Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Gina Dokke

Dynamic Simulation of a Motor Rig Used for Educational Purposes

Master's thesis in Mechanical Engineering
Supervisor: Bjørn Haugen
Co-supervisor: Terje Rølvåg
July 2023

NTNU
Norwegian University of
Science and Technology

Gina Dokke

Dynamic Simulation of a Motor Rig Used for Educational Purposes

Master's thesis in Mechanical Engineering
Supervisor: Bjørn Haugen
Co-supervisor: Terje Rølvåg
July 2023

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

NTNU
Norwegian University of
Science and Technology

Abstract

Rotor dynamics plays a crucial role in numerous applications, and Professor Terje Rølvåg at NTNU is developing a course for students focusing on this subject. One particular aspect covered in the course is oscillations and forces related to a test rig known as Bently Nevada RK0. This rig consists of an electric motor driving a shaft connected by two bearings, with a flywheel placed at the center of the shaft capable of generating an eccentric load.

The objective of this project is to simulate the system using two different software platforms and compare the results with those obtained from the Bently Nevada RK0 test rig. The analysis involves studying a shaft with an eccentric load using NX and Fedem software, along with an analytical solution implemented in Python. The study examines the influence of bearing properties, such as damping and stiffness, as well as the impact of the maximum time step size in NX simulations.

The results indicate that increasing the stiffness of the bearings leads to larger displacements. However, the differences in eigenfrequency are minimal. Varying the damping properties affects the results, resulting in increased displacement with higher damping values. The eigenfrequency of modes 1 and 2 remain largely unchanged, while modes 3 and 4 exhibit lower values with increasing damping. The variation in the maximum time step size affects both the maximum displacement and eigenfrequency of the system. The analytical solution yields lower frequencies compared to the software simulations. Nevertheless, the frequency responses and Campbell diagram demonstrate similar trends. When comparing the results obtained from the different software platforms, there are discrepancies in eigenfrequencies and displacements. Fedem software produces the highest values for displacement and frequency, while the analytical solution yields the lowest frequency results. Overall, it can be concluded that the NX solver is not yet fully developed or mature enough to effectively solve the specific problem of rotor dynamics analyzed in this study.

Sammendrag

Rotordynamikk spiller en viktig rolle i en rekke applikasjoner, og professor Terje Rølvåg ved NTNU utvikler et emne for studenter med fokus på dette emnet. Et spesielt aspekt som dekkes i kurset er svingninger og krefter knyttet til en testrigg kalt Bently Nevada RK0. Denne riggen består av en elektrisk motor som driver en aksling koblet til to lagre, med et svinghjul plassert midt på akselen med muligheter til å generere eksentrisk belastning.

Hovedmålet med dette prosjektet er å simulere systemet med bruk av forskjellige programvare og sammenligne resultatet med resultatet fra Bently Nevada RK0 testrigg. Analysen involverer analyse av en aksling med eksentrisk last ved bruk av programmene NX og Fedem, sammen med en analytisk løsning implementert i Python. Studien undersøker påvirkning av lageregenskaper som demping og stivhet, samt påvirkningen av maksimal steglengde i tidsplanet i NX simuleringen.

Resultatet indikerer at økning i stivhet i lagrene fører til større forskyvning. Imidlertid er forskjellen i egenfrekvensen minimal. Variasjon av dempningsegenskapene påvirker resultatet, som resulterer i økt forskyvning med høyere dempningsverdier. Egenfrekvensen for svingeformene 1 og 2 forblir stort sett uendret, mens for svingeformene 3 og 4 viser lavere verdier ved økt demping. Variasjonen i den maksimale steglengden påvirker den maksimale forskyvningen og egenfrekvensen av systemet. den analytiske løsningen gir lavere frekvens sammenlignet med resultater fra NX og Fedem. Likevel viser frekvensresponsen og Campbell diagrammet lignende tendenser. Ved sammenligning av resultatet fra de forskjellige programvareplattformene er det avvik i egenfrekvens og forskyvninger. Programvaren Fedem produserer de høyeste verdiene for forskyvning og frekvens, mens den analytiske gir den laveste frekvensen. Totalt sett kan det konkluderes med at NX Nastran ennå ikke er fullt utviklet eller godt nok dokumentert til å effektivt løse rotordynamikk-problemet analysert i denne studien.

Preface

This thesis is the final project within the study of mechanical engineering at NTNU Trondheim during the spring of 2023.

The project has given a lot of knowledge about rotor dynamics. The work has been been demanding and the thesis has required acquiring a lot of new knowledge. The introduction of artificial intelligence (AI) on a large scale opened several opportunities during the work, especially with rewriting sections and fixing grammar.

I would like to express gratitude to my supervisor, Bjørn Haugen and my co-supervisor Terje Rølvåg, for their guidance, support and expertise through this research.

Good reading!

Trondheim, 13 July 2023

Gina Dokke

Table of Contents

List of Figures

List of Tables

Abbreviations

CMS	Component Mode Synthesis
COG	Center of Gravity
DOF	Degree of Freedom
FEM	Finite Element Method
FFT	Fast Fourier Transform
RPM	Rotation per Minute

List of Symbols

a	Distance between bearing1 and disk
b	Distance between bearing2 and disk
β	Angular displacement
c_{xi}	Damping in x at index i
c_{yi}	Damping in y at index i
δ	Short distance/time/angle
E	Elasticity module
ε	Distance from COG and center of shaft
f_{xi}	Forces in x at index i
f_{yi}	Forces in y at index i
I	Momentum of inertia
I_d	Momentum of inertia about x and y
I_p	Polar momentum
k_{xi}	Stiffness in x at index i
k_{yi}	Stiffness in y at index i
λ_i	Eigenvalue at index i
m	Mass of flywheel
m_0	Mass of the screw
M_x	Moment about x
M_y	Moment about y
Ω	Angular velocity about z
ω	Frequency
ω_n	Natural frequency

ϕ	Degree about z
$\dot{\phi}$	Velocity about z
$\ddot{\phi}$	Acceleration about z
ψ	Degree about y
$\dot{\psi}$	Velocity about y
$\ddot{\psi}$	Acceleration about y
r_u	Radial displacement in u-direction
r_v	Axial displacement in v-direction
η	Relative phase angle between radial and axial displacements
θ	Degree about x
$\dot{\theta}$	Velocity about x
$\ddot{\theta}$	Acceleration about x
t	Time
u	Displacement in x direction
$\dot{u}$	Velocity in x direction
$\ddot{u}$	Acceleration in x direction
v	Displacement in y direction
$\dot{v}$	Velocity in y direction
$\ddot{v}$	Acceleration in y direction

List of Matrices

$\mathbf{A}$	Combination of $\Omega\mathbf{G}$, $\mathbf{C}$ and $\mathbf{M}$ matrix
$\mathbf{B}$	Combination of $\mathbf{K}$ and $\mathbf{M}$ matrix
$\mathbf{b}_0$	A vector containing initial conditions for the system
$\mathbf{C}$	Damping matrix
$\mathbf{G}$	Gyroscope matrix
$\mathbf{H}$	The transfer matrix
$\mathbf{K}$	Stiffness matrix
$\mathbf{M}$	Mass matrix
$\mathbf{q}$	A vector containing generalized coordinates
$\mathbf{Q}$	A vector containing forces or torque acting on the rotor
$\mathbf{T}$	The transformation matrix

Chapter 1

Introduction

1.1 Background and Motivation

Rotor dynamics is an important application in many fields of industry. Rotating objects such as jet engines, combustion engines, and water turbines are crucial objects in modern society, and the understanding of rotor dynamics is hence important. Associated with rotating objects is the "critical speed", at which the rotating system experiences large displacements, oscillations, and noise [2]. Just think of the noise and vibrations of the washing machine, which has an eccentric load due to clothes gathering on one side. Many things may go wrong if rotor dynamics are not taken into account. Rotor dynamic is about knowing, understanding, and analyzing the forces, movements, and loads when a component is rotating. Neglecting rotor dynamics can have serious consequences. It can result in high forces in bearings or axle deflection. This may damage machines, reduce operational efficiency, and in the worst case be a danger to human safety.

The background of this project is that Terje Rølvåg is currently developing a course for students that include rotor dynamics at NTNU. An important asset of the course is a rig called "Bently Nevada RK0". It consists of a shaft powered by an electrical motor. A flywheel is attached at the center of the shaft, as seen in Figure 1.1. This rig is supposed to be demonstrated in a lecture, and the students will then simulate the increasing RPM using finite element model (FEM) software to find the critical speed numerically. The aim of the course is that the students should gain a deeper understanding of the rotating system, enabling them to observe shaft deflection and recognize the effects of an unbalanced mass and provide a solid comparison between theory and practice.

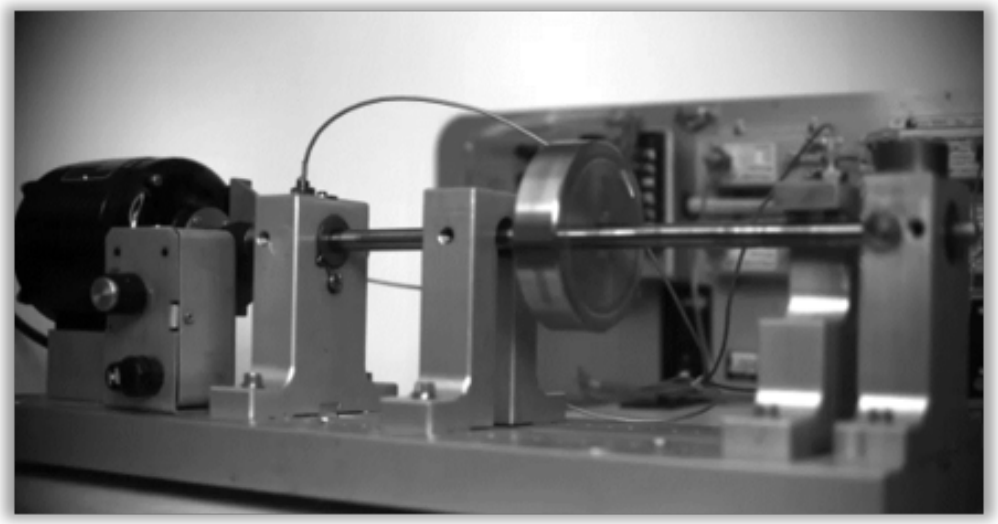


Figure 1.1: Bently Nevada RK0

1.2 Problem Description

The problem in this thesis is that an eccentric load located on the flywheel causes shaft deflections and vibrations. The eccentric load is a screw, placed out of the center that gives an eccentric load in this case. This can result in irregular loading and vibrations in the shaft. This may reduce the lifetime of the shaft and components which are in contact with the shaft and also damage the bearing itself.

When performing analyses of this problem, it is important to use the correct properties of the Bently Nevada rig such as bearing properties, motor properties, and material properties.

The bearing is made of bronze and is a journal bearing. The stiffness and damping properties of this bearing are not known but are estimated. Stiffness and damping affect the shaft. The stiffness is an important property because it makes the bearing resist deformation and vibration. The damping is the property of the bearing that reduces and guides the vibration of the shaft. Bearing properties will hence affect the response of the shaft.

The material of the components in the Bently Nevada rig is steel with Young's modulus of 206GPa, a Poisson's ratio of 0.3, and a density of 7850kg/m^3. The shaft has a diameter of 9.56mm and the length is 353.6mm.

The motor is an electric motor with an adjustable speed ranging from 0 to 10000 RPM.

The shaft on the Bently Nevada rig has a flywheel attached to it. The flywheel has a diameter of 75.6mm and has 16 holes of 4mm placed at a distance of 31 mm from the rotational axis. A steel screw with a mass of 3g is fitted into one of these holes.

1.3 Project Scope

The scope of this project encompasses a numerical simulation of the Bently Nevada rig using the software NX Nastran and Fedem. Furthermore, an analytical solution to the same problem is implemented by using Python. The aim is to compare results from the simulation in NX and Fedem with the analytical model and the physical Bently Nevada RK0 rig.

The main objectives to be studied are as follows:

- To create a model of the Bently Nevada rig in NX Nastran and run rotor analysis
- To create a dynamic model of the rotor assembly in Fedem.
- To create an analytical solution of the rotor assembly in Python.
- To compare the results obtained by the methods above with the physical rig and with each other.

The main focus throughout this project has been NX Nastran, and the work concerning this solver is hence covered more in detail than the other methods used.

This project was started in the spring of 2023 without a preliminary project relative to the subject rotor dynamics. The initial part of this project was a literature review concerning rotor dynamics theory. Next, a model of the rotor assembly was made in NX using beam elements. The bearing properties of the Bently Nevada RK0 rig were not given and it was therefore interesting to perform a parameter study of the bearing properties to match the response of the physical rig. In addition, a transient analysis was performed in NX which resulted in the eigenfrequencies of the system. In the Fedem software, a transient analysis was conducted to determine the eigenfrequency of the shaft. Additionally, an analytical solution was employed to obtain the eigenfrequencies and to solve the frequency response with the eccentric load.

1.3.1 Thesis Structure

Chapter 1 : Introduction presents the thesis and provides the background.

Chapter 2: Theory covers the relevant theory necessary to understand rotor dynamics.

Chapter 3: Method describes the methodologies employed to obtain the result from the Bently Nevada RK0 rig, including the simulation and analytical solution implemented using Python.

Chapter 4: Result presents the outcomes derived from the Bently Nevada RK0 rig, NX Nastran, Fedem, and the analytical solution.

Chapter 5: Discussion is discussing the results presented in Chapter 4.

Chapter 6: Conclusion offers a summary of the results and the enclosing conclusion.

Chapter 7: Further work suggests potential future endeavors based on the findings of this project.

Chapter 2

Theory

2.1 Rotor Dynamics

Rotordynamics is a specialized branch of applied mechanics that deals with the dynamic behavior of a rotating body. In this chapter, the focus is the theory of a rotating system and the equations used to solve this system. Most of the theory described in this chapter is based on chapters 3, 5, and 6 in the book "Dynamics of Rotating Machines" by M.I Friswell (2010). [1].

2.1.1 Coordinates and Degrees of Freedom

A coordinate system is needed in order to define the equations of motion. The coordinate system is either stationary or fixed to the rotor. For a simple and axisymmetric rotor, a stationary coordinate system is the easiest to use.

The stationary coordinate system is a Cartesian coordinate system where the origin is placed at the center of gravity. The z-axis is always placed in the axis direction and is coincident with the center line of the rotor.

Figure 2.1 shows the coordinate system of a rotating system where the z-axis is along the shaft direction. The rotor can translate in x, y and z directions noted as u, v, and w. There are also small rotations around the x- and y-axis, represented by θ and ψ respectively. The angular velocity around the z-axis is denoted as Ω [1].

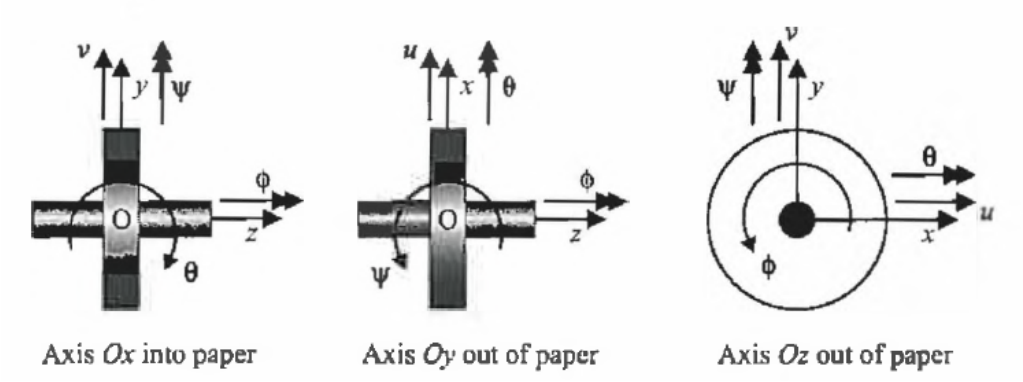


Figure 2.1: Coordinate system of a rotor [1].

2.1.2 Gyroscopic Couples

The effect of the gyroscopic couples arises as a consequence of the angular momentum. The angular momentum of inertia is a product of the polar momentum and the velocity of a given axis. The angular momentum is given as $I_p\Omega$. Rotation about the x-axis is $I_p\Omega$ where I_p is the rotor's polar angular moment of inertia. Figure 2.2 shows the effect when rotating about the y-axis and if this rotated with an angle ψ and a velocity $\dot{\psi}$ over a period δt.

$$M_x \delta t = I_p \Omega \delta\psi \;\; or \;\; M_x = I_p \Omega \dot{\psi} \tag{2.1}$$

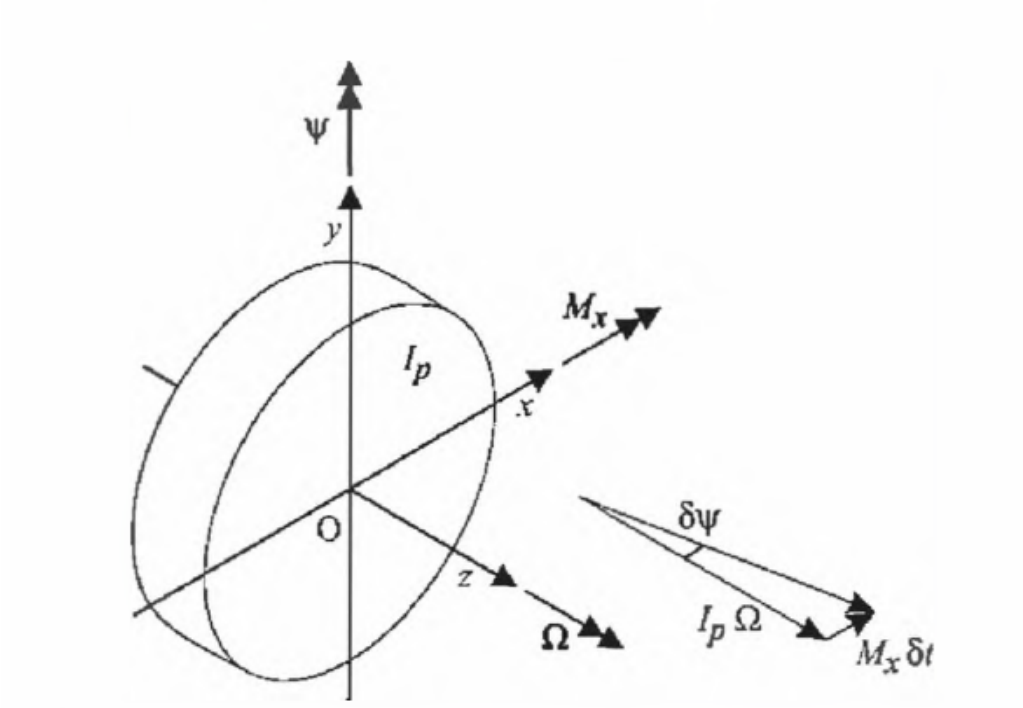


Figure 2.2: Vector diagram with effect of a clockwise momentum about x [1].

2.1.3 Rigid Rotor

Figure 2.3 illustrates a rigid rotor. This rigid rotor is supported by two bearings with no angular stiffness and this is called a short bearing.

Mass of the rotor: m

Bearing stiffness in x and y direction: k_x and k_y

Moment of inertia about x and y axis: I_d

Polar moment of the inertia: I_p

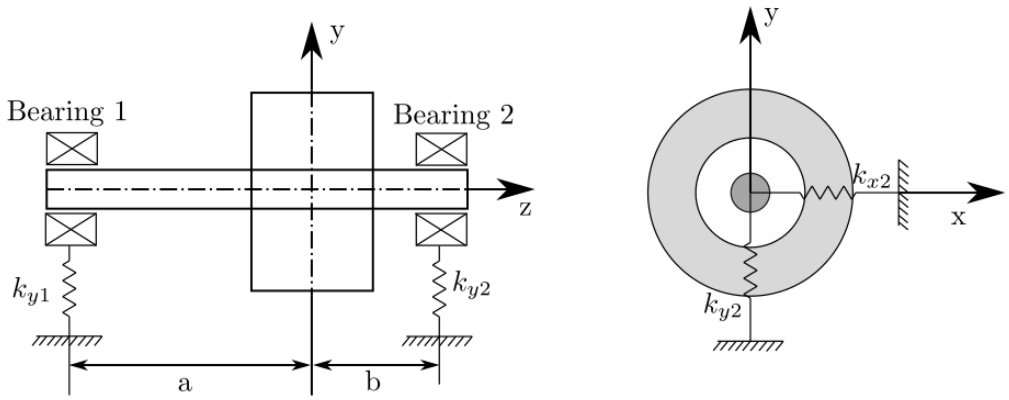


Figure 2.3: A rigid motor on flexible supports.

The equation of motion is represented by using Newton's second law or using an energy method (e.g., Lagrange's equations). A free body diagram is shown in Figure 2.4 where the force on the bearings and also the rotations θ and ψ are presented.

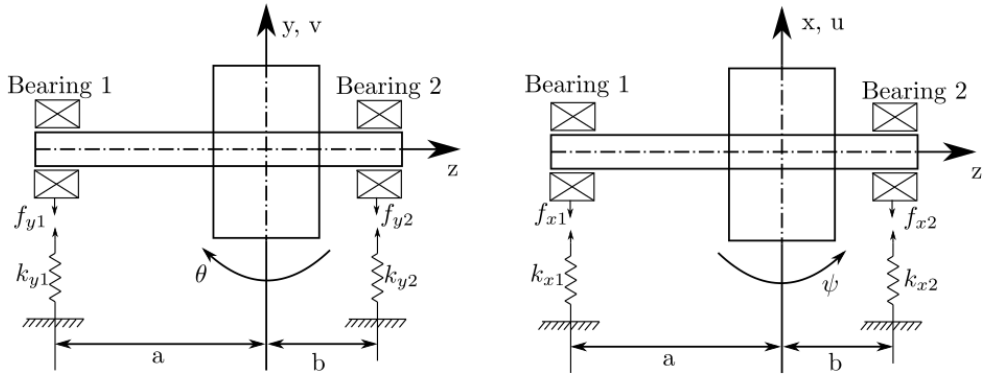


Figure 2.4: Free body diagram with force on the bearing.

The rotor has four degrees of freedom; travel in the x- and y direction is noted as u and v, and rotation about the x- and y axis is noted as θ and ψ respectively.

By using Newton's second law of motion gives the equations:

$$\begin{aligned}
&\text{Force in x direction:} && -f_{x1} - f_{x2} = m\ddot{u} \\
&\text{Force in y direction:} && -f_{y1} + f_{y2} = m\ddot{v} \\
&\text{Moment in } \theta \text{ direction:} && af_{y1} - bf_{y2} = I_d\ddot{\theta} - I_p\Omega\dot{\psi} \\
&\text{Moment in } \psi \text{ direction:} && -af_{x1} - bf_{x2} = I_d\ddot{\psi} + I_p\Omega\dot{\theta}
\end{aligned} \tag{2.2}$$

Assume that the displacement of the rotor is small and then can replace $\sin\psi$ by ψ. Also, assume that the springs are linear and Hooke's law can be used. There are no couplings between the x- and y direction and then the following relations can be defined:

$$f_{x1} = k_{x1}\delta = k_{x1}(u - a\sin\psi) \approx k_{x1}(u - a\psi) \tag{2.3}$$

By using the relationship in Equation 2.3 we get:

$$\begin{aligned}
f_{x1} &= k_{x1}(u - a\psi) \\
f_{x2} &= k_{x2}(u + b\psi) \\
f_{y1} &= k_{y1}(v + a\theta) \\
f_{y2} &= k_{y2}(v - b\theta)
\end{aligned} \tag{2.4}$$

Adding forces from equation 2.4 into equation 2.2 yields:

$$\begin{aligned}
m\ddot{u} + (k_{x1} + k_{x2})u + (-ak_{x1} + bk_{x2})\psi &= 0 \\
m\ddot{v} + (k_{y1} + k_{y2})v + (ak_{y1} - bk_{y2})\theta &= 0 \\
I_d\ddot{\theta} + I_p\Omega\dot{\psi} + (ak_{y1} - bk_{y2})v + (a^2k_{y1} + b^2k_{y2})\theta &= 0 \\
I_d\ddot{\psi} - I_p\Omega\dot{\theta} + (-ak_{x1} + bk_{x2})u + (a^2k_{x1} + b^2k_{x2})\psi &= 0
\end{aligned} \tag{2.5}$$

When using the subscripts T, C, and R that indicate translation, coupling between displacement and rotation gives the following stiffness coefficient:

$$\begin{aligned} k_{xT} &= k_{x1} + k_{x2} & k_{yT} &= k_{y1} + k_{y2} \\ k_{xC} &= -ak_{x1} + bk_{x2} & k_{yC} &= -ak_{y1} + bk_{y2} \\ k_{xR} &= a^2k_{x1} + b^2k_{x2} & k_{yR} &= a^2k_{y1} + b^2k_{y2} \end{aligned} \tag{2.6}$$

When introduce the coefficient in equation 2.6 into the equation 2.5 give:

$$\begin{aligned} m\ddot{u} + k_{xT}u + k_{xC}\psi &= 0 \\ m\ddot{v} + k_{yT}v - k_{yC}\theta &= 0 \\ I_d\ddot{\theta} + I_p\Omega\dot{\psi} - k_{yC}v + k_{yR}\theta &= 0 \\ I_d\ddot{\psi} - I_p\Omega\dot{\theta} + k_{xC}v + k_{xR}\psi &= 0 \end{aligned} \tag{2.7}$$

It is more helpful to write the Equation 2.7 is matrix form and this give equation

$$\mathbf{M}\ddot{\mathbf{q}} + \Omega\mathbf{G}\dot{\mathbf{q}} + \mathbf{K}\mathbf{q} = \mathbf{0} \tag{2.8}$$

Where

$$\mathbf{M} = \begin{bmatrix} m & 0 & 0 & 0 \\ 0 & m & 0 & 0 \\ 0 & 0 & I_d & 0 \\ 0 & 0 & 0 & I_d \end{bmatrix}, \quad \mathbf{G} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & I_p \\ 0 & 0 & -I_p & 0 \end{bmatrix}, \quad \mathbf{K} = \begin{bmatrix} k_{xT} & 0 & 0 & k_{xC} \\ 0 & k_{yT} & -k_{yC} & 0 \\ 0 & -k_{yC} & k_{yR} & 0 \\ k_{xC} & 0 & 0 & k_{xR} \end{bmatrix} \tag{2.9}$$

and

$$\mathbf{q} = \begin{bmatrix} u & v & \theta & \psi \end{bmatrix}^\top$$

The matrix of mass and stiffness are symmetric definite matrices and the gyroscopic matrix is skew-symmetric. This equation can be written in the form:

$$\begin{bmatrix} \Omega\mathbf{G} & \mathbf{M} \\ \mathbf{M} & \mathbf{0} \end{bmatrix} \frac{d}{dt} \begin{bmatrix} \mathbf{q} \\ \dot{\mathbf{q}} \end{bmatrix} + \begin{bmatrix} \mathbf{K} & \mathbf{0} \\ \mathbf{0} & -\mathbf{M} \end{bmatrix} \begin{bmatrix} \mathbf{q} \\ \dot{\mathbf{q}} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \tag{2.10}$$

The equation 2.10 can be written as:

$$\mathbf{A}\dot{\mathbf{x}} + \mathbf{B}\mathbf{x} = \mathbf{0} \tag{2.11}$$

Where:

$$\mathbf{x} = \begin{bmatrix} \mathbf{q} \\ \\ \dot{\mathbf{q}} \end{bmatrix}, \qquad \dot{\mathbf{x}} = \frac{d}{dt} \begin{bmatrix} \mathbf{q} \\ \\ \dot{\mathbf{q}} \end{bmatrix}, \qquad \mathbf{A} = \begin{bmatrix} \Omega\mathbf{G} & \mathbf{M} \\ \\ \mathbf{M} & \mathbf{0} \end{bmatrix}, \qquad \mathbf{B} = \begin{bmatrix} \mathbf{K} & \mathbf{0} \\ \\ \mathbf{0} & -\mathbf{M} \end{bmatrix}$$

Look for a solution in the form $\mathbf{x}(t) = \mathbf{x}_0 e^{st}$ and then $\dot{\mathbf{x}} = \lambda \mathbf{x}_0 e^{st}$ and then the Equation 2.11 can be written as

$$\lambda \mathbf{A}\mathbf{x}_0 = -\mathbf{B}\mathbf{x}_0 \tag{2.12}$$

This is a 2n x 2n eigenvalue problem, which needs to be solved numerically.

2.1.4 Forward and Backward Whirl

When a rotating system enters the "critical speed range", i.e. the rotating speed at which it matches the natural frequency, it will start to resonate causing vibrations. The vibrations start to do a transverse movement. This movement together with the rotation of the axis results in an orbit. The orbit has a rotation direction denoted as a forward- or backward whirl. Eigenvector and eigenvalue are in general complex. The eigenvalue is $\lambda = j\omega_i$ where ω_i is the natural frequency and the natural frequency is real and positive.

The damping affects only the amplitude and not the shape of the orbit or the direction. The free-response in the mode can be written as:

$$\mathbf{x}(t) = \mathbb{R}(\mathbf{x}^{(i)} e^{j\omega_i t}) \tag{2.13}$$

where $x^{(i)}$ is the eigenvector

The direction of the rotation of the mode may be found considering displacement or rotation at a single node. The response can be written as:

$$\begin{bmatrix} u(t) \\ v(t) \end{bmatrix} = \mathbb{R}\left(\begin{bmatrix} r_u e^{j\eta_u} \\ r_v e^{j\eta_v} \end{bmatrix} \quad e^{j\omega_i t} \right) = \begin{bmatrix} r_u \cos(\eta_u + \omega_i t) \\ r_v \cos(\eta_v + \omega_i t) \end{bmatrix} = \mathbf{T} \begin{bmatrix} \cos \omega_i t \\ \sin \omega_i t \end{bmatrix} \tag{2.14}$$

$\mathbf{T}$ is the collected terms of the eigenvector.

$$\mathbf{T} = \begin{bmatrix} r_u \cos \eta_u & -r_u \sin \eta_u \\ r_v \cos \eta_v & -r_v \sin \eta_v \end{bmatrix} \tag{2.15}$$

From Equation 2.14 and modifying the equation yields:

$$\begin{bmatrix} \cos \omega_i t \\ \sin \omega_i t \end{bmatrix} = \mathbf{T}^{-1} \begin{bmatrix} u(t) \\ v(t) \end{bmatrix} \tag{2.16}$$

The orbit (u, v) forms an ellipse, as demonstrated by the fact that the response u and v together constitute an orbit.

$$\begin{bmatrix} u(t) \\ v(t) \end{bmatrix}^{\top} \mathbf{T}^{-\top} \mathbf{T}^{-1} \begin{bmatrix} u(t) \\ v(t) \end{bmatrix} = \cos^2 \omega_i t + \sin^2 \omega_i t = 1 \tag{2.17}$$

The length of the major and minor axes is found by solving the eigenvalue of the matrix $\mathbf{T}\mathbf{T}^{\top}$ and this is denoted as $\mathbf{H}$, where the matrix is symmetric and positively defined. The eigenvalues

have real components λ_1 and λ_2, where $\lambda_1 \geq \lambda_2$. The length of the semiminor and semimajor axis are represented by $\sqrt{\lambda_1}$ and $\sqrt{\lambda_2}$. To define the direction of the orbit can be found by evaluating Equation 2.14. If the time is shifted so that $t \to (t - \eta_u/\omega_i)$, then:

$$\begin{bmatrix} u(t) \\ v(t) \end{bmatrix} = \begin{bmatrix} r_u \cos(\omega_i t) \\ r_v \cos(\eta_v - \eta_u + \omega_i t) \end{bmatrix} \tag{2.18}$$

The direction of rotation depends on the phase difference between the response u and v. The response is given by $\eta_v - \eta_u$. The response is a straight line if $\eta_v = \eta_u$ or $\eta_v = \eta_u + \pi$ and then a point on the rotor's centerline is vibration in one direction. A backward rotation mode exists if $0 < \eta_v - \eta_u < \pi$ and if $-\pi < \eta_v - \eta_u < 0$ a forward rotation exists. If $\eta_v - \eta_u$ is not between $-\pi$ and $-\pi$, then multiplies of 2π are added. The properties of the orbit can be collected by a single parameter κ defined as:

$$\kappa = \pm\sqrt{\lambda_2/\lambda_1} \tag{2.19}$$

Where κ is positive for forward rotation and negative for backward rotation. In cases where $\kappa = \pm 1$, the orbit is circular.

Using small displacements and rotation yield:

$$\begin{aligned} u_1 &= u - a\psi, \quad v_1 = v + a\theta \\ u_2 &= u + b\psi, \quad v_2 = v - b\theta \end{aligned} \tag{2.20}$$

Where u is the displacement in the x direction and v is the displacement in the y direction. The subscript 1 and 2 indicate the displacement at bearing 1 or bearing 2.

2.1.5 Damping

The rotor is assumed to only have damping in each bearing. The damping is modeled parallel to the spring at each bearing location. The free-body diagram of a rotor with damping and spring forces can be seen in Figure 2.5.

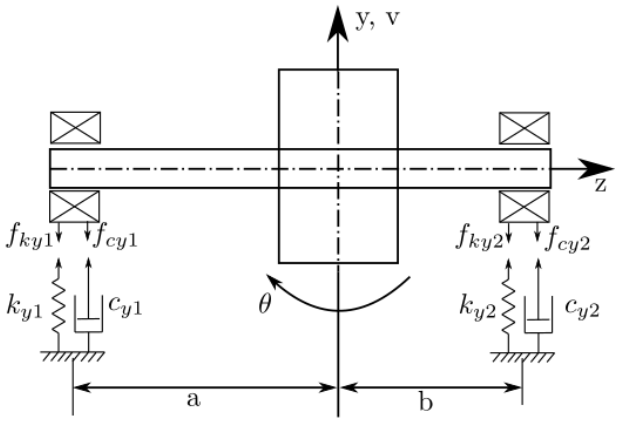


Figure 2.5: Free body diagram of a rotor with damping.

The new equations of the force equilibrium at the bearings from Equation 2.4 become:

$$\begin{aligned} f_{x1} &= k_{x1}(u - a\psi) + c_{x1}(\dot{u} - a\dot{\psi}) \\ f_{x2} &= k_{x2}(u + b\psi) + c_{x2}(\dot{u} + b\dot{\psi}) \\ f_{y1} &= k_{y1}(v + a\theta) + c_{y1}(\dot{v} + a\dot{\theta}) \\ f_{y2} &= k_{y2}(v - b\theta) + c_{y2}(\dot{v} - b\dot{\theta}) \end{aligned} \tag{2.21}$$

Where c is the viscous-damping coefficient and is defined as:

$$\begin{aligned} c_{xT} &= c_{x1} + c_{x2} & c_{yT} &= c_{y1} + c_{y2} \\ c_{xC} &= -ac_{x1} + bc_{x2} & c_{yC} &= -ac_{y1} + bc_{y2} \\ c_{xR} &= a^2c_{x1} + b^2c_{x2} & c_{yR} &= a^2c_{y1} + b^2c_{y2} \end{aligned} \tag{2.22}$$

When using the definition in Equation 2.22 and then substituting Equation 2.21 into Equation 2.2, the complete equation of motion for a damped system are given as:

$$\begin{aligned} m\ddot{u} + c_{xT}\dot{u} + c_{xC}\dot{\psi} + k_{xT}u + k_{xC}\psi &= 0 \\ m\ddot{v} + c_{yT}\dot{v} - c_{yC}\dot{\theta} + k_{yT}v - k_{yC}\theta &= 0 \\ I_d\ddot{\theta} + I_p\Omega\dot{\psi} - c_{yC}\dot{v} + c_{yR}\dot{\theta} - k_{yC}v + k_{yR}\theta &= 0 \\ I_d\ddot{\psi} - I_p\Omega\dot{\theta} + c_{xC}\dot{u} + c_{xR}\dot{\psi} + k_{xC}u + k_{xR}\psi &= 0 \end{aligned} \tag{2.23}$$

Equation 2.23 can be written in matrix form as:

$$\mathbf{M}\ddot{\mathbf{q}} + \mathbf{C}\dot{\mathbf{q}} + \Omega\mathbf{G}\dot{\mathbf{q}} + \mathbf{K}\mathbf{q} = \mathbf{0} \tag{2.24}$$

Where the matrices $\mathbf{M}$, $\mathbf{G}$ and $\mathbf{K}$ is defined in Equation 2.9 and $\mathbf{C}$ is defined as:

$$\mathbf{C} = \begin{bmatrix} c_{xT} & 0 & 0 & c_{xC} \\ 0 & c_{yT} & -c_{yC} & 0 \\ 0 & -c_{yC} & c_{yR} & 0 \\ c_{xC} & 0 & 0 & c_{xR} \end{bmatrix}$$

When making Equation 2.24, symmetry gives the following system of equations:

$$\begin{bmatrix} \Omega\mathbf{G} + \mathbf{C} & \mathbf{M} \\ \mathbf{M} & \mathbf{0} \end{bmatrix} \frac{d}{dt} \begin{bmatrix} \mathbf{q} \\ \dot{\mathbf{q}} \end{bmatrix} + \begin{bmatrix} \mathbf{K} & \mathbf{0} \\ \mathbf{0} & -\mathbf{M} \end{bmatrix} \begin{bmatrix} \mathbf{q} \\ \dot{\mathbf{q}} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \tag{2.25}$$

2.1.6 Flexible Rotor with Rigid Bearings

In the previous section, the rotor is rigid on flexible supports. This means that the rotor shaft is rigid compared to the bearing stiffness. A lot of rotors can not be modeled as a rigid body since they are flexible because of its small diameter compared to the length. A flexible rotor can vibrate, even if the shaft is supported by rigid bearings on rigid supports. Figure 2.6 is show a rotor with a small diameter compared to the length and is a flexible shaft with rigid bearing support.

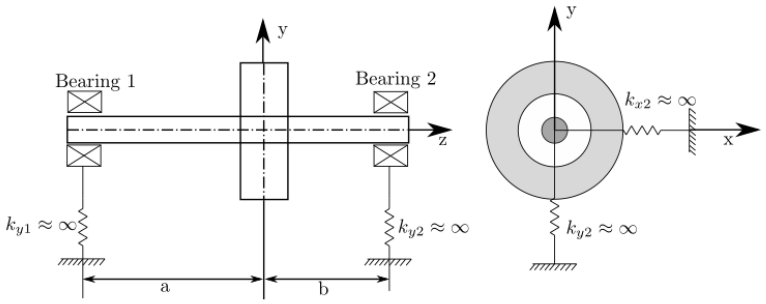


Figure 2.6: Flexible rotor on rigid bearing.

For small displacement and rotation, there is a linear relationship between the force and moment applied to the shaft direction in the x- or y direction. The moment is applied to the shaft in the y- or x direction. This is a result of the respective displacement and rotation of the shaft. For a specific point on the rotor, the force and moments are:

$$\begin{aligned} f_x &= k_{uu}u + k_{u\psi}\psi \\ M_y &= k_{\psi\psi}\psi + k_{\psi u}u \end{aligned} \tag{2.26}$$

In Equation 2.26 is f_x applied to the force in the x direction and M_y is a moment about the y axis. The parameters u and ψ are defined in Figure 2.1, and the coefficients k_{uu}, $k_{\psi u}$ and $k_{u\psi}$ is the stiffness coefficient of the shaft. For a conservative system is $k_{\psi u} = k_{u\psi}$

Using Newton's second law of motion yields:

$$\begin{aligned} &\text{Force on disk in x direction:} && -f_x = m\ddot{u} \\ &\text{Force on disc in y direction:} && -f_y = m\ddot{v} \\ &\text{Moment acting on the disk in } \theta \text{ direction:} && -M_x = I_d\ddot{\theta} + I_p\Omega\dot{\psi} \\ &\text{Moment acting on the disk in } \psi \text{ direction:} && -M_y = I_d\ddot{\psi} - I_p\Omega\dot{\theta} \end{aligned} \tag{2.27}$$

Applying Equation 2.27 on Equation 2.26 give the motion of the flexible rotor as:

$$\begin{aligned} m\ddot{u} + k_{uu}u + k_{u\psi}\psi &= 0 \\ m\ddot{v} + k_{vv}v + k_{u\theta}\theta &= 0 \\ I_d\ddot{\theta} + I_p\Omega\dot{\psi} + k_{\theta v}v + k_{\theta\theta}\theta &= 0 \\ I_d\ddot{\psi} - I_p\Omega\dot{\theta} + k_{\psi u}u + k_{\psi\psi}\psi &= 0 \end{aligned} \tag{2.28}$$

For a circular shaft, the properties are identical in each direction, so the equation in Equation 2.28 can be simplified by using $k_{uu} = k_{vv} = k_T$, $k_{\theta\theta} = k_{\psi\psi} = k_R$ and $k_{u\theta} = -k_{v\theta} = k_C$. The last sign is because of the convention from Figure 2.1. This is reduced to:

$$\begin{aligned} m\ddot{u} + k_T u + k_C\psi &= 0 \\ m\ddot{v} + k_T v - k_C\theta &= 0 \\ I_d\ddot{\theta} + I_p\Omega\dot{\psi} - k_C v + k_R\theta &= 0 \\ I_d\ddot{\psi} - I_p\Omega\dot{\theta} + k_C u + k_R\psi &= 0 \end{aligned} \tag{2.29}$$

These equations are identical to Equation 2.7. The stiffness coefficient for a shaft with length L, with a pinned support and short bearing with a disk that is located a distance a from bearing 1 and b from bearing 2, is given as:

$$\begin{aligned} k_T = k_{uu} = k_{vv} &= \frac{3EI(a^3+b^3)}{a^3b^3} \\ k_C = k_{u\psi} = -k_{v\theta} &= \frac{3EI(a^2-b^2)}{a^2b^2} \\ k_R = k_{\psi\psi} = k_{\theta\theta} &= \frac{3EI(a+b)}{ab} \end{aligned} \tag{2.30}$$

2.1.7 Frequency Response

Frequency response account cause lateral forces. In the previous section, the equation of motion was solved as a free response. This means there were no loads applied to the rotor.

Figure 2.7 explores the synchronous response of a rotor to out-of-balance forces and moments. While the analysis initially focuses on a rigid rotor, it can be extended to flexible rotors. The investigation aims to understand the impact of out-of-balance mass on a circular rotor.

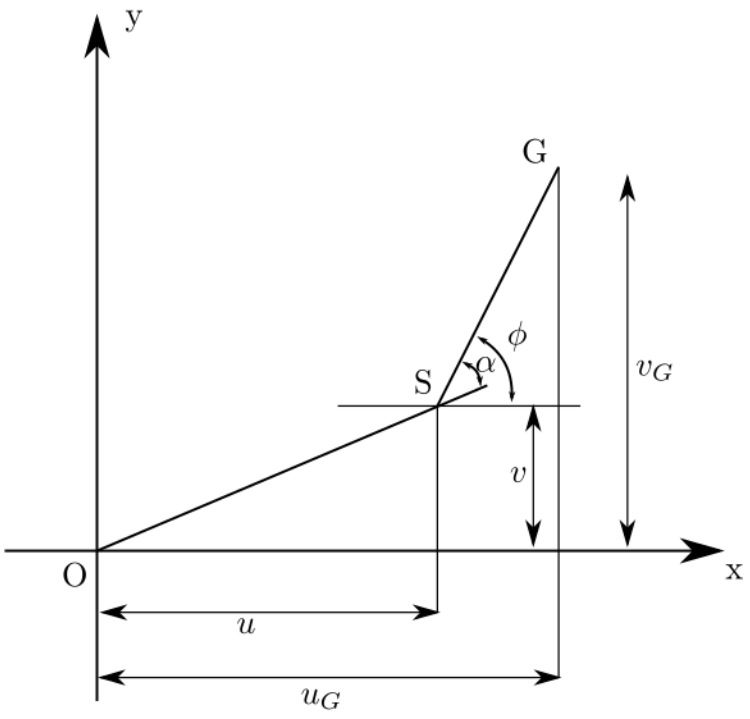


Figure 2.7: Instanteous position of bearing centerline S and rotor mass center G.

In the figure, the equilibrium position (O) is identified, along with the instantaneous position of the disturbed rotor centerline (S) and the position of the mass center of the rotor (G). The distance $|SG|$ represents the initial displacement (ε) of the rotor's center of mass from the shaft centerline at equilibrium.

The angle between the line SG, which represents a line on the rotor, and the Ox axis is denoted as ψ. Additionally, the angle between the line OS and the Ox axis is $\psi - \alpha$. The distance $|OS|$, known as the amplitude of whirl, is calculated during the analysis.

The center of mass has a movement u_G and v_G in the x- and y direction. The centerline of the rotor deflects u and v in corresponding directions at the flexible bearing.

From Figure 2.7, the displacement of the center of mass is found as:

$$\begin{aligned} u_G &= u + \varepsilon \cos\theta \\ v_G &= v + \varepsilon \sin\theta \end{aligned} \tag{2.31}$$

If ε is constant and the equations are differentiated with respect to time, the following can be derived:

$$\begin{aligned} \ddot{u}_G &= \ddot{u} + \varepsilon(-\dot{\theta}^2 \cos\theta - \ddot{\phi} \sin\theta) \\ \ddot{v}_G &= \ddot{v} + \varepsilon(-\dot{\theta}^2 \sin\theta + \ddot{\phi} \cos\theta) \end{aligned} \tag{2.32}$$

If the rotating speed is constant, then $\dot{\psi} = \Omega$ and $\ddot{\psi} = 0$. This gives:

$$\begin{aligned} \ddot{u}_G &= \ddot{u} - \varepsilon\Omega^2 \cos\Omega t \\ \ddot{v}_G &= \ddot{v} - \varepsilon\Omega^2 \sin\Omega t \end{aligned} \tag{2.33}$$

For a system where the center of mass is at an offset from the shaft at a small distance ε, the displacement of the mass is given by u_G and v_G. $\ddot{u}$ and $\ddot{v}$ is replaced by $\ddot{u}_G$ and $\ddot{v}_G$, which give the equation:

$$\begin{aligned} m\ddot{u}_G + c_{xT}\dot{u} + c_{xC}\dot{\psi} + k_{xT}u + k_{xC}\psi &= 0 \\ m\ddot{v}_G + c_{yT}\dot{v} - c_{yC}\dot{\theta} + k_{yT}v - k_{yC}\theta &= 0 \\ I_d\ddot{\theta} + I_p\Omega\dot{\psi} - c_{yC}\dot{v} + c_{yR}\dot{\theta} - k_{yC}v + k_{yR}\theta &= 0 \\ I_d\ddot{\psi} - I_p\Omega\dot{\theta} + c_{xC}\dot{u} + c_{xR}\dot{\psi} + k_{yC}v + k_{yR}\psi &= 0 \end{aligned} \tag{2.34}$$

By substituting for $\ddot{u}_G$ and $\ddot{v}_G$ from Equation 2.34:

$$\begin{aligned} m\ddot{u} + c_{xT}\dot{u} + c_{xC}\dot{\psi} + k_{xT}u + k_{xC}\psi &= m\varepsilon\Omega^2\cos\Omega t \\ m\ddot{v} + c_{yT}\dot{v} - c_{yC}\dot{\theta} + k_{yT}v - k_{yC}\theta &= m\varepsilon\Omega^2\cos\Omega t \\ I_d\ddot{\theta} + I_p\Omega\dot{\psi} - c_{yC}\dot{v} + c_{yR}\dot{\theta} - k_{yC}v + k_{yR}\theta &= 0 \\ I_d\ddot{\psi} - I_p\Omega\dot{\theta} + c_{xC}\dot{u} + c_{xR}\dot{\psi} + k_{yC}v + k_{yR}\psi &= 0 \end{aligned} \tag{2.35}$$

Figure 2.8 shows the effect of an unbalanced load. The figure shows that the center of gravity is moving.

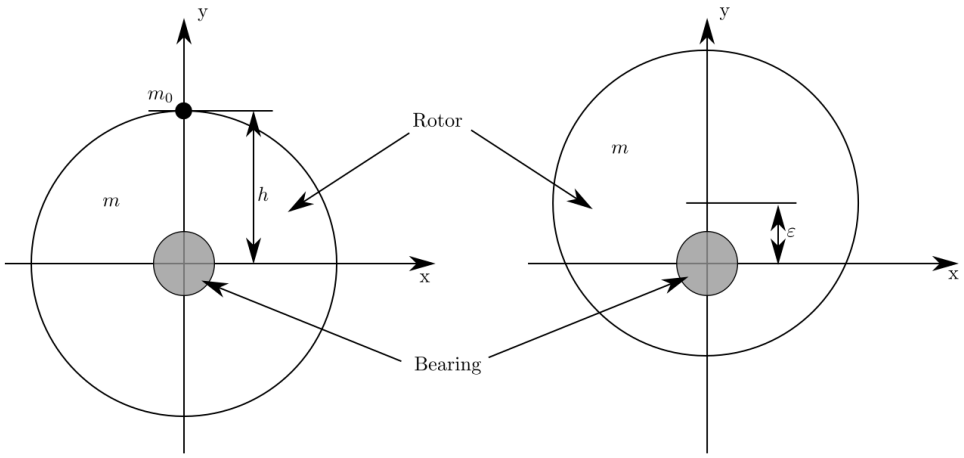


Figure 2.8: Models of out-of-balance causing a lateral force.

The out-of-balance force for the right-hand diagram in Figure 2.8 is given by:

$$F = m\varepsilon\Omega^2 \tag{2.36}$$

The force of the left-hand diagram in Figure 2.8 gives a lateral acceleration of mass m_0 $a\Omega^2$ and the resulting force is given as:

$$F = m_0 h\Omega^2 \tag{2.37}$$

Equation 2.36 and 2.37 is identical if:

$$m\varepsilon = m_0 h \tag{2.38}$$

Figure 2.9 depicts a shaft with a disk where the center of gravity for the disk is displaced at a distance ε.

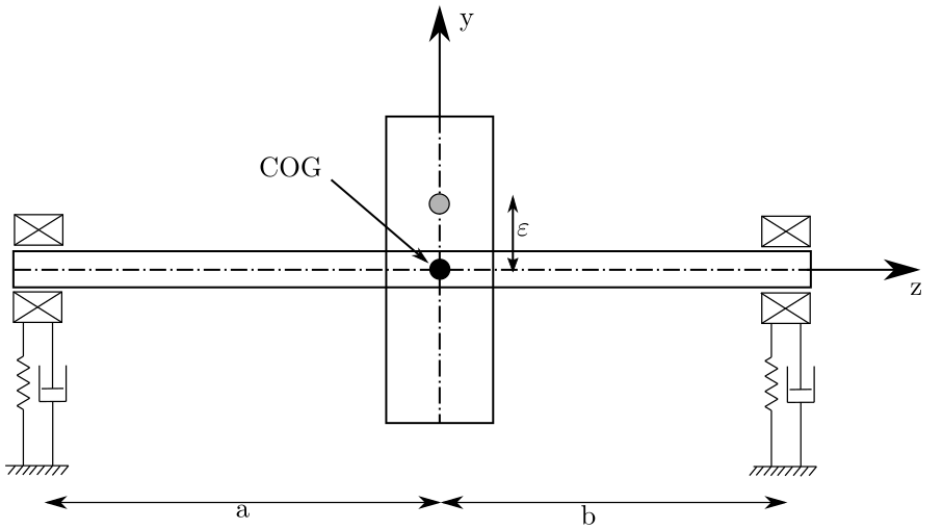


Figure 2.9: Unbalanced rotor. The COG is moved a distance ε from the initial position.

