report

November 11, 2025

1 M6e: Gyroscope with three Axes

by:

- Fatemeh karimi Barikarasfi (3765009)
- Souvik Mukherjee (3780278)

2 Introduction

A gyroscope is a rotating body that exhibits unique dynamical behavior due to the conservation of angular momentum. When subjected to external torques, its axis of rotation undergoes characteristic motions known as precession and nutation. In this experiment, these rotational properties of a gyroscope are investigated in detail.

The moment of inertia of the gyro disk is first determined from measurements of angular acceleration under a known applied torque, applied by a falling mass and from the corresponding rotation speed. Then, the behavior of the gyroscope under no external torques is measured, in order to determine a damping coefficient which will be used in the study of precession and nutation.

Subsequently, the precession and nutation frequencies of the gyroscope are studied as functions of the disk's rotation speed, and the correlations of the same with theoretical relations are examined.

3 Experimental Setup

For the experiment, we used a gyroscope by 3B Scientific with two counterweights, with the following specifications:

Parameter	Symbol	Value	Unit
Diameter of disk	$2R$	250	mm
Mass of disk	m	1500	g
Distance disk–vertical axis	z	165	mm
Counterweight masses	$mG1, mG2$	1400, 50	g
Additional mass pieces	m_z	47	g
Distance additional mass–vertical axis	z_z	275	mm
Diameter of bobbin	$2r$	65	mm

For measurements of the periods of the gyroscope, the stopwatch on a smartphone was used, and for the measurements of the rotation speed, a Laser RPM meter was used, which used a set of 8

reflective stripes on the gyroscope disk.

3.1 Methodology

For the first task to measure the moment of inertia of the gyroscope, firstly a string was wound on the bobbin behind the disk and then attached to a weight of m_Z. The weight was then held at a height $h = 73\,cm$ from the ground, and then released in order to accelerate the disk to a certain measured rotational speed ω. The time t for the weight to reach the ground was also measured. The experiment was repeated ten times for higher accuracy.

For the second task, the gyro was set into rotation at a certain rotational speed on the order of 500 RPM using the spring. Then the decrease of the rotation speed was measured every 10 s over a period of 120 s.

For the third task, initially, the tripod stand was removed, and the gyroscope was brought into equilibrium by shifting the counterweights. Then, the gyroscope was brought to a certain rotation speed, before a mass was hooked onto the end of the gyroscope. Then at the start and at the end of each half precession period, the rotation speed of the gyro disk was measured, and the time for each half period was measured as well. The experiment was repeated for ten different rotation frequencies and for both one and two masses.

Finally for the fourth task, the gyroscope was brought to a certain rotation speed and then a nutation was generated by a brief vertical impulse on the gyro axis. The time for three nutation periods was measured with the stopwatch and the rotation speed was recorded at the start and end. The experiment was repeated for ten different rotation frequencies.

4 Theoretical Background

4.1 Moment of Inertia

From the lab manual, we see that there are in total three ways to calculate the moment of inertia, I_3. Firstly, we can go from a basic definition of the geometry of the disk:

$$I_3 = \frac{1}{2} \cdot m \cdot R^2 \tag{1}$$

Secondly, we can derive an equation using the basic equations of motion:

$$I_3 = m_Z \cdot r \cdot \left(\frac{g}{\alpha} - r\right) \tag{2}$$

Finally, we can derive I_3 using the conservation of energy:

$$I_3 = \frac{2 \cdot m_Z \cdot g \cdot h}{\omega^2} - m_Z \cdot r^2 \tag{3}$$

In our analysis, we shall derive I_3 through these three methods and compare the accuracy of the last 2 methods with the initial geometrical definition

4.2 Frictional Dampening

We can model the damping of the rotational frequency through the following equation:

$$\omega(t) = \omega_0 e^{-\beta t} \tag{4}$$

We can fit this equation by taking the logarithm of both sides:

$$ln\,\omega(t) = ln\,\omega_0 - \beta t \tag{5}$$

As can be clearly seen, this gives us a linear relation with slope $-\beta$ and intercept \$ln: ω_0 \$.

