stiffness
    m = rotor.mass
    spin = speed / 60 * 2 * np.pi
    a = rotor.a
```

```
    div = s**2 * (s**2 * m**2 - a**2) * spin**4 - m * (k1 + k2) * s**2 * spin**2 + k1 * k2
    return (k1 - (m * s**2 + a * s) * spin**2) * F0 / div
```

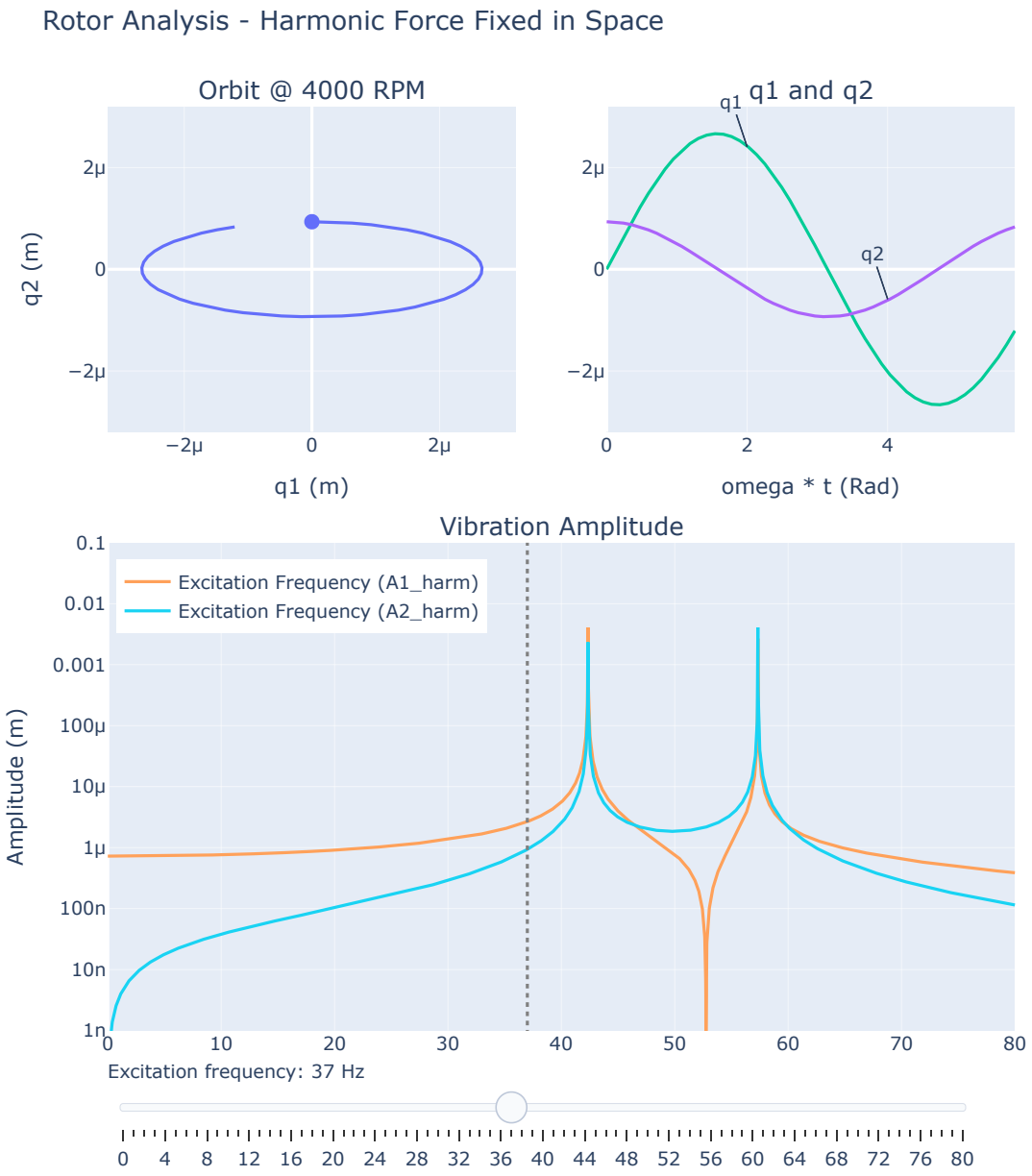
In [47]:

```
# Computing the values
values = []
for speed in speed_range:
    A1 = abs(A1_async(speed, 1, s))
    A2 = abs(A2_async(speed, 1, s))
    values.append(max(A1, A2))

# Generating the Standard Campbell Diagram
campbell_fig = rotor.plot_Campbell()

# Call the function to add the secondary y-axis
rd.add_secondary_yaxis(campbell_fig, values)

#adjusting the legend
campbell_fig.update_layout(
    legend=dict(yanchor="top", y=0.99, xanchor="left", x=0.01),
)

orbit_plot = rd.interactive_orbit_campbell(campbell_fig,
                                           A1_async,
                                           A2_async,
                                           1,
                                           0.5,
                                           initial_speed=2700)
#adjusting the legend
orbit_plot.update_layout(
    legend=dict(yanchor="top", y=0.5, xanchor="left", x=0.01),
)

# Show the updated figure
orbit_plot.show()
```



Força Harmônica fixa no espaço

In [48]:
```
response = rd.plot_vibration_amplitude(A1_harm, A2_harm)

rd.interactive_orbit_fixed_speed(
    response,
    A1_harm,
    A2_harm,
    4000,
    1,
    37
)
```



Rotor Amortecido

In [49]:

```
'''
 Reference:
     Rotordynamics Prediction in Engineering, Second Edition, by Michel
     Lalanne and Guy Ferraris, Chapter 2: Monorotors, Subsection 2.1.6, page 17.
'''
# Defining some parameters
L2 = 2 * L / 3
kxx = Q_(2e5, "N/m")
kzz = Q_(5e5, "N/m")
cxx = 2e5
czz = 5e5

spring = Spring(L2, kxx=kxx, kzz=kzz)
damping = Damping(cxx=cxx, czz=czz, beta=0.015)

rotor = rd.Rotor(shaft, disc, spring=spring, damping=damping)

speed_range = np.linspace(0, 9000, 101)
```

```
print('Frequências Naturais @ 0 RPM: ', round(rotor.omega_0[0], 2), ',', round(rotor.omega_0[1], 2))
rotor.plot_Campbell()
```

```
Frequências Naturais @ 0 RPM:  41.0 , 41.0
```