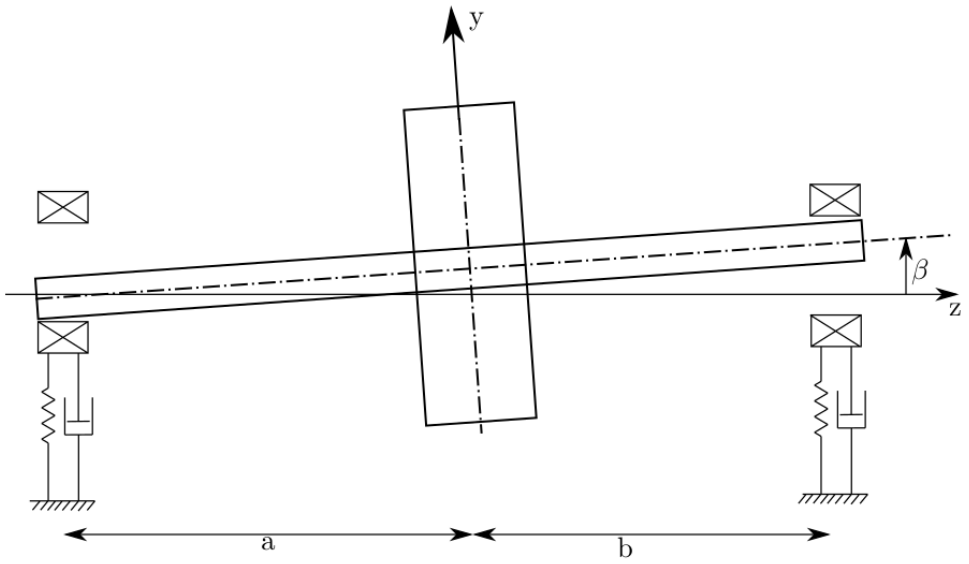


Figure 2.10: Rigid rotor with skewed principal axis of inertia.

Figure 2.10 shows how the angular displacement provides a torque. The torque due to the angular offset is given by:

$$M = \pm(I_d - I_p)\beta\Omega^2 \tag{2.39}$$

The sign in Equation 2.39 is dependent upon the direction of the angle.

By rotating, θ and ψ give the angular position of the rotor as:

$$\begin{aligned} \theta_A &= \theta - \beta \sin \Omega t \\ \psi_A &= \psi + \beta \cos \Omega t \end{aligned} \tag{2.40}$$

This gives an angular velocity of:

$$\begin{aligned} \dot{\theta}_A &= \dot{\theta} - \beta\Omega \cos \Omega t \\ \dot{\psi}_A &= \dot{\psi} - \beta\Omega \sin \Omega t \end{aligned} \tag{2.41}$$

And the angular acceleration becomes:

$$\begin{aligned} \ddot{\theta}_A &= \ddot{\theta} + \beta\Omega^2 \sin \Omega t \\ \ddot{\psi}_A &= \ddot{\psi} - \beta\Omega^2 \cos \Omega t \end{aligned} \tag{2.42}$$

Equation 2.23 is used to define the angular displacement of the rotor. The angular acceleration $\ddot{\theta}$ and $\ddot{\psi}$ is replaced by $\ddot{\theta}_A$ and $\ddot{\psi}_A$. The angular velocity $\dot{\theta}$ and $\dot{\psi}$ is replaced by $\dot{\theta}_A$ and $\dot{\psi}_A$. This gives the following equation:

$$\begin{aligned} m\ddot{u} + c_{xT}\dot{u} + c_{xC}\dot{\psi} + k_{xT}u + k_{xC}\psi &= 0 \\ m\ddot{v} + c_{yT}\dot{v} - c_{yC}\dot{\theta} + k_{yT}v - k_{yC}\theta &= 0 \\ I_d\ddot{\theta}_A + I_p\Omega\dot{\psi}_A - c_{yC}\dot{v} + c_{yR}\dot{\theta} - k_{yC}v + k_{yR}\theta &= 0 \\ I_d\ddot{\psi}_A - I_p\Omega\dot{\theta}_A + c_{xC}\dot{u} + c_{xR}\dot{\psi} + k_{yC}v + k_{yR}\psi &= 0 \end{aligned} \tag{2.43}$$

By substituting $\dot{\theta}_A$, $\dot{\psi}_A$, $\dot{\psi}_A$ and $\ddot{\psi}_A$, Equation 2.41 and 2.42 give:

$$\begin{aligned} m\ddot{u} + c_{xT}\dot{u} + c_{xC}\dot{\psi} + k_{xT}u + k_{xC}\psi &= 0 \\ m\ddot{v} + c_{yT}\dot{v} - c_{yC}\dot{\theta} + k_{yT}v - k_{yC}\theta &= 0 \\ I_d\ddot{\theta} + I_p\Omega\dot{\psi} - c_{yC}\dot{v} + c_{yR}\dot{\theta} - k_{yC}v + k_{yR}\theta &= -(I_d - I_p)\beta\Omega^2\sin\Omega t \\ I_d\ddot{\psi} - I_p\Omega\dot{\theta} + c_{xC}\dot{u} + c_{xR}\dot{\psi} + k_{yC}v + k_{yR}\psi &= (I_d - I_p)\beta\Omega^2\cos\Omega t \end{aligned} \tag{2.44}$$

When combining Equation 2.35 and 2.44, the result is a rotor with out-of-balance forces and moments. The combined equation is:

$$\begin{aligned} m\ddot{u} + c_{xT}\dot{u} + c_{xC}\dot{\psi} + k_{xT}u + k_{xC}\psi &= m\varepsilon\Omega^2\cos(\Omega t + \delta) \\ m\ddot{v} + c_{yT}\dot{v} - c_{yC}\dot{\theta} + k_{yT}v - k_{yC}\theta &= m\varepsilon\Omega^2\sin(\Omega t + \delta) \\ I_d\ddot{\theta} + I_p\Omega\dot{\psi} - c_{yC}\dot{v} + c_{yR}\dot{\theta} - k_{yC}v + k_{yR}\theta &= -(I_d - I_p)\beta\Omega^2\sin(\Omega t + \gamma) \\ I_d\ddot{\psi} - I_p\Omega\dot{\theta} + c_{xC}\dot{u} + c_{xR}\dot{\psi} + k_{yC}v + k_{yR}\psi &= (I_d - I_p)\beta\Omega^2\cos(\Omega t + \gamma) \end{aligned} \tag{2.45}$$

Using matrix notation, Equation 2.45 can be written as:

$$\mathbf{M}\ddot{\mathbf{q}} + (\Omega\mathbf{G} + \mathbf{C})\dot{\mathbf{q}} + \mathbf{K}\mathbf{q} = \mathbf{Q} \tag{2.46}$$

With the $\mathbf{Q}$ vector written as:

$$\mathbf{Q} = \begin{bmatrix} m\varepsilon\Omega^2\cos(\Omega t + \delta) \\ m\varepsilon\Omega^2\sin(\Omega t + \delta) \\ -(I_d - I_p)\beta\Omega^2\sin(\Omega t + \gamma) \\ (I_d - I_p)\beta\Omega^2\cos(\Omega t + \gamma) \end{bmatrix} = \Re \begin{bmatrix} m\varepsilon e^{j\delta} \\ -jm\varepsilon e^{j\delta} \\ j(I_d - I_p)\beta e^{j\gamma} \\ (I_d - I_p)\beta e^{j\gamma} \end{bmatrix} \Omega^2 e^{j\Omega t} \tag{2.47}$$

Assume a response on the form $\mathbf{q}(t) = \Re(\mathbf{q}_0 e^{j\Omega t})$ give that $\mathbf{q} = \mathbf{q}_0 e^{j\Omega t}$, $\dot{\mathbf{q}} = \mathbf{q}_0 \Omega e^{j\Omega t}$ and $\ddot{\mathbf{q}} = -\mathbf{q}_0 \Omega^2 e^{j\Omega t}$. The equation can be written as:

$$(-\Omega^2 \mathbf{M} + j\Omega(\Omega \mathbf{G} + C) + \mathbf{K})\mathbf{q}_0 e^{j\Omega t} = \Omega^2 \mathbf{b}_0 e^{j\Omega t} \tag{2.48}$$

Where:

$$\mathbf{b}_0 = \begin{bmatrix} m\varepsilon e^{j\delta} \\ -jm\varepsilon e^{j\delta} \\ j(I_d - I_p)\beta e^{j\gamma} \\ (I_d - I_p)\beta e^{j\gamma} \end{bmatrix}$$

Solving for $\mathbf{q}_0$:

$$\mathbf{q}_0 = [-\Omega^2 \mathbf{M} + j\Omega(\Omega \mathbf{G} + \mathbf{C} + \mathbf{K}]^{-1} \Omega^2 \mathbf{b}_0 \tag{2.49}$$

This equation is used to find the displacement of the rotor.

2.2 Finite Element Method

2.2.1 Beam Element

In this model is beam element used to simulate. The models with beam elements produce very good results and are solved fast. To define the beam element, two different definitions are used, namely the “Bernoulli” and “Timoskenko” beam elements.

The beam element has two nodes, which connect the element to other elements. The degrees of freedom for an element are described by the q vector as $\mathbf{q} = \begin{bmatrix} u_1 & v_1 & \theta_1 & \psi_1 & u_2 & v_2 & \theta_2 & \psi_2 \end{bmatrix}^\top$. The mass, damping, gyroscopic, and stiffness matrix is represented as an 8x8 form. The local coordinates used for a beam element are defined in Figure 2.11.

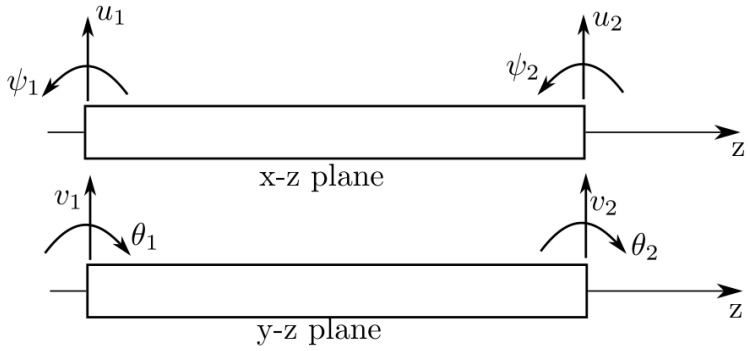


Figure 2.11: Coordinates of a Single Beam Element.

The matrices of shaft elements represent the stiffness, damping, mass, and gyroscopic effect. The gyroscopic effect also arises in the shaft, although this effect is small without a large shaft diameter. For a Bernoulli element is the stiffness matrix defined in Equation 2.50. The matrix for a Timoshenko and Bernoulli element with stiffness, gyroscope, and mass matrix is defined in Appendix A. The Timoshenko formulation has extra terms that account for the shear deformation.

$$K_B = \frac{E_e I_e}{l_e} \begin{bmatrix} 12 & 0 & 0 & 6l_e & -12 & 0 & 0 & 6l_e \\ 0 & 12 & -6l_e & 0 & 0 & -12 & -6l_e & 0 \\ 0 & -6l_e & 4l_e^2 & 0 & 0 & 6l_e & 2l_e^2 & 0 \\ 6l_e & 0 & 0 & 4l_e^2 & -6l_e & 0 & 0 & 2l_e^2 \\ -12 & 0 & 0 & -6l_e & 12 & 0 & 0 & -6l_e \\ 0 & -12 & 6l_e & 0 & 0 & 12 & 6l_e & 0 \\ 0 & -6l_e & 2l_e^2 & 0 & 0 & 6l_e & 4l_e^2 & 0 \\ 6l_e & 0 & 0 & 2l_e^2 & -6l_e & 0 & 0 & 4l_e^2 \end{bmatrix} \tag{2.50}$$

When a system has more than 1 element, the mass matrix must be assembled. Figure 2.12 shows the placement of the local stiffness matrices relative to the global one. The unoccupied spaces in the stiffness matrix are zero.

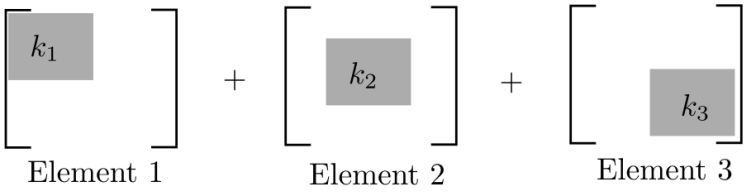


Figure 2.12: Assembly of the global stiffness matrix.

The global stiffness matrix with 3 beam elements is shown in more detail in Figure 2.13, where the stiffness for each degree of freedom is assembled.

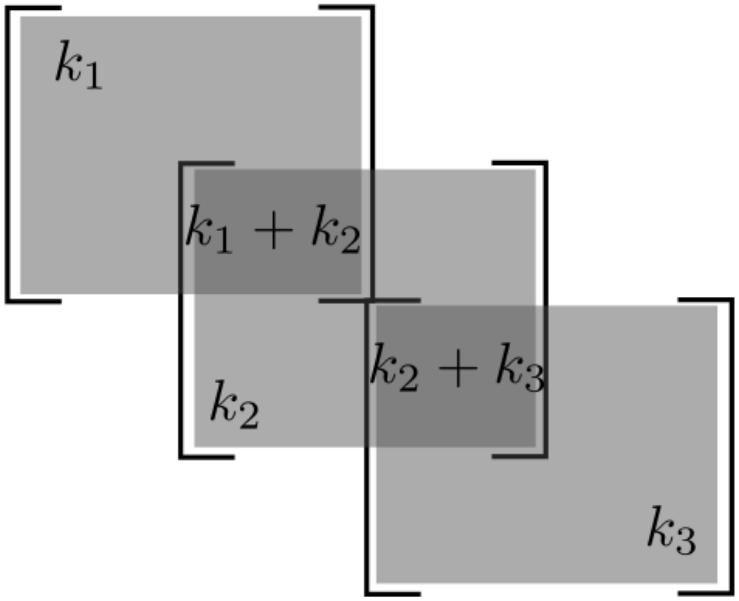


Figure 2.13: Assembled stiffness matrix.

The bearing- and disc matrix are modeled as a lumped matrix parameter. The bearing stiffness is defined in Equation 2.51. These properties are added to the nodes that have the properties of stiffness, gyroscopic effect, or damping.

$$K_{Bearing} = \begin{bmatrix} k_{ui} & 0 & 0 & 0 \\ 0 & k_{vi} & 0 & 0 \\ 0 & 0 & k_{\theta i} & 0 \\ 0 & 0 & 0 & k_{\psi i} \end{bmatrix} \tag{2.51}$$

Chapter 3

Method

In this chapter, the methods for conducting the analysis in Fedem and NX are presented. Also, the analytical method is described as well as the physical rig.

3.1 Physical Rig

To obtain results from the physical rig, it is crucial to connect the screw to the flywheel prior to initiating rotation. The computer must be connected to the sensors, which are positioned on a sensor support attached to the rig. The rig is equipped with a rotating switch that controls the shaft's speed. By manipulating the switch, the desired velocity of the shaft is achieved.

Once the shaft reaches a velocity corresponding to its natural frequency, it begins to produce sound and vibrations. These displacements are recorded by the sensors, which transmit the data to the computer, enabling the generation of plots depicting displacement over time.

Figure 3.1 displays the Bently Nevada RK0 rig used in the analysis. The figure also highlights the key components employed in the simulation and provides guidance on obtaining the desired results.

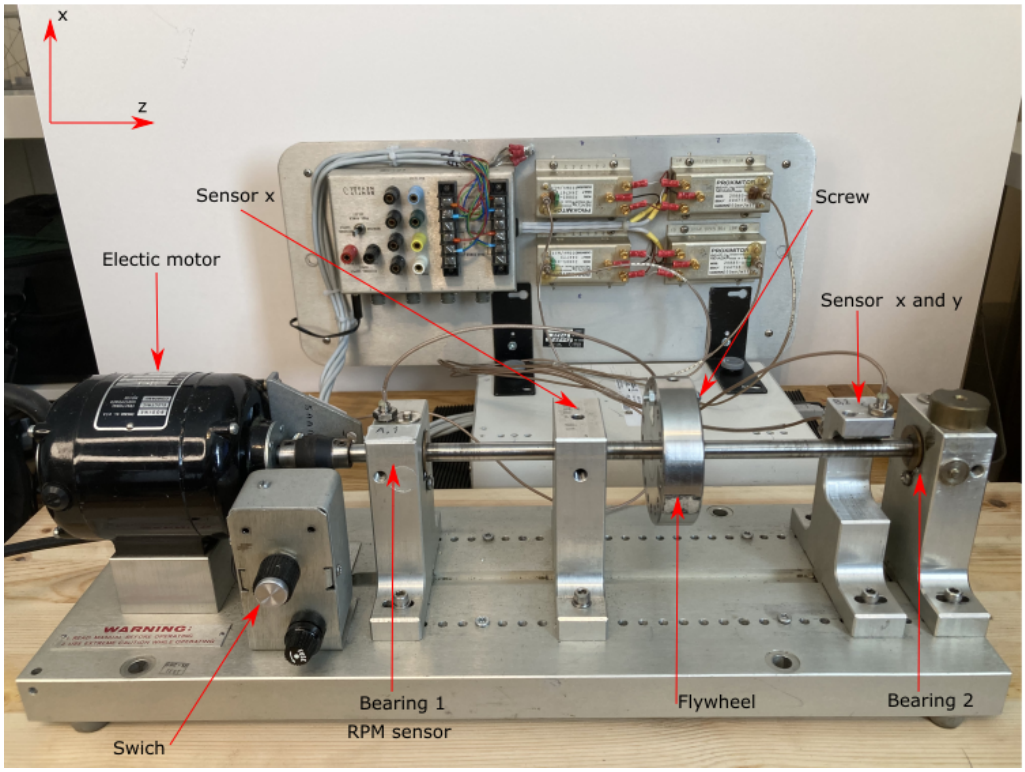


Figure 3.1: Overview of the different components in the physical rig.

3.2 NX Nastran

The simulation in NX aims to obtain results in terms of displacement versus time and the Campbell diagram, which illustrates the relationship between natural frequency and velocity (RPM) to determine critical speed.

3.2.1 1D BEAM Element

To define a 1D element in NX, it is advantageous to consider a rotating axis. As the solver rotates around the z-axis, the simplest approach is to utilize the rotating axis along the z-axis. Figure 3.2 illustrates the model of the element presented in this chapter

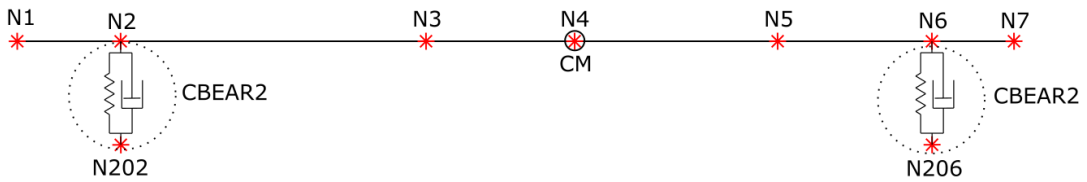


Figure 3.2: Definition of beam element in NX. The node N2 and N202, as well as node N6 and N206 are positioned at same location

Define Nodes

Nodes were manually created in the analysis by specifying their coordinates relative to the start and end points of the shaft. In this case, seven nodes were placed along the shaft axis and connected by beam elements. The specific node shown in Figure 3.2 corresponds to the location where the bearings, mass element, and displacement sensors are positioned. An additional node was necessary at the bearing location to establish the connection between the node from the beam element and the bearing element.

Define CBEAM Element and PBEAML Properties

The CBEAM element is manually created between the specified nodes, such as between N2 and N202, and between N5 and N205 as shown in Figure 3.2. It serves to connect two nodes using a CBEAM element.

In addition to creating the CBEAM element, it is necessary to define its properties. This can be done by accessing the *Physical Properties* and selecting PBEAML as the type of property. By doing so, the properties of the CBEAM element can be created. It is important to define the geometry of the section, which can be achieved by either using a 1D element section to define the section or by accessing the *Show Section Manager* to create a section and define the cross-section properties. Since it is a 1D element, there is no previously defined geometry, and the material property has not been defined yet. The material properties also need to be specified by selecting the appropriate material in the PBEAM properties section.

For each collector, the BEAM property needs to be defined, ensuring that it matches the properties defined in PBEAML. In this collector, all elements will have the same properties as defined. If different properties are required, a new collector must be created to accommodate those specific properties.

Define CBEAR Element and PBEAR Properties

The CBEAR element is defined between two nodes that are positioned at the same location. The first node, known as the source node, represents the fixed boundary condition node, while the second node, known as the target node, is connected to the CBEAM element.

To define the properties and collector for the CBEAR element, you need to select a new collector and then create the physical properties. This allows you to define the stiffness, damping, and mass matrix for the CBEAR element. The properties of the first bearing can be defined within the same collector as the second bearing since they will have the same properties, considering they are part of the same collector.

Define 0D Element

The 0D element must be defined manually since the 0D element on the home page will make a new node that is not connected to the node on the beam. A 0D element is created by going to *Nodes and Element* and then selecting *Choose Element*. The element family is 0D and the element properties are CONM2. To define the properties of this mesh, the command *edit mesh associated data* is first used, and then the mass of the flywheel and the moment of Inertia (Ixx, Iyy, and Izz) are defined. For this simulation the following parameters were used for the flywheel; Ixx of $244.1036kg * mm^2$, Iyy of $244.1036kg * mm^2$ and Izz of $450.3561kg * mm^2$ Ixx and Iyy must be exactly the same because otherwise, the simulation does not run. The flywheel mass is defined at the same node with a value of 0.628944kg.

3.2.2 Simulation of 1D Beam Element

Define Solution and Solution Option

To start the simulation, it is necessary to create a solution in NX Nastran. In this particular problem, a Transient Response Analysis (SOL414,129) and Complex Modal Analysis (SOL414, 110) are used. Additionally, some data must be defined to solve the problem effectively.

Firstly, in the *Case Control* section, the *Bulk Data Echo Request* and *Output Request* must be specified. These settings determine which data will be echoed and what output will be generated during the simulation.

Since this is a rotating problem, the *Rotor Dynamics Solution Parameters* need to be defined. These parameters are specified in the bulk data section. It is possible to create new parameters or select existing ones if they have been defined before. The key parameters to be defined are the *Starting Speed*, *Step Size*, and *Number of Steps*. The starting speed is 0 rev/min, the step size is 60rev/min and the number of steps is 168 this gives the final velocity over 10000RPM. Additionally, the reference system can be set as either a fixed or a rotating coordinate system. In this problem, a fixed coordinate system is used.

Define Boundary Condition

In order to properly simulate the problem, it is necessary to define the boundary conditions. In this case, the bearing point where the shaft is connected needs to be fixed to the ground. This means that the point or node connected to the CBEAR element should have fixed constraints.

At the motor point, it is not possible for the shaft to translate in the x, y, and z directions. Therefore, this point is fixed for all translations and can only rotate about the z-axis.

For the nodes on the beam, the constraints are set to fix the displacement in the z-direction (DOF3) and rotation around the z-axis (DOF6). This means that the nodes are prevented from moving or rotating in these specific directions [3].

Define Rotor Region and Modeling Assembly

The rotor region is defined by selecting the nodes that are rotating. It is important to exclude the fixed node on the bearing since it does not rotate. In this particular problem, a global coordinate system is chosen as the beam element is defined along the z-axis.

To solve this problem, a Rotor Modeling Assembly needs to be defined. The selected rotor region, as previously defined, is included in this assembly.

3.2.3 Transient Analysis of 1D Beam Element

Define Unbalance Mass

The unbalanced mass needs to be defined in order for the solver to run successfully. This is achieved by applying a load with an unbalanced mass. The node where the mass is defined corresponds to the same node where the flywheel is located. The rotor region, as previously defined, is associated with this unbalance mass. The specific mass used is 3g or 0.003kg, with an eccentricity of 31mm.

Define Output Data

To obtain the displacement at nodes 3 and 5, it is important to define a REPORT for these nodes. This can be done by selecting "REPORT" under the Simulation Object Type and then choosing the desired nodes. Since displacement in two directions is of interest, the report should be defined twice, once for each direction. This allows for capturing the displacement data accurately and separately for each specified node.

Subcase - Nonlinear Dynamics

To solve this problem using the Nonlinear Dynamics approach, it is necessary to define certain control parameters. Specifically, the maximum time step (DTMAX) needs to be determined. Equation 3.1 provides guidance for selecting an appropriate value for DTMAX. In the Nonlinear Control Parameters section, under Automatic Time Stepping, DTMAX should be set to a value higher than what is being solved for [3]. In addition to DTMAX, other parameters related to the simulation were set to default values provided by NX.

$$\Delta t_{max} = \frac{1}{20 \times \Omega_{max}} \tag{3.1}$$

In addition, the duration of the simulation known as the "Time Step Definition" should be set to 50 seconds. This will also specify the duration of each time step. The Number of Increments should be set to 1, indicating that the analysis is performed in a single increment. These parameters ensure that the nonlinear dynamics analysis is conducted accurately and efficiently.

Solve

To successfully solve the problem, it is crucial to ensure that all the sections described are active and properly defined. A warning message will appear if any of these sections are not defined in the active solution, and the solution will not run until they are addressed. If any required materials or parameters are missing, they can be easily added by dragging them down to the active solution, provided that they have been defined beforehand. This ensures that all necessary components are included and that the solution can proceed without any issues.

Plotting

To plot the graph of the transient analysis, you need to select the displacement data that was previously defined in the Report section. By selecting this data, you can generate a graph that represents the displacement of the shaft at a specific node over time. Additionally, by applying the Fast Fourier Transform (FFT) to the displacement data, you can obtain a frequency function that shows the dominant frequencies present in the system. This allows for a more comprehensive analysis of the system's dynamic behavior.

3.2.4 Complex Modal Analysis on 1D Beam

To generate a Campbell diagram, the Complex Modal Analysis solver is used. The process for conducting a complex modal analysis is similar to that of a transient analysis. The Bulk Data parameters and Solution Options need to be defined in the same way as in the transient analysis.

Regarding the boundary conditions, they should be applied to the active solution. This can be done by dragging the boundary conditions to the active step in the simulation process.

Outputs Parameters

Complex Modal Analysis is a parameter that needs to be defined in the subcase “Modal Complex” and edited accordingly. Two specific parameters that need to be defined are the Real and Complex Eigenvalue methods. Additionally, the number of Desired Modes should be set to 4 for both methods.

Plotting

To plot the results, select the eigenvalues of interest to obtain the eigenvalue at different speeds and observe the change in speed.

3.3 Fedem

Fedem, which stands for Finite Element Dynamics in Elastic Mechanisms, is a finite element method utilized for analyzing the dynamic behavior and structural simulation of elastic structures and mechanisms. Fedem employs a nonlinear formulation, making it suitable for handling both displacement and rotation. One of the notable features of Fedem is its control system, which allows for the inclusion of sensors and controllers in the analysis [4].

3.3.1 Model Reduction in Fedem

Fedem utilizes Component Mode Synthesis (CMS) to reduce computational time by replacing the internal nodal degrees of freedom (DOF) with a set of static and component modes [4]. This technique allows for the reduction of finite element models into superelements, which consist of external nodes positioned at the interfaces connecting different parts of the mechanism [4]. The model reduction process involves creating superelement mass and stiffness matrices, which are then connected to the overall system’s mass and stiffness matrices during the simulation [4].

3.3.2 Model Setup

Mesh

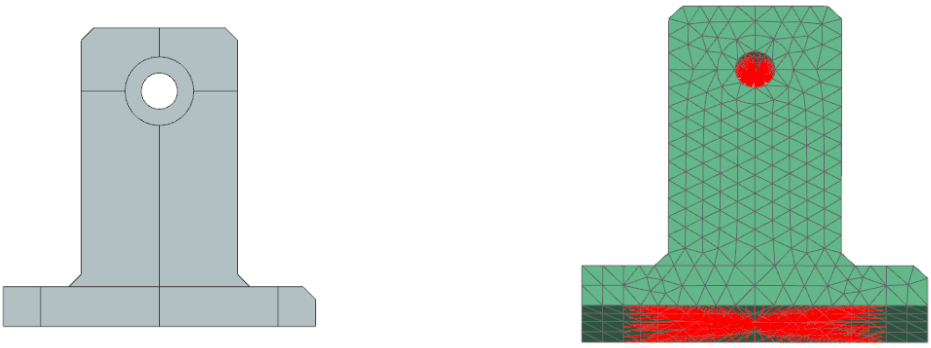
To create a Fedem mesh, the meshing process is performed in NX, and then the mesh is imported into Fedem. The parts excluding the shaft are imported as a Nastran bulk file from NX, while the shaft itself is modeled using beam elements within Fedem.

In Table 3.1, the element size and properties are defined.

Table 3.1: Mesh parameters

Part:	Mesh type:	mesh size
Table	Swept mesh (CQuad4)	8mm
Motor	Swept mesh (CQuad4)	8mm
SupportMotor and SupportEnd	CTETRA(10)	8mm
SupportSlotted	Free mesh CTETRA(10)	8mm
SupportSensor	Free mesh CTETRA(10)	8mm
Flywheel	Free mesh CTETRA(10)	8mm

The SupportEnd and SupportMotor are defined as the same part. Figure 3.3 shows the mesh of this part, including a spider in the hole that defines the bearing connection to the shaft. At the bottom, there is also a spider defined, which is connected to the table support.

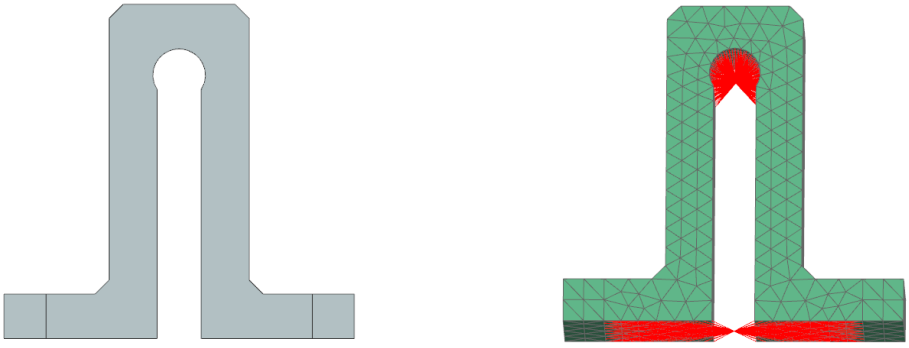


(a) Original SupportEnd and SupportMotor

(b) Mesh with spider

Figure 3.3: Meshed SupportEnd and SupportMotor

The SupportSlotted is shown in Figure 3.4, which displays the mesh of this part. It includes a spider used to connect the part to the table and a spider at the shaft location.

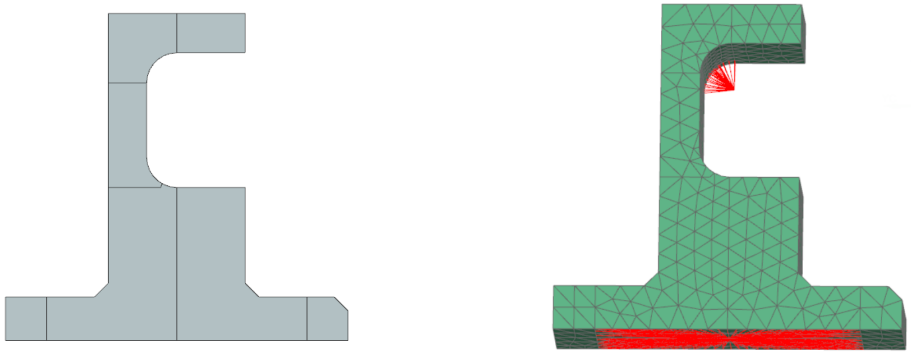


(a) Original SupportSlotted

(b) Mesh with spider

Figure 3.4: Meshed SupportSlotted

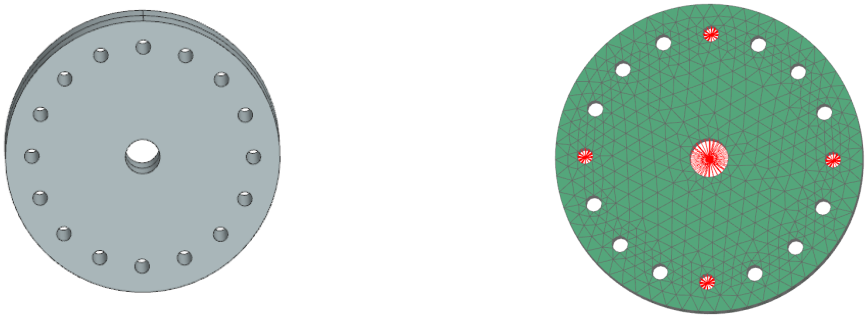
The SupportSensor is meshed with a spider at a surface where the sensor is placed. Figure 3.5 illustrates the meshing of this part. Additionally, there is a spider at the bottom of the SupportSensor, which serves to connect it to the table.



(a) Original SupportSensor (b) Mesh with spider

Figure 3.5: Meshed SupportSensor

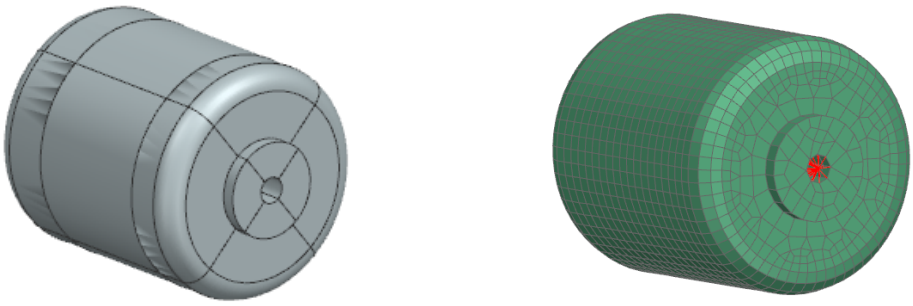
The Flywheel part is meshed with a spider in the shaft hole and in four of the small holes around it. Figure 3.6 displays the defined model, both as a part and the corresponding meshed representation. In one of the spiders located in the upper small hole, a concentrated mass of 3g (0.003kg) is defined.



(a) Original Flywheel (b) Mesh with spider

Figure 3.6: Meshed Flywheel

The motor part is freely meshed and can be seen in Figure 3.7. A spider is defined within the circular area that connects the motor shaft to the motor.



(a) Orginal Motor

(b) Mesh with spider

Figure 3.7: Meshed Motor

The table is defined as a swept mesh and includes spiders at the positions where the supports are connected. This can be seen in Figure 3.8.

(a) Orginal Table

(b) Table mesh with spider

Figure 3.8: Meshed Table

Joints in Fedem

Joints are used in Fedem to connect components together. There are seven different types of joints available in Fedem, numbered from zero to six, depending on the desired joint behavior [4]. These joints are connected using triads.

Figure 3.9 shows the joints used in the Fedem model of Bently Nevada RK0.

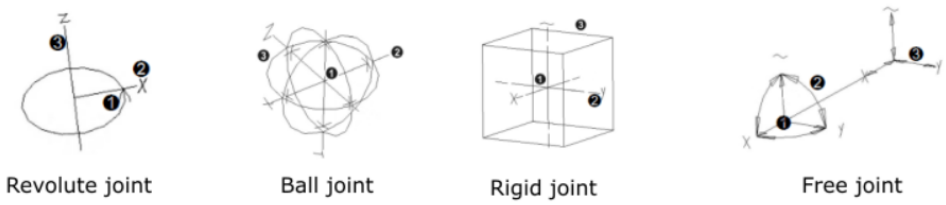


Figure 3.9: Four different joints used in the Bently Nevada rig model in Fedem - retaken from [4].

Revolute Joint in Fedem has one degree of freedom (DOF). This allows one part to rotate about a specified axis relative to another common axis.

Ball Joint in Fedem has three degrees of freedom (DOF). This allows one part to rotate about three specified axes relative to another part.

Rigid Joint in Fedem is used to restrict all displacements between parts, creating a stiff connection. This joint eliminates any relative motion between the connected parts, resulting in a rigid connection.

Free Joint in Fedem is used to allow for various types of mechanism motion. This joint has six degrees of freedom, enabling movement in all directions and rotations around all axes. It provides flexibility in defining complex motions and kinematic behavior within the model.

Mechanical Assembly

The Fedem model for the Bently Nevada RK0 rig is assembled using the joints as described in the previous section. Figure 3.10 illustrates the placement of the different joints within the model. Additionally, a sensor is defined within the model to measure and register the displacement at a specific node.

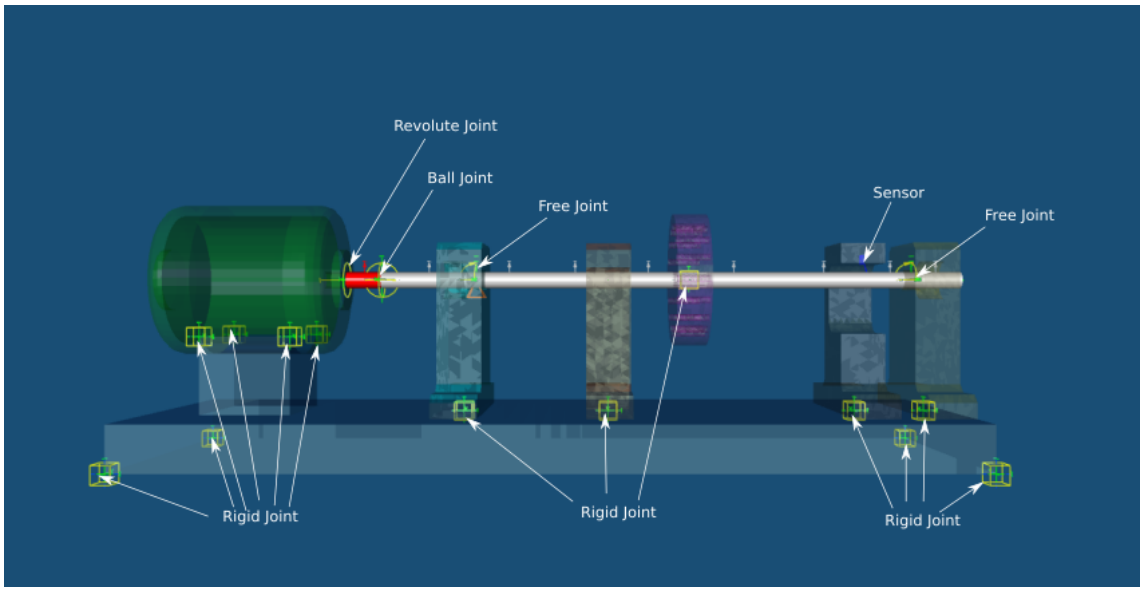


Figure 3.10: The placement of the different joints in the Fedem model.

The ball joint is defined between the motor shaft and the main shaft in the Fedem model. The degrees of freedom (DOFs) of the ball joint at this connection point are described in Table 3.2.

Table 3.2: Ball joint

Joint DOF	Constrain	Spring	Damper
Rx	Free	-	-
Ry	Free	-	-
Rz	Spr/dmp	100e3	0

The Free joint is placed at the bearing locations in the Fedem model, and these free joints are associated with bearing properties such as springs and dampers. The degrees of freedom (DOFs) of the free joints in the model are described in Table 3.3 for the first simulation.

Table 3.3: Free Joint Orginal values

Joints DOF	Bearing 1			Bearing 2		
	Constrain	Spring [N/m]	Damper [Ns/m]	Constrain	Spring [N/m]	Damper [Ns/m]
Tx	Spr/Dmp	10e6	10e3.	Spr/Dmp	10e6	0
Ty	Spr/Dmp	10e6	10e3	Spr/Dmp	10e6	0
Tz	Spr/Dmp	1e6	1000	Spr/Dmp	1e6	10000
Rx	Fixed	-	-	Fixed	-	-
Ry	Fixed	-	-	Fixed	-	-
Rz	Free	-	-	Free	-	-

In Table 3.4, the values of the bearing properties used in NX and the analytical solution are provided. This ensures that the properties of the bearings are consistent and equal in both approaches.

Table 3.4: Free Joint NX values

Joints DOF	Bearing 1			Bearing 2		
	Constrain	Spring [N/m]	Damper [Ns/m]	Constrain	Spring [N/m]	Damper [Ns/m]
Tx	Spr/Dmp	10e6	10e3.	Spr/Dmp	10e6	10e3
Ty	Spr/Dmp	10e6	10e3	Spr/Dmp	10e6	10e3
Tz	Spr/Dmp	1e6	-	Spr/Dmp	1e6	-
Rx	Fixed	-	-	Fixed	-	-
Ry	Fixed	-	-	Fixed	-	-
Rz	Free	-	-	Free	-	-

In Table 3.5, the degrees of freedom (DOF) and the velocity functions associated with the Rz DOF of the revolute joint are provided. The specific functions used for the velocity are described in more detail in the subsequent section.

Table 3.5: Revolute joint

Joint DOF	Velocity
Rz	Reference speed (Function)

The rigid joints are placed between different components and at the corners of the table in the Fedem model.

3.3.3 Control System

The velocity of the shaft is defined to increase linearly over time. This is achieved by setting a slope parameter that determines the rate of increase per second. The details of this parameter can be found in Table 3.6.

Table 3.6: Parameter of the reference speed

Start displacement	0
Slope [rad/s]	104.7
Start of ramp [s]	0.0
End of Ramp [s]	15.0

The linear increase of the shaft velocity occurs during the first 15 seconds of the simulation. The specific velocity profile can be observed in Figure 3.11.

Figure 3.11: Reference speed of Fedem model.

A sensor is placed between two triads to measure the displacement at that specific position.

3.3.4 Analysis Setup

The analyses are conducted using time increments of 0.0002s, starting from time 0s and ending after 12s. For this simulation, the first four eigenmodes were requested. The rest of the parameters were set as default values.

3.4 Analytical

When performing an analytical solution, Python is used for the calculations. The Python script in this thesis is based on code developed by Lars Helge Verde [5], incorporating the theoretical concepts discussed in Chapter 2. The code used in this project is found in Appendix I and includes a setup file for the analysis as well as a system-class file.

3.4.1 Input Parameters

The code requires certain input parameters to be defined in order to obtain accurate results. These parameters provide the necessary information for the calculations and analysis.

Material Parameters:

- E: Young's Modulus
- rho: Density
- v: Poisson's Ratio

Disk

- D: External disk diameters
- d: Internal disk diameter
- th: Disk thickness
- pos: Position of disc
- esp: Initial disc-offset

Shaft

- External diameter

Bearing Stiffness

- kx: [Kx1, Kx2,...]
- ky: [Ky1, Ky2,...]
- kTheta: [KTheta1, KTheta2,...]
- kPsi: [KPsi1, KPsi2,...]

Bearing Damping

- cx: [Cx1, Cx2,...]
- cy: [Cy1, Cy2,...]
- cTheta: [CTheta1, CTheta2,...]
- cPsi: [CPsi1, CPsi2,...]

3.4.2 Extra Parameters

Beam type

- 0 = Bernoulli
- 1 = Thimoshenko

Include Disc

- True (default)
- False

Use Damping

- True (default)
- False

BC Type

- 0 = position at end
- 1 = exactly position
- 2 = closest position

Bearing Position

- Bearing position 1
- Bearing position 2

If the BC type is set to 0, it means that bearings are placed at both ends of the shaft. Although the bearing positions need to be defined for the code to run, they are not directly used in the calculations or analysis.

3.4.3 Make Matrix

The matrices M, K, and G are constructed based on Bernoulli and Timoshenko beam theory for the shaft. The inclusion of the flywheel introduces the gyroscopic effect, which is accounted for in the G matrix. In addition, the stiffness and damping matrices (K and C) are computed for the bearings. These matrices are then added to the global matrix at the nodes where the bearings are located, incorporating their effects into the overall system.

3.4.4 Solve Matrix

After constructing the matrices A and B, the eigenvalue problem is solved to obtain the eigenvalues of the system. Since the eigenvalues are complex, they consist of a real part and an imaginary part. To create the Campbell diagram, the eigenvalues are sorted based on their absolute value, denoted as ω_n. If multiple eigenvalues have the same absolute value, the sorting is done based on the imaginary part. The imaginary part is stored separately for positive and negative values, as the imaginary parts can be positive or negative for the same absolute value. This sorting process allows for a clear representation of the system's eigenfrequencies and critical speeds in the Campbell diagram.

3.4.5 Plot Campbell Diagram

The Campbell diagram is plotted by representing the natural frequencies as a function of the velocity Ω, which is measured in RPM (revolutions per minute). The velocity Ω is defined as a linear function of the parameter n and the natural frequency ω, such that $\Omega = n\omega$. This representation allows for clear visualization of the relationship between the system's natural frequencies and the varying velocity, providing valuable insights into the system's dynamic behavior and critical speeds.

3.4.6 Plot Frequency Response and Operating Deflection

To plot the various graphs, the displacement values for both u and v are solved and obtained by considering the q vector, which represents the displacement and lateral load case b. These displacement values are then used to calculate the amplitude for frequency response analysis and to determine the rotational displacement of the shaft for other plots.

3.4.7 Plot Modeshape

To plot the modeshape, the eigenvalue problem is solved for the specific velocity of interest. This involves determining the eigenvalues and eigenvectors of the system. Once the eigenvalues are obtained, the corresponding eigenvectors represent the modeshapes of the system at that particular velocity. The displacement values for both u and v are then solved using these eigenvectors, and these displacements are plotted to visualize the modeshape of the system at the given velocity.

Chapter 4

Result

This chapter presents the results obtained from the Bently Nevada RK0 rig, as well as the results obtained from the simulations conducted in NX Nastran and Fedem. The NX simulation includes also results from a parameter study. Ultimately, the analytical solution is presented.

4.1 Physical Bently Nevada RK0 Rig

Results from the Bently Nevada RK0 rig are shown in Figure 4.1. It provides data regarding the displacement at various points and the system's response as the velocity increases. The first peak corresponds to the response during an increasing velocity, while the second peak represents the response during a decreasing velocity. The results indicate that the maximum displacement observed is approximately 0.5mm.

Based on the results from the Bently Nevada RK0, an estimated frequency of 75Hz was obtained from the velocity of the shaft.

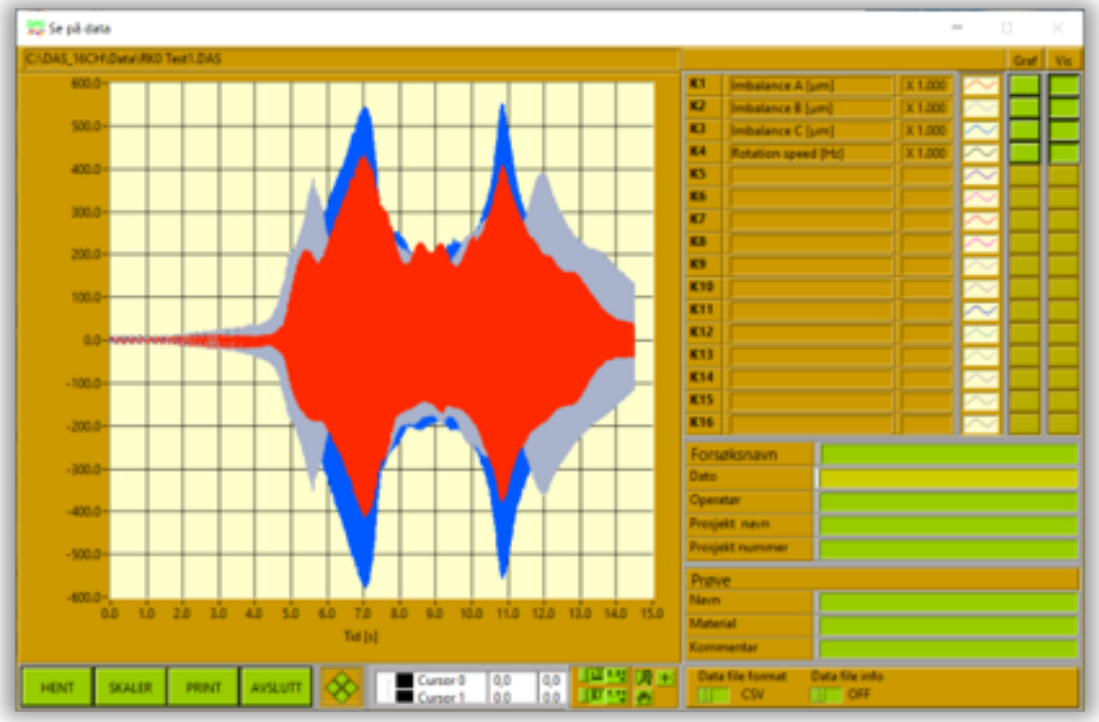


Figure 4.1: Result from Bently Nevada RK0 Rig.

4.2 1D Beam Element in NX Nastran

The results obtained from the analysis conducted using NX Nastran are presented through plots and tables, providing valuable insights into the behavior of the system.

The Campbell diagram in Figure 4.2 illustrates the relationship between the eigenfrequencies and the velocity (RPM). This diagram enables us to observe the critical speed, which occurs at approximately 120 Hz when the velocity intersects the first natural frequency. The eigenvalues obtained from the simulation correspond to the fixed rotation speed

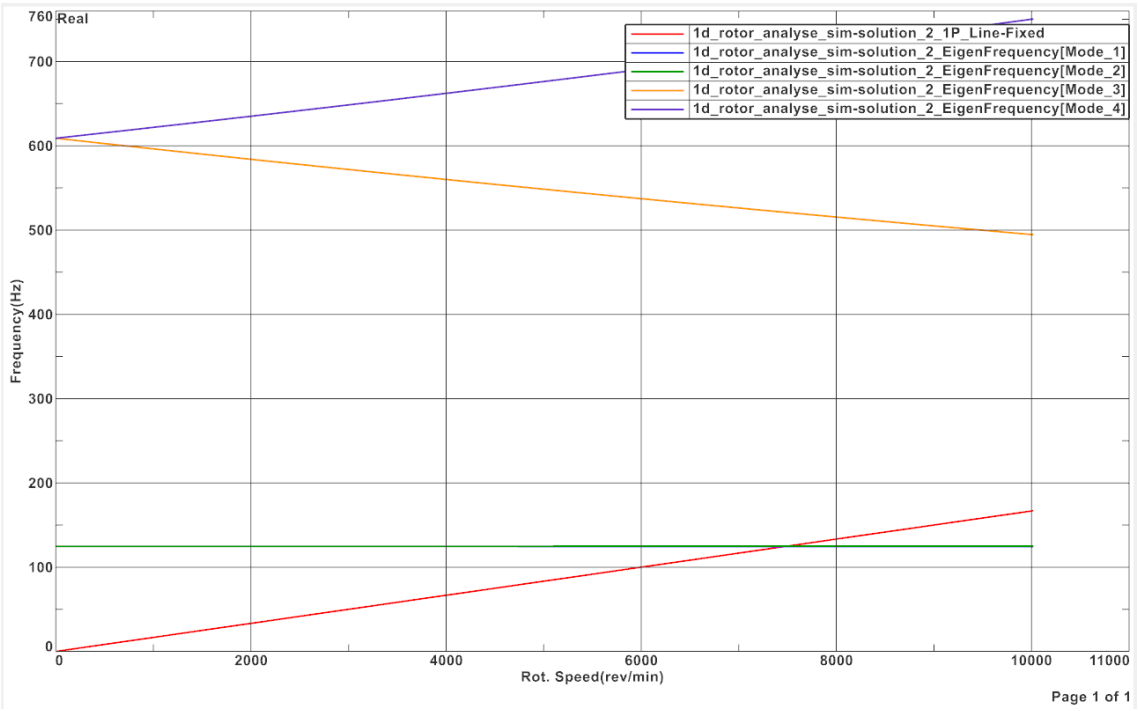


Figure 4.2: Campbell Diagram of 1D Beam Element.