4.3 Precession

When a mass is affixed onto the end of the gyroscope, an external torque $\boldsymbol{\tau}$ acts on the system, and changes the direction of the angular momentum according to

$\boldsymbol{\tau} = \frac{d\mathbf{L}}{dt}$.

The torque is given by:

$\tau = mgr$,

where m is the attached mass, g the acceleration due to gravity, and \$r \$ the perpendicular distance from the pivot to the center of mass of the gyro disk or additional weight. This torque acts perpendicular to the angular momentum vector, causing the axis to move at right angles to both torque and angular momentum. As a result, instead of toppling, the gyroscope's axis slowly rotates around the vertical direction — a motion known as **precession**.

The angular velocity of precession, Ω_p , is obtained by equating the rate of change of angular momentum to the applied torque:

$\tau = \frac{dL}{dt} = L\Omega_p \sin\theta$,

where θ is the inclination angle of the gyro axis. For small angles (or when $\sin\theta \approx 1$), this gives

$\Omega_p = \frac{mgr}{I_3\omega}$.

In terms of measurable rotation and precession frequencies $f = \omega_3/2\pi$ and $f_p = \Omega_p/2\pi$, this becomes $f_p = \frac{mgr}{4\pi^2 I_3} \cdot \frac{1}{f}$.

This expression shows that the precession frequency f_p is inversely proportional to the spin frequency f . A faster-spinning gyroscope experiences a slower precession, illustrating the stabilizing effect of angular momentum.

Experimentally, the precession period is measured for different rotation speeds of the gyro disk. By plotting the precession frequency f_p as a function of the reciprocal of the rotation frequency $1/f$, a linear relationship is obtained with slope

slope $= \frac{mgr}{4\pi^2 I_3}$.

4.4 Nutation

When the gyroscope is slightly disturbed from its equilibrium orientation whilst rotating with angular speed ω, the axis does not simply return smoothly to its original position. Instead, due to the

interplay between the restoring torque and the gyroscope's angular momentum, the axis undergoes a small oscillatory motion called **nutation**.

Physically, nutation can be understood as a *wobbling* of the spin axis about its mean position. When a torque acts briefly on the spinning top, the torque vector changes the direction of the angular momentum vector. However, because the torque and angular momentum are not perfectly aligned, part of the energy goes into producing a small periodic oscillation of the tilt angle. This motion occurs simultaneously with the slower **precession** of the gyroscope about the vertical axis.

For a symmetric top (where $ I_1 = I_2 \neq I_3$), *the nutation angular frequency is given by*

$ \Omega_n = \omega I_3 - I_1 \overline{I_1$}

where $ \omega$ is the spin angular velocity.

Converting to ordinary frequencies $ f_n = \Omega_n / 2\pi$ and $ f = \omega / 2\pi$ yields

$ $\boxed{f_n = kf, \quad \text{where } k = \frac{I_3 - I_1}{I_1}}$ $

This relation shows that the **nutation frequency is directly proportional to the spin frequency**, and the proportionality constant $ k $ depends solely on the gyroscope's geometry and mass distribution.

Experimentally, by measuring the nutation frequency $ f_n $ for various rotation frequencies $ f $ and plotting $ f_n $ versus $ f $, the slope $ k $ can be determined. The transverse moment of inertia $ I_1 $ is then obtained from the known moment of inertia about the spin axis $ I_3 $ using

$ I_1 = I_3 \overline{1+k$}

5 Data Processing and Analysis

The data was collected as aforementioned using a stopwatch and RPM speed meter. This was then inputted into a digital spreadsheet and then processed using Python, using which results and plots were made.

