

report

November 11, 2025

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

1 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

```

[4]: 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

[4]: {'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)}}}

2 Task 2

[5]: 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

return beta, u_beta

def plot(self):

    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))

    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 $\omega [rad/s]$")

```

```

    axs[0].legend()
    axs[0].grid(True)

    axs[1].plot(t, ln_omega, "o", label="$\ln(\omega)$")
    axs[1].plot(t, fit_y, "--", label="Linear fit")
    axs[1].set_xlabel("Time $t$ [s]")
    axs[1].set_ylabel("$\ln(\omega)$")
    axs[1].legend()
    axs[1].grid(True)

    plt.tight_layout()
    plt.show()

```

[6]:

```

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)

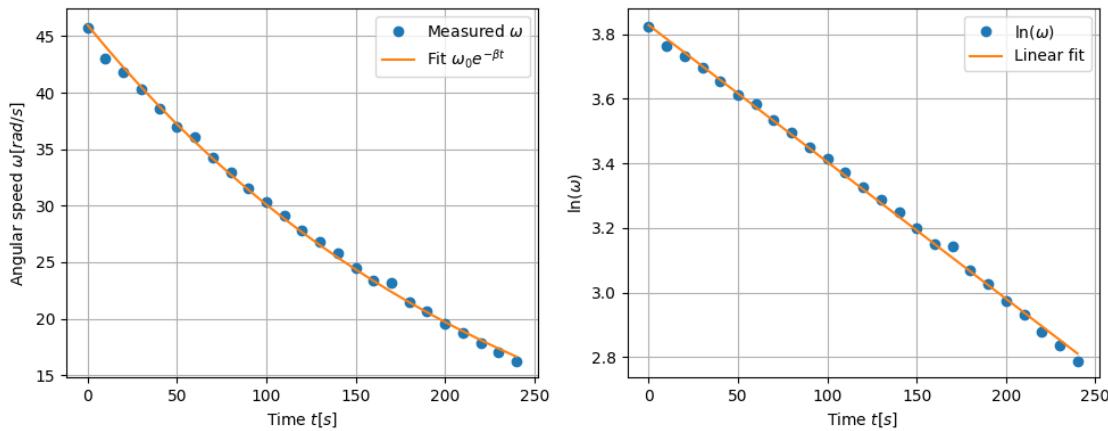
```

[7]:

```

gyro_fric = GyroFriction(I3)
gyro_fric.fit(times, rpms)
gyro_fric.plot()

```



[8]:

```
gyro_fric.k
```

[8]:

```
np.float64(4.964097788024163e-05)
```

[9]:

```
gyro_fric.beta, gyro_fric.u_beta
```

[9]:

```
(np.float64(0.0042355783174267605), np.float64(3.644508366562568e-05))
```

3 Task 3

```
[10]: 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 * np.pi / T

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

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

        # Regression values
        x = tau / omega
        y = Omega
```

```

# Store data in a DataFrame
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:  $\Omega = (1/I_3) * (\tau / \omega)$  (fit through origin)")
        print(f"Slope =  $1/I_3 = {self.slope:.6e} \pm {self.u_slope:.6e}$ ")
        print(f" $I_3 = {self.I3:.6e} \pm {self.u_I3:.6e}$  kg·m2")

    def plot(self):
        """Plot  $\Omega$  vs  $\tau / \omega$  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("$\tau / \omega$ (N·m·s)", fontsize=12)
        plt.ylabel("$\Omega_p$ (rad/s)", fontsize=12)

        # Title
        plt.title("Gyroscope Precession:  $\Omega_p = \tau / \omega$ ", fontweight="bold", fontsize=14)

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

```

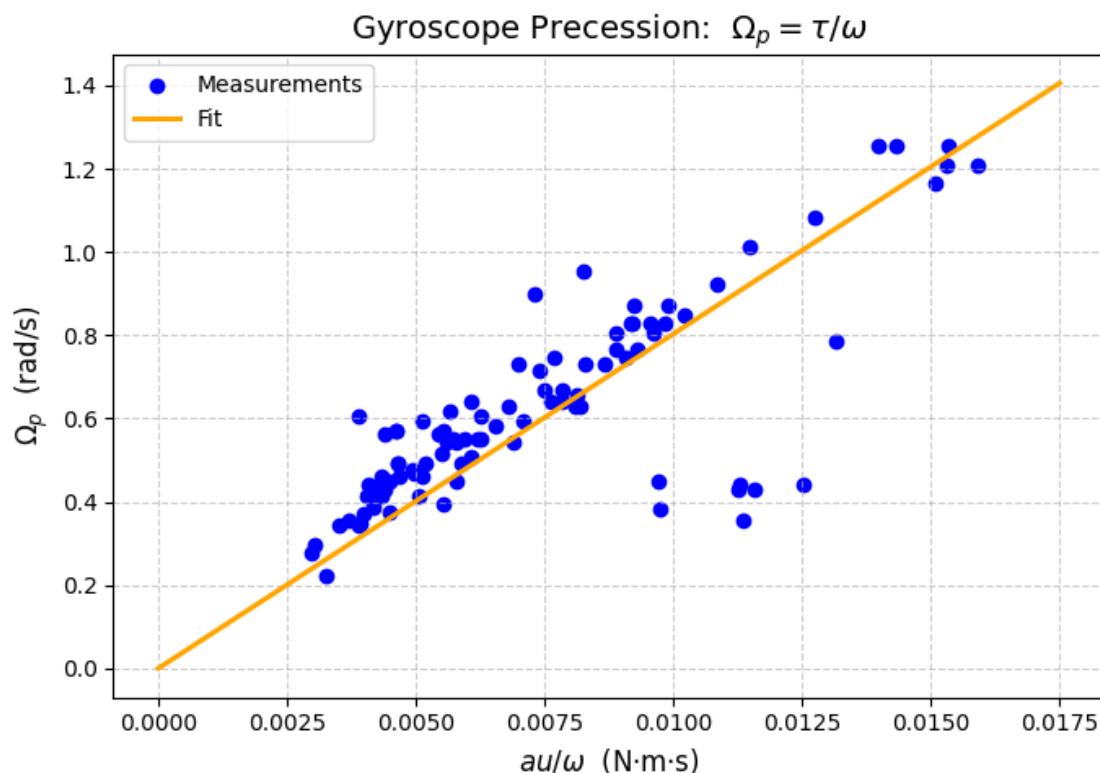
```
[11]: analyzer = (
    PrecessionAnalyzer(
        r_arm=0.275,          # from PDF: 275 mm
        m_piece=0.047,         # 47 g piece
        stripes=8              # disk has 8 reflective stripes
    )
    .load_data("./data/task03.csv")
    .compute()
    .fit()
)

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

Model: $\Omega = (1/I_3) * (\tau/\omega)$ (fit through origin)

Slope = $1/I_3 = 8.027579e+01 \pm 2.122317e+00$

$I_3 = 1.245706e-02 \pm 3.293375e-04 \text{ kg}\cdot\text{m}^2$



4 Task 4