The displacement plot in Figure 4.3 shows the time at which the largest displacements occur. At approximately 36 seconds after the start of acceleration, the maximum displacement is observed, reaching a value of 0.0011 m. Notably, the node closest to the flywheel exhibits the largest displacement in both the x- and y directions.

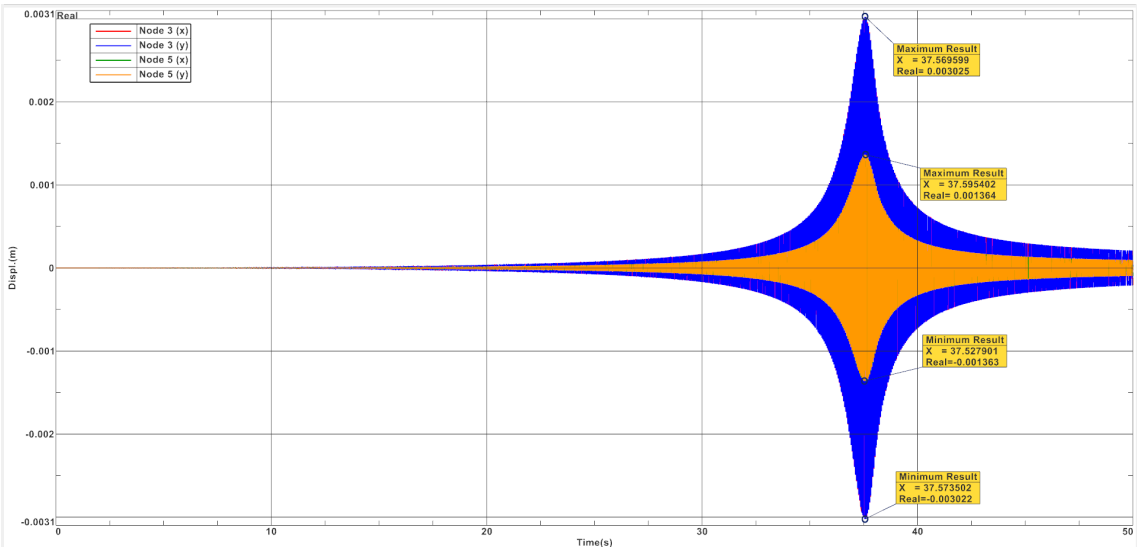


Figure 4.3: Displacement at Node 3 and 5 in x and y direction.

To further analyze the frequency components, a Fast Fourier Transform (FFT) diagram is presented in Figure 4.4. This diagram displays the frequencies and corresponding displacements, revealing a peak displacement at a frequency of 121 Hz.

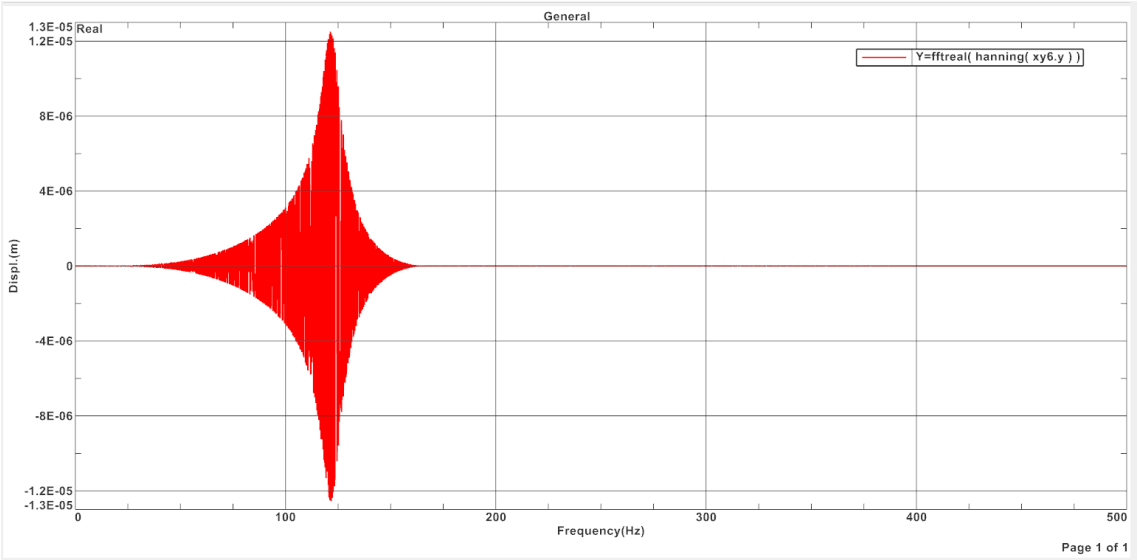


Figure 4.4: FFT diagram

The choice of DTMAX (maximum time step) significantly affects the solution results. Figure 4.5 demonstrates the impact of different DTMAX values on the displacement and vibration levels. Lower DTMAX values result in reduced displacements and increased vibration at various locations.

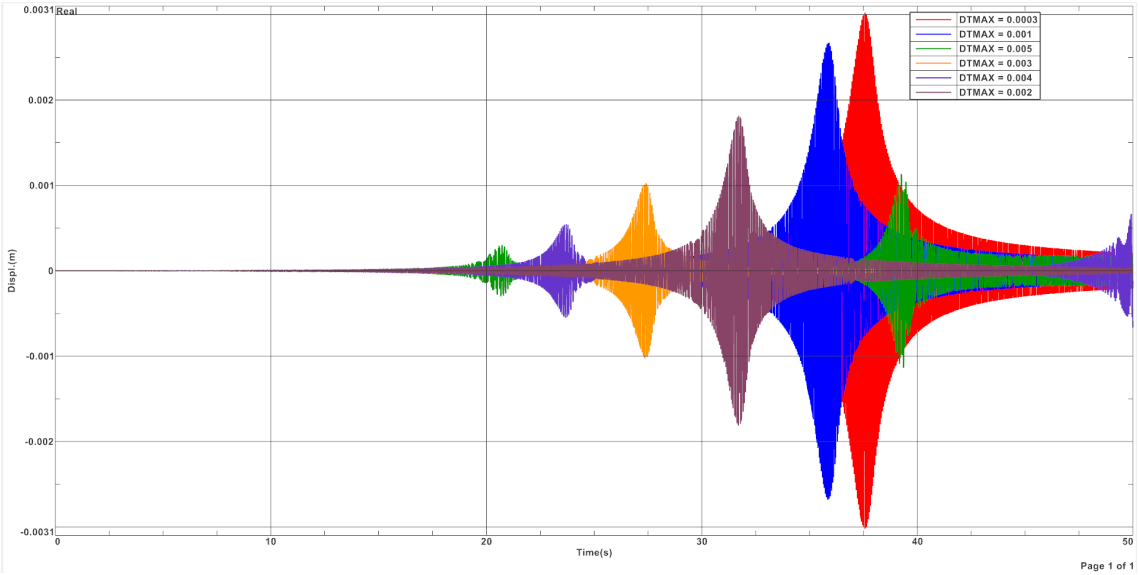


Figure 4.5: Result of 1D Beam element with different DTMAX.

The parameter study of the bearing stiffness and damping provides insights into the time at which the largest displacement occurs. The FFT analysis is used to represent the frequency and displacement in the FFT Diagram.

Table 4.1 presents the results of the damping effects, showcasing the impact on the system's behavior.

Table 4.1: Damping at transient analysis with using DTMAX = 0.001.

Name	K11 and K22 [N/m]	K33 [N/m]	C11 and C22 [Ns/m]	Time [s]	Displacement [m]	Frequency [Hz]
Original [max]	10e6	1e6	10e3	35.909	0.002674	119.78
Original [min]	10e6	1e6	10e3	35.855	-0.002683	120.1
Damp20 [max]	10e6	1e6	20e3	36.238998	0.002771	121.2
Damp20 [min]	10e6	1e6	20e3	36.202	-0.002768	121.02
Damp30 [max]	10e6	1e6	30e3	36.403	0.00393	121.34
Damp30 [min]	10e6	1e6	30e3	36.407001	-0.003393	121.52
Damp40 [max]	10e6	1e6	40e3	36.500999	0.004099	121.7
Damp40 [min]	10e6	1e6	40e3	36.497002	-0.004098	121.74
Damp50 [max]	10e6	1e6	50e3	36.549999	0.004797	121.92
Damp50 [min]	10e6	1e6	50e3	36.554001	-0.004789	121.96
Damp60	10e6	1e6	60e3	-	-	-

Table 4.2 displays the results of the stiffness effects on the bearing. It provides values for the maximum and minimum displacements observed in the system, illustrating the influence of varying stiffness on the behavior of the system.

Table 4.2: Stiffness at transient analysis with using DTMAX = 0.001.

Name	K11 and K22	K33	C11 and C22	Time [s]	Displacement [m]	Frequency [Hz]
Stiff1 [max]	1e6	1e6	10e3	36.279999	0.001035	120.84
Stiff1 [min]	1e6	1e6	10e3	36.243	-0.001035	120.52
Stiff2 [max]	2e6	1e6	10e3	36.132	0.001097	120.84
Stiff2 [min]	2e6	1e6	10e3	36.202	-0.001095	120.52
Stiff3 [max]	3e6	1e6	10e3	36.000	0.00119	120.7
Stiff3 [min]	3e6	1e6	10e3	35.971001	-0.001191	120.38
Stiff4 [max]	4e6	1e6	10e3	35.883999	0.001316	120.7
Stiff4 [min]	4e6	1e6	10e3	35.912998	-0.001315	120.24
Stiff5 [max]	5e6	1e6	10e3	35.883999	0.001469	119.02
Stiff5 [min]	5e6	1e6	10e3	35.885	0.001472	119.54
Stiff6 [max]	6e6	1e6	10e3	35.800999	0.001658	119.02
Stiff6 [min]	6e6	1e6	10e3	35.855	-0.001657	119.54
Stiff7 [max]	7e6	1e6	10e3	35.800999	0.001875	119.02
Stiff7 [min]	7e6	1e6	10e3	35.855	-0.00187	119.54
Stiff8 [max]	8e6	1e6	10e3	35.800999	0.002118	119.78
Stiff8 [min]	8e6	1e6	10e3	35.855	-0.002114	119.54
Stiff9 [max]	9e6	1e6	10e3	35.826	0.002384	119.78
Stiff9 [min]	9e6	1e6	10e3	35.855	-0.002387	119.96
Original [max]	10e6	1e6	10e3	35.909	0.002674	119.78
Original [min]	10e6	1e6	10e3	35.855	-0.002683	120.1
Stiff11 [max]	11e6	1e6	10e3	35.909	0.002998	119.78
Stiff11 [min]	11e6	1e6	10e3	35.880001	-0.002989	120.1
Stiff12 [max]	12e6	1e6	10e3	35.909	0.003325	119.92
Stiff12 [min]	12e6	1e6	10e3	35.938	-0.003332	120.1
Stiff13 [max]	13e6	1e6	10e3	35.966999	0.003682	119.92
Stiff13 [min]	13e6	1e6	10e3	35.995998	-0.003672	120.24
Stiff14 [max]	14e6	1e6	10e3	35.909	0.002674	120.06
Stiff14 [min]	14e6	1e6	10e3	35.855	-0.002683	120.24
Stiff15 [max]	15e6	1e6	10e3	35.909	0.002998	120.06
Stiff15 [min]	15e6	1e6	10e3	35.880001	-0.002989	120.38
Stiff16 [max]	16e6	1e6	10e3	35.909	0.003325	120.2
Stiff16 [min]	16e6	1e6	10e3	35.938	-0.003332	120.38
Stiff17 [max]	17e6	1e6	10e3	35.966999	0.003682	120.2
Stiff17 [min]	17e6	1e6	10e3	35.995998	-0.003672	120.38

4.3 Simulation in Fedem

The results from the Fedem simulation are presented in this section through several plots.

4.3.1 Original Bearing Parameters

Figure 4.6 illustrates the natural frequencies obtained from the simulation. The frequencies are observed at approximately 136.6Hz, two around 186.5Hz, and the last one at 329.3Hz.

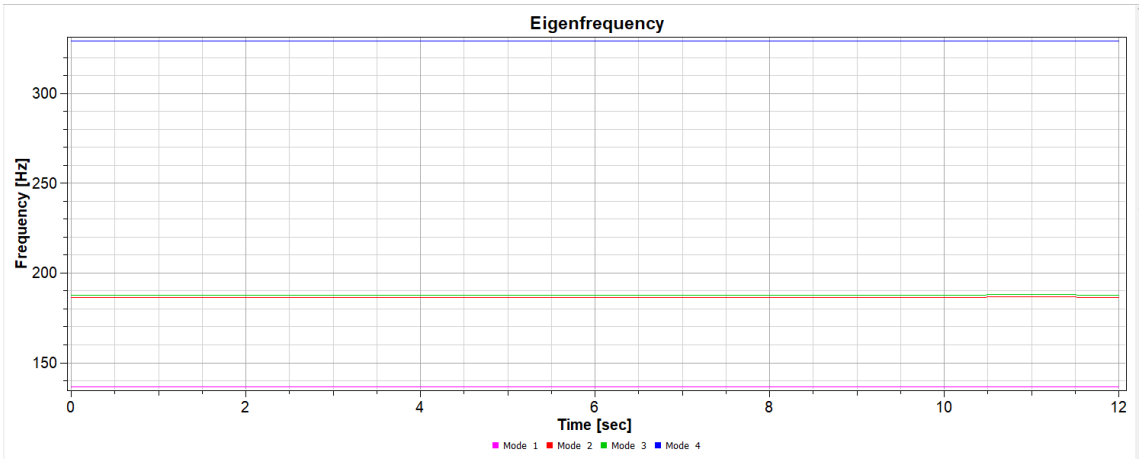


Figure 4.6: Result of the natural frequencies in Fedem for the original bearing properties.

Figure 4.7 depicts the velocity of the rotor as a linear line. The natural frequencies are represented by the horizontal line. This allows us to determine the time when the shaft reaches the same frequencies as the natural frequencies.

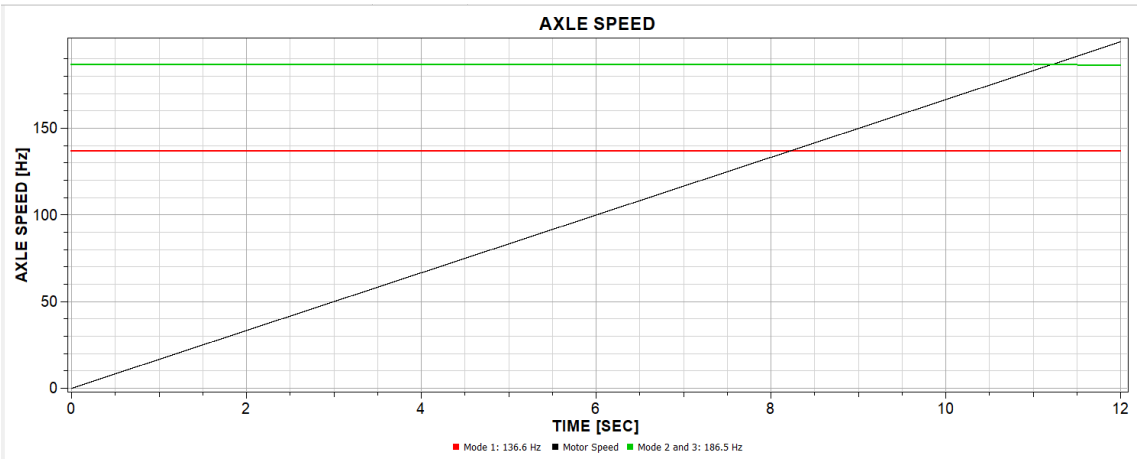


Figure 4.7: Result of Time versus Frequency for the original bearing properties.

In Figure 4.8, the displacement from the Fedem simulation is shown. Comparing this result with Figure 4.7, it can be observed that the displacement occurs at a frequency of 186 Hz.

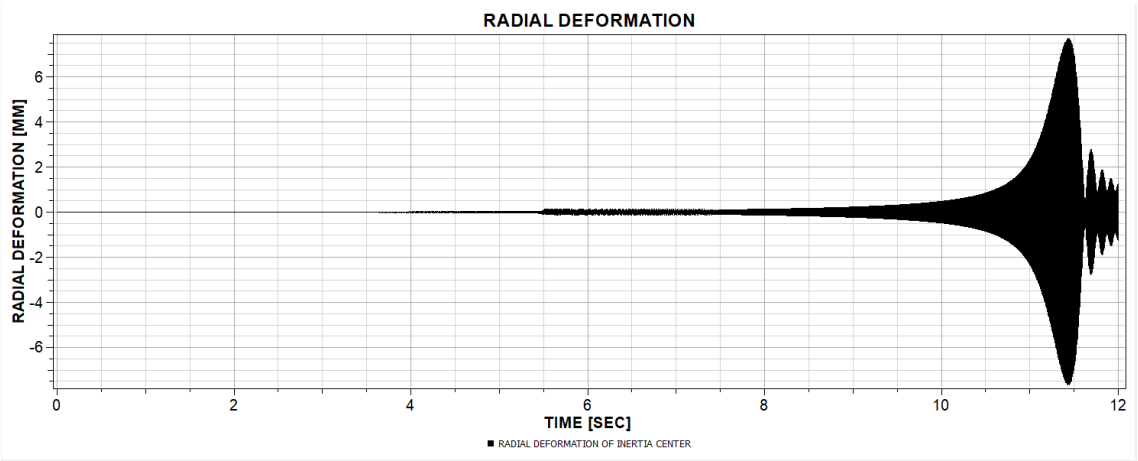


Figure 4.8: Result of displacement in Fedem for the original bearing properties.

In Figure 4.9, the FFT diagram of the displacement shown in Figure 4.8 is presented with respect to frequency. This result indicates a prominent frequency peak around 186Hz.

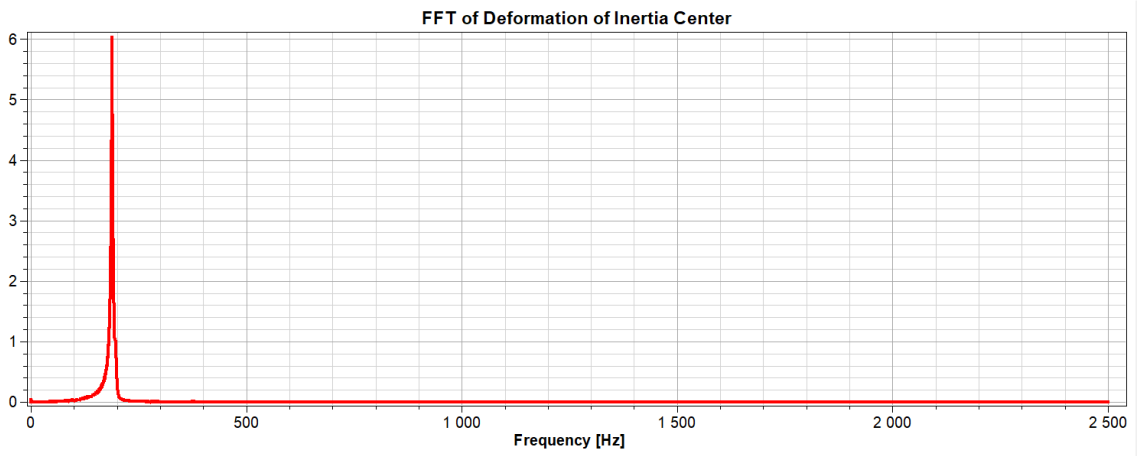


Figure 4.9: Result of the frequency as FFT diagram for the original bearing properties.

4.3.2 NX Bearing Parameters

In Figure 4.10, the natural frequencies from the Fedem simulation are shown. These frequencies are observed at approximately 136.6 Hz, two around 186.5 Hz, and the last one at 329.3 Hz, which are consistent with the original bearing parameters.

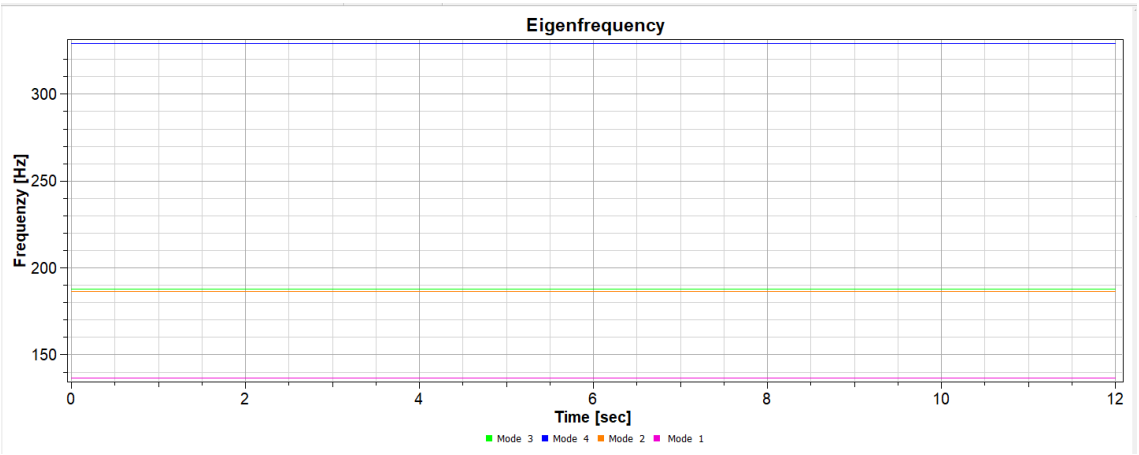


Figure 4.10: Result of the natural frequencies in Fedem for the NX bearing properties.

In Figure 4.11, the velocity of the rotor is represented by a linear line, while the natural frequencies are indicated by horizontal lines. This plot shows the time instances when the shaft reaches the same frequency as the natural frequencies.

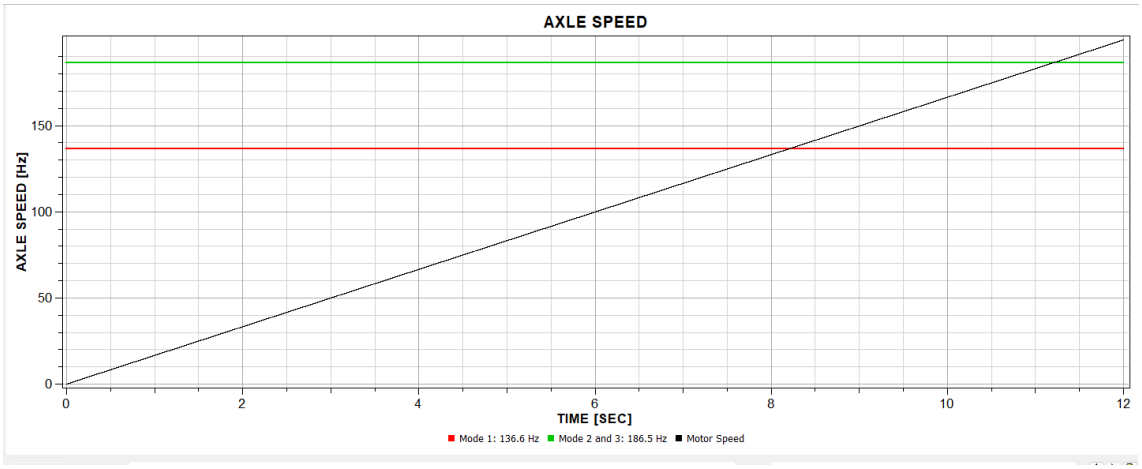


Figure 4.11: Result of Time versus Frequency for the NX bearing properties.

In Figure 4.12, the displacement of the Fedem simulation is represented. Comparing this result with Figure 4.11, it can be observed that the displacement occurs at a frequency of 186.5Hz, which is consistent with the original bearing properties.

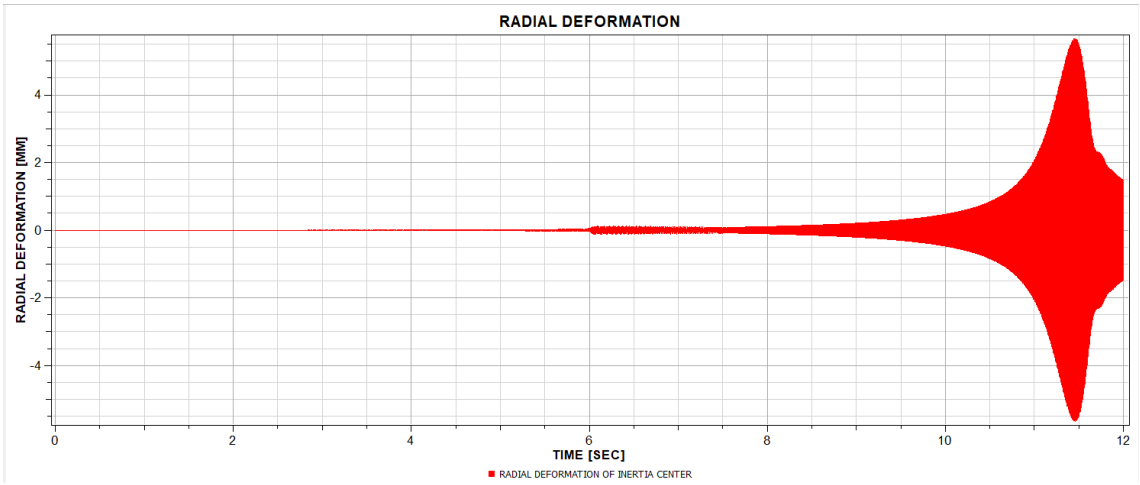


Figure 4.12: Result of displacement in Fedem for the NX bearing properties.

In Figure 4.13, the FFT diagram of the displacement shown in Figure 4.12 is presented with respect to frequency. This result indicates a prominent frequency peak around 186Hz, which aligns with the original bearing parameters.

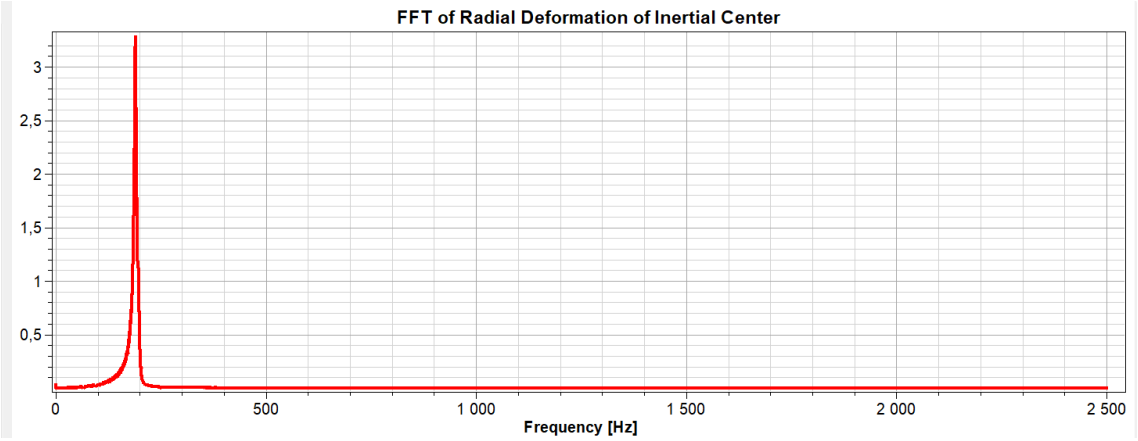


Figure 4.13: Result of displacement in Fedem for the NX bearing properties.

4.4 Analytical Result

In this section, the results of the analytical solution, which was solved using Python, are presented.

In Figure 4.14, the Campbell diagram from the analytical solution is presented. The figure shows that the natural frequencies are observed at approximately 94.0Hz, 160.3Hz, and 161.9Hz.

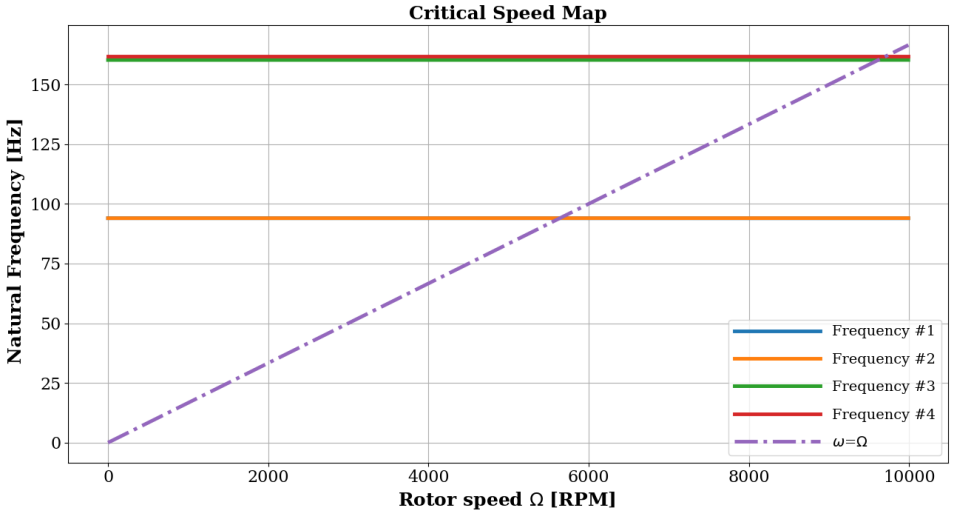


Figure 4.14: Campbell diagram of the analytical solution.

Figure 4.15 and 4.16 present the response at node 3 and 5, respectively, in the system. The results show that these nodes exhibit a maximum amplitude response at approximately 5640RPM, corresponding to the first natural frequency at 94Hz.

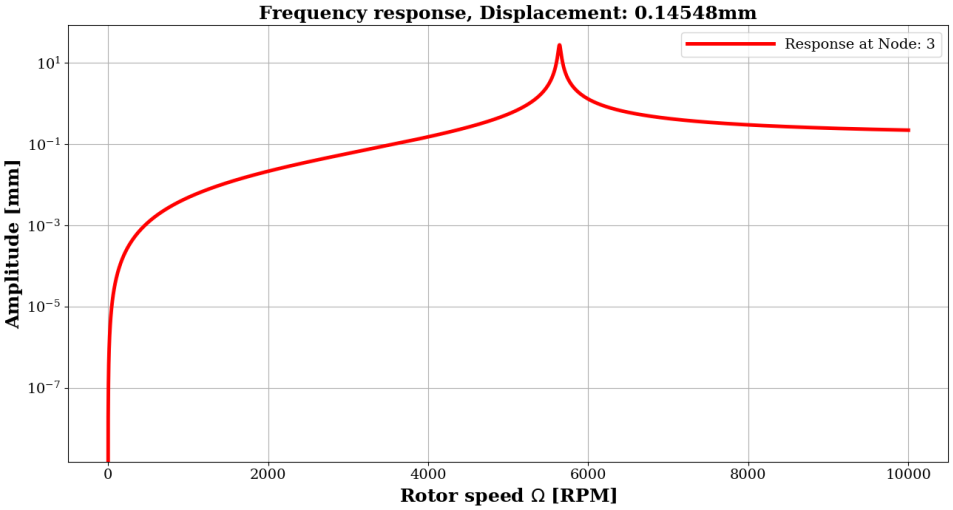


Figure 4.15: Frequency response for analytical solution for node 3.

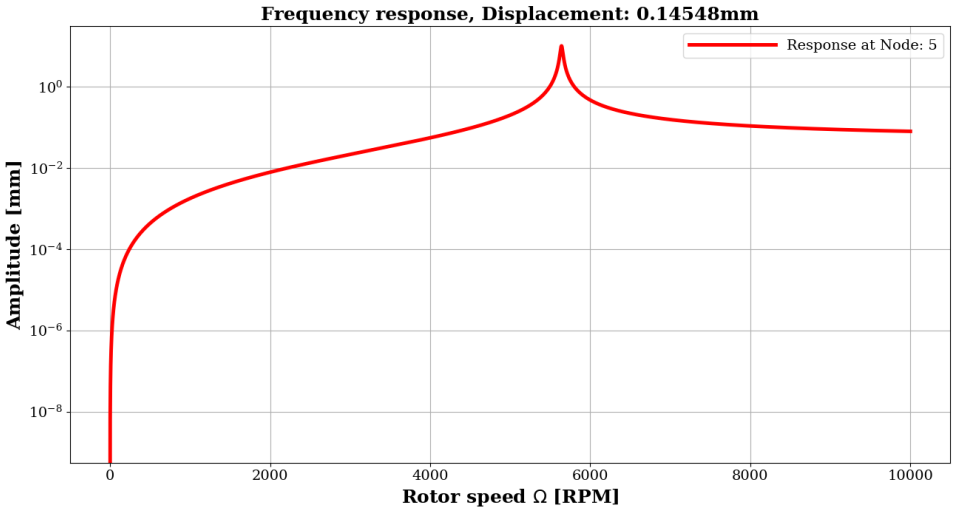


Figure 4.16: Frequency response for analytical solution for node 5.

In Figure 4.17, the deflection of the rotor at a velocity of 5640RPM, representing the critical speed, is displayed.

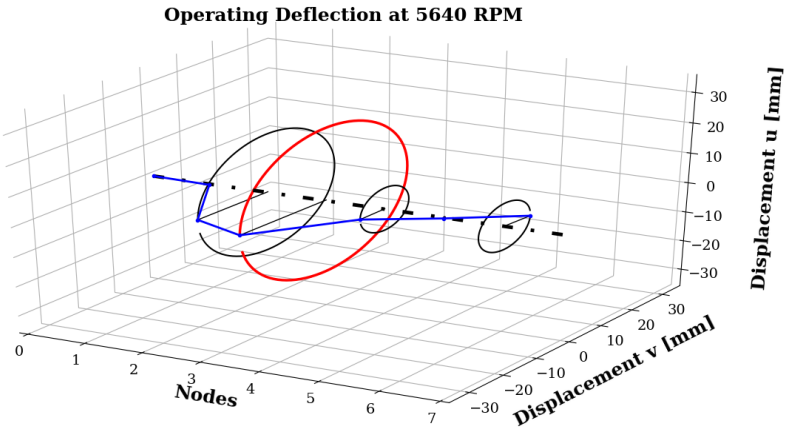


Figure 4.17: Result operating deflection on the analytical solution.

Chapter 5

Discussion

5.1 Bently Nevada RK0

5.1.1 Displacement

The displacement, as measured by the sensors, reached a maximum value of approximately 1.1 mm. This displacement is visually noticeable and not too small to be observed.

5.1.2 Frequency

The frequency is determined by analyzing the displacement and velocity of the shaft. This allows for the identification of bending modes. It is important to note that the accuracy of the frequency measurement can be affected if the sensors are not properly calibrated. Calibration plays a crucial role in obtaining accurate and reliable results.

5.2 NX

5.2.1 Effect of the Bearing Stiffness and Damping

The bearing stiffness and damping properties play a significant role in the behavior of a rotating system. In this section, the effects of these properties on the system's dynamics are discussed, particularly focusing on the rotation aspect.

Bearing Stiffness

The analysis of bearing stiffness produces two types of results: transient response and a Campbell diagram. These results provide valuable insights into the system's behavior and characteristics.

During the transient analysis, the impact of bearing stiffness on the rotor system was investigated. It was observed that increasing the stiffness led to larger displacements. However, it was also noted that excessively high stiffness values, exceeding 17e6N/m, caused the simulation to terminate prematurely before reaching the peak displacement. This limitation occurs due to the challenges in accurately simulating the system with extremely high stiffness values.

Furthermore, it was observed that as the bearing stiffness increased, the frequency initially decreased before eventually increasing again. However, the magnitude of this frequency variation was less than 1Hz, making it challenging to draw definitive conclusions from these slight changes.

The results from the Campbell diagram for different stiffness values indicate that lower stiffness (below 1e6N/m) leads to noticeable changes in the eigenfrequencies of the system. This observation is expected, as lower stiffness allows for more flexibility and leads to lower natural frequencies. However, higher stiffness values did not have a significant effect on the eigenfrequencies, as shown in the Campbell diagram in Appendix H.

The small change can be a result of the shaft stiffness being the most dominant compared to the stiffness of the bearings. The deflection of the shaft is hence governed by the stiffness of the shaft.

Bearing Damping

When the bearing damping is altered, changes are observed in both the Campbell diagram and the transient response.

In the analysis of the transient response, it has been observed that there is a difference in the magnitude of displacement between node 3 and node 5. The FFT diagram shows the same frequencies for both nodes, which is expected since the largest displacement occurs at the same time for both nodes. The difference in displacement magnitude between these two points can be attributed to the fact that node 5 is closer to bearing 2 and has a longer distance from the flywheel compared to node 3. Additionally, the displacement from bearing 1 to node 3 is longer than the displacement from bearing 1 to node 5, resulting in a larger displacement at node 3 compared to node 5.

When the damping is increased, a larger displacement is observed in the transient response. However, this increase in displacement does not result in a significant change in the frequencies as shown in the FFT diagram. The frequencies remain around 120-122Hz. There could be several reasons for this minimal change in frequencies. One possibility is that the value of DTMAX does not coincide with the point of maximum displacement or it occurs during a later oscillation. Ad-

ditionally, it should be noted that the simulation may crash when the damping value becomes too high. The last successful simulation was performed using a damping value of 50e3Ns/m.

When examining the Campbell diagrams, it can be observed that increasing the damping does not significantly alter the eigenfrequencies of Mode 1 and Mode 2. However, it does have an impact on Mode 3 and Mode 4, causing a decrease in frequency. The results from Modes 1 and 2 indicate that there is no substantial change in frequencies in the transient analysis. It is worth noting that the Campbell diagram shows slightly higher frequencies around 124 Hz compared to the frequencies obtained from the transient response analyses, which are around 120-122 Hz. One possible reason for the higher frequencies in the Campbell diagram is that the complex modal analysis does not account for the eccentric load.

5.2.2 Boundary Condition

The decision to fix DOF3 and DOF6 in the analysis is based on the boundary conditions given in Tutorial 6 and relevant models related to the theme [3]. By fixing these degrees of freedom, it ensures that the system remains in a constrained state and prevents free-body motion. Additionally, without fixing these DOFs, the result may yield a frequency of zero in the Campbell diagram. One possible explanation for this is that the DOFs have a prescribed velocity that affects the stiffness of the shaft, and fixing them helps maintain the stability and consistency of the analysis. It is also worth noting that in this analysis, the motor is considered a fixed reference system while rotating.

5.2.3 Effect of DTMAX

The parameter DTMAX represents the maximum time step size that can be used in the simulation. The value of DTMAX is crucial in determining the accuracy and behavior of the system in the transient response simulation.

A high value of DTMAX can potentially result in missing important details, as the time step may be too large to capture rapid changes in the system. On the other hand, a low value of DTMAX can lead to longer computation times, as more time steps are required to simulate the desired time period.

Figure 4.5 illustrates that a higher DTMAX value leads to an earlier response, possibly because the response occurs at an earlier time and is then rounded to fit within the defined RPM steps. In the FFT diagram, a lower DTMAX value is associated with lower frequencies.

In cases where the velocity is high, it is advisable to reduce the value of DTMAX to obtain more result points and ensure a more accurate representation of the system's behavior.

5.2.4 Learning Material

During the simulation process, it was encountered that there was a lack of material available specifically for the NX solver. The user material that was found seemed to be designed for an older solver version, and may not have been compatible with SOL 414, which was used in this thesis. While the theory described in the guide may still be relevant, there was a need for specific learning material tailored to the SOL 414 solver.

Obtaining learning material and support for performing simulations in NX has proven to be challenging. Efforts have been made to reach out to Siemens and connect with individuals experienced in using NX. Acquiring relevant models related to the thesis problem has been beneficial in verifying the correctness of the undertaken steps. However, certain aspects, such as defining BERGSET as mentioned in the "old" user guide, have posed difficulties in understanding. Additionally, comprehending the appropriate boundary conditions has also presented challenges in the process.

When attempting to solve the transient response, the accurate determination of DTMAX proved to be challenging. As described in Section 3.2, DTMAX was defined using a formula. However, due to the lack of relevant simulations, it was difficult to fully understand the impact and significance of this parameter, as previously discussed.

It was also challenging to determine the velocity of the shaft in the transient analysis. As a result, it was difficult to ascertain the velocity of the system's response accurately. Unfortunately, no graphs or plots were available to visualize the velocity at specific points in time.

Towards the end of this thesis, a video on Xalrator Academy was found, despite difficulties accessing it. Although it is believed that the video could have been beneficial in terms of improving the results and gaining a better understanding, its late discovery limited its impact on the overall thesis.

5.3 Fedem

In this section, the results and findings from the analysis conducted using Fedem are presented and discussed. The focus is on the displacement patterns and the frequencies observed in the system.

5.3.1 Displacement

The observed higher displacement in the Fedem simulation compared to the results from the physical rig and NX analysis could be attributed to various factors.

One possible reason for the difference in displacement between the Fedem simulation and the physical rig or NX analysis could be the time integration in the simulation. The choice of interaction method in Fedem can have an impact on the results, potentially leading to higher displacements.

It is worth mentioning that the Fedem simulation utilized the HHT-alpha method with an alpha value of 0.1 for the numerical integration. This choice of numerical integration scheme can have an impact on the simulation results, including the calculated displacement values.

5.3.2 Natural Frequency

The frequency analysis conducted in Fedem yielded higher values for the bending mode compared to the results obtained from NX, as observed in both the Campbell diagram and the transient response analysis. While the first mode in Fedem is relatively close to the results from NX, it is important to note that this mode represents a different mode shape and exhibits higher frequencies.

5.3.3 Different in Bearing Properties

The difference in bearing properties, whether using given values or the values used in Fedem, does not have a significant impact on the frequency response, as observed in Figures 4.8 and 4.12. The results obtained using the original bearing values in Fedem show minimal damping in bearing 2, resulting in a small response at higher speeds, occurring after surpassing the critical speed.

5.4 Analytical Solution Using Python

5.4.1 Frequency Response and Natural Frequencies

The frequency response analysis shows a prominent response at a velocity of approximately 5640 RPM, with an amplitude exceeding 10 mm. The Campbell diagram from the analytical solution indicates a lower frequency compared to the results obtained from Fedem and NX. This discrepancy in frequencies between the analytical solution and the estimated value in the physical Bently Nevada RK0 suggests that the analytical solution may provide higher frequency values in this particular case.

The two first modes in the analytical solution are observed to be positioned between 90 Hz and 100 Hz, which is lower than the frequencies obtained from the simulation. When comparing the frequency response and the Campbell diagram, it can be seen that the response occurs at the same velocity where the rotor speed crosses the eigenfrequencies. However, it is worth noting that there may be errors in the code used for solving the eigenvalues and eigenvectors of the A and B matrices, particularly in cases where the shaft is present between the bearings.

5.4.2 Geometry and Model

The model used in this analysis assumed a solid disk or flywheel without any holes. However, it should be noted that the presence of 16 holes in the actual physical system was not considered in the Python code. Although the holes are relatively small compared to the size of the flywheel, their presence could potentially have some effect on the behavior of the shaft. Therefore, it is possible that not accounting for these holes in the model could result in some discrepancies between the predicted results and the actual system response.

The script was designed to set the displacement values in the v and u directions to zero for node 1, indicating that this node was fixed in the analysis. However, it seems that the fixation of the node was not implemented correctly in the modal analysis. This means that the modeshape analysis might not accurately represent the fixed boundary condition at node 1.

The solution of the shaft between two bearings did not yield the desired result because the eigenvalues were not successfully computed from the A and B matrices. This issue likely prevented the accurate determination of the natural frequencies and mode shapes of the system.

5.4.3 Gyroscopic Effect

The gyroscopic effect on the shaft is considered in the analytical solution of this project. However, the value of the gyroscopic effect is found to be near zero, indicating that its contribution to the system dynamics is negligible. This is likely due to the small cross-section of the shaft, which does not generate a significant gyroscopic effect.

It is important to note that neglecting the gyroscopic effect in the analytical solution is valid for this specific case. In situations where the shaft has a larger diameter cross-section, the gyroscopic effect may become more significant and should be included in the analysis. Careful consideration of the shaft's geometry and its influence on the gyroscopic effect is crucial to accurately model the system dynamics.

The reason for neglecting the gyroscopic effect in this analysis can be explained based on the theory presented in Section 2.1.2. The gyroscopic effect is proportional to the angular momentum, which in turn depends on the polar moment of inertia. For small cross-sections, the polar moment of inertia is small, resulting in a small angular momentum and consequently a negligible gyroscopic effect. Therefore, for the specific case of small cross-sections in this project, the gyroscopic effect can be safely neglected.

5.5 Comparison of the Results

When comparing the results from different simulations and analytical solutions, it is important to consider certain factors that can introduce errors or discrepancies. One such factor is the estimation of bearing properties, which may not be known exactly and can result in variations in the simulation results. Additionally, the accuracy of the sensors used to measure displacement can also introduce errors in the results.

In comparing the results from NX, Fedem, and the analytical solution, it is expected that they should exhibit similar properties and be compatible with each other. However, as discussed earlier in this chapter, there are differences in frequencies and displacement between the simulations. These differences could be attributed to various factors such as modeling assumptions, numerical methods, and simulation parameters used in each software.

5.5.1 Element Size

The element size used in the simulation can indeed impact the results, and it is important to ensure consistency in element size when comparing results between different software or analysis methods. In the case of NX and Python, using the same element size allows for a more direct comparison of the results and increases their comparability.

Regarding the analytical solution, the way the plots are generated and presented can affect the interpretation of the results. If the plot is based on node data rather than position, it may result in a less smooth and less intuitive representation of the results. This can make it more challenging to analyze and interpret the results accurately.

Furthermore, the difference in element size between Fedem and NX can also contribute to variations in the results. Different element sizes can lead to different levels of discretization and approximation, potentially affecting the accuracy and behavior of the system being analyzed.

Chapter 6

Conclusion

Based on the results obtained from the Bently Nevada RK0 rig, the estimated frequency response cannot be directly compared to the results of other simulations. Additionally, the exact bearing properties used in the rig are unknown, which further adds to the difficulty of making direct comparisons.

Based on the obtained results, there were certain limitations and challenges encountered when using the NX solver SOL414. Some of these issues included difficulties in obtaining all the relevant results in a convenient format, such as the absence of velocity plots. These limitations indicate that the solver may require further development and improvements to enhance its usability and user-friendliness.

The results from the analyses indicate that when the bearing stiffness becomes too high, it is easier to obtain a solution by applying fixed boundary conditions in the x and y directions. This approach helps stabilize the system and enables the simulation to proceed successfully.

The increase in bearing damping does not have a significant effect on the frequencies of the bending mode. However, it does impact the displacement, with higher damping resulting in higher displacement values. The Campbell diagram, on the other hand, shows more noticeable changes in frequencies, particularly for modes 3 and 4. This suggests that bearing damping can influence the dynamic behavior of the system, particularly for higher-order modes.

Furthermore, it is important to consider the appropriate boundary conditions, such as fixing the translation in the x and y directions and fixing the rotation about the z-axis.

Choosing an appropriate value for DTMAX is crucial as it can significantly impact the simulation results. A high value of DTMAX may cause important details to be missed, while a low value can result in longer computation times. Finding the right balance is essential to ensure accurate and efficient simulations.

The analysis in Fedem results in high displacement and frequency values, which can possibly be attributed to the damping properties.

In the analytical solution, there is no need to include a gyroscopic effect for the shaft, especially when the shaft has a thin cross-section. The gyroscopic effect is minimal for such cases and can be safely neglected in the analysis.

The simulation in NX and the analytical solution should be comparable since they both use the same properties, such as bearing stiffness and beam element size. However, it is important to note that there may be some differences or discrepancies between the two methods due to various factors, such as numerical approximations, assumptions, or simplifications made in the analytical solution. Therefore, while the results can provide valuable insights and trends, it is necessary to interpret them with caution and consider the limitations and assumptions of each method.

Chapter 7

Further Work

Since there were a lot of issues in NX, there is a lot of possible further work.

Firstly, an analysis performed in an alternative solver such as Abaqus and Ansys would give valuable insight into the problem, as the is a lot of literature on these solvers. This can be used to compare the different results in Fedem and from the physical Bently Nevada RK0 rig with the chosen solver.

Additionally, it is possible to go back to the analysis in NX once the documentation is better. This includes doing the simulation with beam elements and solid elements.

Another thing to do is obtain more precise bearing properties. This should help when comparing the result with the physical Bently Nevada RK0 rig.

Furthermore, the Fedem model should be improved by looking at the integration properties used in the model.

Lastly, focusing on enhancing the Python script would be valuable, so that results can be more easily read. This can be done by changing element size and improving plotting.

References

[1] M.I Friswell. *Dynamics of rotating machines*. eng. Vol. 28. Cambridge aerospace series. Cambridge: Cambridge University Press, 2010. ISBN: 9780521850162.

[2] Mohammad Hadi Jalali et al. 'Dynamic analysis of a high speed rotor-bearing system'. In: *Measurement* 53 (2014), pp. 1–9.

[3] *Transient response of a 1D model at run-up under Unbalance*. English. SIEMENS. 2023. 14 pp.

[4] *User's Guide*. English. Version Release 2211. SAP. 2022. 392 pp.

[5] L.H. Veldre. 'Mechanical Design of Electric Demonstrator Motor for E-Fan X Hybrid Aeroplane and a Study of Dynamics of Rotating Bodies'. MA thesis. NTNU, Norwegian University of Science, Technology Faculty of Engineering Department of Mechanical and Industrial Engineering, 2020.