[1]:
```
import numpy as np
import pandas as pd
import math
from IPython.display import display, Latex
import matplotlib.pyplot as plt
```

6 Task1

[2]:
```
class GyroInertia:

    g = 9.81 # m/s^2

    def __init__(self, m_disk, R_disk, r_bobbin, m_weight,
                 u_m_disk=None, u_R_disk=None):
```

```
        self.m_disk = m_disk
        self.R_disk = R_disk
        self.r_bobbin = r_bobbin
        self.m_weight = m_weight
        self.u_m_disk = u_m_disk
        self.u_R_disk = u_R_disk

    # Formula
    def I_geometry(self):
        """Moment of inertia and its uncertainty by formula"""
        I = 0.5 * self.m_disk * self.R_disk**2

        if self.u_m_disk is not None and self.u_R_disk is not None:
            dI_dm = 0.5 * self.R_disk**2
            dI_dR = self.m_disk * self.R_disk
            u_I = math.sqrt((dI_dm * self.u_m_disk)**2 + (dI_dR * self.
↪u_R_disk)**2)
        else:
            u_I = None
        return I, u_I

    # Alpha method (using t)
    def I_from_time(self, h, t):
        a = 2.0 * h / t**2
        alpha = a / self.r_bobbin
        return self.m_weight * self.r_bobbin * (self.g / alpha - self.r_bobbin)

    @staticmethod
    def rpm_to_rad_s(rpm):
        return rpm * 2 * math.pi / 60.0

    # Energy method (using rpm)
    def I_from_rpm(self, h, rpm_meter, stripes=8):
        rpm_disk = rpm_meter / stripes
        omega = self.rpm_to_rad_s(rpm_disk)
        return (2 * self.m_weight * self.g * h) / omega**2 - self.m_weight *␣
↪self.r_bobbin**2

    # Compute statistics from multiple measurements
    def analyze(self, h, times, rpms, stripes=8):

        """
        Compute I for each measurement using both methods,
        then return mean, std, and percent deviation from geometry.
        """
        I_geom = self.I_geometry()
```

```
        # compute arrays
        I_time = np.array([self.I_from_time(h, t) for t in times])
        I_rpm = np.array([self.I_from_rpm(h, rpm, stripes) for rpm in rpms])

        # stats
        def stats(arr):
            return {
                "mean": arr.mean(),
                "std": arr.std(ddof=1)
            }

        return {
            "geometry": I_geom,
            "from_time": stats(I_time),
            "from_rpm": stats(I_rpm),
        }
```

[3]:
```
df_task1 = pd.read_csv("./data/task01.csv")
times = df_task1['t (s)'].to_numpy()
rpms = df_task1['w (rpm)'].to_numpy()

R_disk = 250 / 2.0 * 0.001 #mm
u_R_disk = 1.0 / 2.0 * 0.001 #mm
m_disk = 1500 * 0.001 #gg
u_m_disk = 1 * 0.001 #gr
h = 73 * 0.01 #cm
u_h = 1 * 0.01 #cm
r_bobbin = 65.0 / 2.0 * 0.001 # mm
m_weight = 47 * 0.001 #gr
```

[6]:
```
gyro = GyroInertia(m_disk, R_disk, r_bobbin, m_weight, u_m_disk, u_R_disk)
result_1 = gyro.analyze(h, times, rpms)
result_1
```

[6]:
```
{'geometry': (0.01171875, 9.407495764681481e-05),
 'from_time': {'mean': np.float64(0.008503542657465755),
  'std': np.float64(0.0009116935653116154)},
 'from_rpm': {'mean': np.float64(0.01486052964385364),
  'std': np.float64(0.0016250304785011263)}}
```

6.1 Moment of Inertia

In the first task, we calculated the moment of inertia I_3 in the three different ways as aforementioned. The results are displayed in the table below:

Method	Value
Geometrical	$(1.1719 \pm 0.0094) \times 10^2$ kg·m^2
Equations of Motion	$(8.50 \pm 0.91\ ) \times 10^3$ kg·m^2
Conservation of Energy	$(1.49 \pm 0.16) \times 10^2$ kg·m^2

As seen, both the measured values of I_3 are seen to be relatively close to the geometrically derived value. The minor divergence arises from inaccuracies and the uncertainties in the measurement of both the RPM of the disk and the time taken for the mass to fall to the ground. Overall, this shows high accuracy of the experimental task.