```
[12]: 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 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

        #  $\Omega_n = 2 / T_{mean}$ 
        Omega_n = 2 * np.pi / T_mean

        # Convert tacho RPM → disk RPM →
        rpm_disk = rpm_vals / self.stripes
        omega = 2 * np.pi * 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  $\Omega_n = s * \omega$  and compute  $I_1$ ."""
    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 + ")
    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,
    capsized=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"$\omega$ (rad/s)")
plt.ylabel(r"$\Omega_n$ (rad/s)")
plt.title(r"Nutation: $\Omega_n = s \cdot \omega$, $I_1 = \frac{I_3}{1+s}$")
plt.grid(True, linestyle="--", alpha=0.6)
plt.legend()
plt.tight_layout()
plt.show()

return self

```

[13]: I3_value = 0.01172

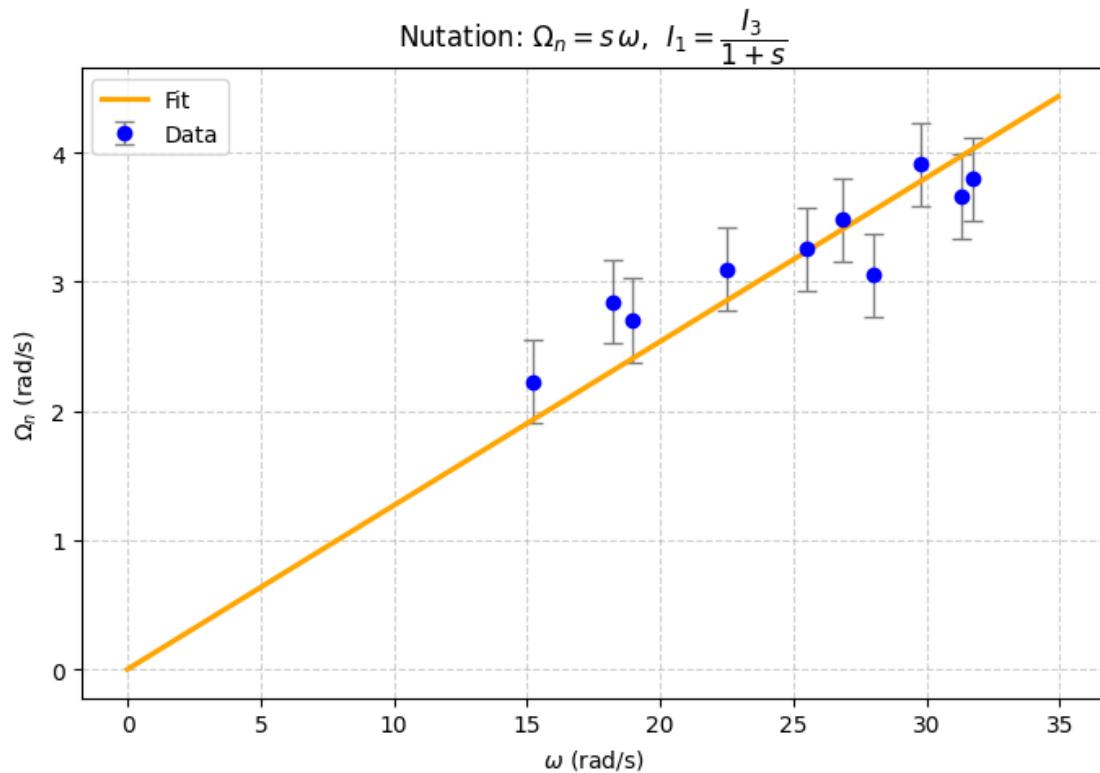
```

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

```

Fit model: $\Omega_n = s \cdot$

slope s = 1.268605e-01 ± 4.018869e-03
I1 = 1.040058e-02 ± 3.709293e-05 kg·m²



[] :