Appendix A

Bernoulli and Timoshenko Matrices

A.1 Bernoulli

$$K_B = \frac{E_e I_e}{l_e} \begin{bmatrix} 12 & 0 & 0 & 6l_e & -12 & 0 & 0 & 6l_e \\ 0 & 12 & -6l_e & 0 & 0 & -12 & -6l_e & 0 \\ 0 & -6l_e & 4l_e^2 & 0 & 0 & 6l_e & 2l_e^2 & 0 \\ 6l_e & 0 & 0 & 4l_e^2 & -6l_e & 0 & 0 & 2l_e^2 \\ -12 & 0 & 0 & -6l_e & 12 & 0 & 0 & -6l_e \\ 0 & -12 & 6l_e & 0 & 0 & 12 & 6l_e & 0 \\ 0 & -6l_e & 2l_e^2 & 0 & 0 & 6l_e & 4l_e^2 & 0 \\ 6l_e & 0 & 0 & 2l_e^2 & -6l_e & 0 & 0 & 4l_e^2 \end{bmatrix} \tag{A.1}$$

$$M_B = \frac{\rho_e A_e l_e}{420} \begin{bmatrix} 156 & 0 & 0 & 22l_e & 54 & 0 & 0 & -13l_e \\ 0 & 156 & -22l_e & 0 & 0 & 54 & 13l_e & 0 \\ 0 & -22l_e & 4l_e^2 & 0 & 0 & -13l_e & -3l_e^2 & 0 \\ 22l_e & 0 & 0 & 4l_e^2 & 13l_e & 0 & 0 & -3l_e^2 \\ 54 & 0 & 0 & 13l_e & 156 & 0 & 0 & -22l_e \\ 0 & 54 & -13l_e & 0 & 0 & 156 & 22l_e & 0 \\ 0 & 13l_e & -3l_e^2 & 0 & 0 & 22l_e & 4l_e^2 & 0 \\ -13l_e & 0 & 0 & -3l_e^2 & -22l_e & 0 & 0 & 4l_e^2 \end{bmatrix} \tag{A.2}$$

$$G_B = \frac{\rho_e I_e}{15 l_e} \begin{bmatrix} 0 & 36 & -3l_e & 0 & 0 & -36 & -3l_e & 0 \\ -36 & 0 & 0 & -3l_e & 36 & 0 & 0 & -3l_e \\ 3l_e & 0 & 0 & 4l_e^2 & -3l_e & 0 & 0 & -l_e^2 \\ 0 & 3l_e & -4l_e^2 & 0 & 0 & -3l_e & l_e^2 & 0 \\ 0 & -36 & 3l_e & 0 & 0 & 36 & 3l_e & 0 \\ 36 & 0 & 0 & 3l_e & -36 & 0 & 0 & 3l_e \\ 3l_e & 0 & 0 & -l_e^2 & -3l_e & 0 & 0 & 4l_e^2 \\ 0 & 3l_e & l_e^2 & 0 & 0 & -3l_e & -4l_e^2 & 0 \end{bmatrix} \tag{A.3}$$

A.2 Timoshenko

$$B_T = \frac{E_e l_e}{(1+\Phi_e) l_e^3} \begin{bmatrix} 12 & 0 & 0 & 6l_e & -12 & 0 & 0 & 6l_e \\ 0 & 12 & -6l_e & 0 & 0 & -12 & -6l_e & 0 \\ 0 & -6l_e & l_e^2(4+\Phi_e) & 0 & 0 & 6l_e & l_e^2(2-\Phi_e) & 0 \\ 6l_e & 0 & 0 & l_e^2(4+\Phi_e) & -6l_e & 0 & 0 & l_e^2(2-\Phi_e) \\ -12 & 0 & 0 & -6l_e & 12 & 0 & 0 & -6l_e \\ 0 & -12 & 6l_e & 0 & 0 & 12 & 6l_e & 0 \\ 0 & -6l_e & l_e^2(2-\Phi_e) & 0 & 0 & 6l_e & l_e^2(4+\Phi_e) & 0 \\ 6l_e & 0 & 0 & l_e^2(2-\Phi_e) & -6l_e & 0 & 0 & l_e^2(4+\Phi_e) \end{bmatrix} \tag{A.4}$$

$$\begin{aligned} M_T = & \frac{\rho_e A_e l_e}{840(1+\Phi_e)^2} \begin{bmatrix} m_1 & 0 & 0 & m_2 & m_3 & 0 & 0 & m_4 \\ 0 & m_1 & -m_2 & 0 & 0 & m_3 & -m_4 & 0 \\ 0 & -m_2 & m_5 & 0 & 0 & m_4 & m_6 & 0 \\ m_2 & 0 & 0 & m_5 & -m_4 & 0 & 0 & m_6 \\ m_3 & 0 & 0 & -m_4 & m_1 & 0 & 0 & -m_2 \\ 0 & m_3 & m_4 & 0 & 0 & m_1 & m_2 & 0 \\ 0 & -m_4 & m_6 & 0 & 0 & -m_2 & m_5 & 0 \\ m_4 & 0 & 0 & m_6 & -m_2 & 0 & 0 & m_5 \end{bmatrix} \\ & + \frac{\rho_e I_e}{30(1+\Phi_e)^2} \begin{bmatrix} m_7 & 0 & 0 & m_8 & -m_7 & 0 & 0 & m_8 \\ 0 & m_7 & -m_8 & 0 & 0 & -m_7 & -m_8 & 0 \\ 0 & -m_8 & m_9 & 0 & 0 & m_8 & m_{10} & 0 \\ m_8 & 0 & 0 & m_9 & -m_8 & 0 & 0 & m_{10} \\ -m_7 & 0 & 0 & -m_8 & m_7 & 0 & 0 & -m_8 \\ 0 & -m_7 & m_8 & 0 & 0 & m_7 & m_8 & 0 \\ 0 & -m_8 & m_{10} & 0 & 0 & m_8 & m_9 & 0 \\ m_8 & 0 & 0 & m_{10} & -m_8 & 0 & 0 & m_9 \end{bmatrix} \end{aligned} \tag{A.5}$$

$$\begin{aligned} m_1 &= 312 + 588\Phi_e + 280\Phi_e^2, & m_6 &= -(6 + 14\Phi_e + 7\Phi_e^2) l_e^2, \\ m_2 &= (44 + 77\Phi_e + 35\Phi_e^2) l_e, & m_7 &= 36, \\ m_3 &= 108 + 252\Phi_e + 140\Phi_e^2, & m_8 &= (3 - 15\Phi_e) l_e, \\ m_4 &= -(26 + 63\Phi_e + 35\Phi_e^2), & m_9 &= (4 + 5\Phi_e + 10\Phi_e^2) l_e^2, \\ m_5 &= (8 + 14\Phi_e + 7\Phi_e^2), & m_{10} &= (-1 - 5\Phi_e + 5\Phi_e^2) l_e^2 \end{aligned}$$

$$G_T = \frac{\rho_e I_e}{15(1+\Phi_e)^2 l_e} \begin{bmatrix} 0 & g_1 & -g_2 & 0 & 0 & -g_1 & -g_2 & 0 \\ -g_1 & 0 & 0 & -g_2 & g_1 & 0 & 0 & -g_2 \\ g_2 & 0 & 0 & g_3 & -g_2 & 0 & 0 & g_4 \\ 0 & g_2 & -g_3 & 0 & 0 & -g_2 & -g_4 & 0 \\ 0 & -g_1 & g_2 & 0 & 0 & g_1 & g_2 & 0 \\ g_1 & 0 & 0 & g_2 & -g_1 & 0 & 0 & g_2 \\ g_2 & 0 & 0 & g_4 & -g_2 & 0 & 0 & g_3 \\ 0 & g_2 & -g_4 & 0 & 0 & -g_2 & -g_3 & 0 \end{bmatrix} \tag{A.6}$$

$$\begin{array}{ll} g_1 = 36, & g_3 = (4 + 5\Phi_e + 10\Phi_e^2) l_e^2 \\ g_2 = (3 - 15\Phi_e) l_e, & g_4 = (-1 - 5\Psi_e + 5\Phi_e^2) l_e^2 \end{array}$$

Appendix B

Modeshapes in NX

B.1 Mode 1

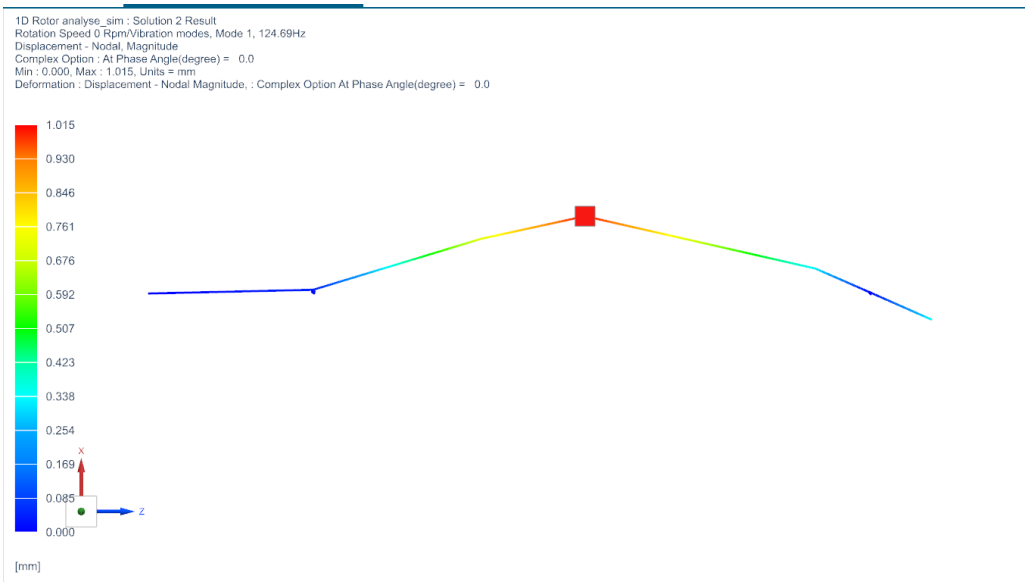


Figure B.1: Mode 1

B.2 Mode 2

Figure B.2: Mode 2

B.3 Mode 3

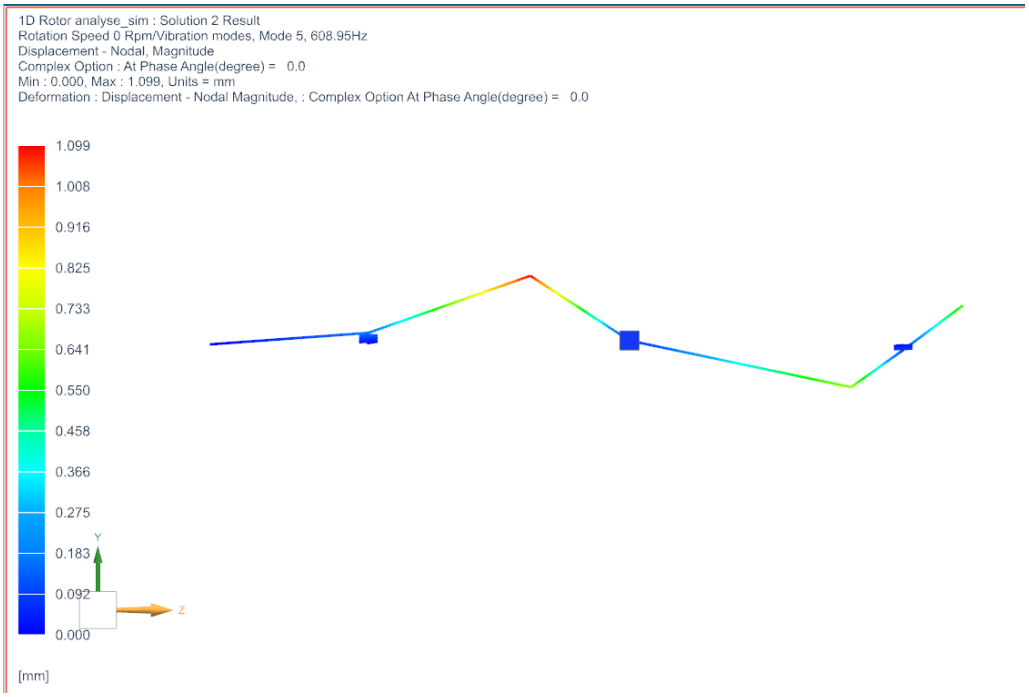


Figure B.3: Mode 3 xz

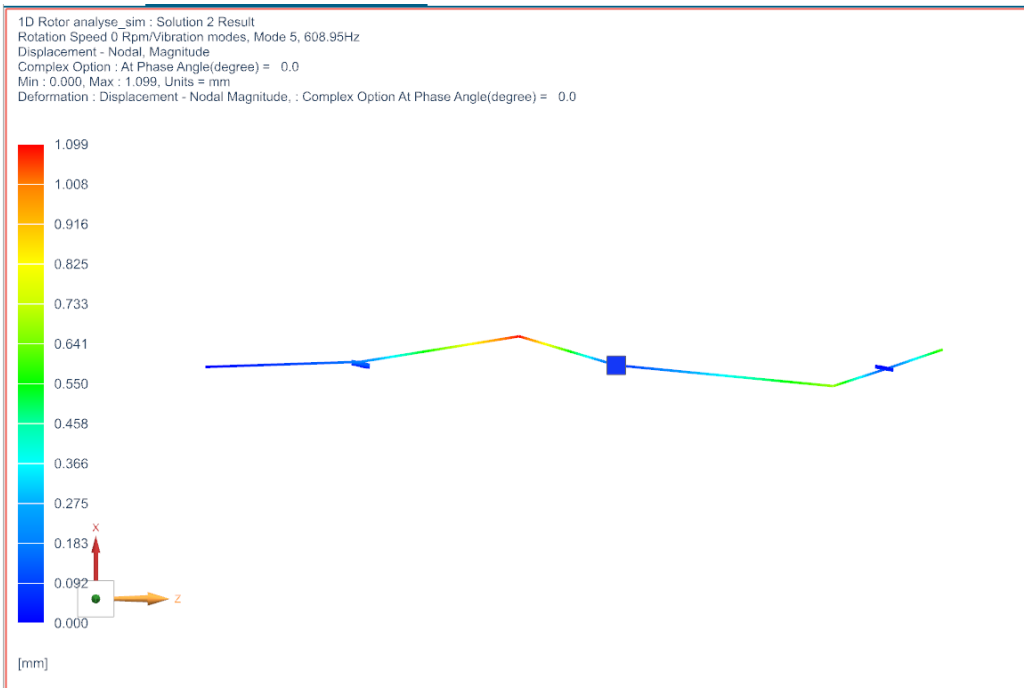


Figure B.4: Mode 3 yz

B.4 Mode 4

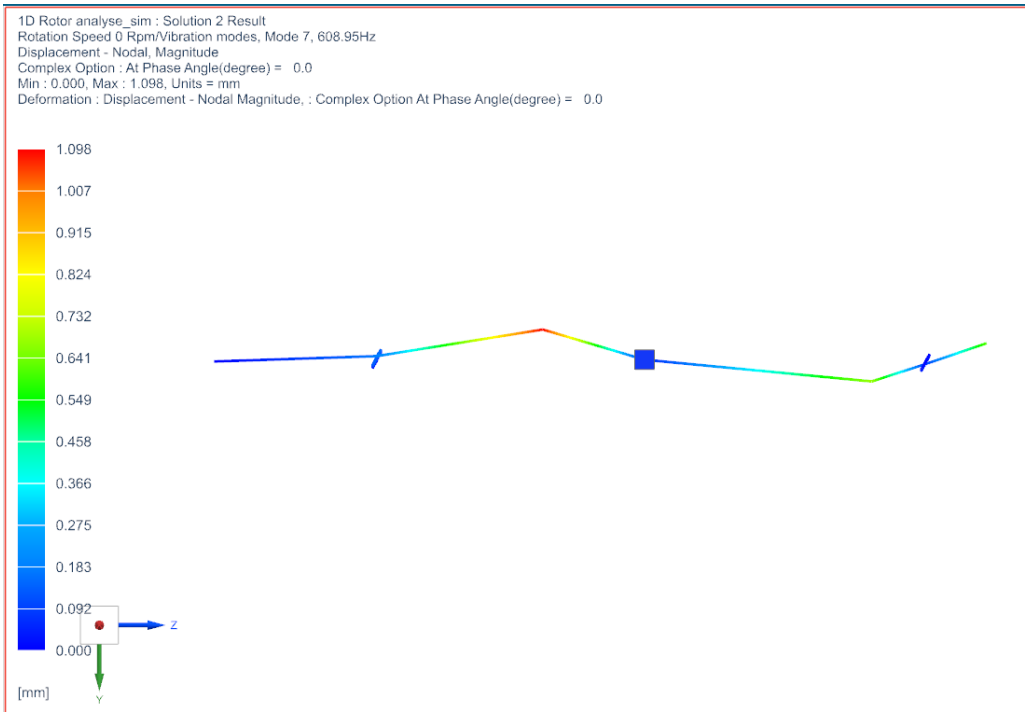


Figure B.5: Mode 4 xz

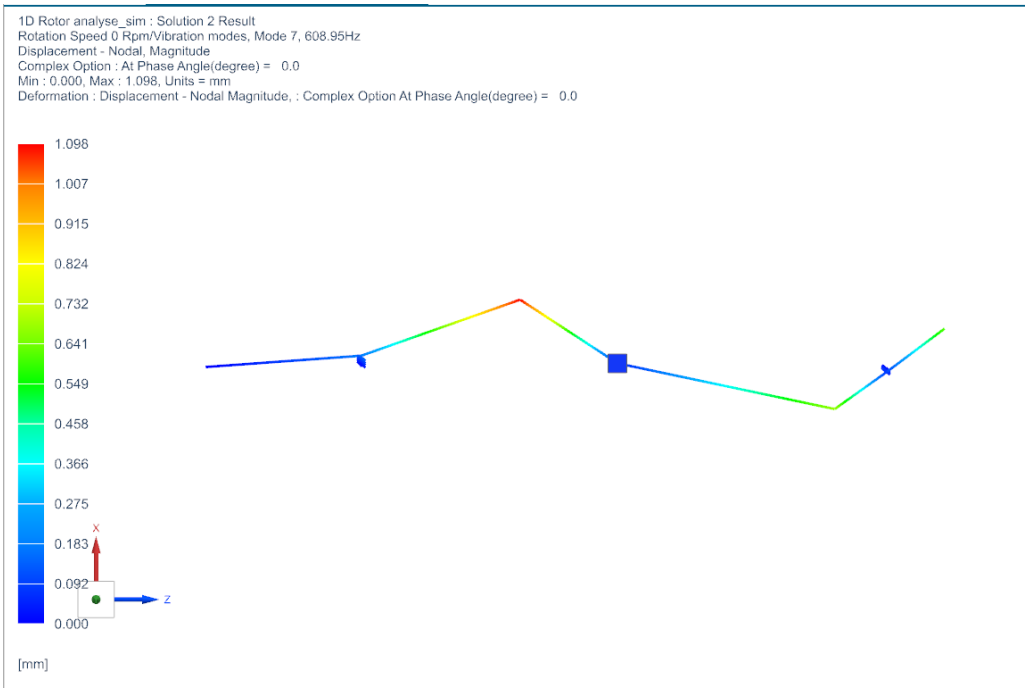


Figure B.6: Mode 4 yz

Appendix C

Modeshape in Python

C.1 Mode 1

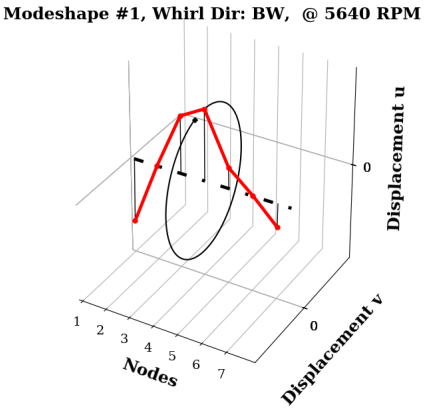


Figure C.1: Mode 1

C.2 Mode 2

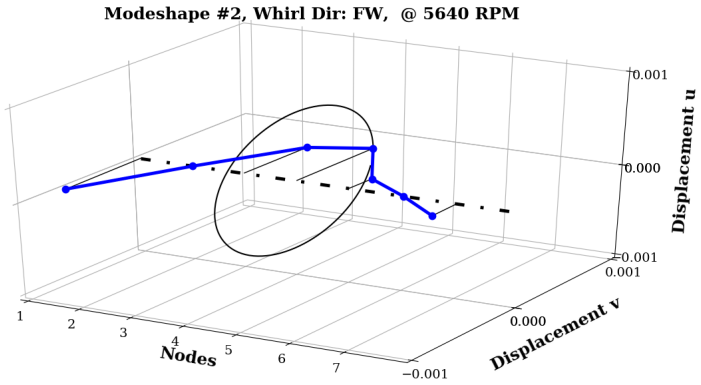


Figure C.2: Mode 2

C.3 Mode 3

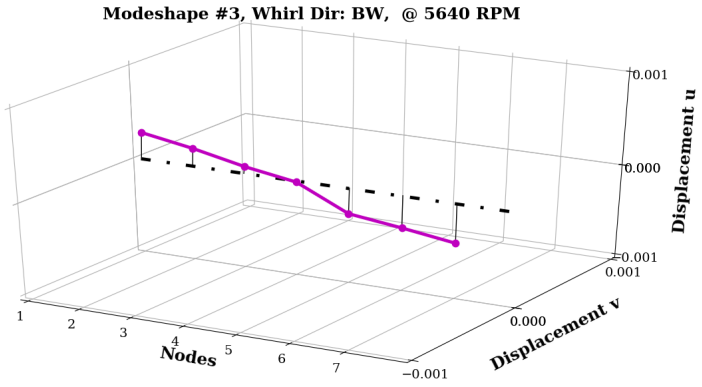


Figure C.3: Mode 3

C.4 Mode 4

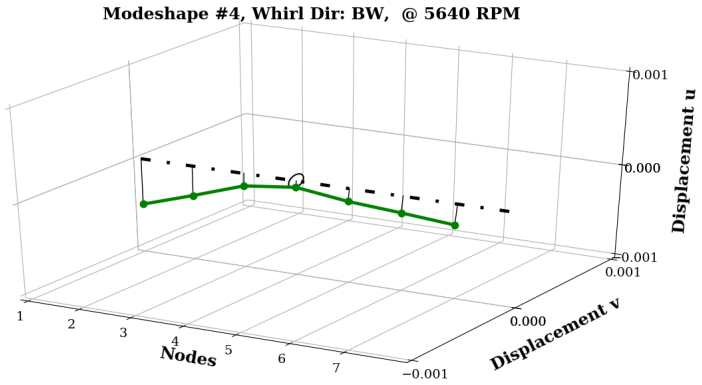


Figure C.4: Mode 4

Appendix D

FFT of DTMAX

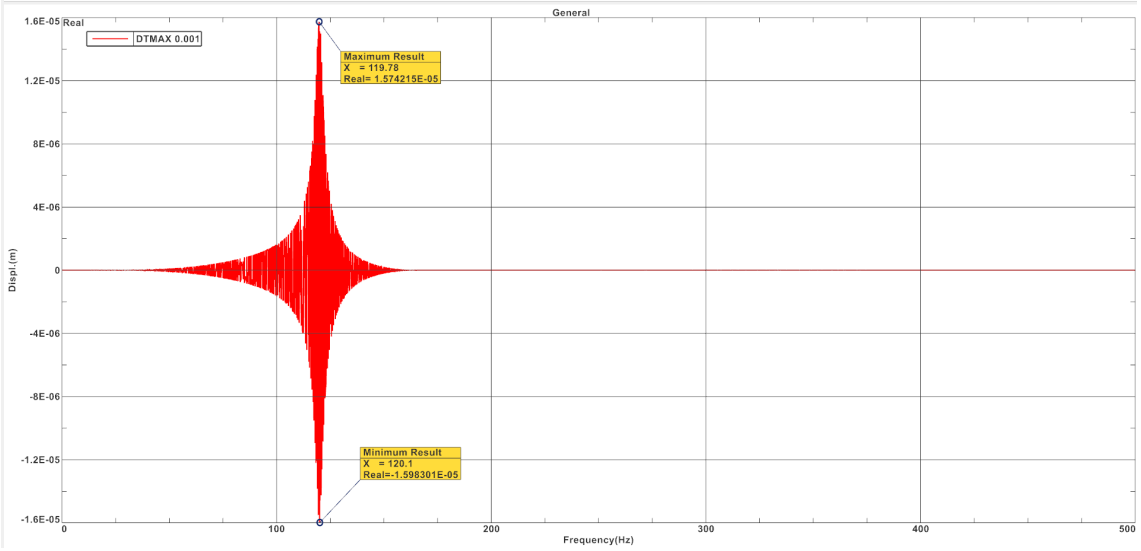


Figure D.1: DTMAX = 0.001s

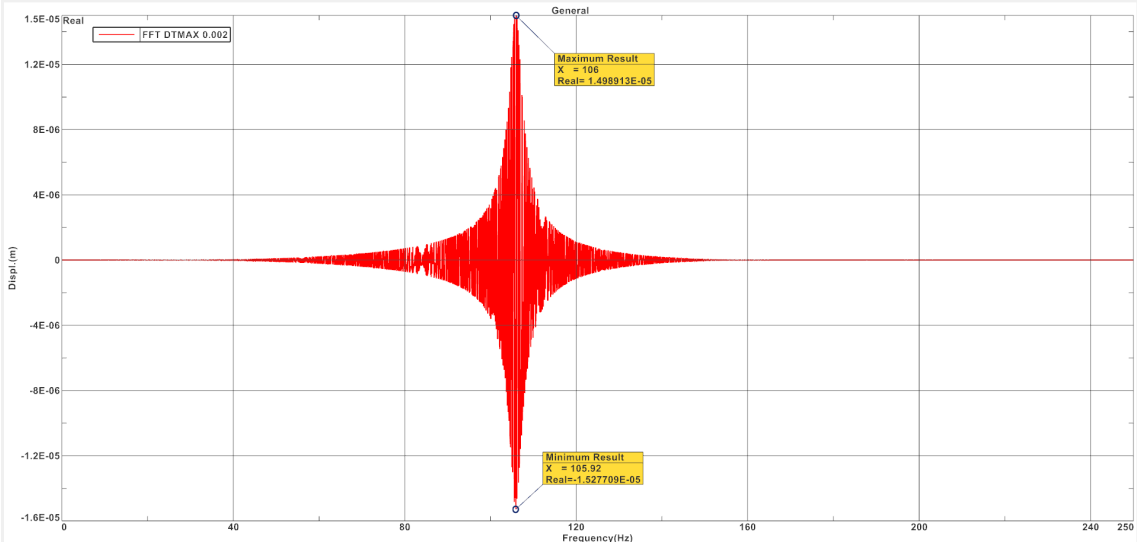


Figure D.2: DTMAX = 0.002s

Appendix E

Damping Plot Using Transient Response

In this plot, only the damping coefficient, c, is varied. The stiffness properties are set as follows: the radial translation stiffness (K11 = K22) is set to 10e6 N/m, and the axial translation stiffness (K33) is set to 1e6 N/m. The purpose is to observe the effect of changing the damping coefficient on the system's behavior.