7 Task 2

[53]:
```
class GyroFriction:

    def __init__(self, I3, u_I3=None):

        self.I3 = I3
        self.u_I3 = u_I3
        self.data = None
        self.beta = None
        self.u_beta = None
        self.k = None
        self.u_k = None
        self.omega0 = None


    @staticmethod
    def rpm_to_rad_s(rpm):
        return rpm * 2 * math.pi / 60.0

    def fit(self, times, rpms, stripes=8):

        omegas = self.rpm_to_rad_s(rpms / stripes)

        y = np.log(omegas)
        coeffs = np.polyfit(times, y, 1, cov=True)
        slope, intercept = coeffs[0]
        cov = coeffs[1]
        u_slope, u_intercept = np.sqrt(np.diag(cov))

        beta = -slope
        u_beta = u_slope
        omega0 = np.exp(intercept)


        self.data = {
```

```
            "t": times,
            "omega": omegas,
            "ln_omega": y,
            "fit_y": intercept + slope * times
        }
        self.beta = beta
        self.u_beta = u_beta
        self.omega0 = omega0


        if self.I3 is not None:
            self.k = self.beta * self.I3
            if self.u_I3 is not None:
                self.u_k = self.k * math.sqrt(
                    (self.u_beta / self.beta)**2 + (self.u_I3 / self.I3)**2
                )
            else:
                self.u_k = None

        print(beta,u_beta, intercept, u_intercept)
        return beta, u_beta


    def plot(self):

        """Plot ω(t) and ln(ω) vs t with fit."""

        if self.data is None:
            raise ValueError("No data fitted yet. Run fit() first.")
        t, omega, ln_omega, fit_y = (
            self.data["t"],
            self.data["omega"],
            self.data["ln_omega"],
            self.data["fit_y"],
        )

        fig, axs = plt.subplots(1, 2, figsize=(10, 4))

        # ω(t)
        axs[0].plot(t, omega, "o", label="Measured $\\omega$")
        #axs[0].plot(t, np.exp(fit_y), "-", label="Fit $\\omega_0 e^{-\\beta
↪t}$")
        axs[0].set_xlabel("Time t [s]")
        axs[0].set_ylabel("Angular speed ω [rad/s]")
        axs[0].legend()
        axs[0].grid(True)
```

```
        # ln ω vs t
        axs[1].plot(t, ln_omega, "o", label="ln ω(t)")
        axs[1].plot(t, fit_y, "-", label="Linear fit $ln\\: \\omega (t) = ln\\:␣
↪\\omega_0 -\\beta t$")
        axs[1].set_xlabel("Time t [s]")
        axs[1].set_ylabel("ln(ω(t))")
        axs[1].legend()
        axs[1].grid(True)

        plt.tight_layout()
        plt.show()
```

[49]:
```
df_task2 = pd.read_csv('./data/task02.csv')
times = df_task2['t (s)'].to_numpy()
rpms = df_task2['w (rpm)'].to_numpy()
I3 = round(result_1['geometry'][0], 5)
```

[54]:
```
gyro_fric = GyroFriction(I3)
gyro_fric.fit(times, rpms)
gyro_fric.plot()
```

```
0.00423557831742676 3.644508366562658e-05 3.828206378678316 0.005102311713187719
```

Shown above are two plots. One is a direct plot of the angular velocity versus the time, and exhibits logarithmic behavior, as expected from the theoretical model. The second plot on the right is a plot of the logarithm of the angular velocity versus the time.

In the second plot, the logarithmic models as described in the Theoretical Background section has been fitted. The model is described as such:

Model Equation - $ln\,\omega(t) = ln\,\omega_0 - \beta t$;