E.1 Damping at Node 3

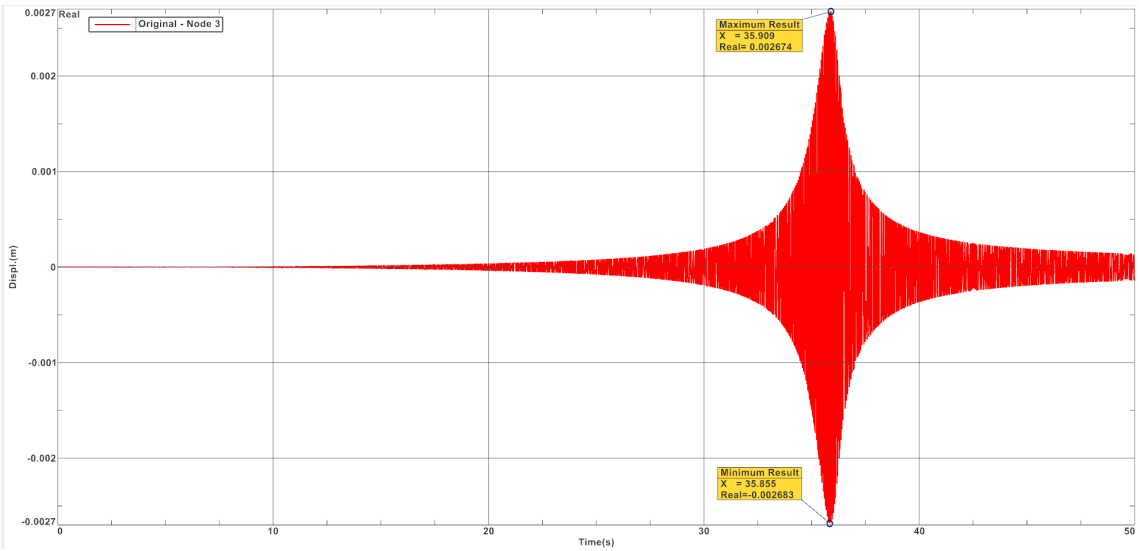


Figure E.1: c = 10e3Ns/m

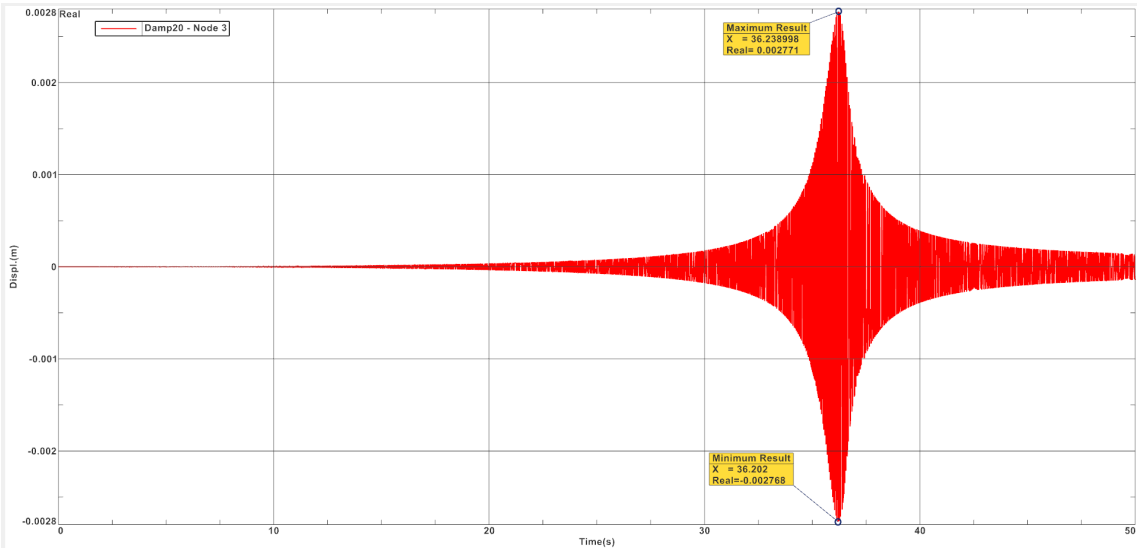


Figure E.2: c = 20e3Ns/m

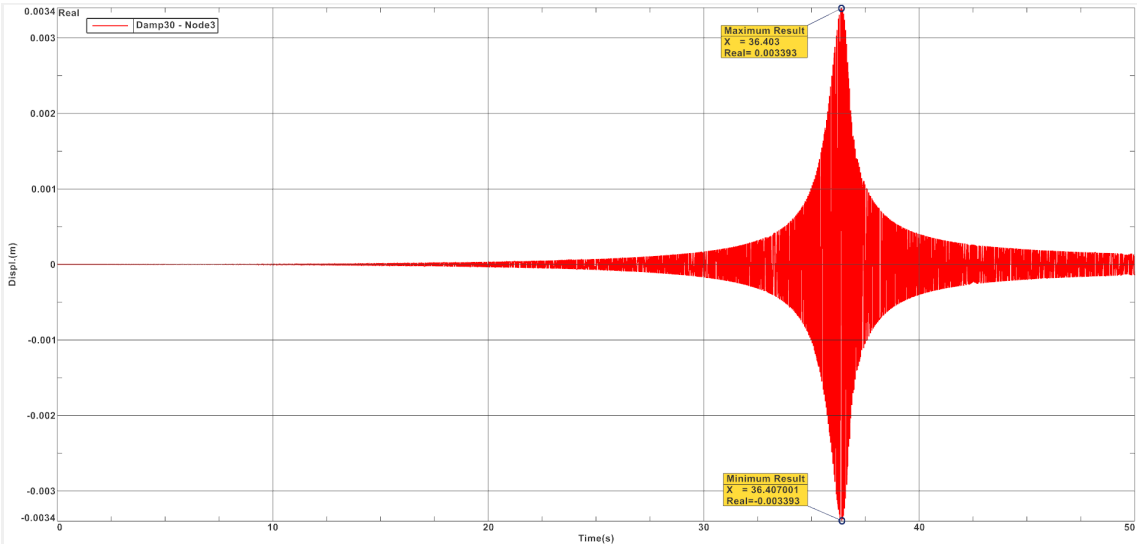


Figure E.3: c = 30e3Ns/m

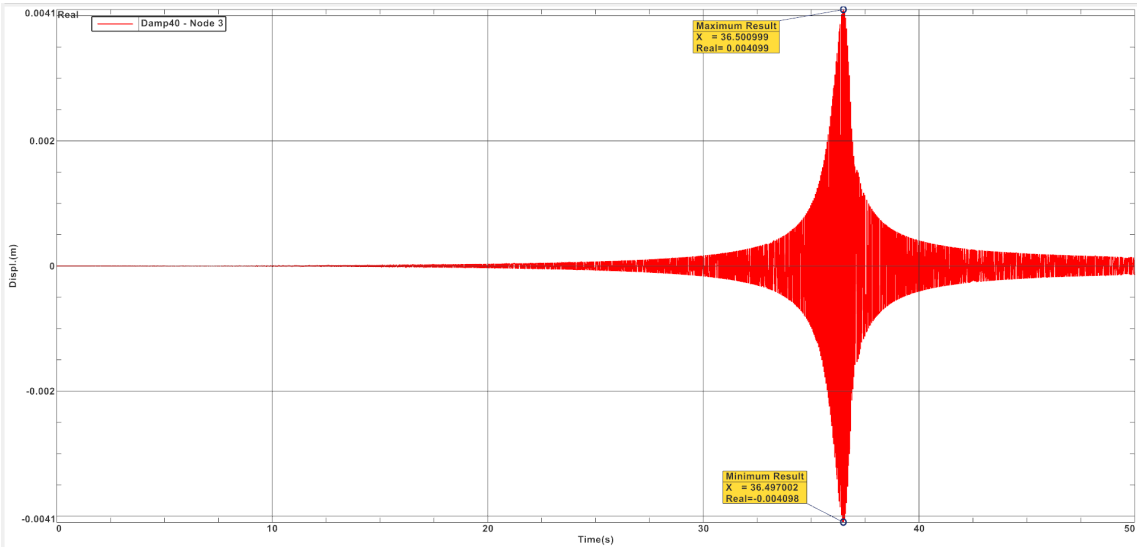


Figure E.4: c = 40e3Ns/m

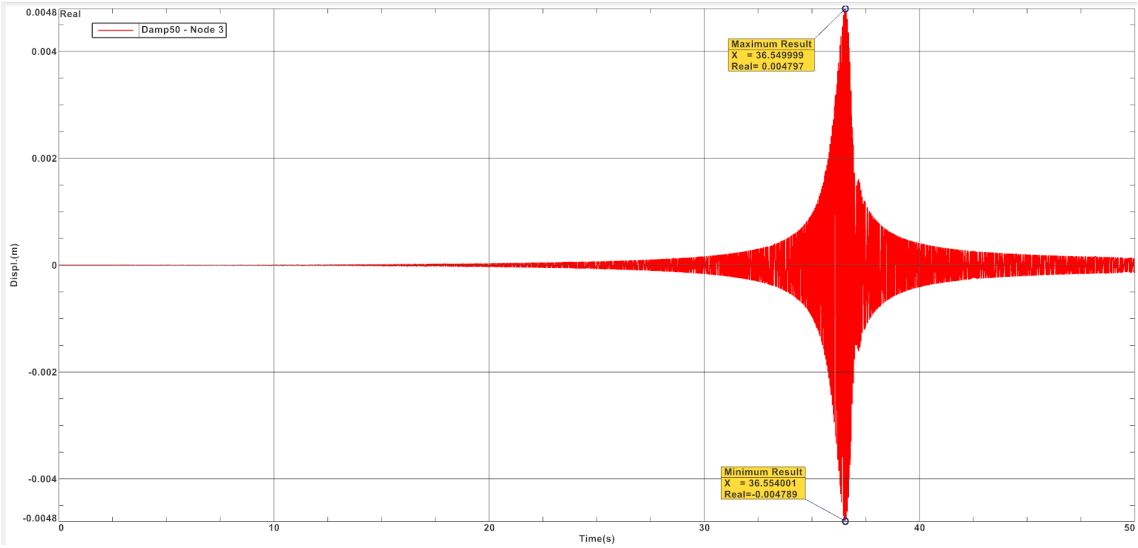


Figure E.5: c = 50e3Ns/m

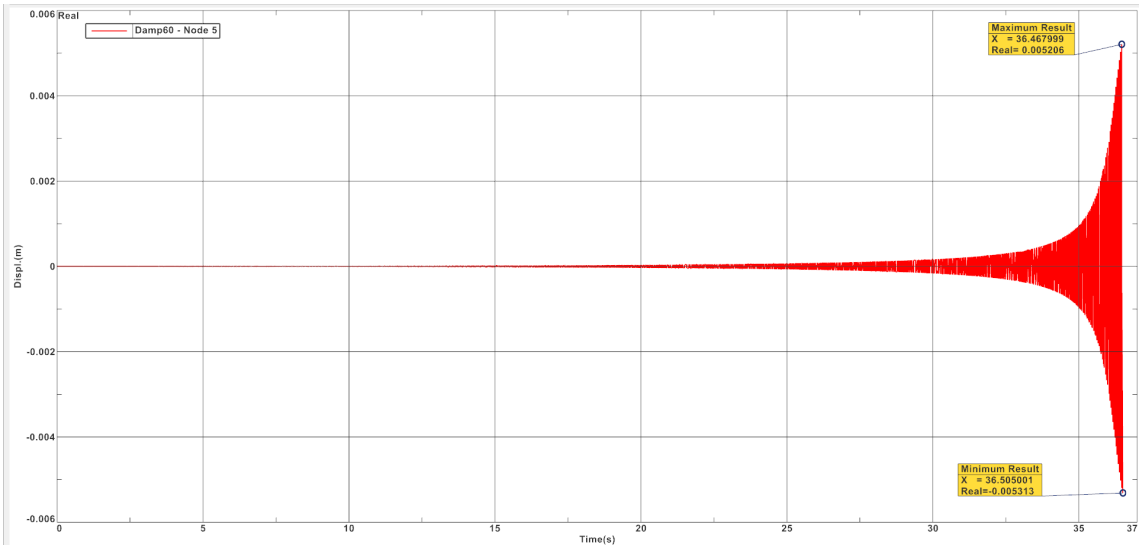


Figure E.6: c = 60e3Ns/m

E.2 FFT at Node 3

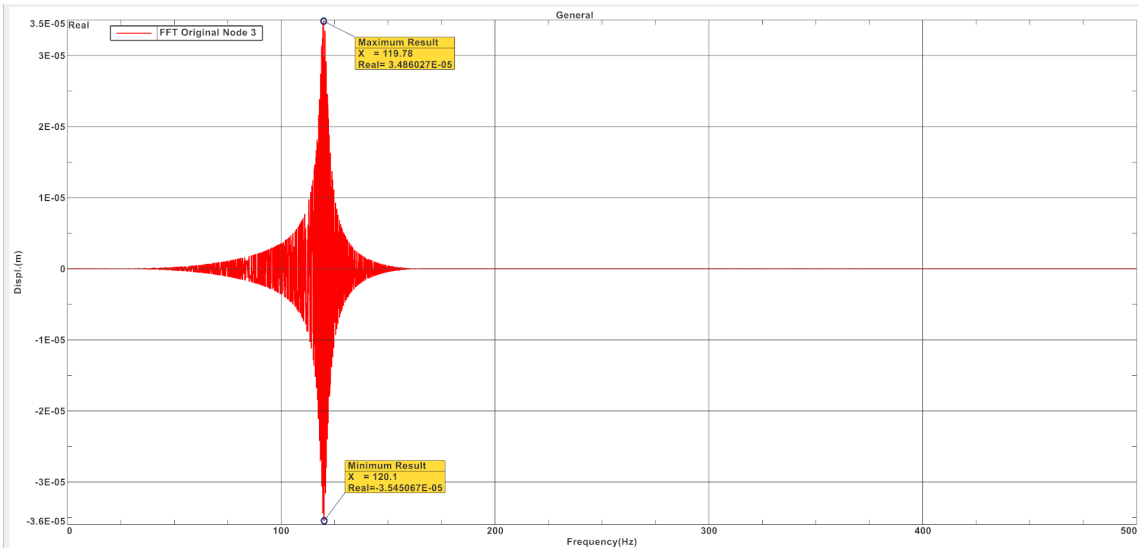


Figure E.7: c = 10e3Ns/m

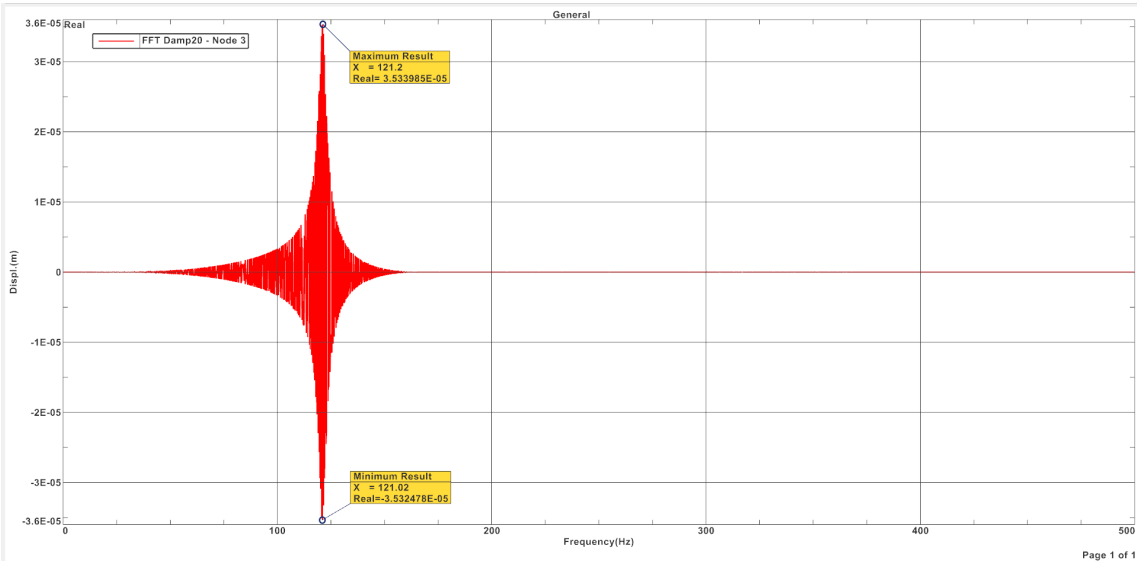


Figure E.8: c = 20e3Ns/m

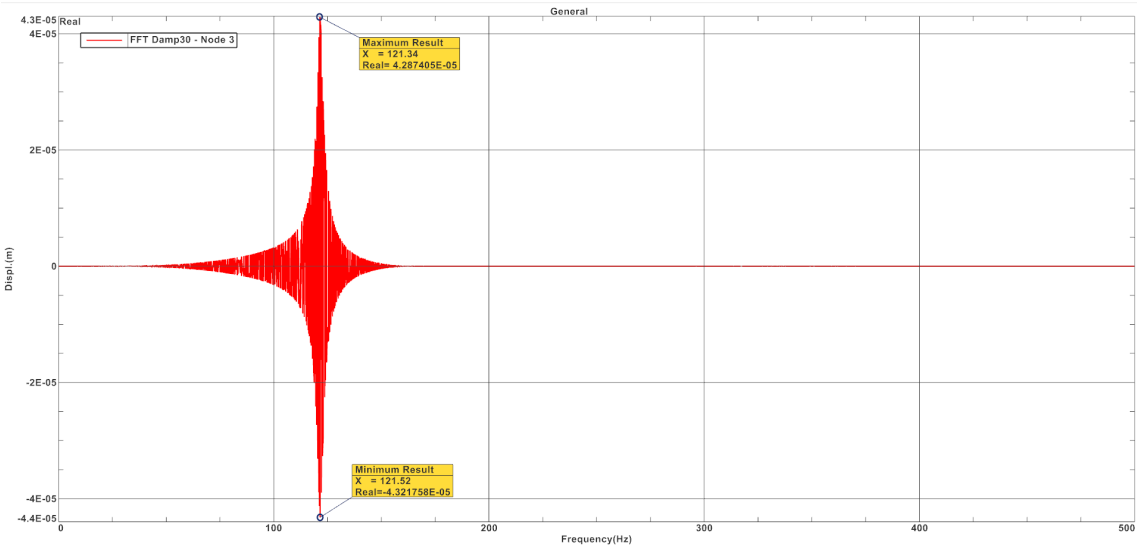


Figure E.9: c = 30e3Ns/m

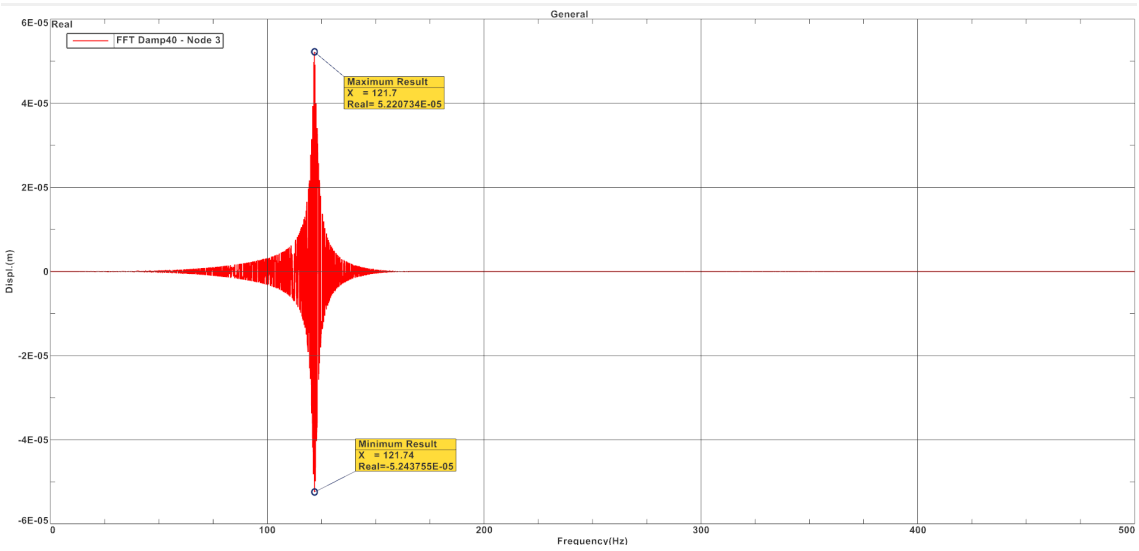


Figure E.10: c = 40e3Ns/m

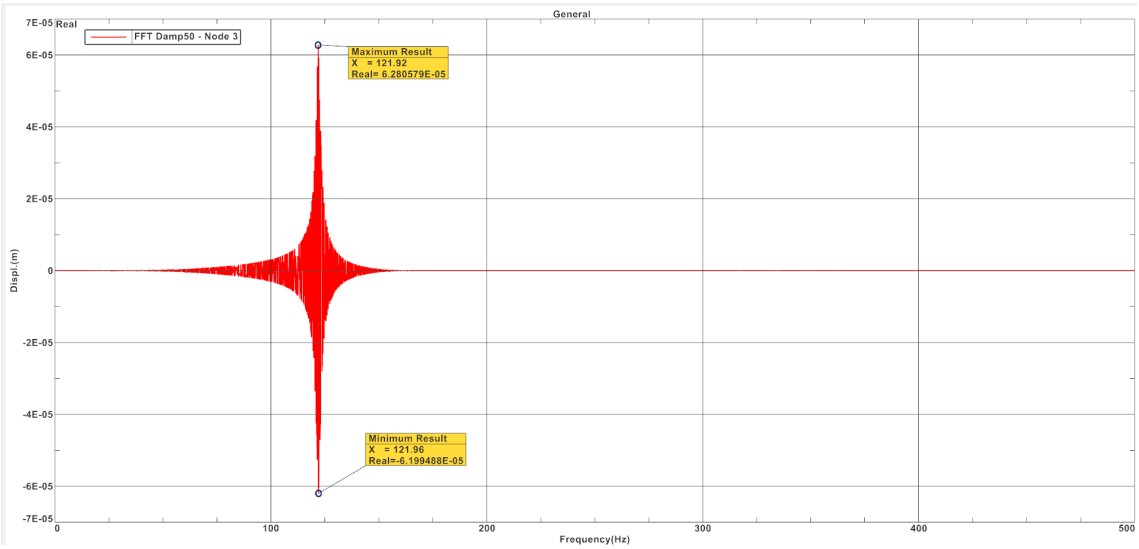


Figure E.11: c = 50e3Ns/m

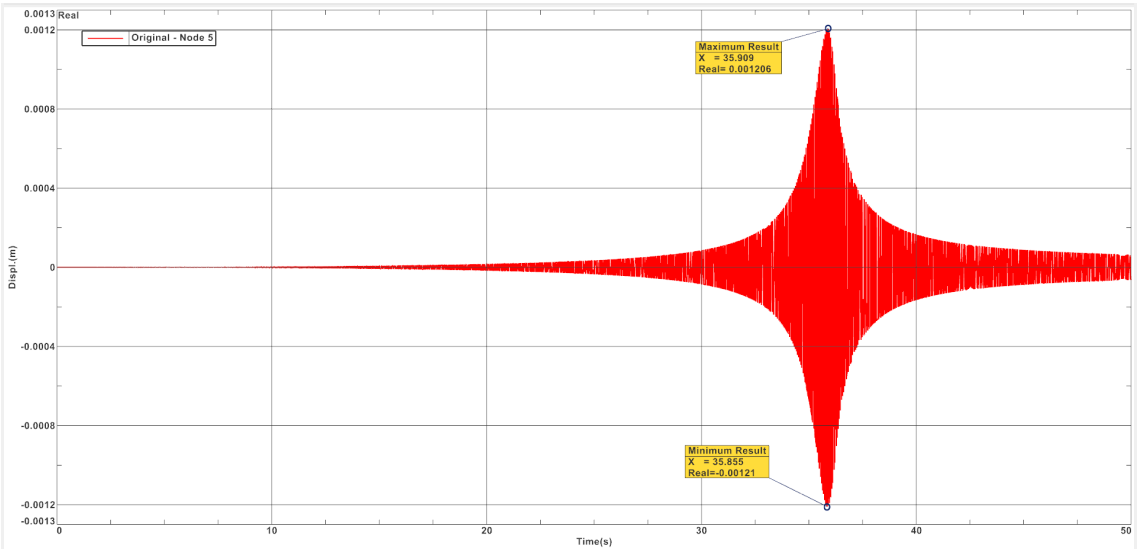


Figure E.12: c = 10e3Ns/m

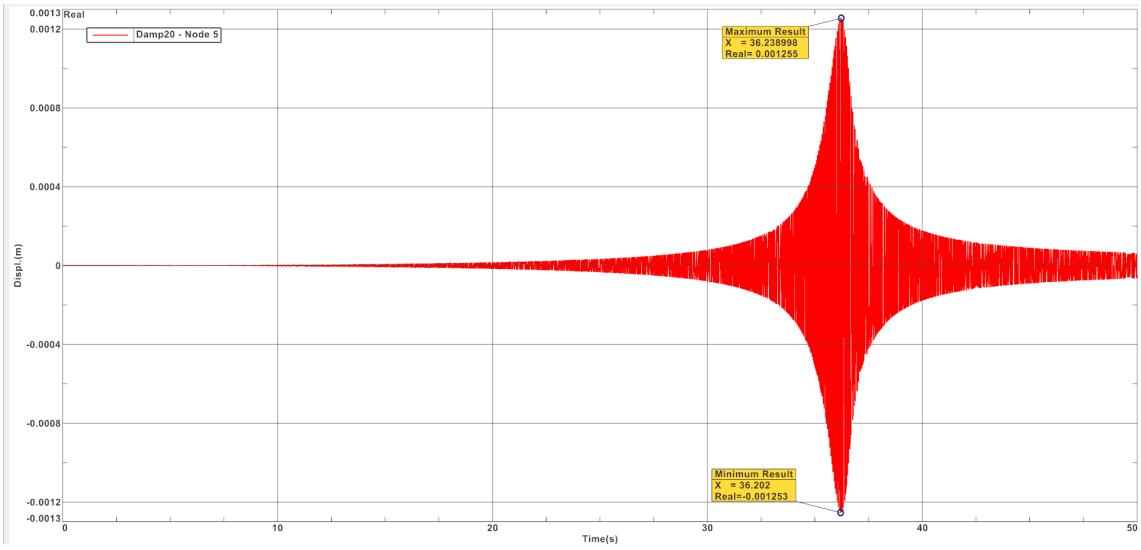


Figure E.13: c = 20e3Ns/m

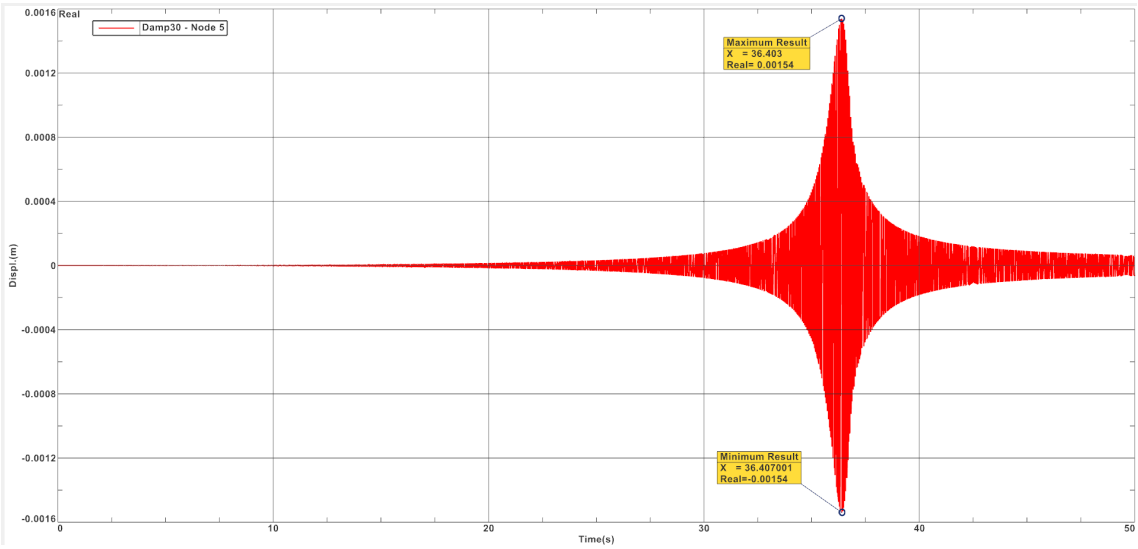


Figure E.14: c = 30e3Ns/m

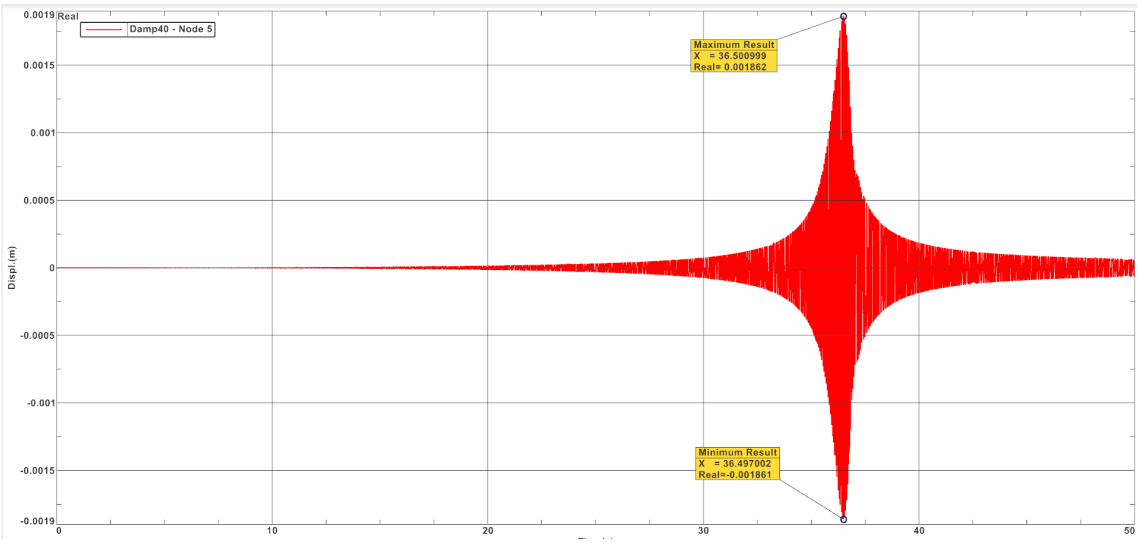


Figure E.15: c = 40e3Ns/m

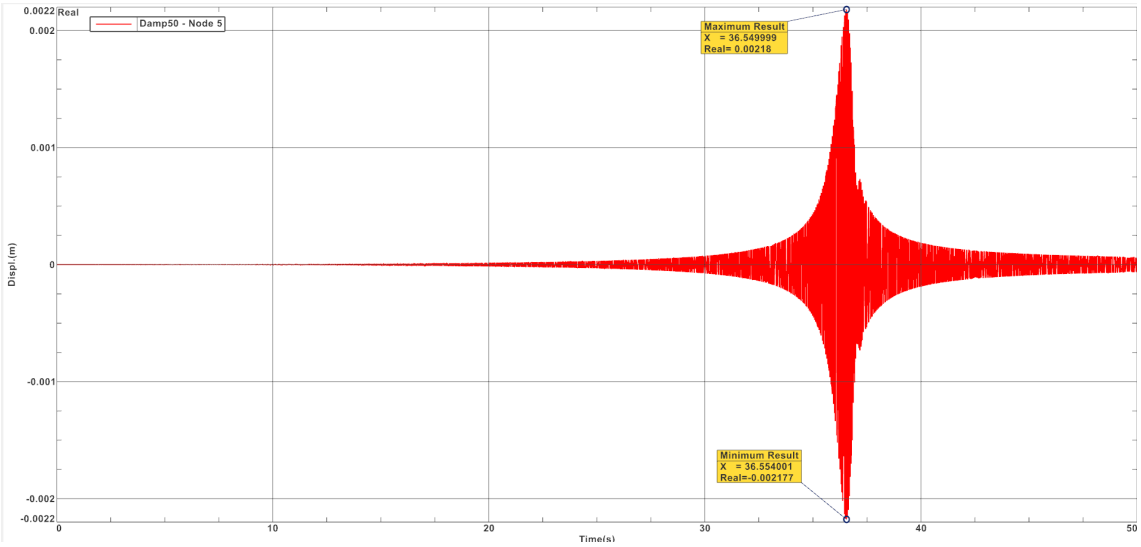


Figure E.16: c = 50e3Ns/m

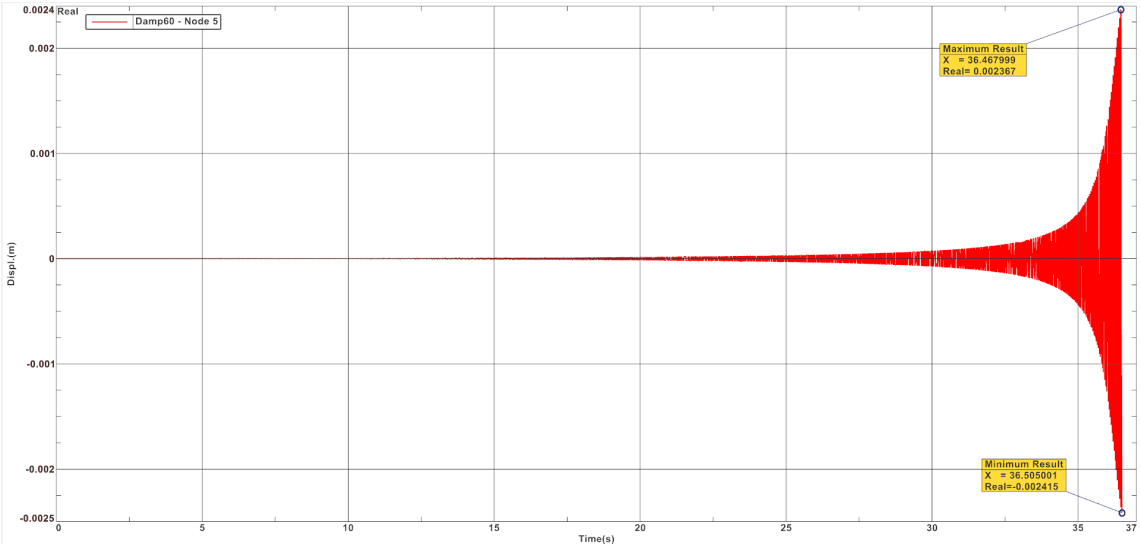


Figure E.17: c = 60e3Ns/m

Appendix F

Stiffness Plot Using Transient Response

In these plots, the transient response of the system is presented. The damping coefficient (C11 = C22) is kept constant at 10e3Ns/m, and the axial stiffness (K33) is set to 1e6N/m. The radial stiffness, k, is varied to observe its effect on the system's transient behavior.

F.1 Stiffness at Node 3

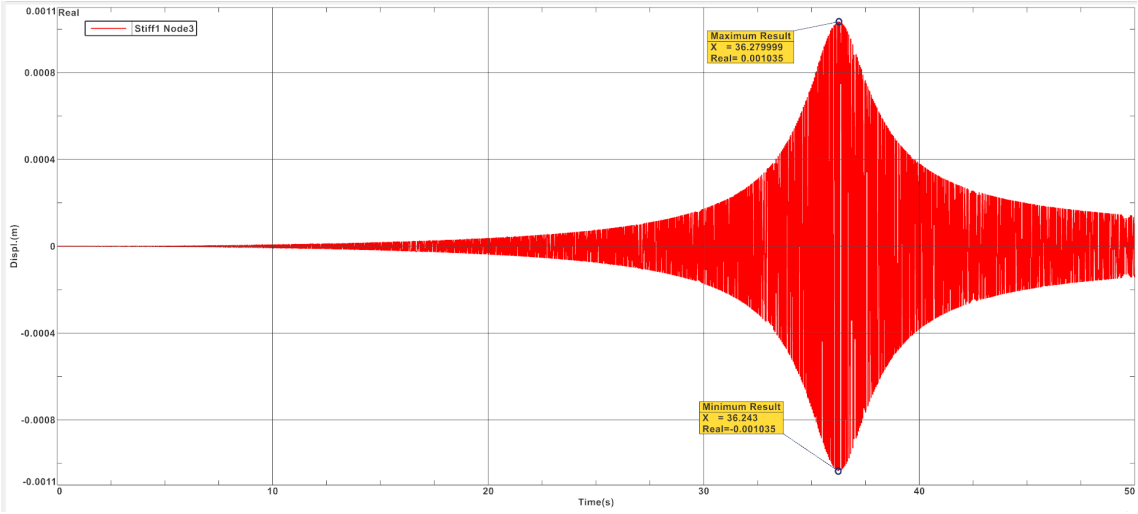


Figure F.1: K = 1e6N/m

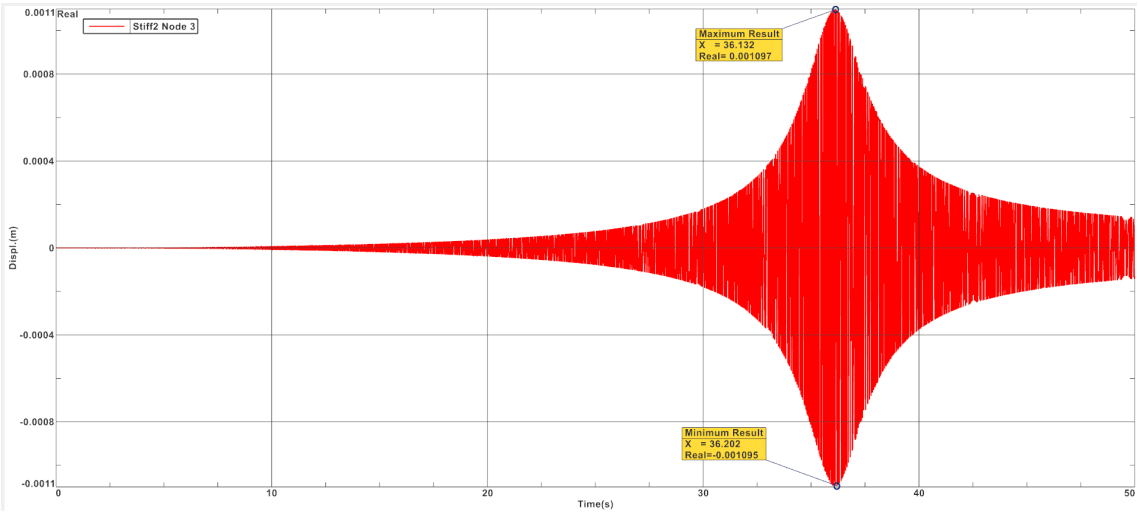


Figure F.2: K = 2e6N/m

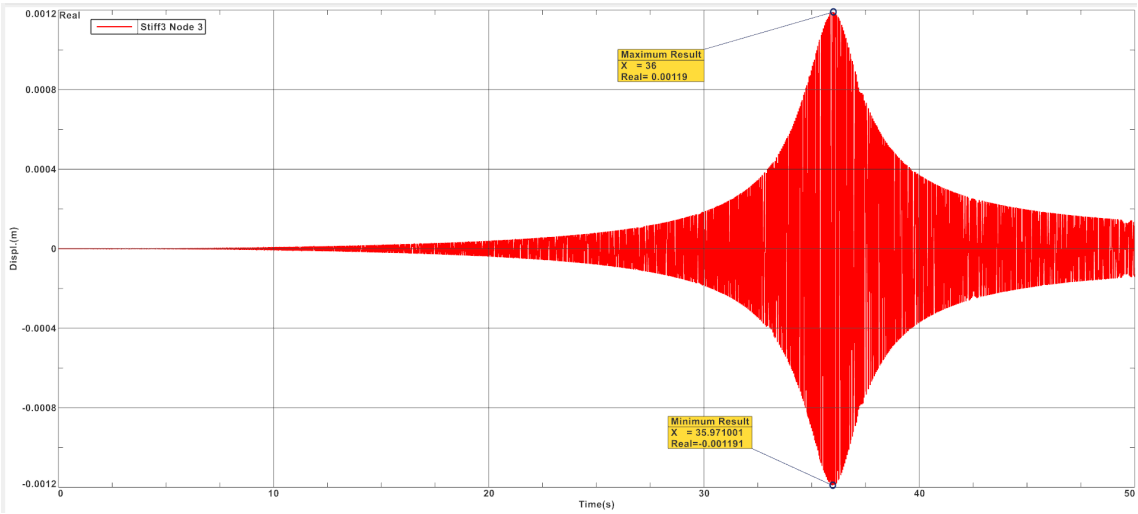


Figure F.3: K = 3e6N/m

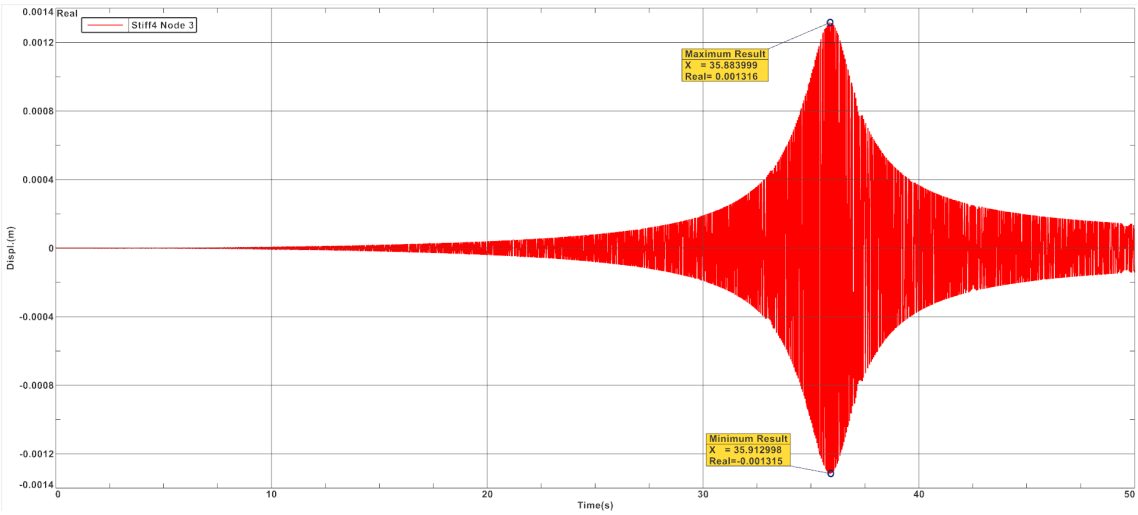


Figure F.4: K = 4e6N/m

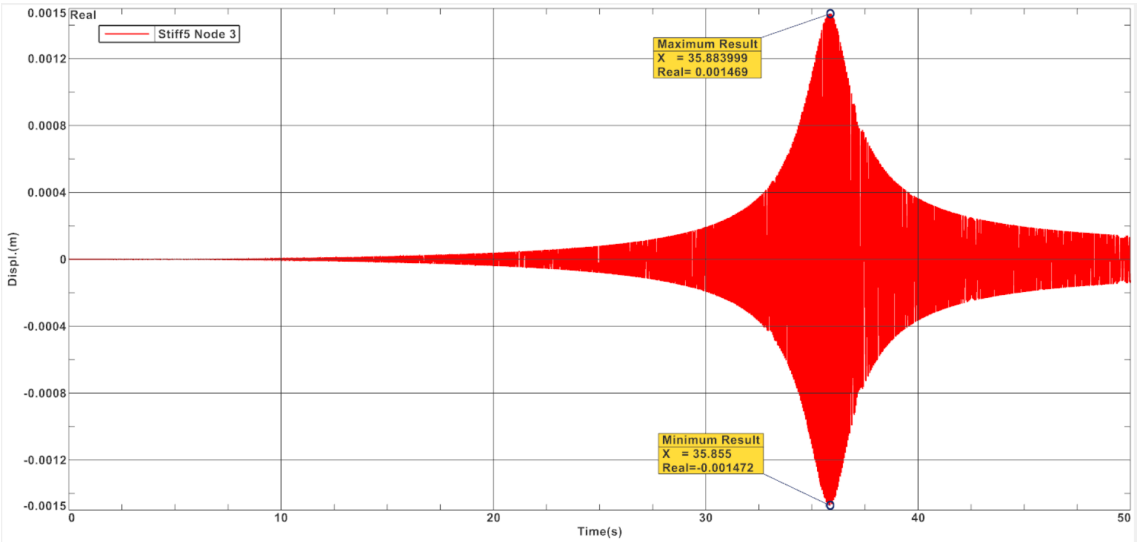


Figure F.5: K = 5e6N/m

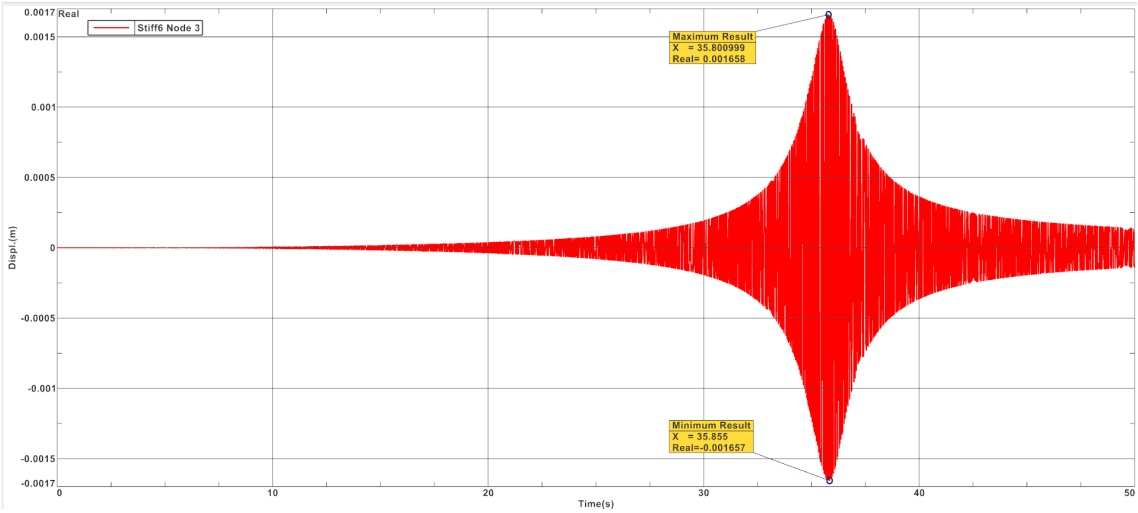


Figure F.6: K = 6e6N/m

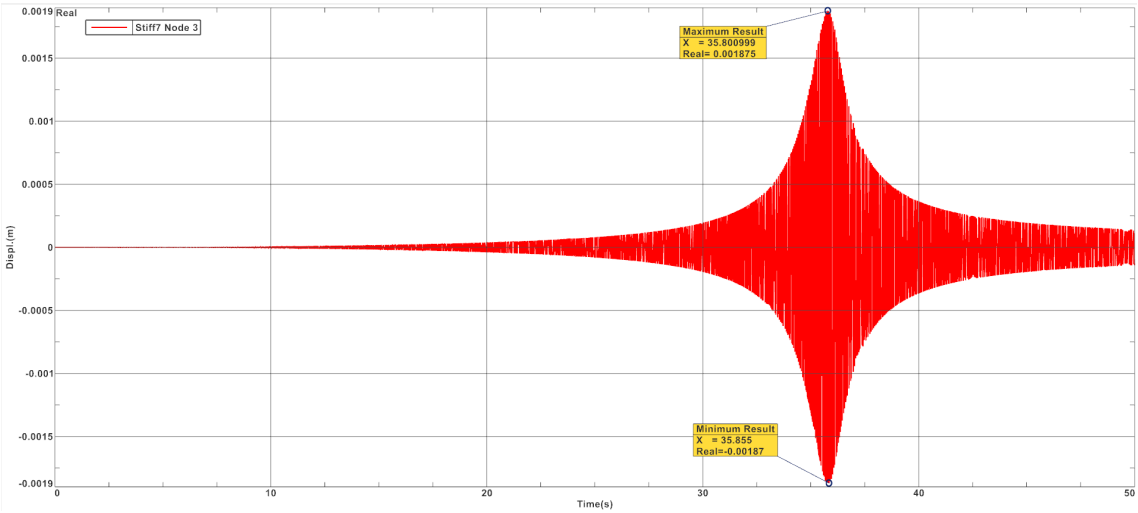


Figure F.7: K = 7e6N/m

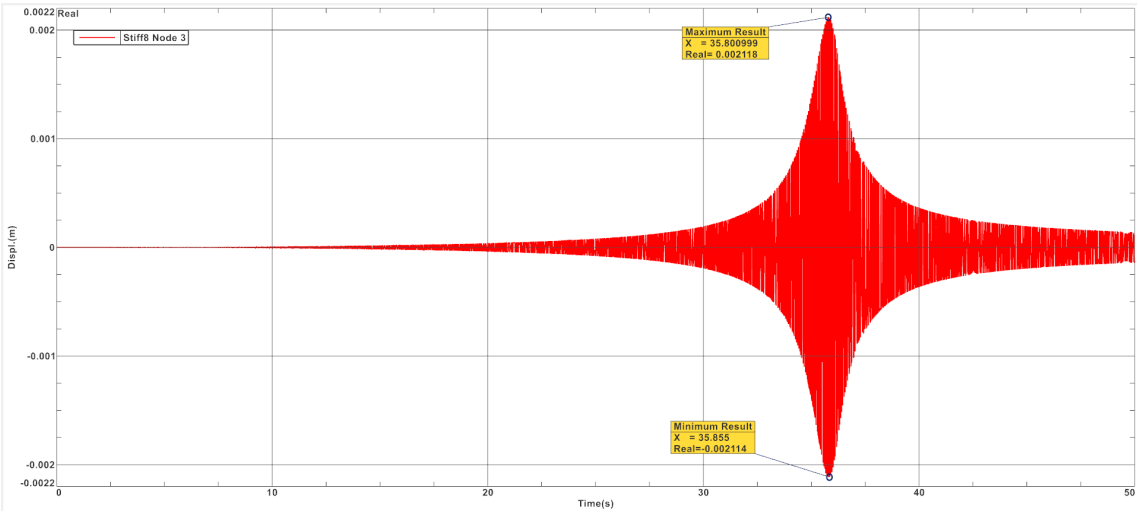


Figure F.8: K = 8e6N/m

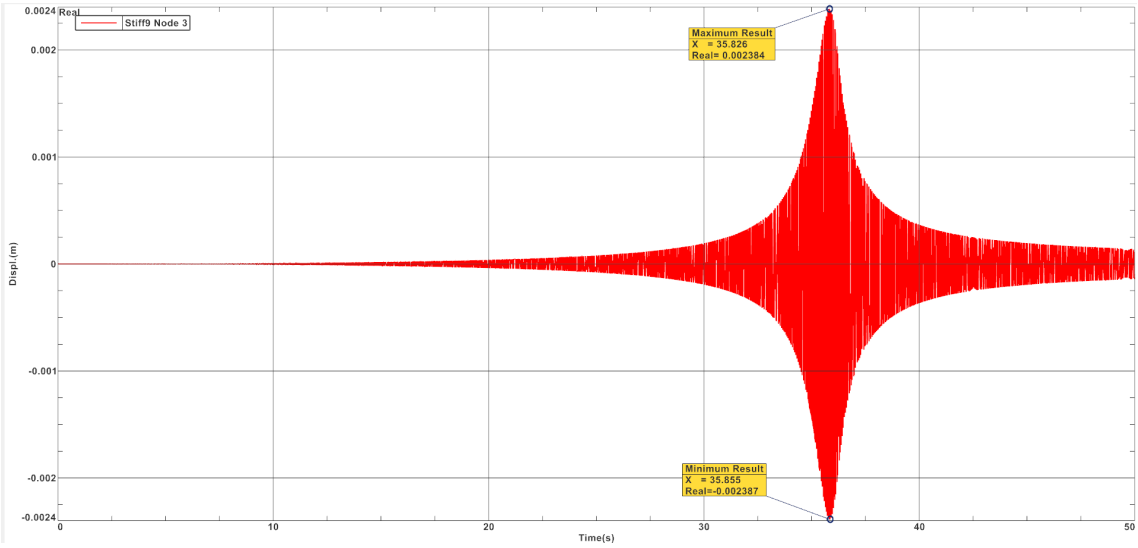


Figure F.9: K = 9e6N/m

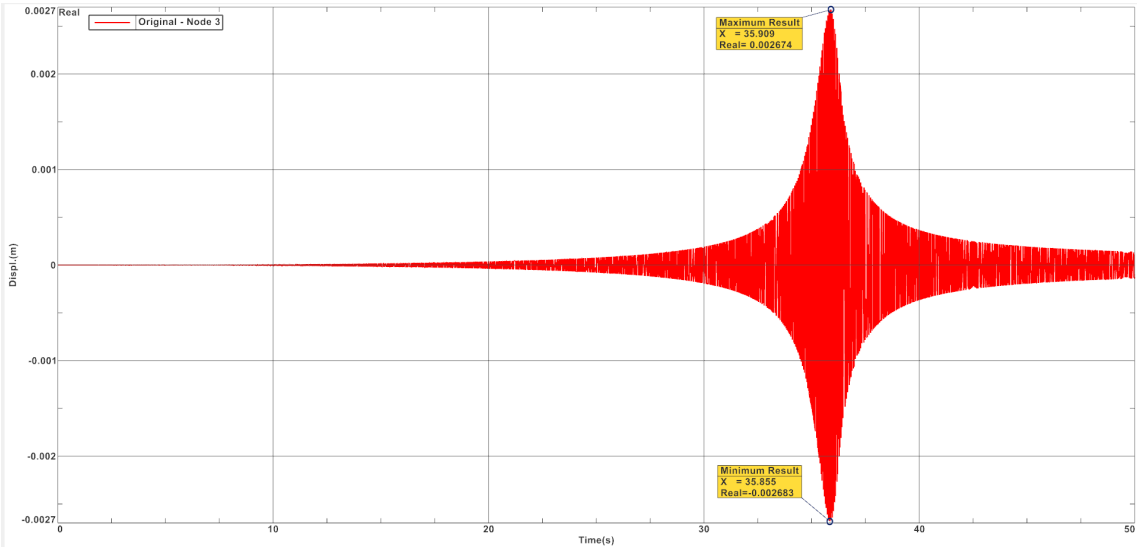


Figure F.10: K = 10e6N/m

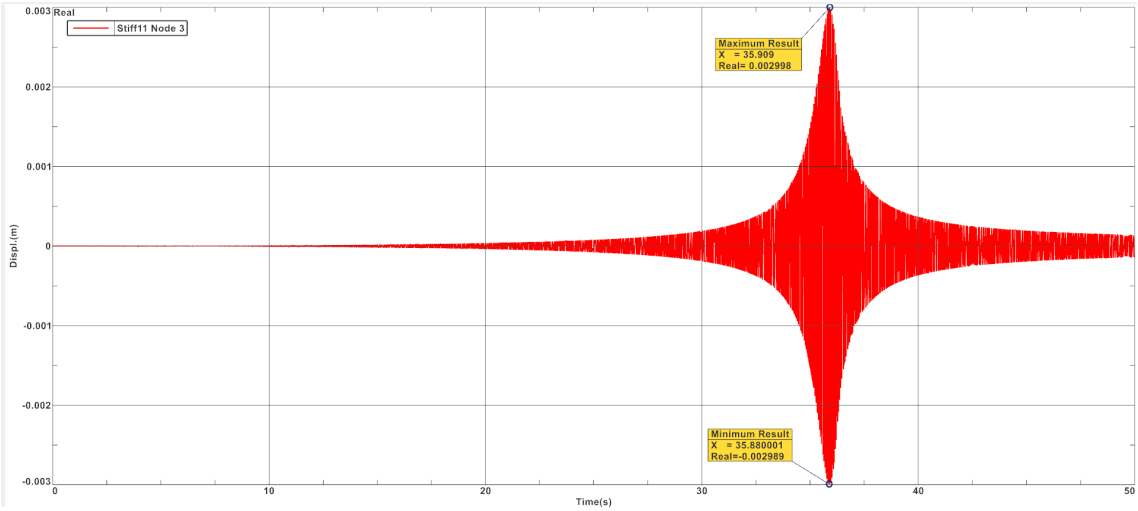


Figure F.11: K = 11e6N/m

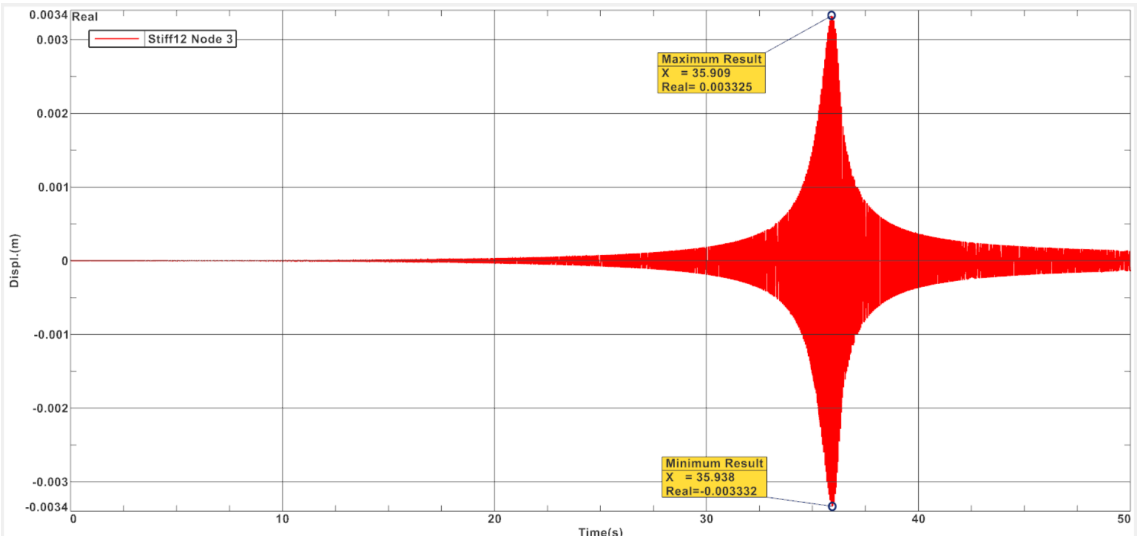


Figure F.12: K = 12e6N/m

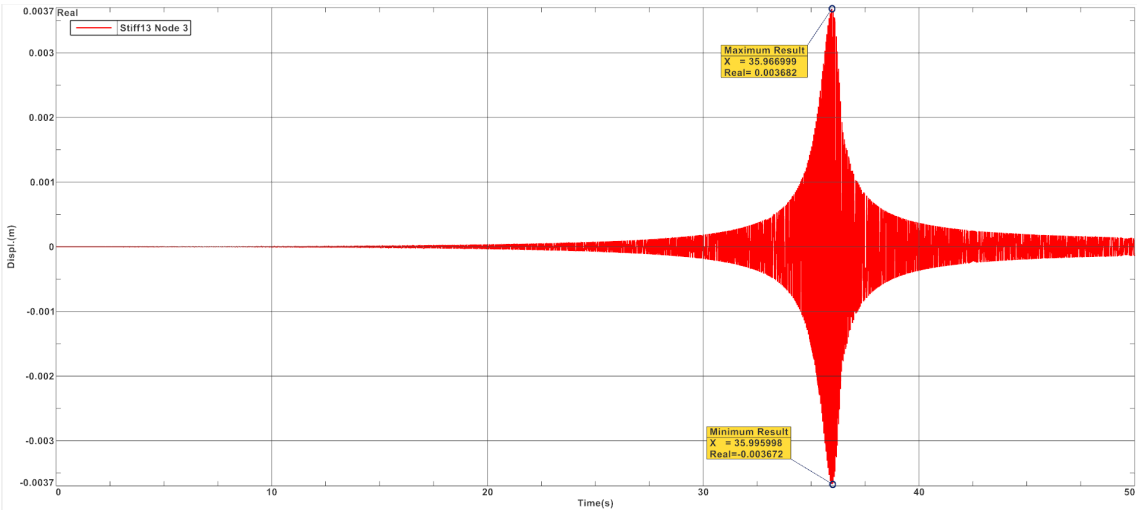


Figure F.13: K = 13e6N/m

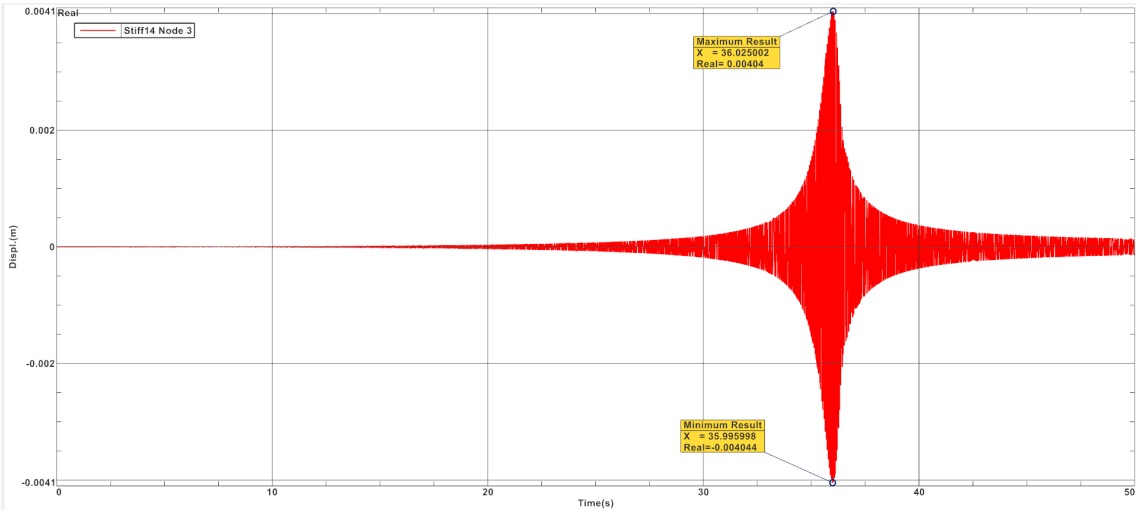


Figure F.14: K = 14e6N/m

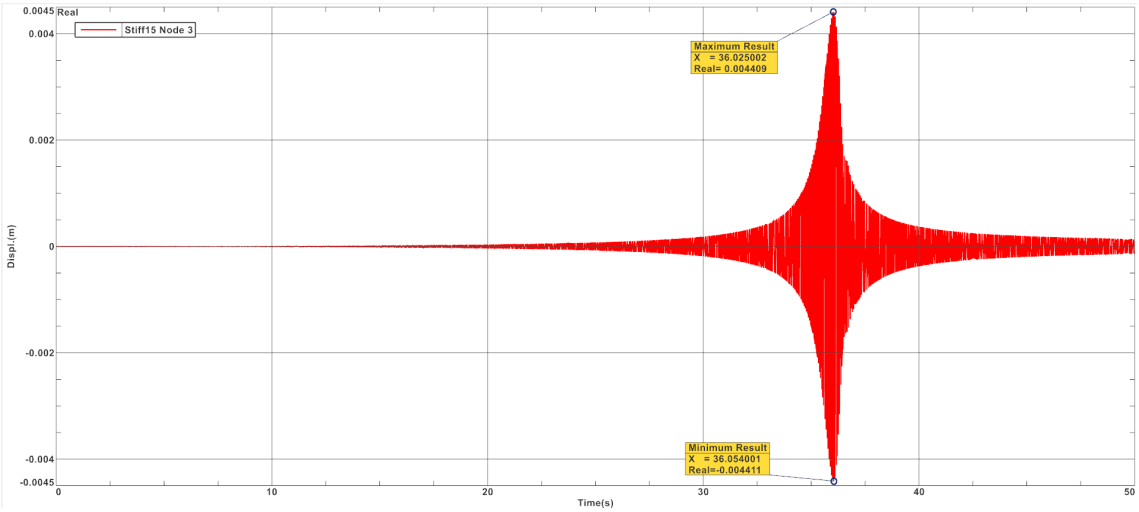


Figure F.15: K = 15e6N/m

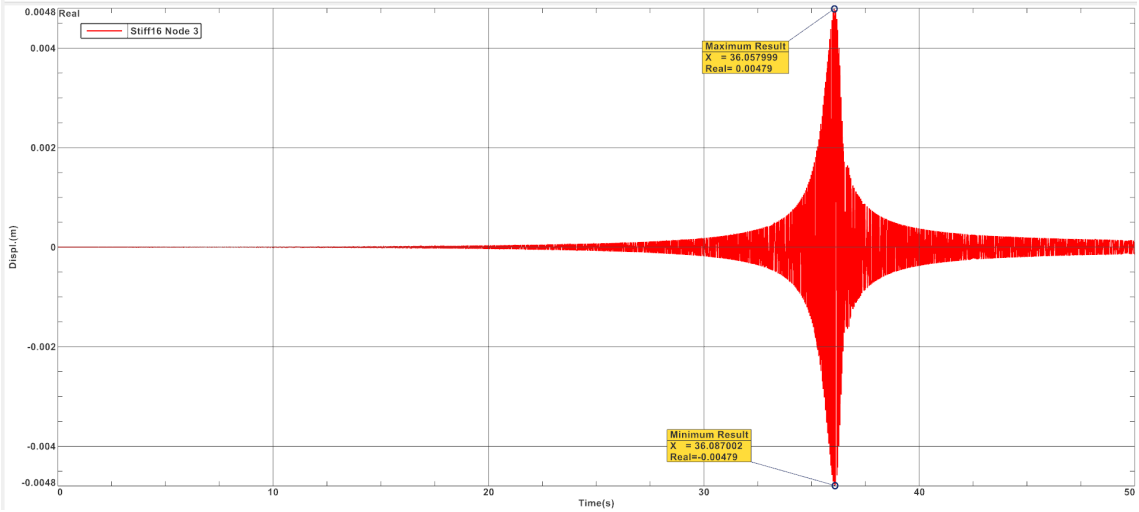


Figure F.16: K = 16e6N/m

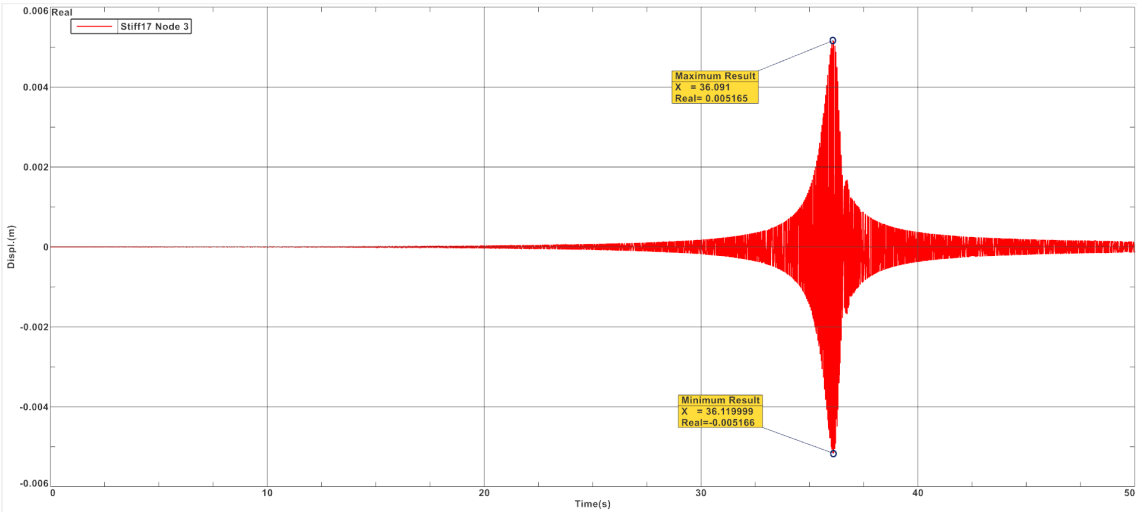


Figure F.17: K = 17e6N/m

F.2 FFT at Node 3

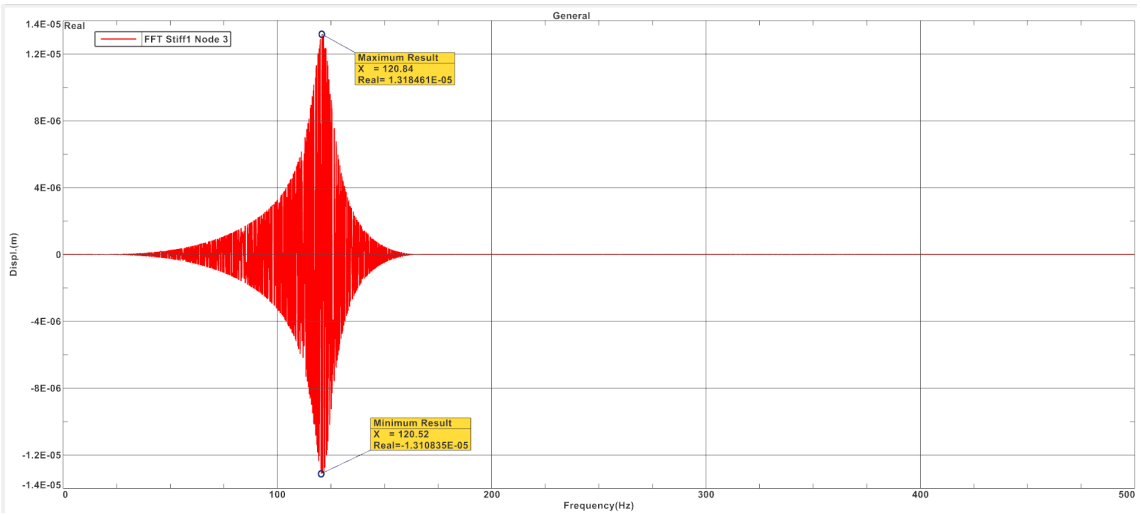


Figure F.18: K = 1e6N/m

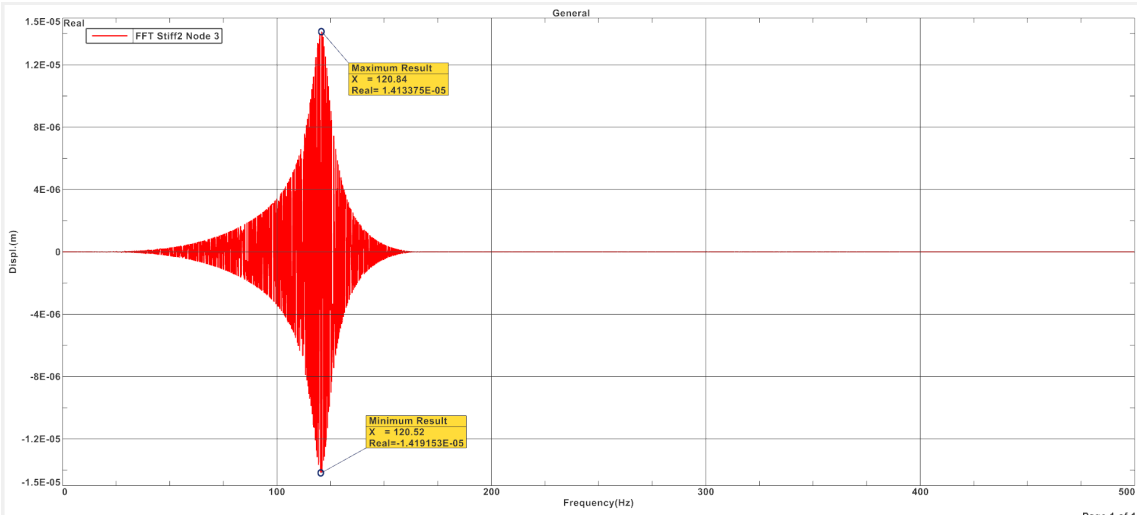


Figure F.19: K = 2e6N/m

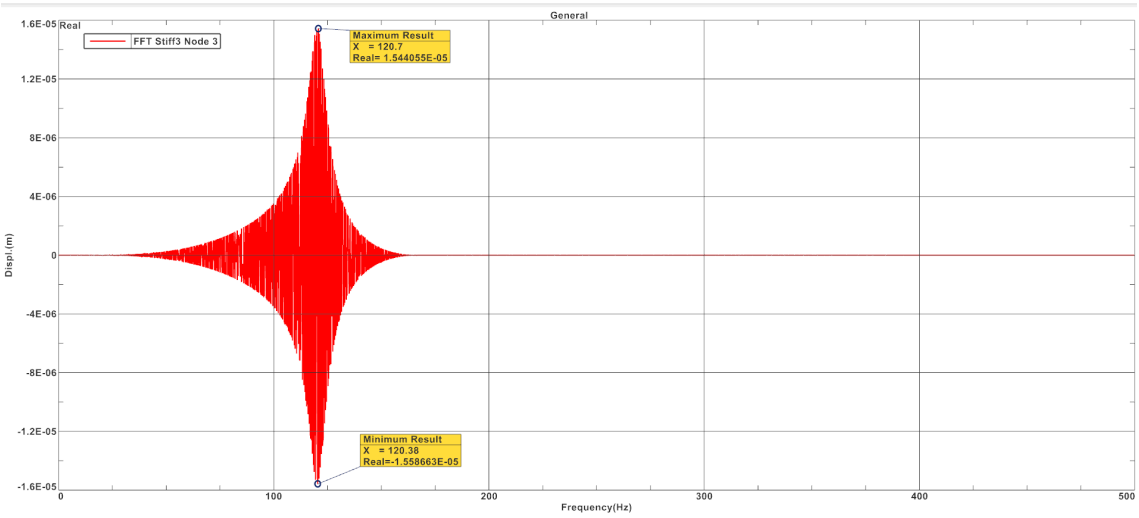


Figure F.20: K = 3e6N/m

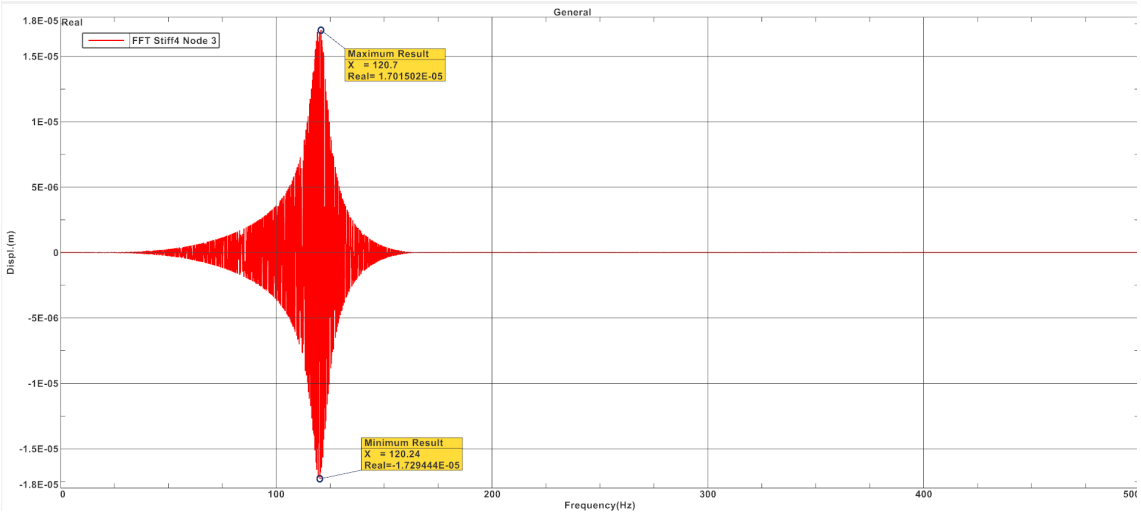


Figure F.21: K = 4e6N/m

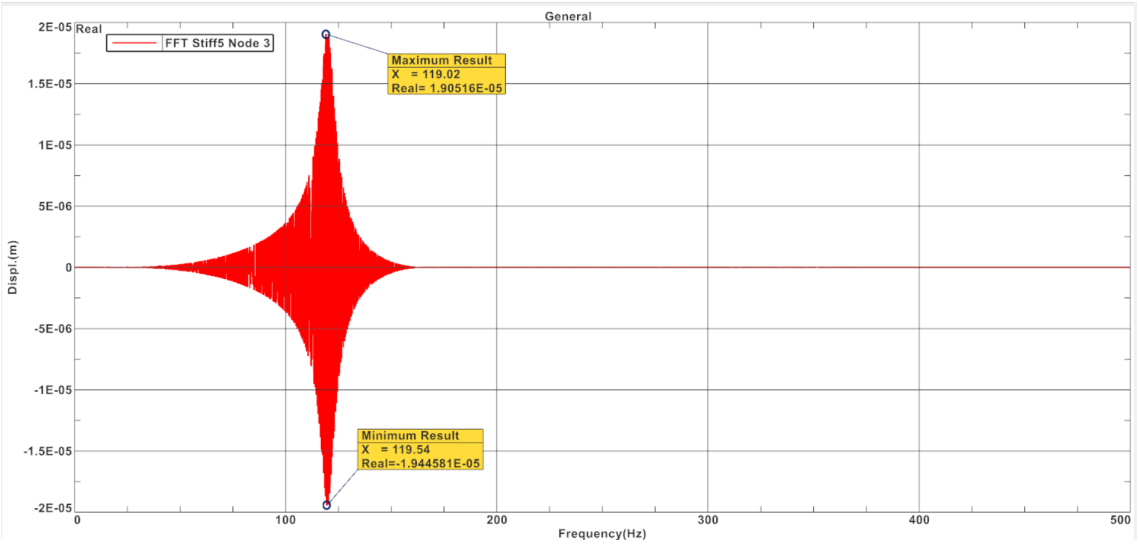


Figure F.22: K = 5e6N/m

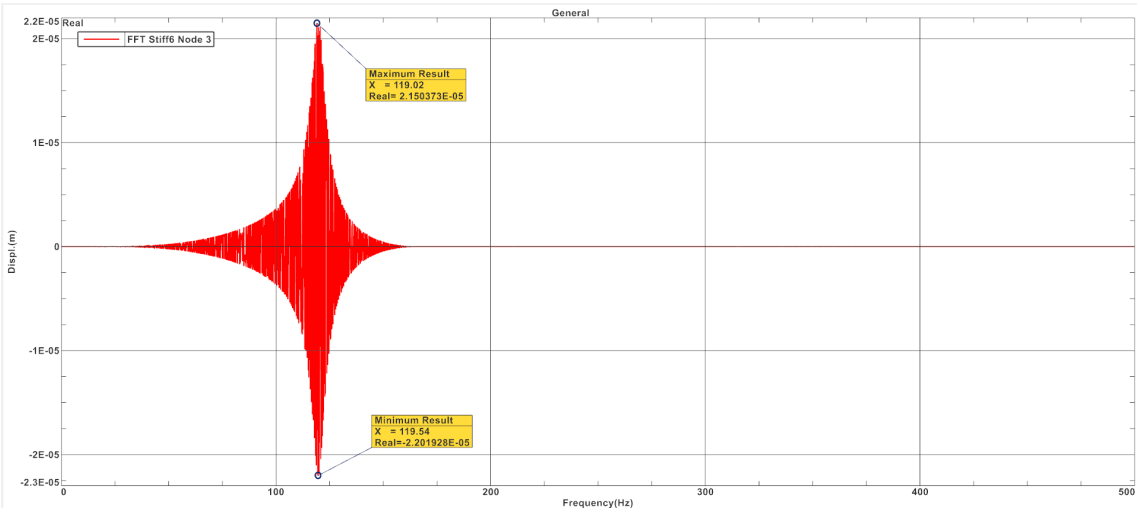


Figure F.23: K = 6e6N/m

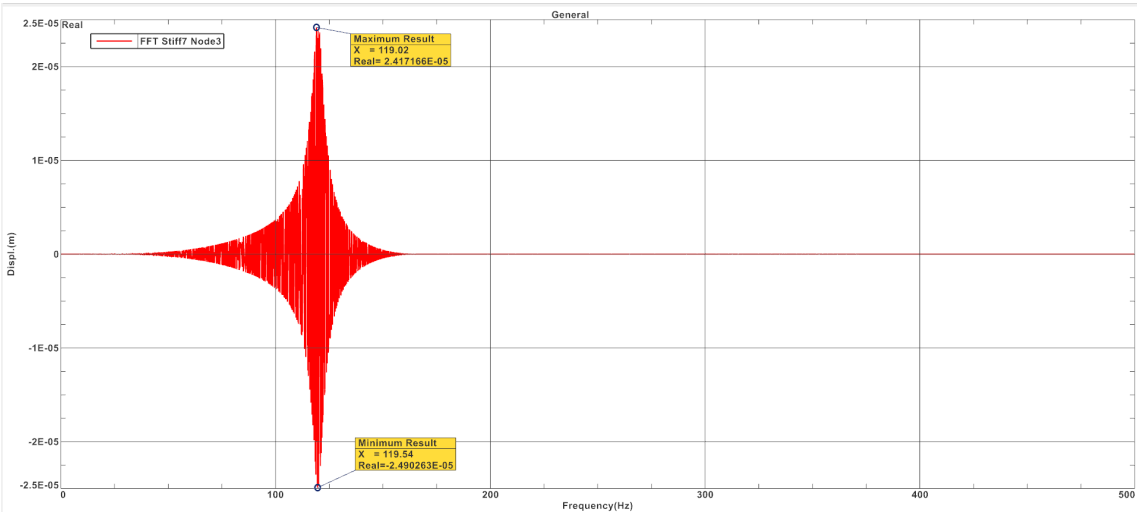


Figure F.24: K = 7e6N/m

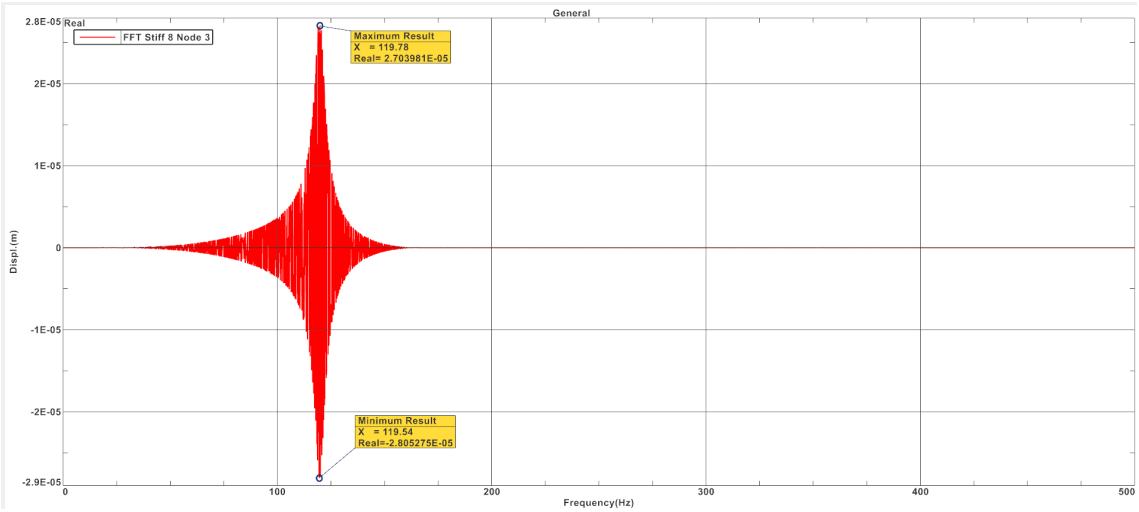


Figure F.25: K = 8e6N/m

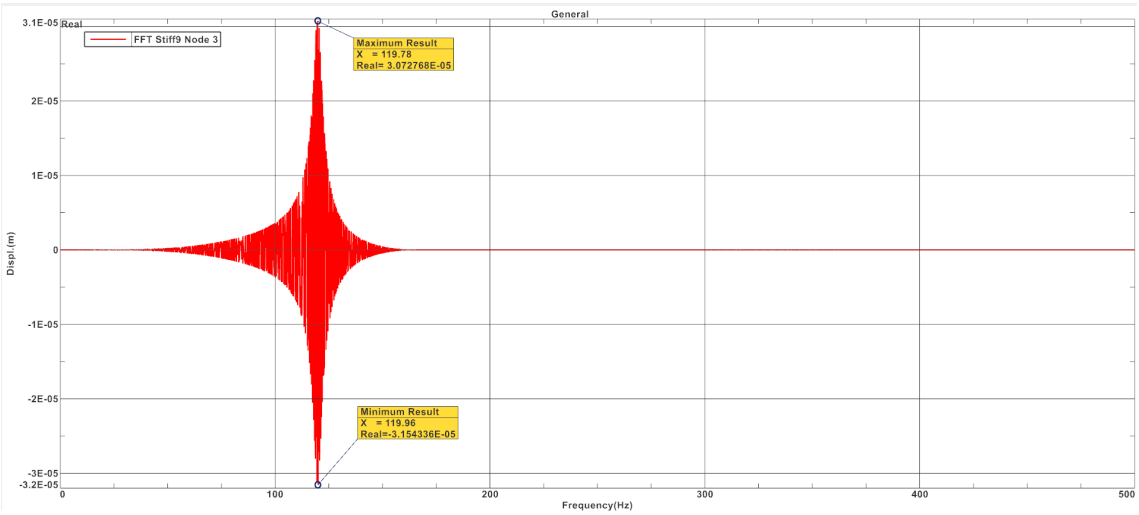


Figure F.26: K = 9e6N/m

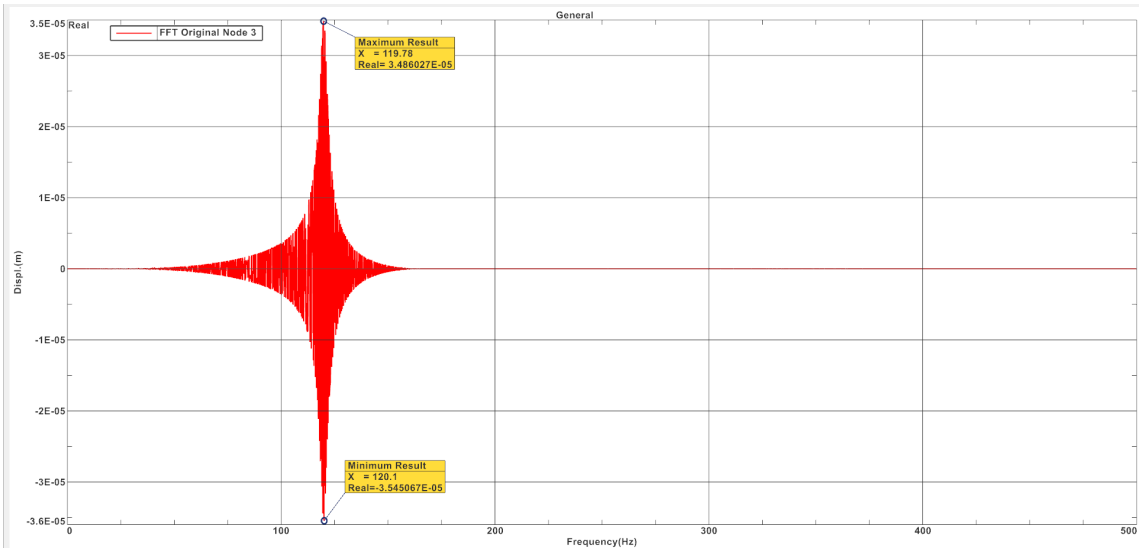


Figure F.27: K = 10e6N/m

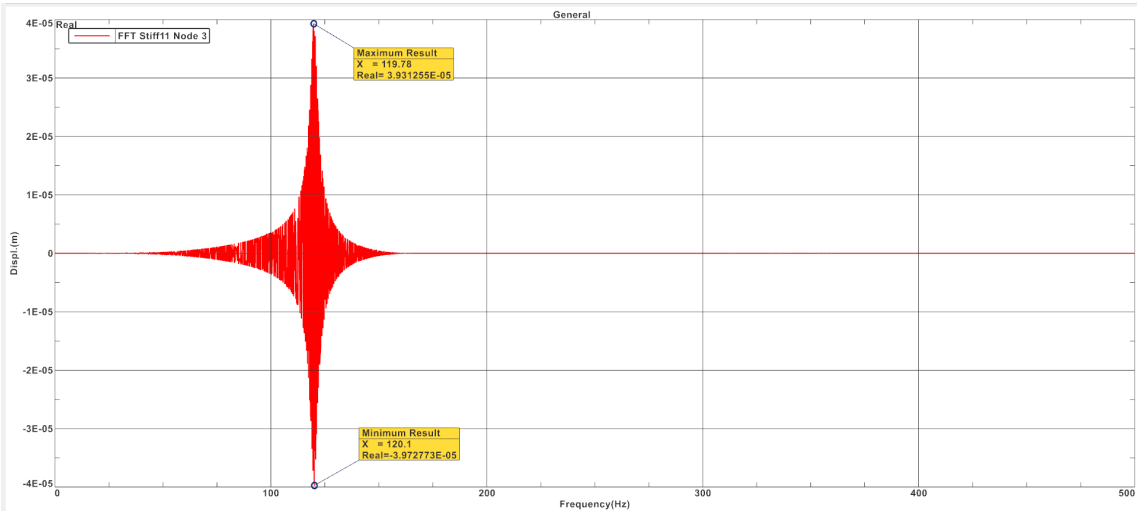


Figure F.28: K = 11e6N/m

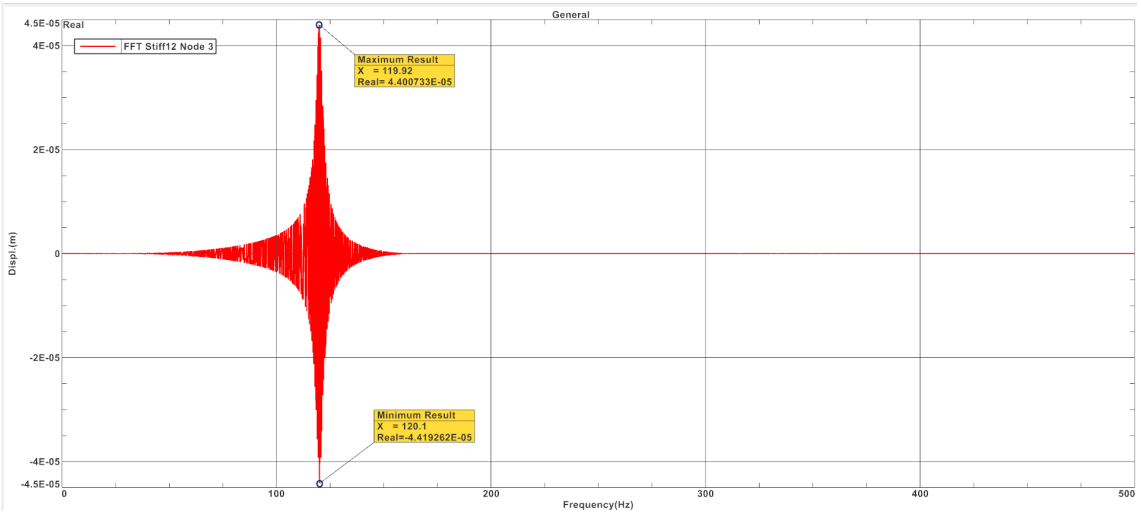


Figure F.29: K = 12e6N/m

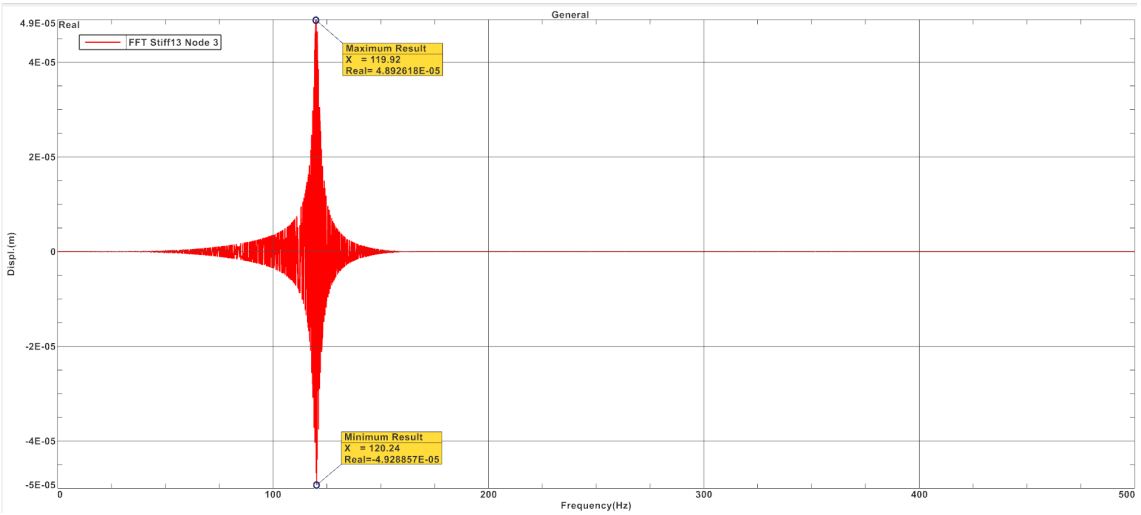


Figure F.30: K = 13e6N/m

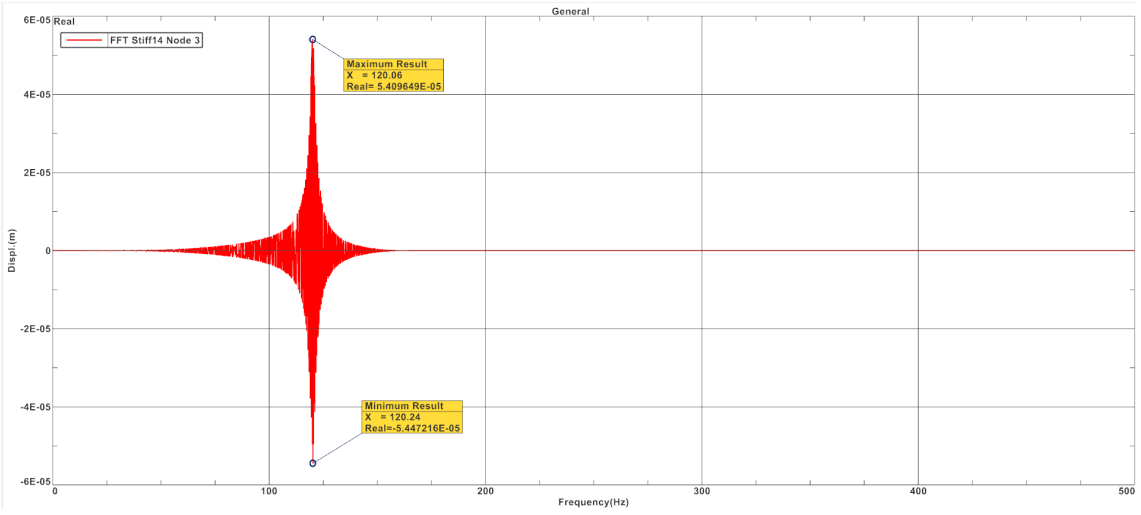


Figure F.31: K = 14e6N/m

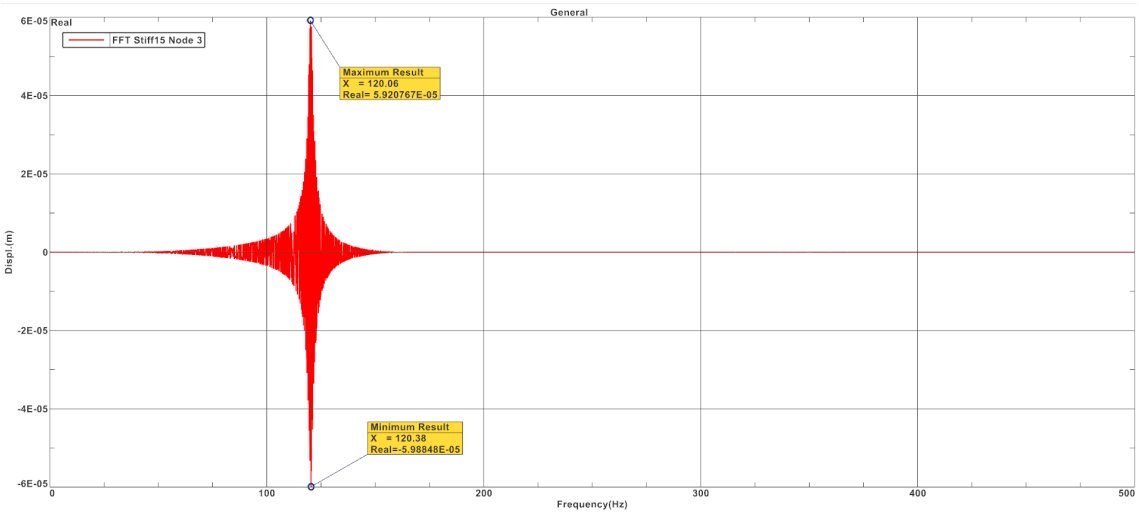


Figure F.32: K = 15e6N/m

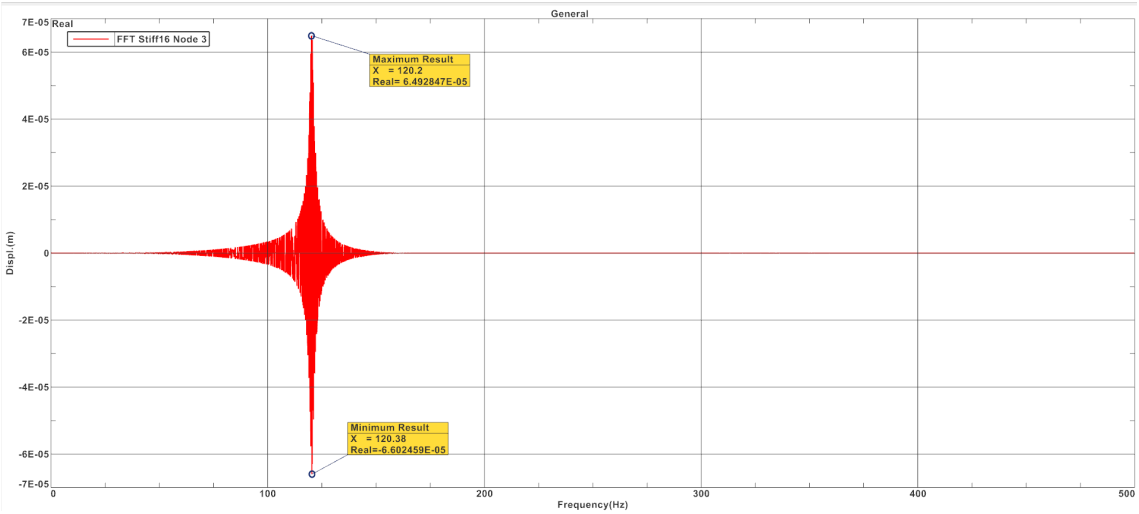


Figure F.33: K = 16e6N/m

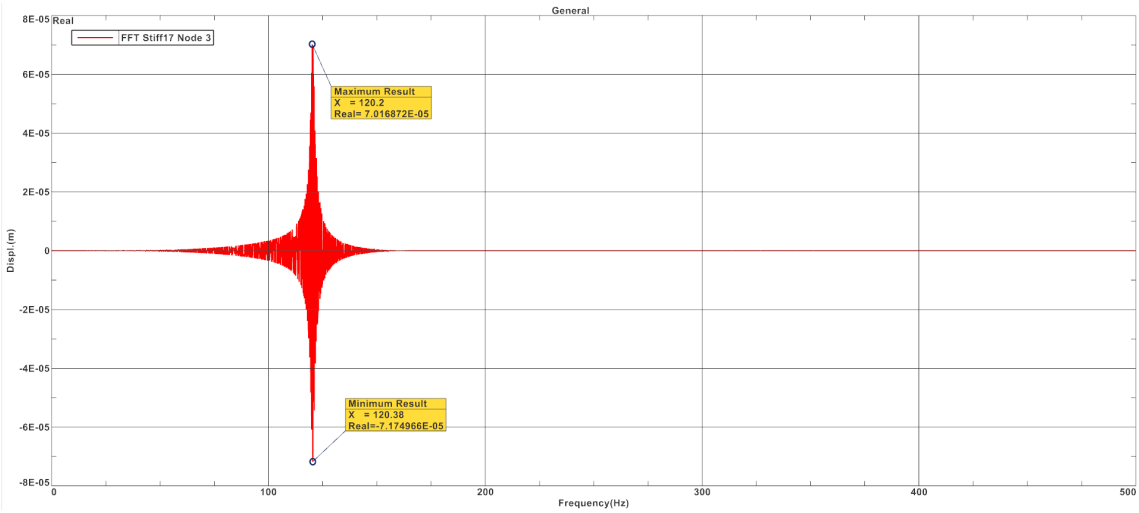


Figure F.34: K = 17e6N/m

Appendix G

Damping Plot Using Complex Modal Analysis

This appendix presents the results of the complex modal analysis in the form of a Campbell diagram. In these plots, the radial stiffness translation is held constant at 10e6N/m, while the axial translation is maintained at a constant value of 1e6N/m. The damping coefficient C11 and C22 are varied using a common coefficient, denoted as c. The Campbell diagram illustrates the relationship between the rotor speed (expressed in RPM) and the natural frequencies of the system for different damping coefficients.