$\beta = (4.236 \pm 0.036) \times 10^3$; $ln\,\omega_0 = 3.828 \pm 0.005$

8 Task 3

[85]:
```
class PrecessionAnalyzer:
    g = 9.81 #kg/m^s

    def __init__(self, r_arm, m_piece, stripes=8):
```

```
        self.r_arm = r_arm
        self.m_piece = m_piece
        self.stripes = stripes

        self.df = None
        self.results = None
        self.slope = None
        self.I3 = None
        self.u_slope = None
        self.u_I3 = None
        self.R2 = None
        self.residuals = None

    def load_data(self, path):
        self.df = pd.read_csv(path)
        return self

    def compute(self):
        if self.df is None:
            raise ValueError('Call load_data() first!')

        # Convert columns to NumPy arrays
        halfT = self.df['half_period_s'].to_numpy(dtype=float)
        rpm_m = self.df['rpm_meter'].to_numpy(dtype=float)
        k = self.df['mass_count'].to_numpy(dtype=float)

        # Full period
        T = 2.0 * halfT
        Omega = 2.0 / T

        # Angular velocity
        rpm_disk = rpm_m / float(self.stripes)
        omega1 = (2.0 * np.pi / 60.0) * rpm_disk
        omega = rpm_disk/60

        # Torque
        tau = k * self.m_piece * self.g * self.r_arm

        # Regression values
        #x  = tau / omega
        x= tau/(4*np.pi*np.pi*omega)
        y = Omega

        # Store data in a DataFram
        self.results = pd.DataFrame({
            'dataset': self.df['dataset'],
            'mass_count': k,
```

```
            'half_period_s': halfT,
            'T_full': T,
            'rpm_meter': rpm_m,
            'rpm_disk': rpm_disk,
            'omega': omega,
            'Omega': Omega,
            'tau': tau,
            'x': x,
            'y': y
        })

        return self



    def fit(self):
        if self.results is None:
            raise ValueError("Call compute() first.")
        X = self.results["x"].to_numpy(dtype=float)
        Y = self.results["y"].to_numpy(dtype=float)

        # Through-origin least squares
        Sxx = np.sum(X*X)
        Sxy = np.sum(X*Y)
        slope = Sxy / Sxx

        # Residuals and origin-anchored R^2
        Yhat = slope * X
        resid = Y - Yhat
        n = len(X)
        dof = max(n - 1, 1)
        SSE = np.sum(resid**2)
        SST0 = np.sum(Y**2)  # sum of squares about 0 for origin-anchored model
        R2 = 1.0 - SSE / SST0 if SST0 > 0 else np.nan

        # Standard error of slope and uncertainty of I3
        sigma2 = SSE / dof
        var_slope = sigma2 / Sxx
        u_slope = np.sqrt(var_slope)
        I3 = 1.0 / slope
        u_I3 = u_slope / (slope**2)

        # Store
        self.slope = slope
        self.I3 = I3
        self.u_slope = u_slope
        self.u_I3 = u_I3
```

```
        self.R2 = R2
        self.residuals = resid
        return self

    def report(self):
        if self.I3 is None:
            raise ValueError("Call fit() first.")
        print("\nModel:  f_p = (1/I3) * (τ/ω)  (fit through origin)")
        print(f"Slope = 1/I3 = {self.slope:.6e} ± {self.u_slope:.6e}")
        print(f"I3 = {self.I3:.6e} ± {self.u_I3:.6e}  kg·m²")

    def plot(self):
        """Plot Ω vs τ/ω with units and orange fit line."""
        if self.results is None or self.slope is None:
            raise ValueError("Call compute().fit() first.")

        X = self.results["x"].to_numpy(float)
        Y = self.results["y"].to_numpy(float)

        plt.figure(figsize=(7, 5))

        # Scatter measurement points
        plt.scatter(X, Y, color="blue", label="Measurements")

        # Fit line
        x_line = np.linspace(0, X.max() * 1.1, 200)
        y_line = self.slope * x_line
        plt.plot(x_line, y_line, color="orange", linewidth=2.2, label="Fit")

        # Axes labels with physical units
        plt.xlabel("$\\tau / 4\\pi^2f \: (N·m·s^{-1})$", fontsize=12)
        plt.ylabel("$f_p$  (Hz)", fontsize=12)

        # Title
        plt.title("Gyroscope Precession:  $f_p = \\tau / 4\\pi^2 I_3 f$",␣
↪fontsize=14)

        plt.grid(True, linestyle="--", alpha=0.6)
        plt.legend()
        plt.tight_layout()
        plt.show()
```

[86]:
```
analyzer = (
    PrecessionAnalyzer(
        r_arm=0.275,      # from PDF: 275 mm
        m_piece=0.047,    # 47 g piece
        stripes=8         # disk has 8 reflective stripes
```

```
    )
    .load_data("./data/task03.csv")
    .compute()
    .fit()
)

analyzer.report()
analyzer.plot()
```

```
Model:  f_p = (1/I3) * (τ/ω)  (fit through origin)
Slope = 1/I3 = 1.605516e+02 ± 4.244634e+00
I3 = 6.228528e-03 ± 1.646687e-04  kg·m²
```

In the above plot, we see that the precession frequency fits quite well to the rotational frequency. From the linear fit, using the theoretical background, we get a slope of:

Slope $= (1.606 \pm 0.042) \cdot 10^2$

As we know that the slope is equal to $\frac{1}{I_3}$, we can get the value of I_3 as:

$I_3 = (6.229 \pm 0.165) \cdot 10^{-3}\ kg\ m^2$

We see that there is significant divergence of the value of I_3 from the geometrically determined values in Task 1. We can see that this is potentially caused by the presence of a set of around 5 stray data points below the line of best fit, which potentially affected the slope. There are additional considerations on the experimental side which will be discussed in the Analysis section.

9 Task 4

[ ]:
```
class NutationAnalyzer:

    def __init__(self, I3, stripes=8):

        self.I3 = float(I3)
        self.stripes = int(stripes)

        self.df = None
        self.summary = None
        self.slope = None
        self.u_slope = None
```

```
        self.I1 = None
        self.u_I1 = None
        self.R2 = None

    def load(self, path):
        df = pd.read_csv(path)

        # Normalize column names
        df.columns = [c.strip() for c in df.columns]

        required = {"set", "T_s", "rpm_raw"}
        if not required.issubset(df.columns):
            raise ValueError(
                f"Input file must contain columns: {required}, but found {df.
↪columns.tolist()}"
            )

        self.df = df.dropna().copy()

        if self.df.empty:
            raise ValueError("Input dataframe is empty after cleaning.")

        return self

    def compute(self):
        """Group by dataset and compute ω and Ω_n."""
        if self.df is None:
            raise ValueError("Call load() first.")

        results = []

        for set_id, group in self.df.groupby("set"):

            # Convert columns safely
            T_vals = pd.to_numeric(group["T_s"], errors="coerce").dropna().
↪to_numpy()
            rpm_vals = pd.to_numeric(group["rpm_raw"], errors="coerce").dropna().
↪to_numpy()

            if len(T_vals) == 0 or len(rpm_vals) == 0:
                continue

            T_mean = np.mean(T_vals)
            if T_mean <= 0:
                continue

            # Ω_n = 2π / T_mean
```

```
            Omega_n = 1 / T_mean

            # Convert tacho RPM → disk RPM → ω
            rpm_disk = rpm_vals / self.stripes
            omega = np.mean(rpm_disk) / 60.0

            results.append(
                {
                    "set": int(set_id),
                    "T_mean_s": T_mean,
                    "omega": omega,
                    "Omega_n": Omega_n,
                }
            )

        if not results:
            raise ValueError("No valid data groups found in file.")

        self.summary = pd.DataFrame(results).sort_values("set").
↪reset_index(drop=True)
        return self

    def fit(self):
        """Fit Ω_n = s * ω and compute I1."""
        if self.summary is None:
            raise ValueError("Call compute() first.")

        X = self.summary["omega"].to_numpy()
        Y = self.summary["Omega_n"].to_numpy()

        if len(X) < 2:
            raise ValueError("At least two datasets required for linear fit.")

        Sxx = np.sum(X * X)
        Sxy = np.sum(X * Y)

        if Sxx == 0:
            raise ValueError("All omega values are zero - cannot compute slope.")

        slope = Sxy / Sxx

        # Fit residuals
        Y_fit = slope * X
        resid = Y - Y_fit

        dof = max(len(X) - 1, 1)
        SSE = np.sum(resid**2)
```

```
        sigma2 = SSE / dof
        u_slope = np.sqrt(sigma2 / Sxx)

        # Moment of inertia
        I1 = self.I3 / (1 + slope)
        u_I1 = (self.I3 / (1 + slope) ** 2) * u_slope

        # R²
        SST = np.sum((Y - Y.mean()) ** 2)
        R2 = 1 - SSE / SST if SST > 0 else np.nan

        # Store
        self.slope = slope
        self.u_slope = u_slope
        self.I1 = I1
        self.u_I1 = u_I1
        self.R2 = R2

        return self

    def report(self):
        print("\nFit model:  Ω_n = s · ω")
        print(f"slope s   = {self.slope:.6e} ± {self.u_slope:.6e}")
        print(f"I1        = {self.I1:.6e} ± {self.u_I1:.6e}  kg·m²")
        return self

    def plot(self):
        if self.summary is None or self.slope is None:
            raise ValueError("Call compute().fit() first.")

        X = self.summary["omega"].to_numpy()
        Y = self.summary["Omega_n"].to_numpy()
        Y_fit = self.slope * X

        # Standard deviation of points around the fit
        sigma = np.sqrt(np.sum((Y - Y_fit) ** 2) / max(len(X) - 1, 1))

        plt.figure(figsize=(7, 5))

        # Data points + error bars
        plt.errorbar(
            X,
            Y,
            yerr=sigma,
            fmt="o",
            color="blue",
            ecolor="gray",
```

```
            elinewidth=1,
            capsize=4,
            label="Data",
        )

        # Fit line
        xx = np.linspace(0, X.max() * 1.1, 200)
        plt.plot(xx, self.slope * xx, color="orange", linewidth=2.3, label="Fit")

        plt.xlabel(r"$f$ (Hz)")
        plt.ylabel(r"$f_n$ (Hz)")
        plt.title(r"Nutation: $f_n = s\,f$,  $I_1 = \dfrac{I_3}{1+s}$")
        plt.grid(True, linestyle="--", alpha=0.6)
        plt.legend()
        plt.tight_layout()
        plt.show()

        return self
```