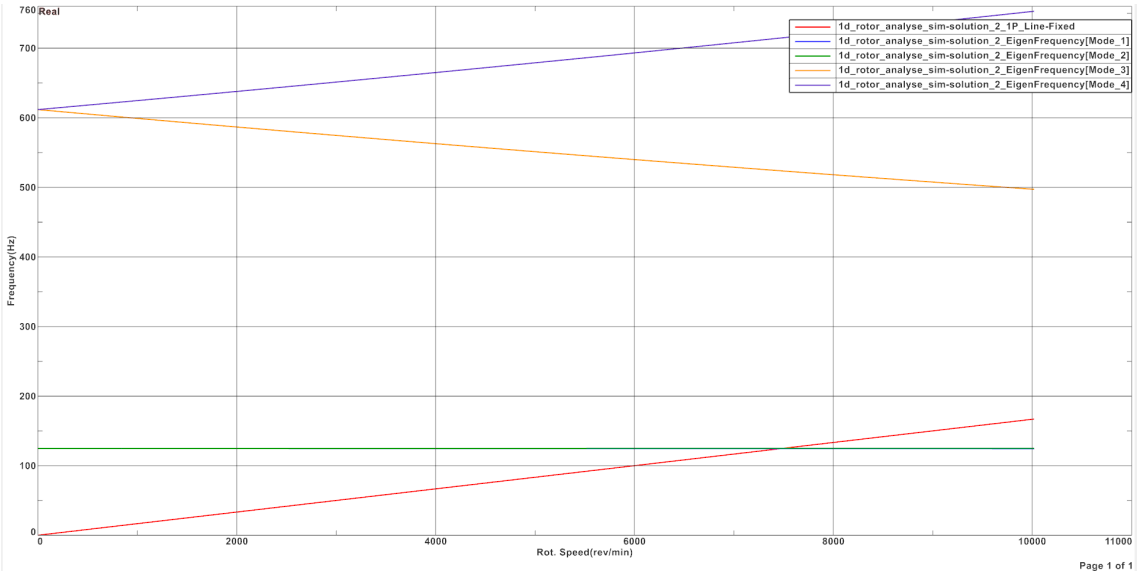


Figure G.1: c = = 1e3Ns/m

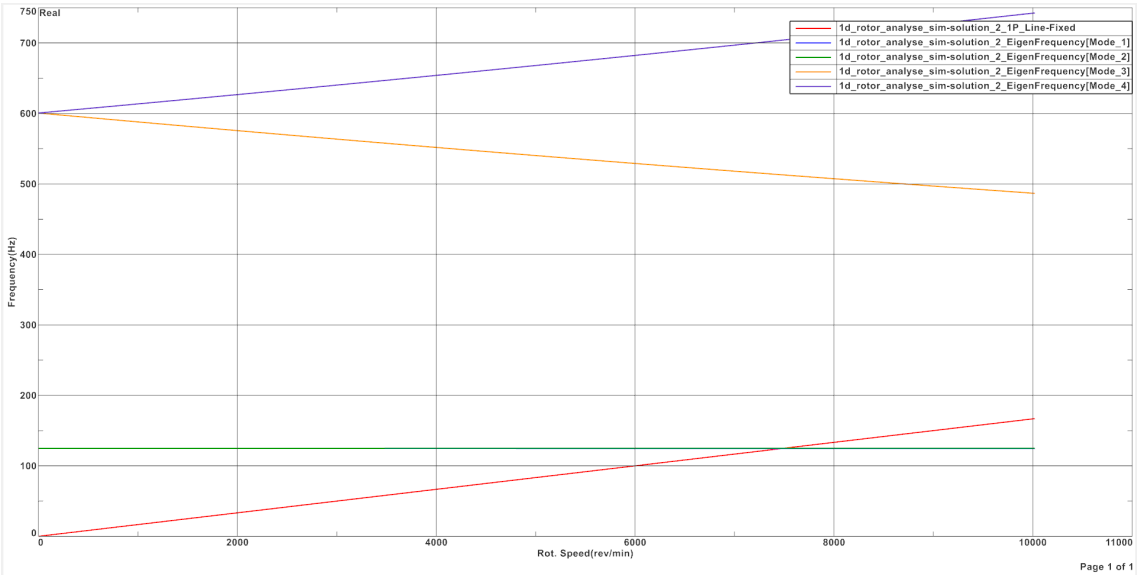


Figure G.2: c = 20e3

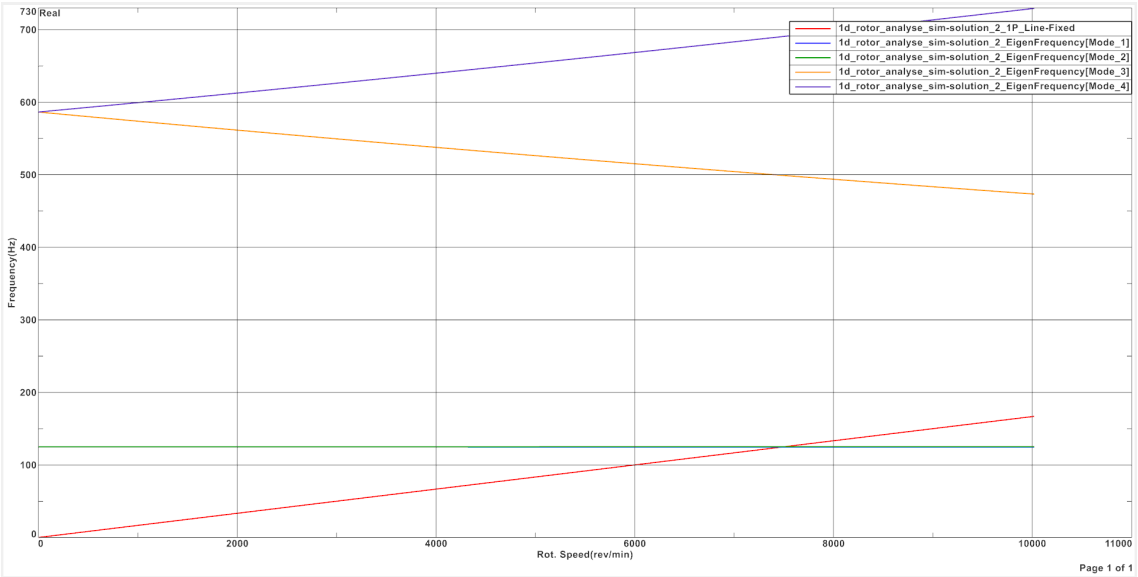


Figure G.3: c = 30e3Ns/m

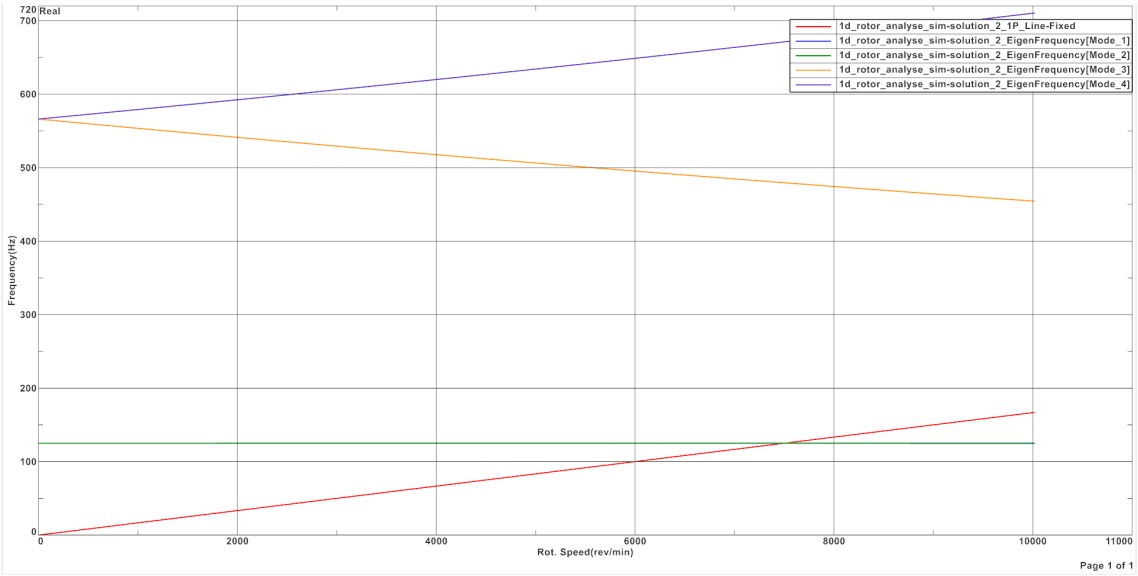


Figure G.4: c = 40e3Ns/m

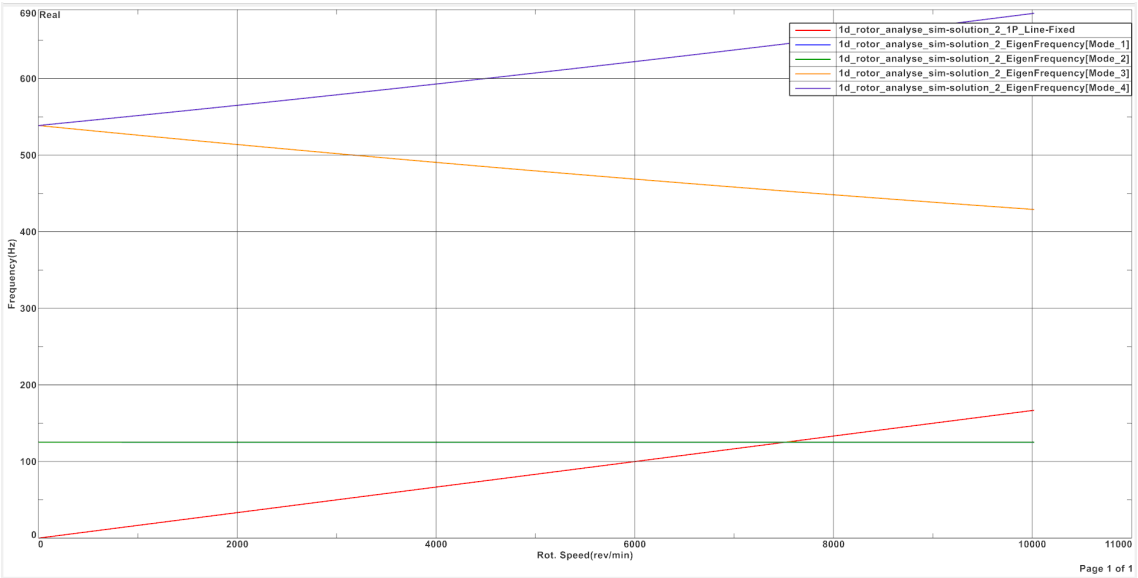


Figure G.5: c = 50e3Ns/m

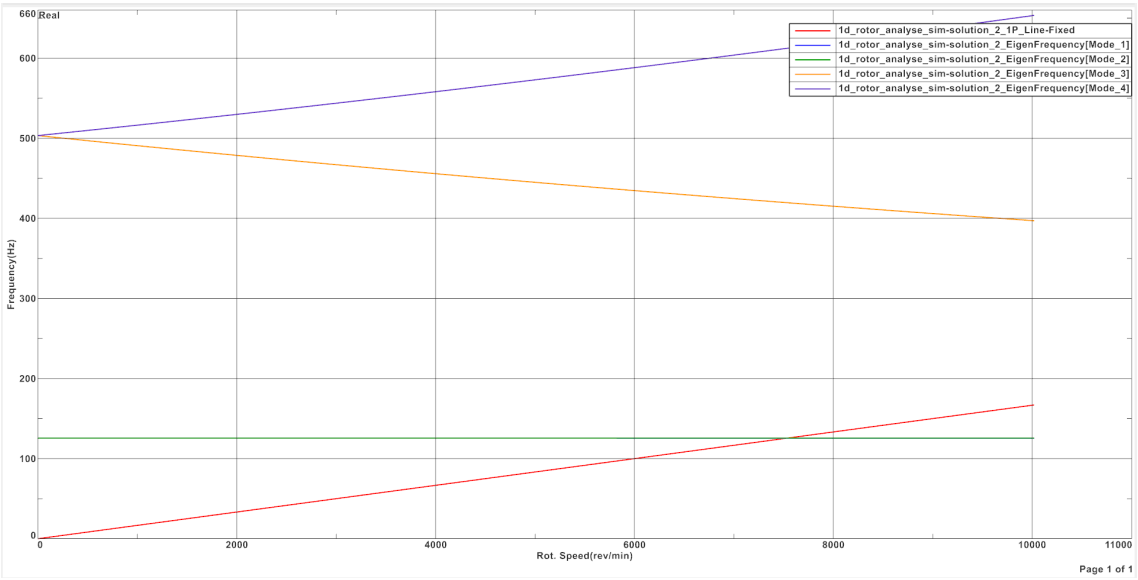


Figure G.6: c = 60e3Ns/m

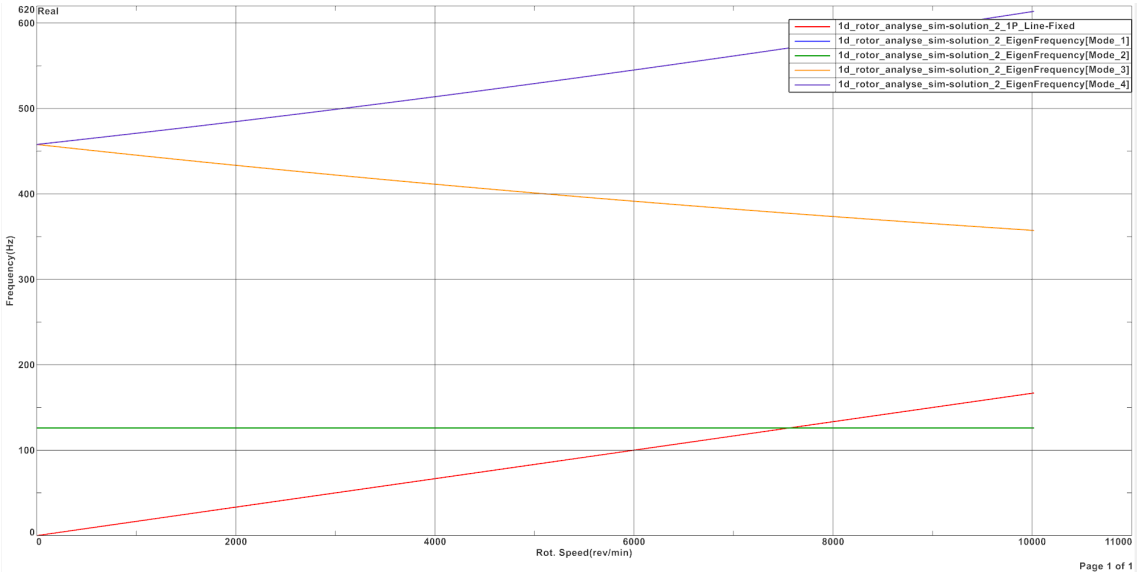


Figure G.7: c = 70e3Ns/m

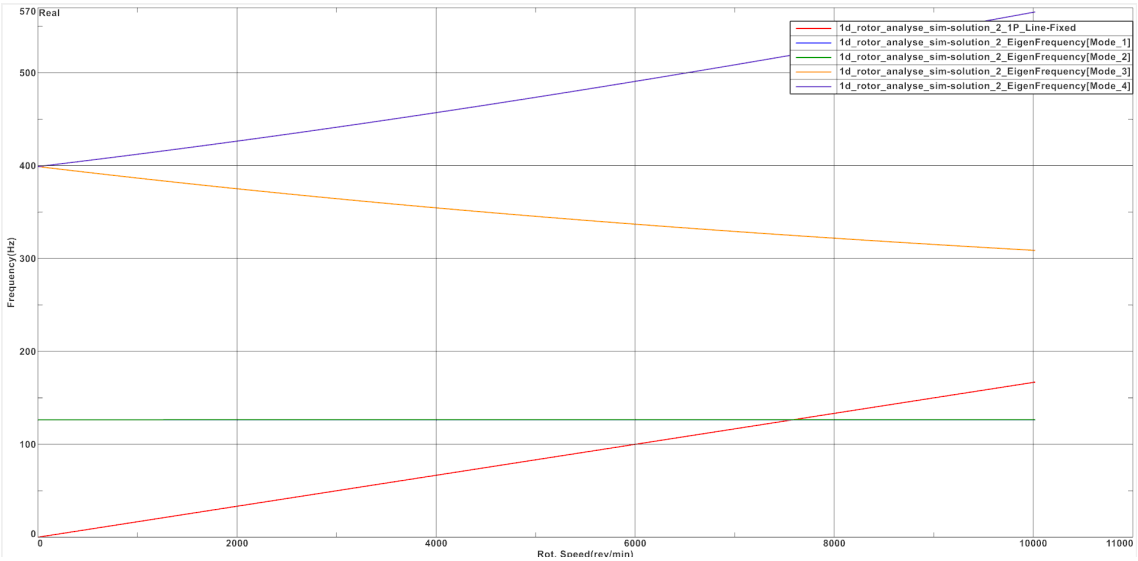


Figure G.8: c = 80e3Ns/m

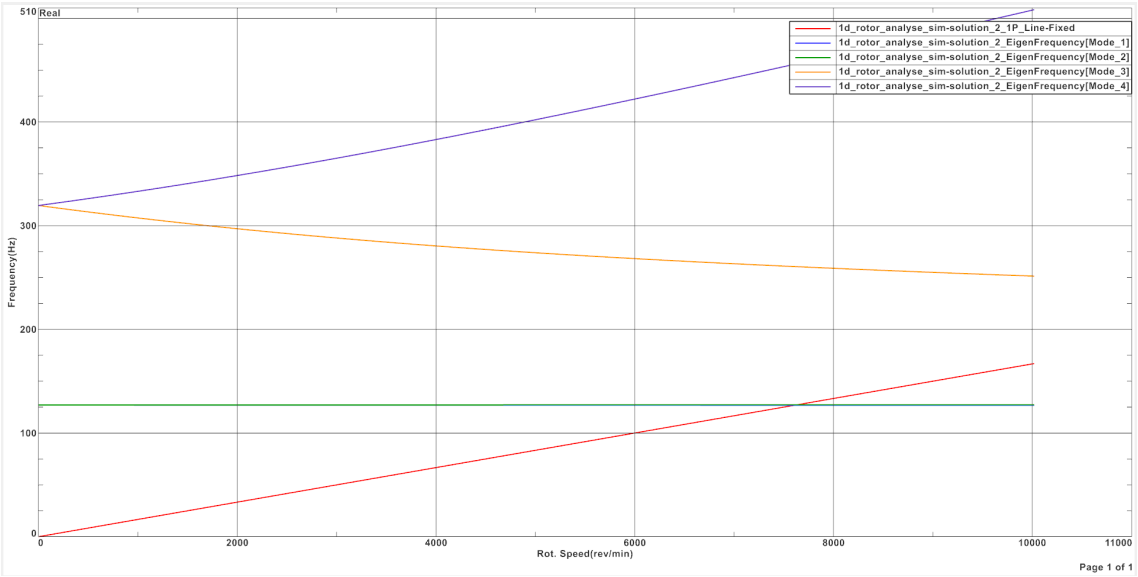


Figure G.9: c = 90e3Ns/m

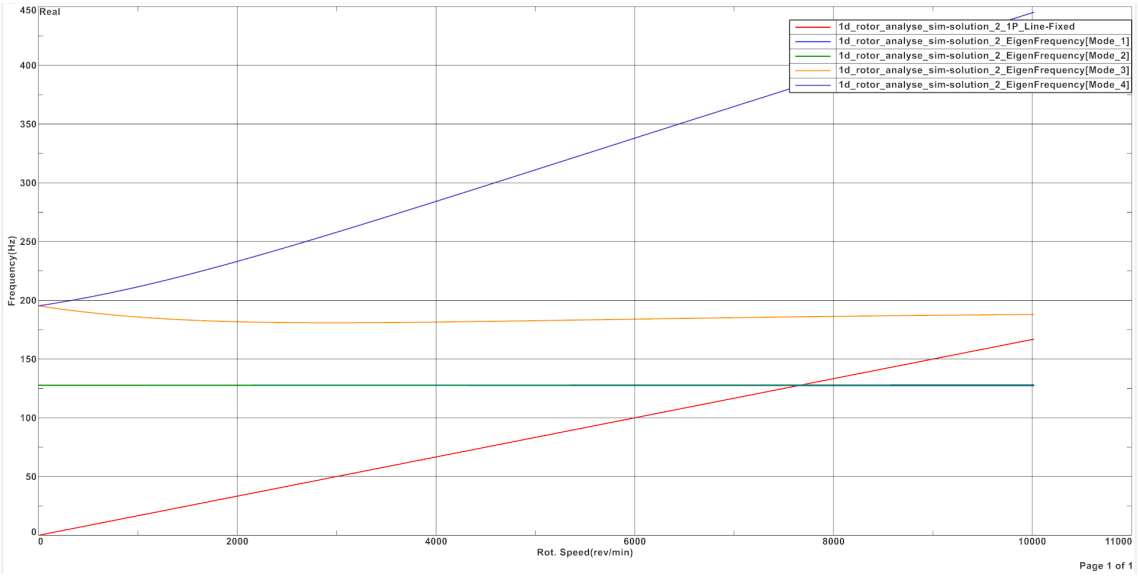


Figure G.10: c = 100e3Ns/m

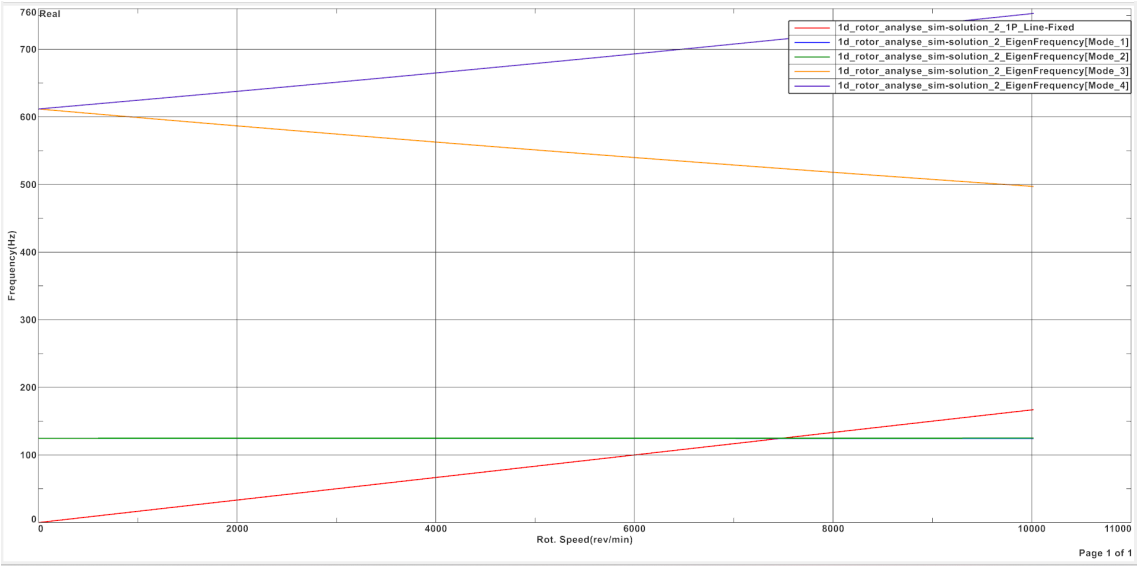


Figure G.11: c = 1Ns/m

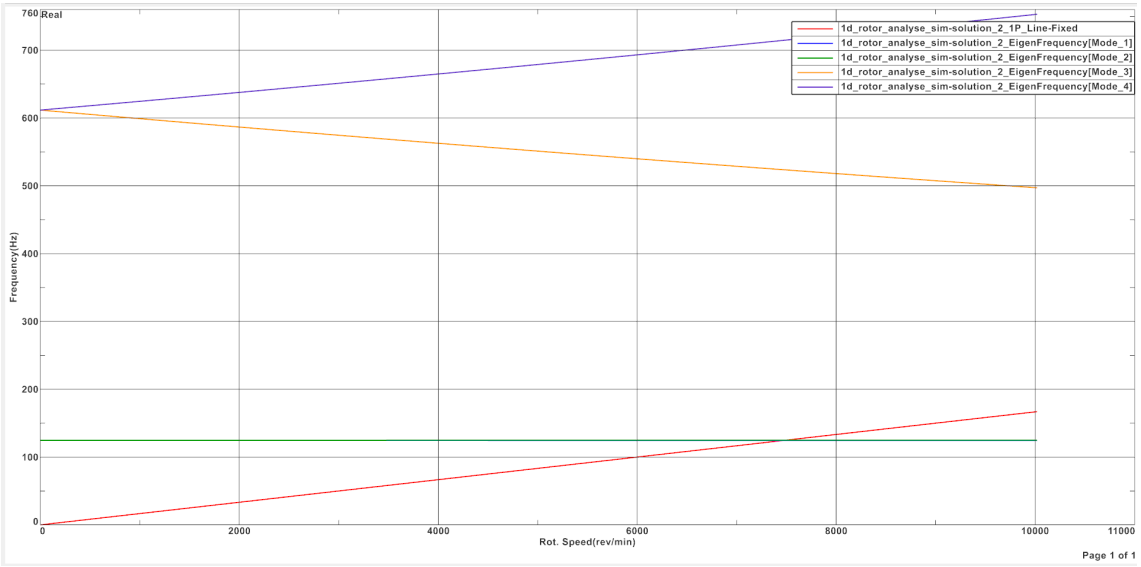


Figure G.12: c = 1e1Ns/m

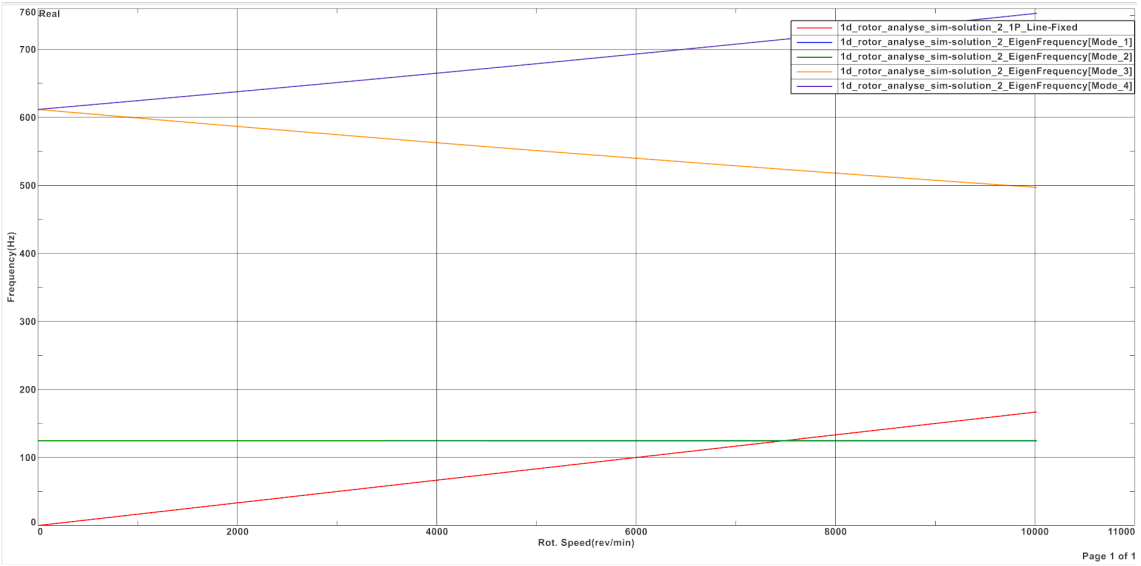


Figure G.13: c = 1e2Ns/m

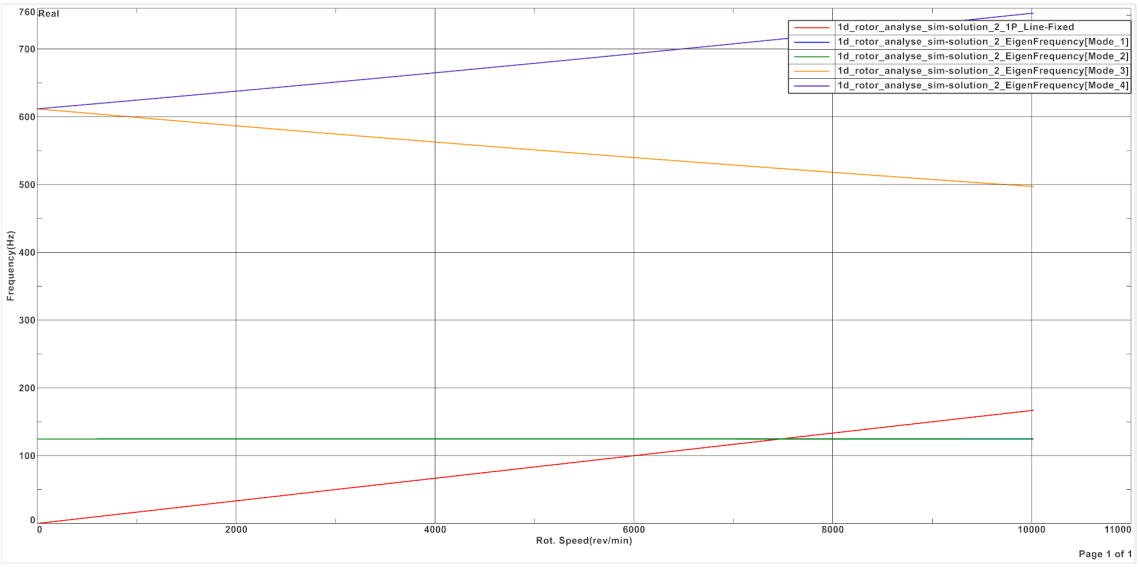


Figure G.14: c = 1e3Ns/m

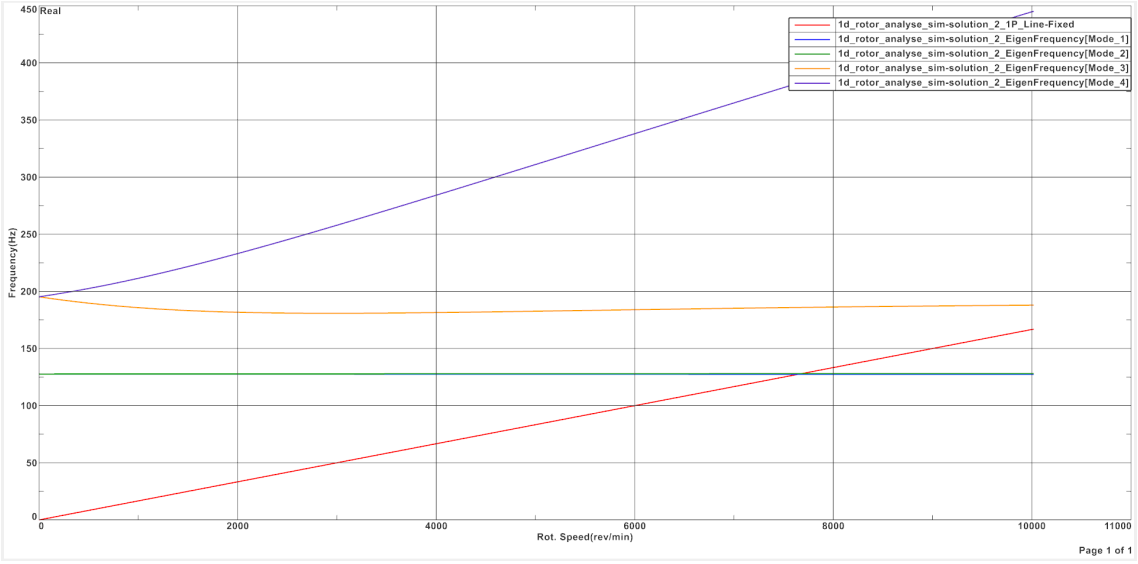


Figure G.15: c = 1e5

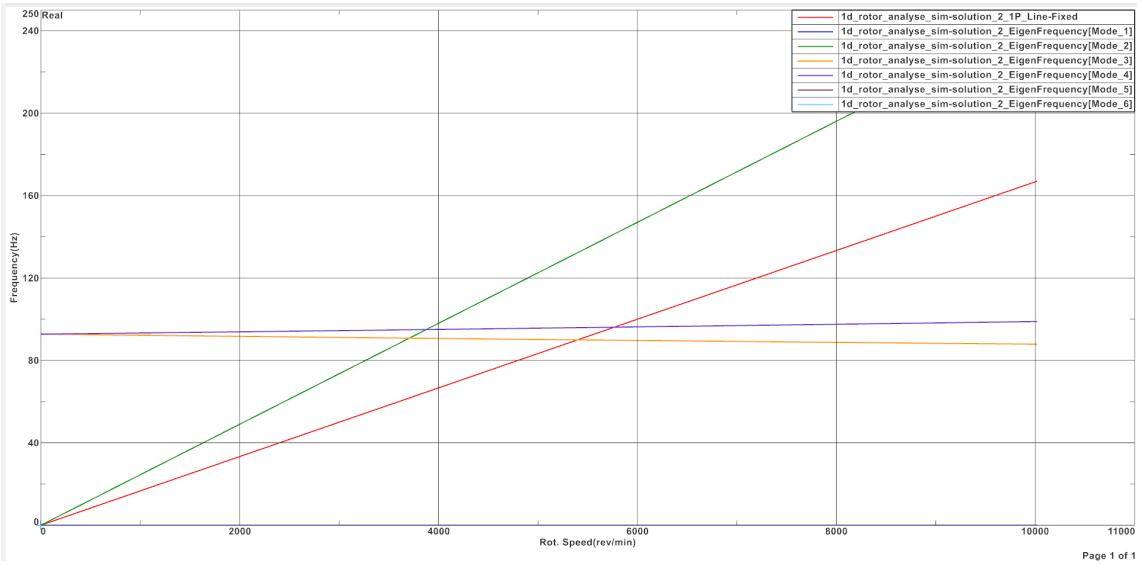


Figure G.16: c = 1e6Ns/m

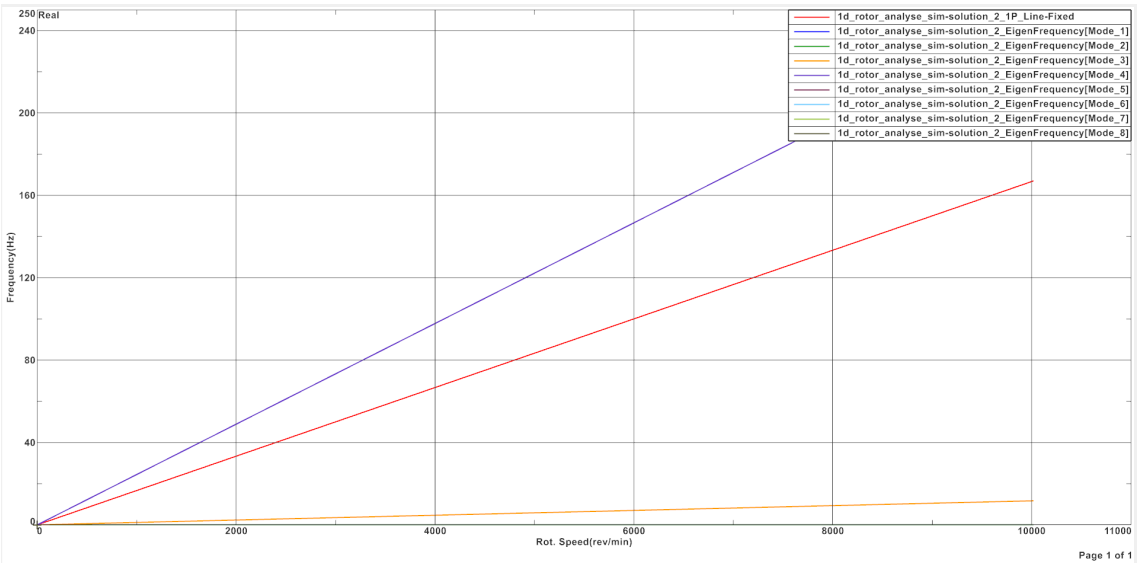


Figure G.17: c = 1e7Ns/m

Appendix H

Stiffness Plot Using Complex Modal Analysis

In this appendix, the complex modal analysis results are presented as a Campbell diagram. The damping coefficient (C11 = C22) is kept constant at 10e3Ns/m, while the axial stiffness (K33) is set to 1e6N/m. The radial stiffness, k, is varied to observe its effect on the system's natural frequencies.

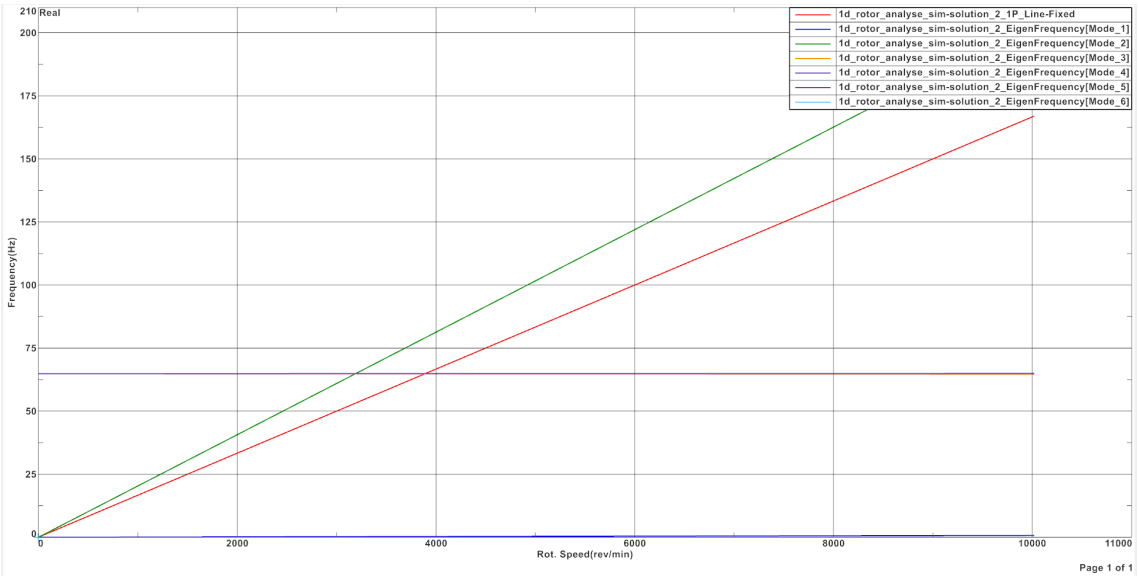


Figure H.1: k = 1e6N/m

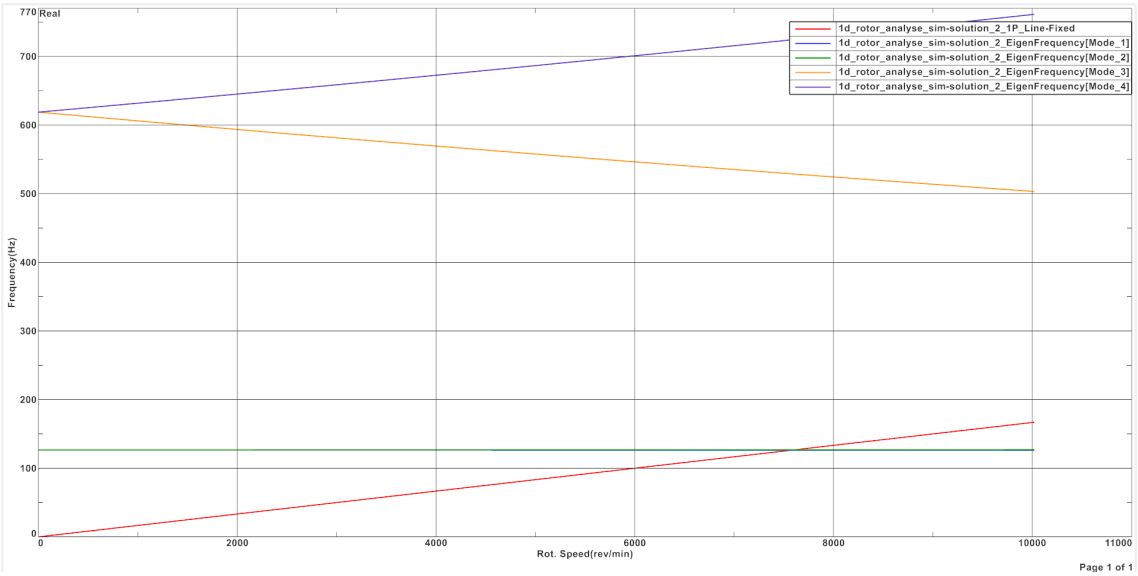


Figure H.2: k = 20e6N/m

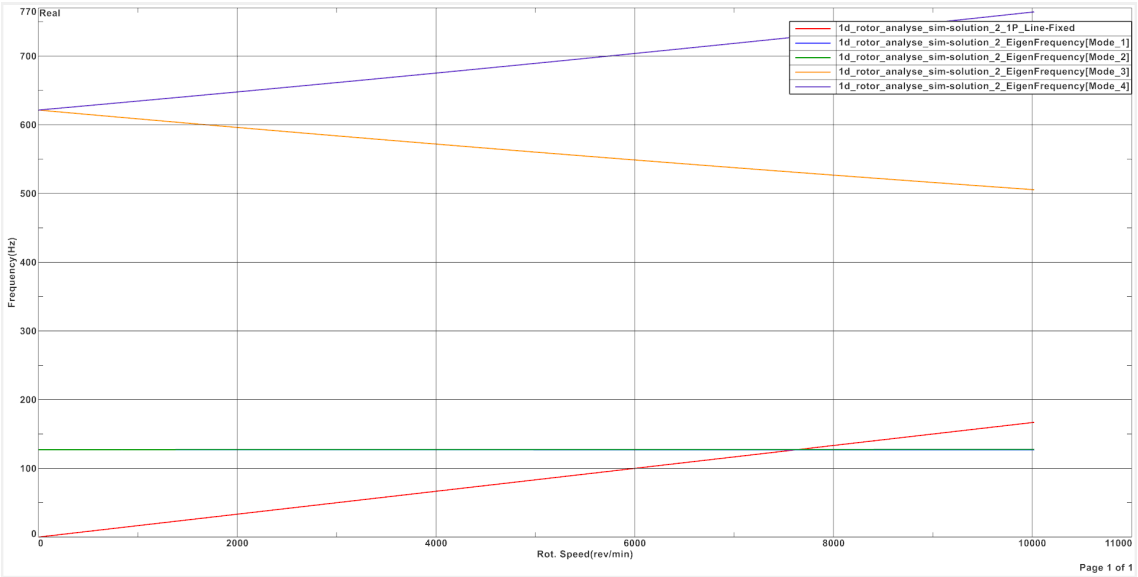


Figure H.3: k = 30e6N/m

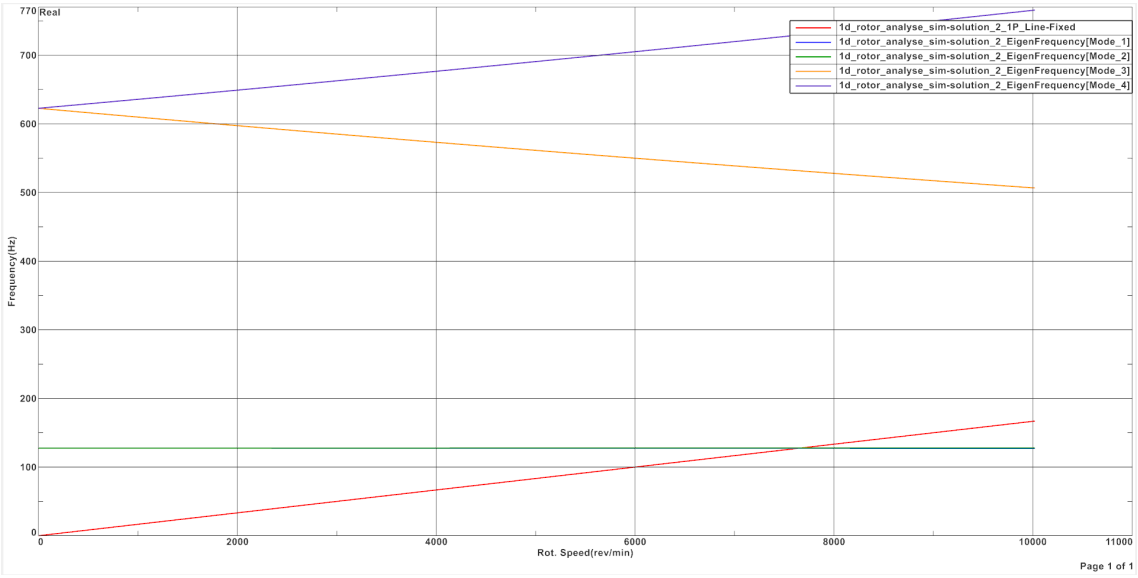


Figure H.4: k = 40e6N/m

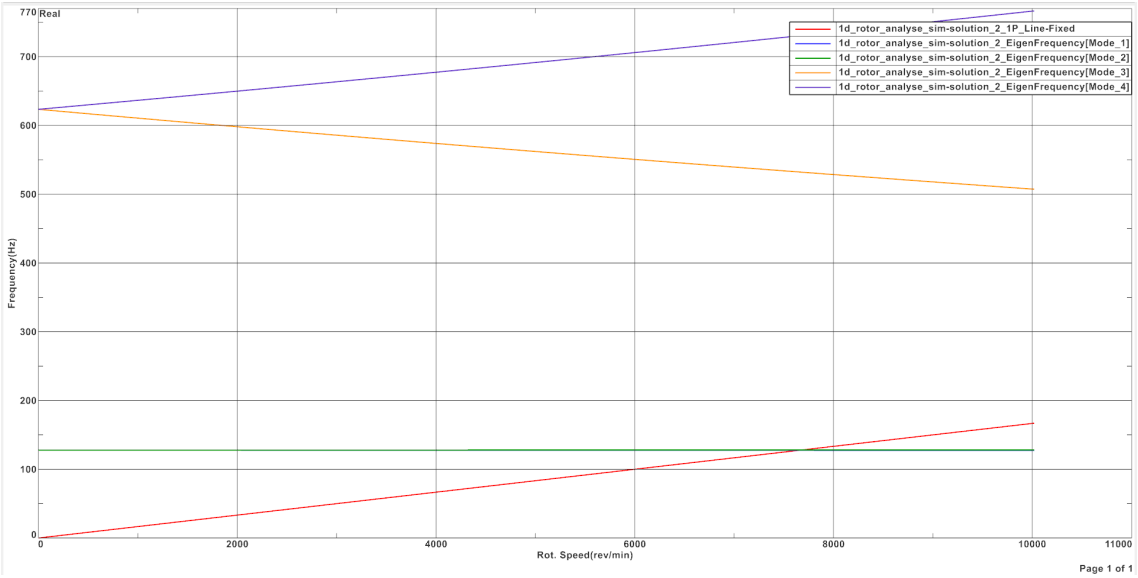


Figure H.5: k = 50e6N/m

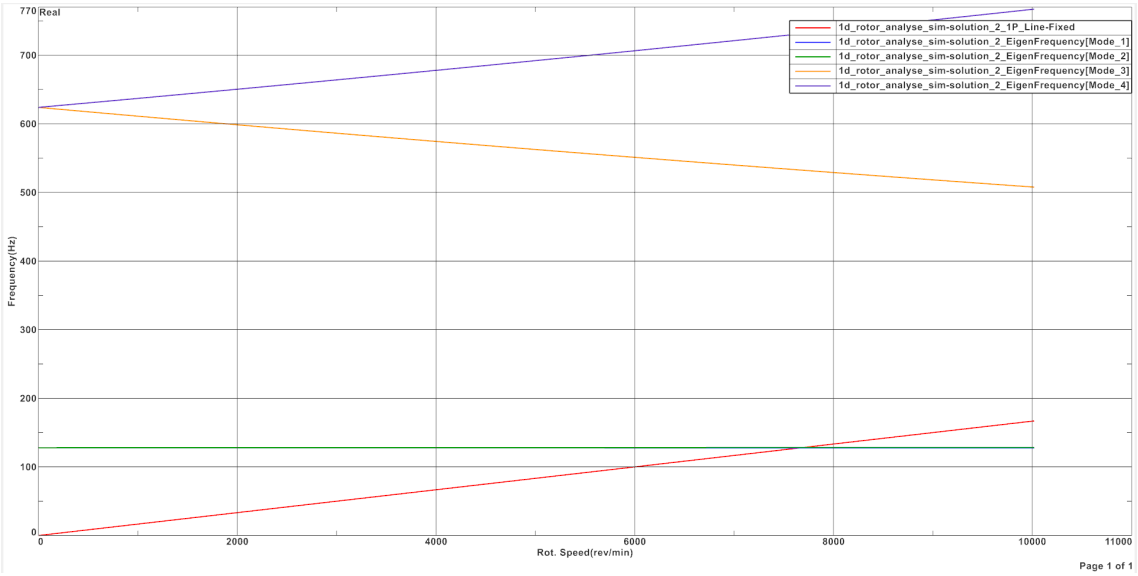


Figure H.6: k = 60e6N/m

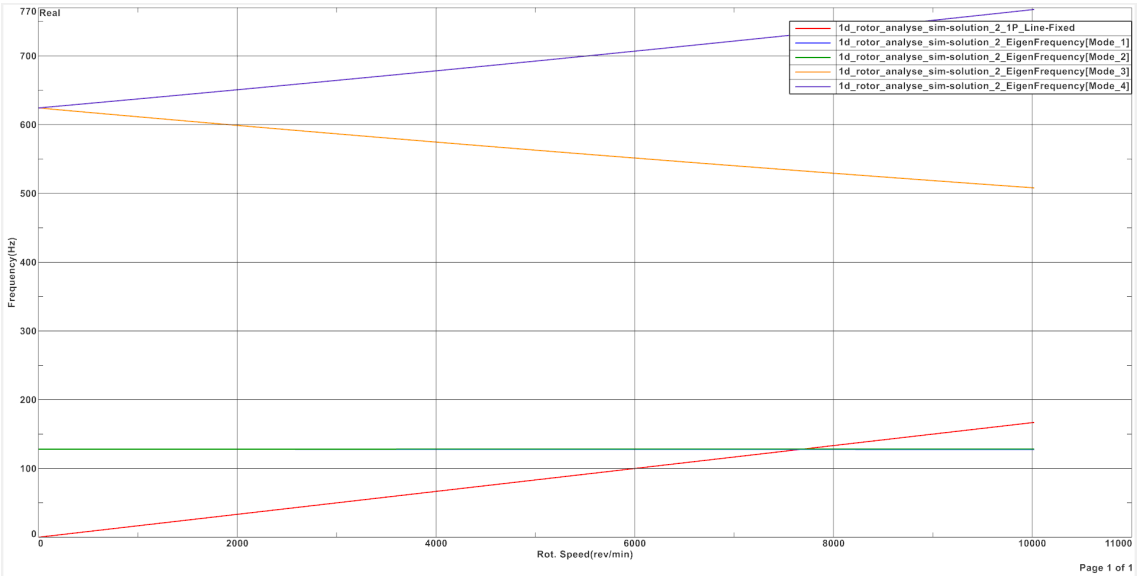


Figure H.7: k = 70e6N/m

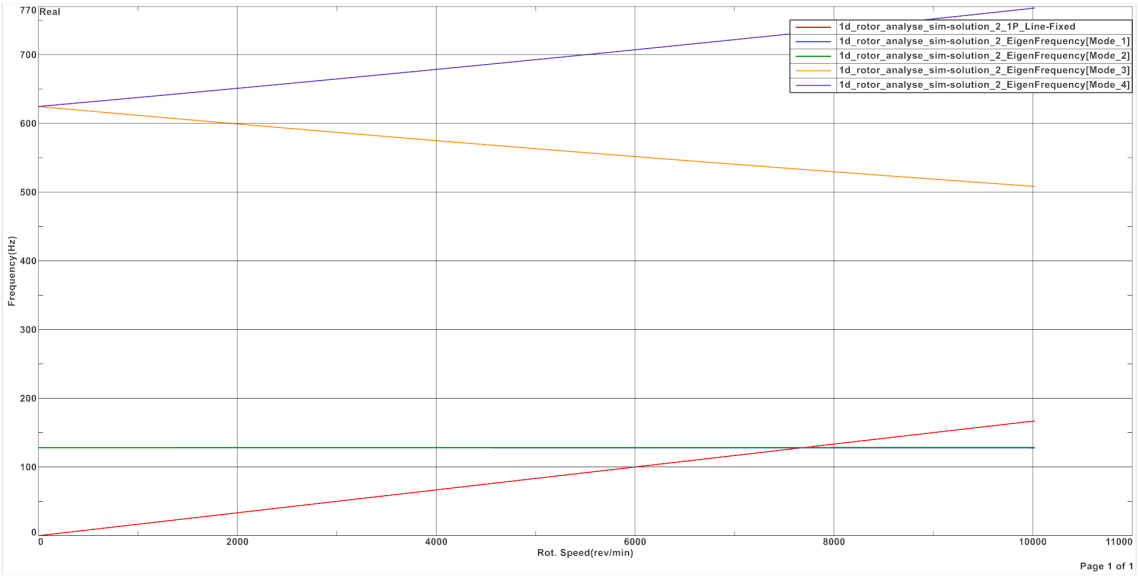


Figure H.8: k = 80e6N/m

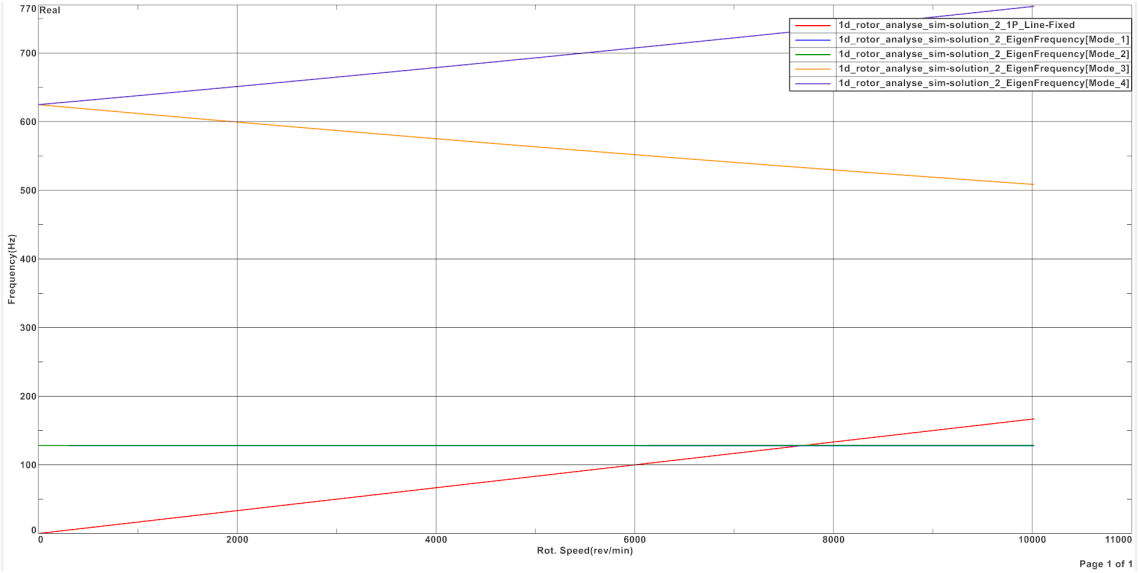


Figure H.9: k = 90e6N/m

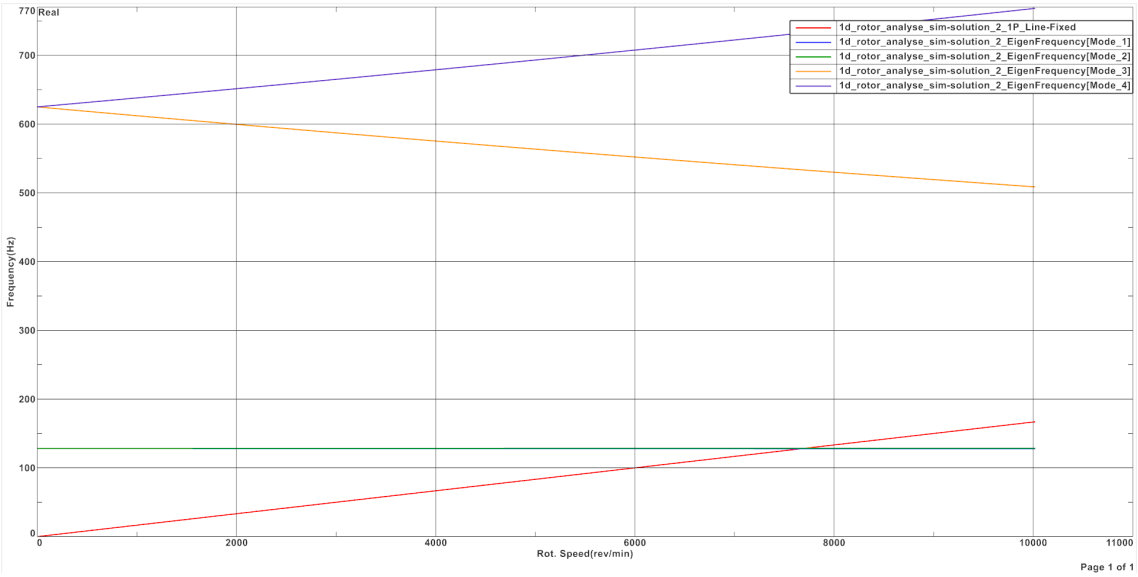


Figure H.10: k = 100e6N/m

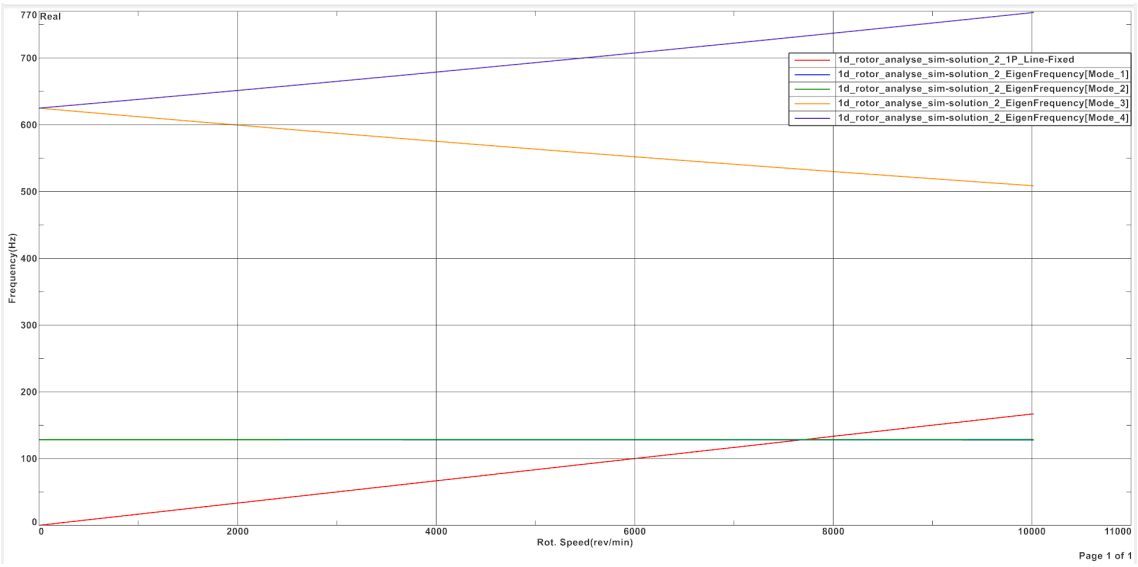


Figure H.11: k = 1e8N/m

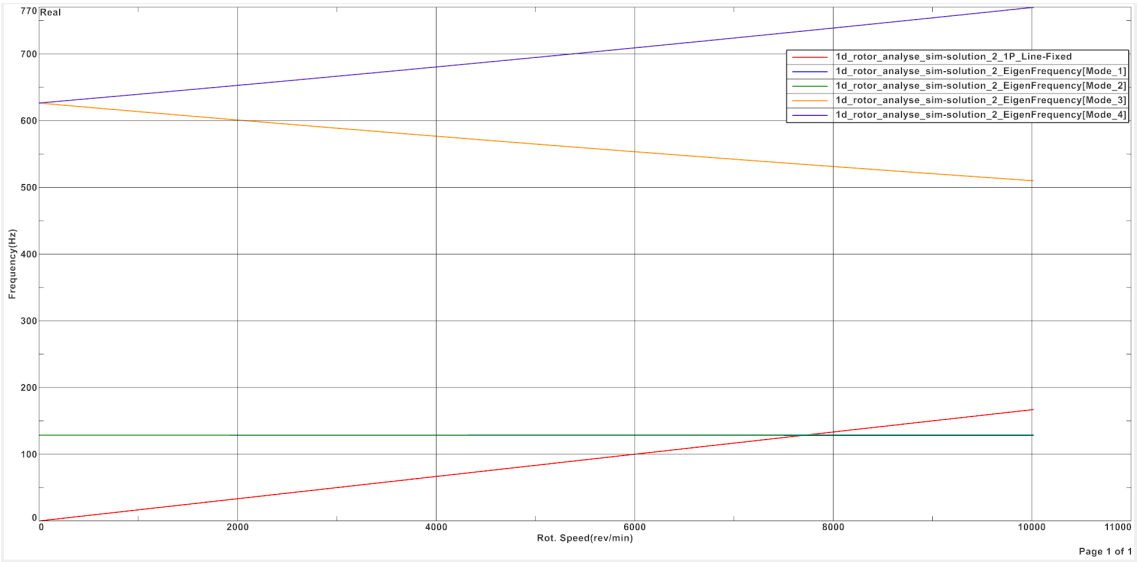


Figure H.12: k = 1e9N/m

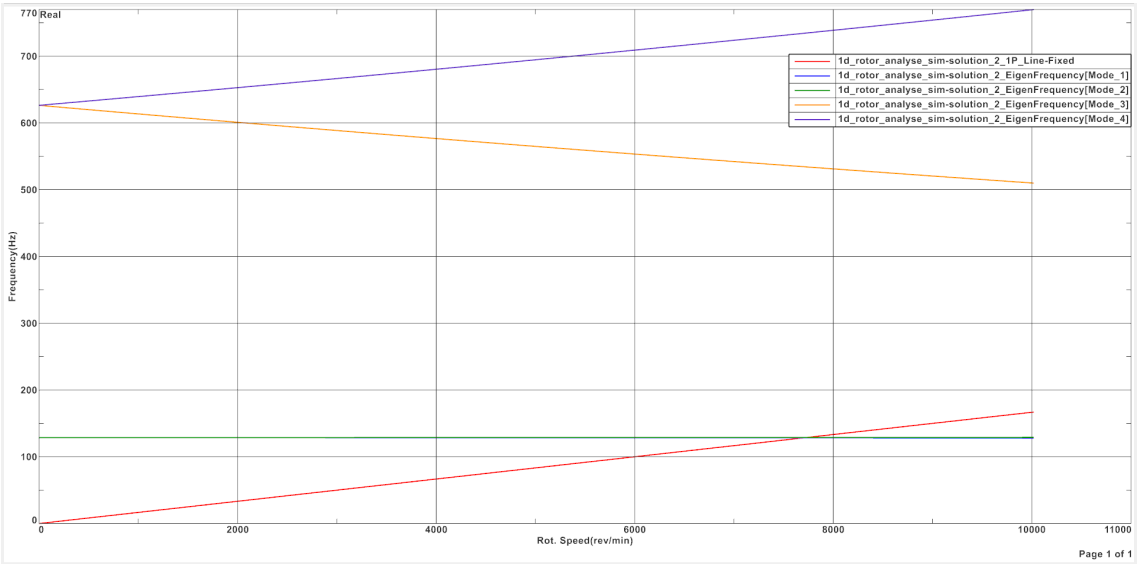


Figure H.13: k = 1e10N/m

Appendix I

Python Code Used in the Analytical Solution

Below is the code used to obtain the analytical solution:

I.1 rotor.py

```
import SystemClass as SC
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.rcParams['lines.linewidth'] = 3.5
mpl.rcParams['legend.labelspacing'] =1
mpl.rcParams["legend.handlelength"] = 6
mpl.rcParams["legend.fontsize"] = 14
mpl.rcParams['xtick.labelsize'] = 16
mpl.rcParams['ytick.labelsize'] = 16
mpl.rcParams['axes.titlesize'] = 18
mpl.rcParams['axes.labelsize'] = 18
```

```
Dis = 0 #Change the position of bearing
```

```
#Rotor Parameters
shaft_param = np.array([9.56]) #Shaft External Diameter, Shaft Internal diameter
material = {'E':206000,'ro' : 7.850e-9,'ve':0.3}
disk = {'D':[75],'d':[9.56],'th':[19],'pos':[197.1-Dis],'eps':[0.14548]}
# {'D' : [Diamter 1, Diameter..], 'th':[thicknes..],
# 'pos': [position from bearing 1],
# 'eps': displacement at disk



bearing1_position = 74.5-Dis
bearing2_position = 325.5-Dis
sensor1_position = 150.5-Dis
sensor2_position = 301.0-Dis
disk_position = disk['pos']
axes_length = 353.6



z_coords = [0.0,]

#Add all parameters to a x coordinates of the rotor:
z_coords.append(bearing1_position)
z_coords.append(bearing2_position)
z_coords.append(sensor1_position)
z_coords.append(sensor2_position)
z_coords.append(axes_length)
z_coords.extend(disk_position)
z_coords.sort()

#Define bering position is needed to define. If BC_Type =0 it will not be used,
↪  but will not run without.
Bearing_position = [bearing1_position, bearing2_position]

#Rotor Coordinates
#z_coords = np.arange(0,353.,1)
y=[1]*len(z_coords)
```

```
print(z_coords)

#Bearing Properties in N and mm
kx1 = 10e3
kx2 = 10e3
ky1 = 10e3
ky2 = 10e3
cx1 = 10.0e0
cx2 = 10.0e0
cy1 = 10.0e0
cy2 = 10.0e0


Kb ={'kx':[kx1,kx2],'ky':[ky1,ky2],'kTheta':[0,0],'kPsi':[0,0]}
Cb ={'cx':[cx1,cx2],'cy':[cy1,cy2],'cTheta':[0,0],'cPsi':[0,0]}

#Rotor instance
param = [material,shaft_param, disk,Kb,Cb]
coords = [z_coords,y]
rotor = SC.RotorSystem(coords, param, Bearing_position ,beam_type=0
↪  ,include_disk= True, damping=True,BC_Type=1, fixed=False)

#Rotational intervals
O = [o for o in np.arange(0,10000,1)]
O_campell = [o for o in np.arange(0,10000,10)]



#Plot
fig1,ax1 =
↪  rotor.plot_CampBell_Diagram(O_campell,n_frequencies=4,n_multiple_omega=1,
↪  wD=False,wN=True,plot_dampratio=False,include_mode=False)
ax1.set_yticks(np.arange(0,700,100))
ax1.set_xlim(0,10000)
ax1.set_ylim(0,200)

fig2,ax2 =rotor.plot_modeshape(0,5640)
```

```
fig5,ax5 = rotor.plot_freqResponse(0,response_type=0, pos = 150.5)



plt.show()
```

I.2 SystemClass.py

```
import numpy as np
import matplotlib.pyplot as plt
from math import *
from numpy import linalg as LA
from scipy.sparse.linalg import eigsh
from mpl_toolkits import mplot3d

plt.rcParams['axes.labelsize'] = 14
plt.rcParams['axes.labelweight'] = 'bold'
plt.rcParams['axes.titleweight'] = 'bold'
plt.rcParams['font.family'] = 'serif'


class RotorSystem:

    def __init__(self, coords, param, Bearing_position,beam_type
    ↪  =1,include_disk=True, damping=False,BC_Type=2, fixed=False,**kwargs):
        r'''This class creates a rotor instance which is used to calculate
        ↪  critical speeds, mode shapes, damping ratios, frequency
        responses, Operational deflections shapes, whirl orbits, natural
↪  frequencies

        :parameter
        coords: 2D List of rotor coordinates on the form [z,y]=
↪  [[100,200,300..],[0,0,0..]
        param: List of parameters:
               Material: Dictionary on the form {'E':, 'ro':, 've':, }
               Shaft Param: List of External and Internal diameter of Shaft, ex:
↪  [50,0]
```

```
            Disk Param: Dictionary of disk paramters on the form:{'D' :
↪ [Diamter 1, Diameter..],

↪ 'th':[thicknes..], 'pos': [position from bearing 1]
                                                        'eps':
↪ displacement at disk }
            Kb: Dictionary containing bearing stiffness propertis on the
↪ form: {'kx':,'ky':,'kTheta':,'kPsi':}
            Cb: Dictionary containing bearing damping propertis on the form:
↪ {'cx':,'cy':,'cTheta':,'cPsi':}
        beam_type: Integere (0 or 1) to choose between Bernoulli(0) or
↪ Timoshenko(1) beam element, default =1
        include_disk: Boolean value to choose to include disk or not, default =
↪ True
        damping: Boolean value to choose to include damping or not, default =
↪ False'''
        # General parameters
        self.kwargs = kwargs
        self.z_coords = coords[0]
        self.y_coords = coords[1]
        self.use_disk = include_disk
        self.use_damping = damping
        self.beam_type = beam_type #Beamtype=0 : Bernoulli Beam, Beamtype = 1,
        ↪ Timoshenko Beam #Default is 1
        self.BC_Type = BC_Type # BC_Type =0 : Bearing at end. BC_Type = 1: exsakt
        ↪ node, BC_Type = 2: nearest node.


        self.fixed_end = fixed

        #Material Parameters:
        self.material_param = param[0]
        self.ro = self.material_param['ro']
        self.E = self.material_param['E']
        self.ve = self.material_param['ve']

        self.bearing1 = Bearing_position[0]
        self.bearing2 = Bearing_position[1]

        # Shaft parameters
```

```
        self.shaft_params = param[1]
        self.nNodes = len(self.z_coords)
        self.Ds = self.shaft_params[0]  # External diameter of shaft
        self.A = np.pi * (self.Ds ** 2) / 4
        self.I = np.pi * (self.Ds ** 4) / 64

        # Disk parameters
        self.disk_dict = param[2]

        # Bearing parameters
        self.Kb = param[3]  # Kx1,Kx2,Ky1,Ky2
        self.Cb = param[4]  # Cx1,Cx2,Cy1,Cy2




    def get_Element_lengths(self):
        """Method to create list containing all element length from coordinates.
        :return Le: list of element lengths"""
        Le = []
        for elem in range(len(self.z_coords) - 1):
            le = np.sqrt((self.z_coords[elem + 1] - self.z_coords[elem]) ** 2 + (
                        self.y_coords[elem + 1] - self.y_coords[elem]) ** 2)
            Le.append(le)
            #print ('Le',Le)
        return Le

    def get_Edofs(self):
        '''Method to create an Nx8 list containig element degrees of freedom. N
        ↪  is the number of elements,
        and the DOFs are zero indexed. Ex:
↪   [[0,1,2,3,4,5,6,7][4,5,6,7,8,9,10,11][...]]
        :returns Edofs: Nx8 list containing element dofs '''
        nElem = self.nNodes - 1
        Edofs = np.zeros((nElem, 8), dtype=int)
        for i in range(nElem):
            Edofs[i, :] = np.array([0, 1, 2, 3, 4, 5, 6, 7], dtype=int) + i * 4
        return Edofs
```

```
    def get_K_sys_Bernoulli(self):
        '''Method which creates system stiffness matrix with bernoulli elements
        ↪  and lumped bearing stiffness,
                :returns K_sys: System stiffness matrix including lumped bearing
↪   stiffness '''

        nDofs = 4 * self.nNodes
        K_sys = np.zeros((nDofs, nDofs))
        # le = self.L / nElem
        L_elements = self.get_Element_lengths()
        Edofs = self.get_Edofs()
        for iEl in range(self.nNodes - 1):
            le = L_elements[iEl]
            Edof = Edofs[iEl]
            # print(np.ix_(Edof,Edof))
            Ke = np.array([[12, 0, 0, 6 * le, -12, 0, 0, 6 * le],
                           [0, 12, -6 * le, 0, 0, -12, -6 * le, 0],
                           [0, -6 * le, 4 * le ** 2, 0, 0, 6 * le, 2 * le ** 2,
                           ↪  0],
                           [6 * le, 0, 0, 4 * le ** 2, -6 * le, 0, 0, 2 * le **
                           ↪  2],
                           [-12, 0, 0, -6 * le, 12, 0, 0, -6 * le],
                           [0, -12, 6 * le, 0, 0, 12, 6 * le, 0],
                           [0, -6 * le, 2 * le ** 2, 0, 0, 6 * le, 4 * le ** 2,
                           ↪  0],
                           [6 * le, 0, 0, 2 * le ** 2, -6 * le, 0, 0, 4 * le **
                           ↪  2]]) * ((self.E * self.I) / (le ** 3))
            K_sys[np.ix_(Edof, Edof)] += Ke

        if self.fixed_end == True:
            K_sys[0]+=self.K_fixed
            K_sys[1]+=self.K_fixed
        #bc1_loc is index of
        BC = self.get_BC()
        be1_loc = BC[0:4]
        be2_loc = BC[4:8]
        Kbe = self.get_bearing_Ke()
        K_sys[np.ix_(be1_loc, be1_loc)] += Kbe[0]
```

```
        K_sys[np.ix_(be2_loc, be2_loc)] += Kbe[1]
        return K_sys

    def get_K_sys_Timoshenko(self):
        '''Method which creates system stiffness matrix with Timoshenko elements
        ↪  and lumped bearing stiffness,
                :returns K_sys: System stiffness matrix including lumped bearing
↪   stiffness '''

        nDofs = 4 * self.nNodes
        K_sys = np.zeros((nDofs, nDofs))
        # le = self.L / nElem
        L_elements = self.get_Element_lengths()
        Edofs = self.get_Edofs()
        for iEl in range(self.nNodes - 1):
            le = L_elements[iEl]
            Edof = Edofs[iEl]

            G = self.E / (2 * (1 + self.ve))
            kke = 6 * (1 + self.ve ** 2) / (7 + 12 * self.ve * 4 * self.ve ** 2)
            PSI = (12 * self.E * self.I) / (kke * G * self.A * le ** 2)
            kei = (self.E * self.I) / ((1 + PSI) * le ** 3)
            Ke = np.array([[12, 0, 0, 6 * le, -12, 0, 0, 6 * le],
                           [0, 12, -6 * le, 0, 0, -12, -6 * le, 0],
                           [0, -6 * le, le ** 2 * (4 + PSI), 0, 0, 6 * le, le **
                           ↪  2 * (2 - PSI), 0],
                           [6 * le, 0, 0, le ** 2 * (4 + PSI), -6 * le, 0, 0, le
                           ↪  ** 2 * (2 - PSI)],
                           [-12, 0, 0, -6 * le, 12, 0, 0, -6 * le],
                           [0, -12, 6 * le, 0, 0, 12, 6 * le, 0],
                           [0, -6 * le, le ** 2 * (2 - PSI), 0, 0, 6 * le, le **
                           ↪  2 * (4 + PSI), 0],
                           [6 * le, 0, 0, le ** 2 * (2 - PSI), -6 * le, 0, 0, le
                           ↪  ** 2 * (4 + PSI)]]) * kei
            K_sys[np.ix_(Edof, Edof)] += Ke

        if self.fixed_end == True:
            K_sys[0]+=self.K_fixed
            K_sys[1]+=self.K_fixed
```

```
        BC = self.get_BC()
        be1_loc = BC[0:4]
        be2_loc = BC[4:8]
        Kbe = self.get_bearing_Ke()
        K_sys[np.ix_(be1_loc, be1_loc)] += Kbe[0]
        K_sys[np.ix_(be2_loc, be2_loc)] += Kbe[1]

        return K_sys

    def get_M_sys_Bernoulli(self):
        '''Method which creates system mass matrix with bernoulli elements
        ↪  including lumped disk masses and inertias,
           :returns M_sys: System mass matrix including lumped disk
↪  properties'''

        nElem = self.nNodes - 1
        nDofs = 4 * self.nNodes
        M_sys = np.zeros((nDofs, nDofs))
        # le = self.L / nElem
        L_elements = self.get_Element_lengths()
        Edofs = self.get_Edofs()
        for iEl in range(nElem):
            le = L_elements[iEl]
            Edof = Edofs[iEl]
            Me = np.array([[156, 0, 0, 22 * le, 54, 0, 0, -13 * le],
                           [0, 156, -22 * le, 0, 0, 54, 13 * le, 0],
                           [0, -22 * le, 4 * le ** 2, 0, 0, -13 * le, -3 * le **
                           ↪  2, 0],
                           [22 * le, 0, 0, 4 * le ** 2, 13 * le, 0, 0, -3 * le **
                           ↪  2],
                           [54, 0, 0, 13 * le, 156, 0, 0, -22 * le],
                           [0, 54, -13 * le, 0, 0, 156, 22 * le, 0],
                           [0, 13 * le, -3 * le ** 2, 0, 0, 22 * le, 4 * le ** 2,
                           ↪  0],
                           [-13 * le, 0, 0, -3 * le ** 2, -22 * le, 0, 0, 4 * le
                           ↪  ** 2]]) * (
                          (self.ro * self.A * le) / 420)
            M_sys[np.ix_(Edof, Edof)] += Me
```

```
        if self.use_disk == True:
            DiskDOFS = self.get_DiskNodeDOFS()
            Md = self.get_disk_Md()
            for i in range(len(DiskDOFS)):
                M_sys[np.ix_(DiskDOFS[i], DiskDOFS[i])] += Md[i]
            return M_sys
        else:
            return M_sys


    def get_M_sys_Timoshenko(self):
        '''Method which creates system mass matrix with Timoshenko elements
        ↪  including lumped disk masses and inertias,
           :returns M_sys: System mass matrix including lumped disk
↪  properties'''
        nElem = self.nNodes - 1
        nDofs = 4 * self.nNodes
        M_sys = np.zeros((nDofs, nDofs))
        # le = self.L / nElem
        L_elements = self.get_Element_lengths()
        Edofs = self.get_Edofs()
        for iEl in range(nElem):
            le = L_elements[iEl]
            Edof = Edofs[iEl]
            G = self.E / (2 * (1 + self.ve))
            kke = 6 * (1 + self.ve ** 2) / (7 + 12 * self.ve + 4 * self.ve ** 2)
            PSI = (12 * self.E * self.I) / (kke * G * self.A * le ** 2)
            m1 = (312 + 588 * PSI + 280 * PSI ** 2)
            m2 = (44 + 77 * PSI + 35 * PSI ** 2) * le
            m3 = (108 + 252 * PSI + 140 * PSI ** 2)
            m4 = -(26 + 63 * PSI + 35 * PSI ** 2) * le
            m5 = (8 + 14 * PSI + 7 * PSI ** 2) * le ** 2
            m6 = -(6 + 14 * PSI + 7 * PSI ** 2) * le ** 2
            m7 = (36)
            m8 = (3 - 15 * PSI) * le
            m9 = (4 + 5 * PSI + 10 * PSI ** 2) * le ** 2
            m10 = (-1 - 5 * PSI + 5 * PSI ** 2) * le ** 2

            met = self.ro * self.A * le / (840 * (1 + PSI) ** 2)
```

```
            MEt = np.array([[m1, 0, 0, m2, m3, 0, 0, m4],
                            [0, m1, -m2, 0, 0, m3, -m4, 0],
                            [0, -m2, m5, 0, 0, m4, m6, 0],
                            [m2, 0, 0, m5, -m4, 0, 0, m6],
                            [m3, 0, 0, -m4, m1, 0, 0, -m2],
                            [0, m3, m4, 0, 0, m1, m2, 0],
                            [0, -m4, m6, 0, 0, m2, m5, 0],
                            [m4, 0, 0, m6, -m2, 0, 0, m5]]) * met
            mei = self.ro * self.I / (30 * ((1 + PSI) ** 2) * le)
            MEi = np.array([[m7, 0, 0, m8, -m7, 0, 0, m8],
                            [0, m7, -m8, 0, 0, -m7, -m8, 0],
                            [0, -m8, m9, 0, 0, m8, m10, 0],
                            [m8, 0, 0, m9, -m8, 0, 0, m10],
                            [-m7, 0, 0, -m8, m7, 0, 0, -m8],
                            [0, -m7, m8, 0, 0, m7, m8, 0],
                            [0, -m8, m10, 0, 0, m8, m9, 0],
                            [m8, 0, 0, m10, -m8, 0, 0, m9]]) * mei


            Me = MEi + MEt
            M_sys[np.ix_(Edof, Edof)] += Me
        if self.use_disk == True:
            DiskDOFS = self.get_DiskNodeDOFS()
            Md = self.get_disk_Md()
            for i in range(len(DiskDOFS)):
                M_sys[np.ix_(DiskDOFS[i], DiskDOFS[i])] += Md[i]
            return M_sys
        else:
            return M_sys



    def get_G_sys_Bernoulli(self):
        '''Method which creates system gyroscopic matrix with bernoulli elements
        ↪  including lumped disk properties,
           :returns G_sys: System mass matrix including lumped disk
↪  properties'''

        nElem = self.nNodes - 1
        nDofs = 4 * self.nNodes
```

```
        G_sys = np.zeros((nDofs, nDofs))
        # le = self.L / nElem
        L_elements = self.get_Element_lengths()
        Edofs = self.get_Edofs()
        for iEl in range(nElem):
            le = L_elements[iEl]
            Edof = Edofs[iEl]
            Ge = np.array([[0,          36,     -3 * le,    0,              0,
            ↪  -36,    -3 * le,        0],
                           [-36,        0,      0,          -3 * le,        36,
                           ↪  0,      0,          -3 * le],
                           [3 * le,     0,      0,          4 * le ** 2,    -3 *
                           ↪  le,    0,      0,          -le ** 2],
                           [0, 3 * le, -4 * le ** 2, 0, 0, -3 * le, le ** 2, 0],
                           [0, -36, 3 * le, 0, 0, 36, 3 * le, 0],
                           [36, 0, 0, 3 * le, -36, 0, 0, 3 * le],
                           [3 * le, 0, 0, -le ** 2, -3 * le, 0, 0, 4 * le ** 2],
                           [0, 3 * le, le ** 2, 0, 0, -3 * le, -4 * le ** 2, 0]])
                           ↪  * (
                          (self.ro * self.I) / (15 * le))
            G_sys[np.ix_(Edof, Edof)] += Ge#*0
        # G_sys = np.zeros((nDofs, nDofs))
        if self.use_disk == True:
            DiskDOFS = self.get_DiskNodeDOFS()
            Gd = self.get_disk_Gd()
            for i in range(len(DiskDOFS)):
                G_sys[np.ix_(DiskDOFS[i], DiskDOFS[i])] += Gd[i]
            return G_sys
        else:
            return G_sys


    def get_G_sys_Timoshenko(self):
        '''Method which creates system gyroscopic matrix with Timoshenko elements
        ↪  including lumped disk properties,
             :returns G_sys: System mass matrix including lumped disk
↪  properties'''

        nElem = self.nNodes - 1
```

```
        nDofs = 4 * self.nNodes
        G_sys = np.zeros((nDofs, nDofs))
        # le = self.L / nElem
        L_elements = self.get_Element_lengths()
        Edofs = self.get_Edofs()
        for iEl in range(nElem):
            le = L_elements[iEl]
            Edof = Edofs[iEl]
            G = self.E / (2 * (1 + self.ve))
            kke = 6 * (1 + self.ve ** 2) / (7 + 12 * self.ve + 4 * self.ve ** 2)
            PSI = (12 * self.E * self.I) / (kke * G * self.A * le ** 2)

            gei = (self.ro * self.I) / ((15 * ((1 + PSI) ** 2) * le))
            g1 = 36
            g2 = (3 - 15 * PSI) * le
            g3 = (4 + 5 * PSI + 10 * PSI ** 2) * le ** 2
            g4 = (-1 - 5 * PSI + 5 * PSI ** 2) * le ** 2

            Ge = np.array([[0, g1, -g2, 0, 0, -g1, -g2, 0],
                           [-g1, 0, 0, -g2, g1, 0, 0, -g2],
                           [g2, 0, 0, g3, -g2, 0, 0, g4],
                           [0, g2, -g3, 0, 0, -g2, -g4, 0],
                           [0, -g1, g2, 0, 0, g1, g2, 0],
                           [g1, 0, 0, g2, -g1, 0, 0, g2],
                           [g2, 0, 0, g4, -g2, 0, 0, g3],
                           [0, g2, -g4, 0, 0, -g2, -g3, 0]]) * gei
            G_sys[np.ix_(Edof, Edof)] += Ge
        if self.use_disk == True:
            DiskDOFS = self.get_DiskNodeDOFS()
            Gd = self.get_disk_Gd()
            for i in range(len(DiskDOFS)):
                G_sys[np.ix_(DiskDOFS[i], DiskDOFS[i])] += Gd[i]
            return G_sys
        else:
            return G_sys


    def get_C_sys(self):
```

```
        '''Method which creates system damping matrix with lumped damping
        ↪  properties of bearings,
            :returns C_sys: System damping matrix '''
        nDofs = 4 * self.nNodes
        C_sys = np.zeros((nDofs, nDofs))
        if self.use_damping == True:
            BC = self.get_BC()
            be1_loc = BC[0:4]
            be2_loc = BC[4:8]
            Cbe = self.get_bearing_Ce()
            C_sys[np.ix_(be1_loc, be1_loc)] += Cbe[0]
            C_sys[np.ix_(be2_loc, be2_loc)] += Cbe[1]


            return C_sys

        else:

            return C_sys

    def get_DiskNodeDOFS(self):
        '''Method to get the degrees of freedom of the nodes which have disks
        ↪  attached to them.
          :returns DiskDofs: List of DOF's of disks'''

        locations = self.disk_dict['pos']
        Edofs = self.get_Edofs()
        DiskDofs = []
        for i in range(len(locations)):
            # node_idx = list(self.x_coords).index(self.disk_loc)
            node_idx = list(self.z_coords).index(locations[i])
            if node_idx != Edofs.shape[0]:
                DiskDofs.append(Edofs[node_idx][0:4])
            else:
                DiskDofs.append(Edofs[node_idx-1][4:8])  # If disk at last node,
                ↪  need to go one node back and
                                                         # use the last DOFs
                                                         ↪  of the element
                                                         ↪  instead
```

```
        return DiskDofs

    def get_BC(self):
        '''Method to get the degrees of freedom of the nodes which have bearings
        ↪  attached to them.
                :returns BC: List of DOF's of Bearings'''
        if self.BC_Type == 0:

            Edofs = self.get_Edofs()

            BC1 = Edofs[0][0:int(len(Edofs[0]) / 2)]  # First node
            ↪  #print("BC1", BC1)
            BC2 = Edofs[-1][int(len(Edofs[0]) / 2):8]  # Last Node
            BC = np.hstack([BC1, BC2])

        elif self.BC_Type == 1:
            Edofs = self.get_Edofs()
            bearing_node_idx_1 = list(self.z_coords).index(self.bearing1)
            bearing_node_idx_2 = list(self.z_coords).index(self.bearing2)

            BC1 = Edofs[bearing_node_idx_1][0:int(len(Edofs[0]) / 2)]  # First
            ↪  bearing node
            BC2 = Edofs[bearing_node_idx_2][0:int(len(Edofs[0]) / 2)]  # Second
            ↪  bearing node
            BC = np.hstack([BC1, BC2])

        elif self.BC_Type == 2:
            Edofs = self.get_Edofs()

            node_idx_1 = min(range(len(self.z_coords)), key=lambda x:
            ↪  abs(self.z_coords[x] - self.bearing1))
            node_idx_2 = min(range(len(self.z_coords)), key=lambda x:
            ↪  abs(self.z_coords[x] - self.bearing2))

            BC1 = Edofs[node_idx_1][0:4]  # First bearing node
            BC2 = Edofs[node_idx_2][0:4]  # Second bearing node
            BC = np.hstack([BC1, BC2])

        return BC
```

```
    def get_diskMass(self):
        '''Method which calculates the mass of the disks,
        :returns md: List of disk masses'''
        md = []
        Dd = self.disk_dict['D']
        ddi = self.disk_dict['d']
        th = self.disk_dict['th']
        for i in range(len(Dd)):
            # md = self.ro * np.pi * self.Dd ** 2 * self.t / 4
            md.append((self.ro * np.pi *( Dd[i] ** 2 -ddi[i]**2)* th[i] / 4))

        return md

    def get_diskIp2(self):
        '''Method which calculates the polar inertia of the disks,
            :returns Ip: List of disk polar inertias'''
        m = self.get_diskMass()
        Dd = self.disk_dict['D']
        ddi = self.disk_dict['d']
        Ip = []
        for i in range(len(m)):
            Ip.append(m[i] * (Dd[i] ** 2) / 8)
            # Ip = m * self.Dd ** 2 / 8
        return Ip

    def get_diskIp(self):
        '''Method which calculates the polar inertia of the disks,
            :returns Ip: List of disk polar inertias'''
        m = self.get_diskMass()
        Dd = self.disk_dict['D']
        ddi = self.disk_dict['d']
        Ip = []
        for i in range(len(m)):
            Ip.append(m[i] * (Dd[i] ** 2+ddi[i]**2) / 8)
            # Ip = m * self.Dd ** 2 / 8
        return Ip
```

```
    def get_diskId2(self):
        '''Method which calculates the diametral inertia of the disks,
            :returns Ip: List of disk diametral inertias'''
        m = self.get_diskMass()
        Ip = self.get_diskIp()
        th = self.disk_dict['th']
        ddi = self.disk_dict['d']
        Id = []
        for i in range(len(m)):
            Id.append(Ip[i] / 2 + m[i] * th[i] ** 2 / 12)

            # Id = Ip / 2 + m * self.th ** 2 / 12
        return Id


    def get_diskId(self):
        '''Method which calculates the diametral inertia of the disks,
            :returns Ip: List of disk diametral inertias'''
        m = self.get_diskMass()
        #Ip = self.get_diskIp()
        #th = self.disk_dict['th']
        Dd = self.disk_dict['D']
        ddi = self.disk_dict['d']
        Id = []
        for i in range(len(m)):
            Id.append(m[i]*(Dd[i]**2-ddi[i]**2) / 16) #MR^2/4

            # Id = Ip / 2 + m * self.th ** 2 / 12
        return Id

    def get_disk_Md(self):
        '''Method which creates the disk mass matrices,
            :returns Md: nx4x4 matrix of the disk mass matrix '''

        m = self.get_diskMass()
        Id = self.get_diskId()
        Md = []
        for i in range(len(m)):
            Md.append(np.array([[m[i], 0, 0, 0],
```

```
                                [0, m[i], 0, 0],
                                [0, 0, Id[i], 0],
                                [0, 0, 0, Id[i]]]))

        return Md

    def get_disk_Gd(self):
        '''Method which creates the disk gyroscopic matrices,
            :returns Gd: nx4x4 matrix of the disk gyroscopic matrix '''

        Ip = self.get_diskIp()
        Gd = []
        for i in range(len(Ip)):
            Gd.append(np.array([[0, 0, 0, 0],
                                [0, 0, 0, 0],
                                [0, 0, 0, Ip[i]],
                                [0, 0, -Ip[i], 0]]))

        return Gd

    def get_bearing_Ke(self):
        '''Method which creates the lumped bearing stiffness matrices,
           :returns KB: nx4x4 matrix of the bearing stiffness matrix '''

        Kb_param = self.Kb
        KB = []
        for i in range(len(Kb_param['kx'])):
            KB.append(np.array([[Kb_param['kx'][i], 0, 0, 0],
                                [0, Kb_param['ky'][i], 0, 0],
                                [0, 0, Kb_param['kTheta'][i], 0],
                                [0, 0, 0, Kb_param['kPsi'][i]]]))


        return KB

    def get_bearing_Ce(self):
        '''Method which creates the lumped bearing damping matrices,
           :returns KB: nx4x4 matrix of the bearing damping matrix '''
```

```
        Cb_param = self.Cb
        CB = []
        for i in range(len(Cb_param['cx'])):
            CB.append(np.array([[Cb_param['cx'][i], 0, 0, 0],
                                [0, Cb_param['cy'][i], 0, 0],
                                [0, 0, Cb_param['cTheta'][i], 0],
                                [0, 0, 0, Cb_param['cPsi'][i]]]))
        return CB

    def get_A(self, omega):
        r'''Method which creates matrix A of the subspace formulation A\dot{x} +
        ↪  Bx = 0,
                :parameter: Omega: Rotational Speed in RPM
               :returns A: 2n*8 x 2n*8 matrix on the form A = [[G*Omega + C, M],
                                                               [M,           0]]'''
        omega_rad = omega*(2*np.pi/60)
        if self.beam_type == 0:
            G_sys = self.get_G_sys_Bernoulli()
            M_sys = self.get_M_sys_Bernoulli()
            C_sys = self.get_C_sys()
            MatSize = len(M_sys)
        elif self.beam_type == 1:
            G_sys = self.get_G_sys_Timoshenko()
            M_sys = self.get_M_sys_Timoshenko()
            C_sys = self.get_C_sys()
            MatSize = len(M_sys)
        A = np.zeros((2 * MatSize, 2 * MatSize))  # A = [[G*omega, M],[M,0]]
        A[0:MatSize, 0:MatSize] = G_sys * omega_rad + C_sys
        A[MatSize:2 * MatSize, 0:MatSize] = M_sys
        A[0:MatSize, MatSize:2 * MatSize] = M_sys
        return A

    def get_B(self):
        r'''Method which creates matrix B of the subspace formulation A\dot{x} +
        ↪  Bx = 0,
           :returns B: 2n*8 x 2n*8 matrix on the form A = [[K, 0],
                                                          [0, -M]]'''
```

```
        if self.beam_type == 0:
            M_sys = self.get_M_sys_Bernoulli()
            K_sys = self.get_K_sys_Bernoulli()
            MatSize = len(M_sys)
        elif self.beam_type == 1:
            M_sys = self.get_M_sys_Timoshenko()
            K_sys = self.get_K_sys_Timoshenko()
            MatSize = len(M_sys)



        B = np.zeros((2 * MatSize, 2 * MatSize))  # B = [[K, 0],[0,-M]]
        B[0:MatSize, 0:MatSize] = K_sys
        B[MatSize:2 * MatSize, MatSize:2 * MatSize] = -M_sys
        return B

    def sort_eig(self,w):
        '''# Method used to generate an index that will sort eigenvalues and
        ↪  eigenvectors
            based on the imaginary (w) part of the eigenvalues.
            Positive eigenvalues will be positioned at the first half of the
↪   array.
            Parameters
            ----------
            w: Array with the eigenvalues.

            :returns idx: Sorted indices'''

        evals_truncated = np.around(w, decimals=10)
        a = np.imag(evals_truncated)  # First column
        b = np.absolute(evals_truncated)  # Second column



        #ind = np.lexsort((b, a))  # Sort by imag (wd), then by absolute (wn)
        # Positive eigenvalues first
        #positive = [i for i in ind[len(a) // 2:]]
        #negative = [i for i in ind[: len(a) // 2]]
        #idx = np.array([positive, negative]).flatten()
```

```
        ind = np.lexsort((a, b)) #Sort by Absolut and then imag

        positive = [i for i in ind[1::2]] #positive first
        negative = [i for i in ind[: :2]]

        idx = np.array([positive, negative]).flatten()
        return idx

    def solve_Eigproblem(self, omega=0,output='rads'):
        r'''Method which solves the eigenproblem A\dot{x} = -Bx
        :parameter: Omega: Rotational Speed in RPM
                    output: Output unit, standard is Rad/s
           :returns w: List of sorted eigenvalues as per method sort_eig,
                    v: Sorted eigenvectors where the vectors are the columns'''

        A = self.get_A(omega)
        B = self.get_B()

        w, v = LA.eig(-LA.inv(A) @ B) # First Order: Ax' + Bx = 0
        #w, v = LA.eig(LA.solve(A, -B))  # First Order: Ax' + Bx = 0
        idx = self.sort_eig(w)
        w = w[idx]
        v = v[:,idx]
        if output=='hz':
            w = w/2/np.pi

        return w, v


    def plot_modeshape(self,modes,omega = 0, maxlines = True,only_diskpos =True):
        '''Method to plot the 3D modeshapes at resonance for a given RPM. Plots
        ↪  the REAL mode shape only.
         :parameter
            mode: Modeshape(s) to be plotted, either int or list
            omega: Rpm at which the modeshape is calculated, default is  zero.

        :returns
        fig,ax: 3D plot of the modeshape
             '''
```

```
        # omega_rad= omega*2*np.pi/60
        evals, evecs = self.solve_Eigproblem(omega)
        colors = ['red', 'blue', 'm', 'green']
        if type(modes)!=list:

            whirl_mode = self.get_whirl_modes(4, omega)
            mode = modes
            u = np.zeros((len(self.z_coords), 100))
            v = np.zeros((len(self.z_coords), 100))
            w_0 = evals[mode]
            T = 2 * np.pi / (w_0)
            t = np.linspace(0, T * (0.96), 100)
            for i in range(len(self.z_coords)):
                u[i, :] = (evecs[4 * i, mode] * np.e ** (1j * w_0 * t)).real
                v[i, :] = (evecs[4 * i + 1, mode] * np.e ** (1j * w_0 * t)).real


            fig = plt.figure()
            ax = plt.axes(projection='3d')
            ampU = np.max((u[:, :])) * 1.3
            ampV = np.max((v[:, :])) * 1.3
            amp = np.max([ampU, ampV])
            z0 = np.ones(len(self.z_coords))
            x0 = np.zeros(len(self.z_coords))
            y0 = np.zeros(len(self.z_coords))
            ax.set_ylim(-amp, amp)
            ax.set_zlim(-amp, amp)
            ax.set_xlim(0, len(self.z_coords))
            for i in range(len(self.z_coords)):
                x0[i] = u[i, 0]
                y0[i] = v[i, 0]
                z0[i] *= i
                x = u[i, :]
                y = v[i, :]
                z = i * np.ones(100)
                if only_diskpos == True:
                    if self.z_coords[i] in self.disk_dict['pos']:
                        ax.plot(z, x, y, linewidth=1.5, color='black')
                else:
```

```
                ax.plot(z, x, y, linewidth=1.5, color='black')
            if maxlines == True:
                ax.plot([i, i], [x0[i], min(abs(x))], [y0[i], min(abs(y))],
                ↪  '-', color='Black', linewidth=1)
        ax.plot([0, z0[-1] + 1], [0, 0], [0, 0], linestyle=(0, (3, 5, 1, 5)),
        ↪  color='Black')

        # ax.plot([0, z0[-1] + 1], [0, 0], [0, 0], linestyle=(0, (3, 5, 1,
        ↪  5)), color='Black')

        ax.plot(z0, x0, y0, '.-', linewidth=3.5, color=colors[modes],
        ↪  markersize=15)
        ax.tick_params(axis='both', which='major', labelsize=14)
        ax.set_title('Modeshape #{}, Whirl Dir: {},  @ {}
        ↪  RPM'.format(modes+1, whirl_mode[modes], omega))
        ax.set_xlabel('Nodes', labelpad=10)
        # ax.set_zticklabels([])
        # ax.set_yticklabels([])
        ax.set_ylim(-amp, amp)
        ax.set_zlim(-amp, amp)
        ax.set_xlim(0, len(self.z_coords))

        ax.set_zticks(np.round(np.linspace(-amp, amp, 4), 3))
        ax.set_yticks(np.round(np.linspace(-amp, amp, 4), 3))
        ax.set_xticks(np.arange(0,len(self.z_coords)))
        ax.set_xticklabels(np.arange(1,len(self.z_coords)+1))
        #
        ax.set_zlabel('Displacement u', labelpad=15)
        ax.set_ylabel('Displacement v', labelpad=15)
        ax.w_xaxis.set_pane_color((1.0, 1.0, 1.0, 1.0))
        ax.w_yaxis.set_pane_color((1.0, 1.0, 1.0, 1.0))
        ax.w_zaxis.set_pane_color((1.0, 1.0, 1.0, 1.0))
        return fig, ax

    else:
        fig = plt.figure()
        whirl_mode = self.get_whirl_modes(len(modes), omega)
        k =0
        for mode in modes:
```

```
        #print(mode)
        u = np.zeros((len(self.z_coords), 100))
        v = np.zeros((len(self.z_coords), 100))
        w_0 = evals[mode]
        T = 2 * np.pi / (w_0)
        t = np.linspace(0, T * (0.96), 100)
        for i in range(len(self.z_coords)):
            u[i, :] = (evecs[4 * i, mode] * np.e ** (1j * w_0 * t)).real
            v[i, :] = (evecs[4 * i + 1, mode] * np.e ** (1j * w_0 *
            ↪  t)).real
        if len(modes)>2:

            ax = fig.add_subplot(221 + k, projection='3d')
        else:
            ax = fig.add_subplot(121 + k, projection='3d')

        ampU = np.max((u[:, :])) * 1.3
        ampV = np.max((v[:, :])) * 1.3
        amp = np.max([ampU, ampV])
        z0 = np.ones(len(self.z_coords))
        x0 = np.zeros(len(self.z_coords))
        y0 = np.zeros(len(self.z_coords))

        for i in range(len(self.z_coords)):
            x0[i] = u[i, 0]
            y0[i] = v[i, 0]
            z0[i] *= i
            x = u[i, :]
            y = v[i, :]
            z = i * np.ones(100)
            if only_diskpos == True:
                if self.z_coords[i] in self.disk_dict['pos']:
                    ax.plot(z, x, y, linewidth=1.5, color='black')
                    ax.scatter(i, x[-1], y[-1], marker='D',
                    ↪  color='black', s=15)
                if maxlines == True:
                    ax.plot([i, i], [x0[i], min(abs(x))], [y0[i],
                    ↪  min(abs(y))], '-', color='Black', linewidth=1)
            else:
```

```
                    ax.plot(z, x, y, linewidth=1.5, color='black')

                    ax.scatter(i, x[-1], y[-1],marker='D',color = 'black', s=
                    ↪  15)
                    if maxlines == True:
                        ax.plot([i, i], [x0[i], min(abs(x))], [y0[i],
                        ↪  min(abs(y))], '-', color='Black', linewidth=1)

                ax.plot([0, z0[-1] + 1], [0, 0], [0, 0], linestyle=(0, (3, 5, 1,
                ↪  5)), color='Black')


                ax.plot(z0, x0, y0, '.-', linewidth=3.5, color=colors[k],
                ↪  markersize=10)
                ax.tick_params(axis='both', which='major', labelsize=14)
                ax.set_title('Modeshape #{}, Whirl Dir: {},  @ {}
                ↪  RPM'.format(mode+1, whirl_mode[k], omega))
                ax.set_xlabel('Nodes', labelpad=10)

                ax.set_ylim(-amp, amp)
                ax.set_zlim(-amp, amp)
                ax.set_xlim(0, len(self.z_coords))

                ax.set_zticks(np.round(np.linspace(-amp, amp, 4), 2))
                ax.set_yticks(np.round(np.linspace(-amp, amp, 4), 2))
                ax.set_xticks(np.arange(0, len(self.z_coords)))
                ax.set_xticklabels(np.arange(1, len(self.z_coords) + 1))

                ax.set_zlabel('Displacement u', labelpad=15)
                ax.set_ylabel('Displacement v', labelpad=15)
                ax.w_xaxis.set_pane_color((1.0, 1.0, 1.0, 1.0))
                ax.w_yaxis.set_pane_color((1.0, 1.0, 1.0, 1.0))
                ax.w_zaxis.set_pane_color((1.0, 1.0, 1.0, 1.0))
                k+=1

        return fig,ax

def get_Hmat(self,freq_index=0,node=0,omega=0,T_dict = False):
```

```
        '''Method which creates the H matrix used to find the major and minor
        ↪  orbits of the whirl ellipse.
        :parameter
            freq_index: Index of which natural frequency to use, default = 0
            node: Node to use, zero indexed, i.e. first node = 0, default = 0
            omega: Rotational speed at which to calculate H
            T_dict: If True, returns Dictionary with parameters to create H
↪   matrix and H, if False, returns only H. '''
        evals,evec =self.solve_Eigproblem(omega)
        vector = evec[4*node:4*node+2,freq_index]
        u0,v0 = vector[0],vector[1]
        ru, rv = np.absolute(u0), np.absolute(v0)
        nu, nv = np.angle(u0), np.angle(v0)
        T = np.array([[ru * np.cos(nu), -ru * np.sin(nu)],
                      [rv * np.cos(nv), -rv * np.sin(nv)]])
        Tdic = {'ru': ru, 'rv': rv, 'nu': nu, 'nv': nv}

        H = T @ T.T

        if T_dict:
            return H, Tdic
        else:
            return H



    def get_Kappa(self,freq_index=0,node=0,omega=0,kappa_dict = False):
        '''Method which is used to calculate the kappa value of the whirl orbit.
        ↪  The kappa value is used to
        determine whirl direction.
        :parameter
        freq_index: Index of which natural frequency to use, default = 0
        node: Node to use, zero indexed, i.e. first node = 0, default = 0
        omega: Rotational speed at which to calculate kappa
        kappa_dict: If True, returns dictionary with major and minor axis of
↪   ellipse and kappa value,
                    if False, returns only Kappa value. '''

        H, Tdic = self.get_Hmat(freq_index,node,omega, T_dict=True)
```

```
        nu, nv = Tdic['nu'], Tdic['nv']
        lam = LA.eig(H)[0]
        major = np.sqrt(max(lam))
        if min (lam)<0:
            minor = np.sqrt(-min(lam))
        else:
            minor = np.sqrt(min(lam))
        minor = np.sqrt(min(lam))
        kappa = minor / major

        diff = nv - nu
        if diff < -np.pi:
            diff += 2 * np.pi
        elif diff > np.pi:
            diff -= 2 * np.pi

        if diff == 0 or diff == np.pi:
            kappa = 0
        elif 0 < diff < np.pi:
            kappa *= -1

        if kappa_dict:
            Kdict = {'major': major.real, 'minor': minor.real, 'kappa':
            ↪  kappa.real}

            return Kdict
        else:

            return kappa


    def get_Kappa_list(self,freq_index=0,omega=None,detailed = False):
        '''Method which creates a list of kappa values of each node.
        :parameter
            freq_index:Index of which natural frequency to use,default =0
            omega: Either list of rotational speeds at which to calculate the
↪   kappa value,
                    or integer value at which to calculate kappa values,
```

```
            detiled: If True, returns 2D list of kappa values at each node for
↪  every speed in the omega list,
                    If False: Returns list of kappa values at the mean rotational
↪  speed or the value given. '''


        nodes = self.z_coords

        if type(omega) == list and detailed == True:
            kappa_vals = np.empty((len(omega),len(nodes)))
            for o in omega:
                for i in range(len(nodes)):
                    kappa = self.get_Kappa(freq_index,i,omega=o)
                    kappa_vals[omega.index(o)][i] = kappa

            return kappa_vals

        elif type(omega) == list and detailed==False:
            len_omega =len(omega) //2
            omega_val = omega.index(omega[len_omega])
            kappa_vals = []
            for i in range(len(nodes)):
                kappa_vals.append(self.get_Kappa(freq_index,i,omega_val))
            return np.array(kappa_vals)
        else:
            kappa_vals = []
            for i in range(len(nodes)):
                kappa_vals.append(self.get_Kappa(freq_index, i, omega))
            return np.array(kappa_vals)

    def get_whirl_modes(self,n_modes,omega=None):
        '''Method which creates a list of string values indicating the whirl
        ↪  rotation direction.
         :parameter
            n_modes: Integer, number of modes to include
            omega: List or Integer with rotational speeds'''

        modes = []
        for mode in range(n_modes):
```

```
            kappa_values = self.get_Kappa_list(mode,omega)
            if all(kappa >= -1e-3 for kappa in kappa_values):
                # print('Mode #{} is a Forward Whirl'.format(mode + 1))
                modes.append('FW')
            elif all(kappa <= 1e-3 for kappa in kappa_values):
                # print('Mode #{} is a Backward Whirl'.format(mode + 1))
                modes.append('BW')
            else:
                # print('Mode #{} is a mixed mode'.format(mode + 1))
                modes.append('Mixed')

        return modes



    def get_unbalance(self, eps, bet=0, DEL=0, GAM=0):
        '''Method to create the b matrix containing the unbalance forces
        ↪  functions at a given node
        :parameter
            eps: float, Nodal displacement in millimeters of disk COG from
↪   centerline of rotor
            bet: Shaft angle, default =0
            DEL: Angle in u-direction of bent rotor at t = 0.
            GAM: Angle in v-direction of bent rotor at t = 0
            n_disk_unbalance: number of disks in unbalance, default = 1.

        NOTE: only 1 disk is implemented. '''

        md = (self.get_diskMass()[0])
        Ip = self.get_diskIp()[0]
        Id = self.get_diskId()[0]
        b = np.array([[md * eps * np.e ** (1j * DEL)],
                      [-1j * md * eps * np.e ** (1j * DEL)],
                      [1j * (Id - Ip) * bet * np.e ** (1j * GAM)],
                      [(Id - Ip) * bet * np.e ** (1j * GAM)]])


        return b
```

```
    def get_b0(self):
        '''Method which creates the load vector for the unbalanced rotor
        :returns
            b0: Vector containing loads'''

        if len(self.disk_dict['eps']) ==1:
            eps = self.disk_dict['eps'][0]
            b = self.get_unbalance(eps)
            Edof = self.get_Edofs()

            disk_pos = self.disk_dict['pos'][0]
            b0 = np.zeros((self.nNodes * 4, 1)) * 1j
            if list(self.z_coords).index(disk_pos) +1 ==
            ↪  len(list(self.z_coords)):
                b0[Edof[list(self.z_coords).index(disk_pos)-1][4:8]] = b
            else:
                b0[Edof[list(self.z_coords).index(disk_pos)][0:4]] = b

            return b0

        else:
            for i in range(len(self.disk_dict['eps'])):
                eps = self.disk_dict['eps'][i]
                b = self.get_unbalance(eps)
                Edof = self.get_Edofs()

                disk_pos = self.disk_dict['pos'][i]

                b0 = np.zeros((self.nNodes * 4, 1)) * 1j
                b0[Edof[list(self.z_coords).index(disk_pos)][0:4]] = b
            return b0

    def get_nodeCoords_U_V(self, Omega):
        '''Method which creates a 3D matrix containing containing the responses
        ↪  in u and v direction of the rotor at
        a given rotational speed for a given displacement of the COG of the
↪   disk.
```

```
        The time is the time it takes the rotor to do one revolution. The
↪   responses are used
        to create the frequency response plot, the whirl orbit plot and the
↪   operational deflection shapes.
        :parameter
            Omega: Rotational speed in RPM
        :returns
            u: 3D matrix
        3D matrix'''


        O_rads = np.array(Omega) * 2 * np.pi / 60
        if self.beam_type==0:
            M, K, C, G = self.get_M_sys_Bernoulli(), self.get_K_sys_Bernoulli(),
            ↪  self.get_C_sys(), self.get_G_sys_Bernoulli()
        elif self.beam_type == 1:
            M, K, C, G = self.get_M_sys_Timoshenko(),
            ↪  self.get_K_sys_Timoshenko(), self.get_C_sys(),
            ↪  self.get_G_sys_Timoshenko()

        b0 = self.get_b0()
        DOF = self.nNodes * 4
        q0 = np.zeros((DOF, len(O_rads))) * 1j
        u = np.zeros((self.nNodes, len(O_rads), 100))
        v = np.zeros((self.nNodes, len(O_rads), 100))


        for i in range(1, len(O_rads)):
            T = 2 * np.pi / (O_rads[i])
            t = np.linspace(0, T * (.96), 100)
            a = (np.linalg.inv(-O_rads[i] ** 2 * M + 1j * O_rads[i] * (O_rads[i]
            ↪  * G + C) + K) @ (O_rads[i] ** 2 * b0))
            q0[:, i] = a[:, 0]

            for k in range(self.nNodes):

                if k == 0 and self.fixed_end == True:
                    u[k, i , :] = 0
                    v[k, i , :] = 0
```

```
                else:
                    u[k, i, :] = (q0[4 * k, i] * np.e ** (1j * O_rads[i] *
                    ↪  t)).real
                    v[k, i, :] = (q0[4 * k + 1, i] * np.e ** (1j * O_rads[i] *
                    ↪  t)).real

        return u, v


    def get_freqResponse(self, Omega):
        '''Method used to create the frequency response amplitudes in x-, and
        ↪  y-directions as well as the total
        amplitude of the unbalance "eps" at a given rotational speed.
        :parameter
            Omega: List Rotational speed
        :returns
            ampX: Amplitude in X-direction
            ampY: Amplitude in Y-direction
            amp: Total ampltidue np.sqrt(ampX**2 + ampY**2)'''


        u, v = self.get_nodeCoords_U_V(Omega)

        S = u.shape

        nodes = S[0]

        Omax = S[1]

        ampX = np.zeros((nodes, Omax))
        ampY = np.zeros((nodes, Omax))

        for i in range(nodes):
            for j in range(Omax):
                ampX[i, j] = max(abs(u[i, j, :]))
                #print ("ampX", ampX)
                ampY[i, j] = max(abs(v[i, j, :]))

        amp = np.sqrt(ampX ** 2 + ampY ** 2)
```

```
        return ampX, ampY, amp

    def plot_freqResponse(self, Omega, response_type=1, pos = None):
        '''Method used to create frequency response plot:
        :parameter
            Omega: Rotational speed In RPM
            response type 1: Amplitude in X and Y at Disk position 1
            response type 2: Amplitude in X and Y at Disk position 2
            response type 3: Absolute amplitude in X and Y at Disk position 1 &
↪   2'''

        if self.use_disk and pos == None:
            positions = self.disk_dict['pos']
        else:
            positions = [pos]
        nodes = self.z_coords
        ampX, ampY, amp = self.get_freqResponse(Omega)
        fig, ax = plt.subplots()
        ax.tick_params(axis='both', which='major', labelsize=14)

        if response_type == 0:
            ax.semilogy(Omega, amp[list(nodes).index(positions[0]), :],
            ↪  color='Red',
                        label='Response at Node:
                        ↪  {}'.format(list(nodes).index(positions[0])+1))

        elif response_type == 1:
            ax.semilogy(Omega, ampX[list(nodes).index(positions[0]), :],
            ↪  color='Red',
                        label='Response in X - dir at Node:
                        ↪  {}'.format(list(nodes).index(positions[0])+1))
            ax.semilogy(Omega, ampY[list(nodes).index(positions[0]), :],
            ↪  color='Green',
                        label='Response in Y - dir at Node:
                        ↪  {}'.format(list(nodes).index(positions[0])+1))
        elif response_type == 2:
            ax.semilogy(Omega, ampX[list(nodes).index(positions[1]), :],
            ↪  color='Red',
```

```
                    label='Responsein X - dir at Node:
                    ↪ {}'.format(list(nodes).index(positions[1])+1))
            ax.semilogy(Omega, ampY[list(nodes).index(positions[1]), :],
            ↪ color='Green',
                    label='Response in Y - dir at Node:
                    ↪ {}'.format(list(nodes).index(positions[1])+1))
        elif response_type == 3:
            for i in range(len(positions)):
                if i == 0:
                    color = 'Red'
                    line='-'
                elif i == 1:
                    color = 'Green'
                    line='-.'
                ax.semilogy(Omega, amp[list(nodes).index(positions[i]), :],line,
                ↪ color=color,
                            label='Response at Node:
                            ↪ {}'.format(list(nodes).index(positions[i])+1))
                # ax.semilogy(Omega, amp[list(nodes).index(positions[i]), :],
                ↪ color='Green',
                #             label='Response at Node:
                ↪ {}'.format(list(nodes).index(positions[i])))

        ax.legend()
        ax.set_title('Frequency response, Displacement:
        ↪ {}mm'.format(self.disk_dict['eps'][0]))
        ax.set_xlabel(r'Rotor speed $\Omega$ [RPM]')
        ax.set_ylabel('Amplitude [mm]')
        ax.grid(which='Both')

        # plt.show()
        return fig,ax




    def plot_CampBell_Diagram(self, Omega, n_frequencies=6,
    ↪ n_multiple_omega=1,wD=True,wN=False,plot_dampratio=False, include_mode =
    ↪ False):
```

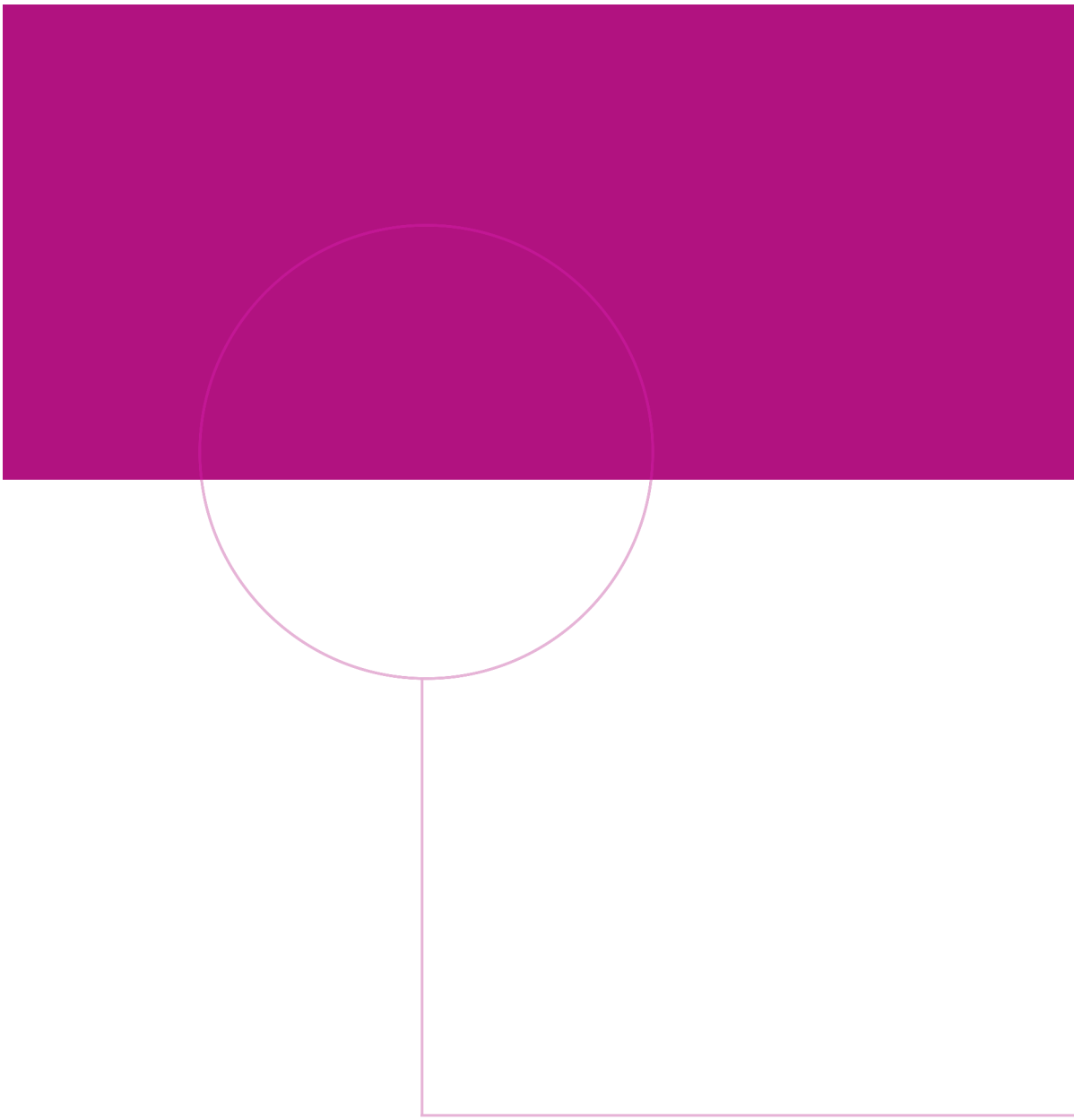
```
        r'''Campbell Diagram.
        This method returns a plot of the natural damped or undamped frequencies
↪   as a function
        of rotational speed of a rotor system.
        :parameter
        Omega: list of rotor speeds in RPM (Will be converted to RAD/S)
        n_frequencies: Integer of number of desired frequencies, default = 6
        n_multiple_omega: Integer of number of multiples of Omega, i.e. load
↪   frequency, default = 1
        wD: Returns plot of damped frequencies, default = True
        wN: Returns plot of undamped frequencies, default = False
        plot_dampratio: Returns plot of the damping ratios as a function of rotor
↪   speed, default = False'''


        modes = self.get_whirl_modes(n_frequencies,Omega) #List of calculated
        ↪  whirl directions
        wd = np.empty((n_frequencies,(len(Omega))))
        wn = np.empty((n_frequencies,(len(Omega))))
        zi = np.empty((n_frequencies,(len(Omega))))
        for o in Omega:
            # w, v = self.solve_Eigproblem(2 * np.pi * o / 60)
            w, v = self.solve_Eigproblem(o)

            w_len = len(w) // 2
            zi_i = ((-np.real(w) / np.absolute(w)))[0:w_len]
            #print("zi_i:", zi_i)
            for i in range(n_frequencies):

                wd[i][Omega.index(o)] = w[i].imag/2/np.pi
                #wn[i][Omega.index(o)] = np.absolute(w[i])/2/np.pi
                wn[i][Omega.index(o)] = np.abs(w[i])/(2*np.pi)

                zi[i][Omega.index(o)] = zi_i[i]

        fig, ax = plt.subplots(1, 1)
        a = 1

        for i in range(n_frequencies):
```

```
            if wD:
                ax.plot(Omega,wd[i,:],label = 'Frequency #{}'.format(i+1) )
                ax.text(max(wd[i]/2),(max(Omega)//2),modes[i])
                if include_mode:
                    a*=-1
                    ax.text(((max(Omega)//2)), (max(wd[i])+min(wd[i]))/2+4*a,
                    ↪  modes[i],fontsize=14)

            elif wN:
                ax.plot(Omega,wn[i,:].real,label = 'Frequency #{}'.format(i+1) )
                if include_mode:
                    a*=-1
                    ax.text(((max(Omega)//2)), (max(wd[i])+min(wd[i]))/2+4*a,
                    ↪  modes[i],fontsize=14)

        if n_multiple_omega == 1:
            ax.plot(Omega, np.array(Omega) * (1 / (60)), '-.',
            ↪  label=r'$\omega$=$\Omega$')
        elif n_multiple_omega == 2:
            ax.plot(Omega, np.array(Omega) * (1 / (60)), '-.',
            ↪  label=r'$\omega$=$\Omega$')
            ax.plot(Omega, np.array(Omega) * (2 / (60)), '-.',
            ↪  label=r'$\omega$=$2\times\Omega$')
        elif n_multiple_omega == 3:
            ax.plot(Omega, np.array(Omega) * (1 / (60)), '-.',
            ↪  label=r'$\omega$=$\Omega$')
            ax.plot(Omega, np.array(Omega) * (2 / (60)), '-.',
            ↪  label=r'$\omega$=$2\times\Omega$')
            ax.plot(Omega, np.array(Omega) * (3 / 60), '-.',
            ↪  label=r'$\omega$=$3\times\Omega$')
        # ax.tick_params(axis='x', which='major', labelsize=14)
        # ax.tick_params(axis='y', which='major', labelsize=18)



        if wD:
            ax.set_ylabel('Damped Natural Frequency [Hz]')
        elif wN:
            ax.set_ylabel(' Natural Frequency [Hz]')
```

```
        ax.set_xlabel(r'Rotor speed $\Omega$ [RPM] ')
        ax.set_title('Critical Speed Map')
        # ax.set_xticks(np.linspace(0,max(Omega),20))
        ax.legend(loc = 'best')
        #ax.legend(loc='upper center', shadow=True, fontsize='x-large')
        #ax.legend(loc=4, shadow=False, fontsize='medium')
        ax.grid()
        # plt.show()
        if plot_dampratio:
            fig2,ax2 = plt.subplots(1,1)
            for i in range(n_frequencies):
                ax2.plot(Omega, zi[i, :], label=r'Damping Ratio $\xi$
                ↪  #{}'.format(i + 1), linewidth=2)


            ax2.legend()
            ax2.grid(which = 'Both')
            ax2.set_xlabel(r'Rotor speed $\Omega$ [RPM] ')
            ax2.set_ylabel(r'Damping Ratio, $\xi$')
            ax2.set_title("Damping Ratio's")

        return fig,ax



    def plot_OperatingDeflection(self, Omega,RPM,only_disk = False,maxlines =
    ↪  False):
        '''Method used to plot the operational deflection of the rotor as well as
        ↪  the whirl orbit of the nodes.
        :parameter
            Omega: List of rotational speeds
            RPM: Operational RPM to plot
            Maxlines: Creates lines from the rotor node to the maximal orbit
↪  radius.
            '''
        O = Omega.index(RPM)
        u,v = self.get_nodeCoords_U_V(Omega)
        n_nodes = (len(self.z_coords))
```

```
        z0 = np.ones(n_nodes)
        x0 = np.zeros(n_nodes)
        y0 = np.zeros(n_nodes)
        # if len(self.disk_dict['pos']) > 1:
        disk_node_pos_lst = []
        for i in range(len(self.disk_dict['pos'])):
            disk_node_pos_lst.append(list(self.z_coords)
            ↪  .index(self.disk_dict['pos'][i]))
        # else:
        #     disk_node_pos = list(self.z_coords).index(self.disk_dict['pos'])

        ampU = np.max((u[:, 0, :])) * 1.1
        ampV = np.max((v[:, 0, :])) * 1.1
        amp = max([ampU, ampV])
        fig = plt.figure()
        ax = plt.axes(projection='3d')
        ax.set_ylim(-(amp+0.3*amp), amp+0.3*amp)
        ax.set_zlim(-(amp+0.3*amp), amp+0.3*amp)
        ax.set_xlim(0, n_nodes)

        for i in range(n_nodes):
            if only_disk :
                a=0

            else:
                a=1

            if i in disk_node_pos_lst:
                color = 'Red'
                linewidth = 2.5
                a=1

            else:
                color = 'Black'
                linewidth = 1.5

            x0[i] = u[i, 0, 0]
            y0[i] = v[i, 0, 0]
            z0[i] *= i
```

```
        x = np.squeeze(u[i, 0, :])*a
        y = np.squeeze(v[i, 0, :])*a
        z = i * np.ones(100)
        ax.plot(z, y,x, linewidth=linewidth,color = color)
        # ax.scatter(z[i], x[0], y[0], marker='*', color='Red', s=50)
        # ax.scatter(z[i],x[-1],y[-1],marker= 'D', color = 'c',s = 50)
        if maxlines == True:
            ax.plot([i,i],[y0[i],min(abs(y))],[x0[i],min(abs(x))],
            ↪  '-',color='Black',linewidth = 1)
    ax.plot([0,z0[-1]+1],[0,0],[0,0],linestyle=(0,(3,5,1,5)),color = 'Black')
    ax.plot(z0, y0,x0, '.-', linewidth=2, color='Blue')
    ax.tick_params(axis='both', which='major', labelsize=14)
    ax.set_title('Operating Deflection at {} RPM'.format(RPM))
    ax.set_xlabel('Nodes',labelpad=10)
    ax.set_zlabel('Displacement u [mm]',labelpad=20)
    ax.set_ylabel('Displacement v [mm]',labelpad=15)
    ax.set_xticks(np.arange(0,len(z0)+1,1))
    #ax.set_xticklabels(np.arange(1,int(len(z0))+1,1))

    ax.w_xaxis.set_pane_color((1.0, 1.0, 1.0, 1.0))
    ax.w_yaxis.set_pane_color((1.0, 1.0, 1.0, 1.0))
    ax.w_zaxis.set_pane_color((1.0, 1.0, 1.0, 1.0))


    # plt.show()
    return fig,ax
```



NTNU

Norwegian University of
Science and Technology