[80]:
```
I3_value = 0.01172

an = (
    NutationAnalyzer(I3=I3_value)
    .load('./data/task04.csv')
    .compute()
    .fit()
    .report()
    .plot()
)
```

```
Fit model:  Ω_n = s · ω
slope s   = 1.268605e-01 ± 4.018869e-03
I1        = 1.040058e-02 ± 3.709293e-05  kg·m²
```

As can be seen from the graph, the linear fit matches the data quite well. The slope is:

Slope $= (1.269 \pm 0.040) \cdot 10^{-1}$

As we know that the slope is equal to $\frac{I_3 - I_1}{I_1}$, using the value of I_3 from Task 1, we can get the value

of I_1 as:

$I_1 = (1.040 \pm 0.004) \cdot 10^{-2}\, kg\, m^2$

10 Analysis

10.1 Moment of Inertia

Initially we see that both the measured values of the Moment of Inertia are on both sides of the Geometrical Moment of Inertia. The Geometrical Method result can be taken as a fixed reference value with which the other two values can be compared, however it is important to note that the geometrical value assumes an ideal gyroscope. However, the real gyroscope includes extra inertia from components such as the shaft, counterweight, etc. Both the values have a moderate value of divergence from the Geometrical MOI value. The divergences may be caused by the measurement uncertainty in measuring the time taken for the mass to fall to the ground, measuring the RPM of the gyroscope, and in measuring the height of the mass drop. Additionally, as seen in Task 2, there is a certain amount of damping, which affects the conversion of energy to the gyroscope, which is assumed to be 100% in the equation.

10.2 Frictional Dampening

As has been seen both in the graph and in the uncertainty of β, the frictional dampening model is an excellent fit to the data. The moderate value of β shows that for small experimental timeframes, the gyroscope can be assumed to be ideal without significant energy loss. However over longer timeframes it would need to be accounted in the theoretical background, for accurate results.

10.3 Precession

As stated in the Data Processing section above, there is significant divergence of the value of I_3 from the geometrically determined values in Task 1. As stated this is potentially caused by the presence of a set of around 5 stray data points below the line of best fit, which potentially affected the slope. There is also the additional reason of experimental difficulties. Due to high precession frequencies, there were difficulties faced with measuring the RPM accurately every half-period. As stated above, there is also an effect from the frictional dampening.

10.4 Nutation

As stated above, the linear fit from the equation matches the data quite well, evidenced by the low uncertainty in the slope and the resulting value of I_1.