Collaborators

I worked with **Adarsh Iyer, Aren Martinian** and **Andrew Binder** to complete this homework.

Problem 1

Revisit Problem 4 on Homework 4, with the following modifications. The chain is now a massless, ideal string. The ends of the string are wrapped around and fixed to two uniform-density cylindrical shafts of mass M_1 and M_2, respectively, as shown in the figure. The string can slide along the wedge without friction. Gravity acts downwards. Find the tension in the string and the (linear) acceleration of each mass.

Solution: First, let x_1 and x_2 denote the positions of the masses down their respective slopes. Then, writing down the Lagrangian, we get:

$$\mathcal{L} = T - U = \frac{1}{2}M_1\dot{x}_1^2 + \frac{1}{2}M_2\dot{x}_2^2 + \frac{1}{2}I_1\omega_1^2 + \frac{1}{2}I_2\omega_2^2 + M_1 g x_1 \sin\alpha + M_2 g x_2 \sin\beta$$

Here, I use the top of the wedge as the point of zero potential. Now, we have the constraint equation:

$$x_1 + x_2 = L + R_1\theta_1 + R_2\theta_2 \implies f = x_1 + x_2 - R_1\theta_1 - R_2\theta_2 = \text{ const.}$$

Therefore, our full Lagrangian with constraint is:

$$\mathcal{L} = T - U = \frac{1}{2}M_1\dot{x}_1^2 + \frac{1}{2}M_2\dot{x}_2^2 + \frac{1}{2}I_1\omega_1^2 + \frac{1}{2}I_2\omega_2^2 + M_1 g x_1 \sin\alpha + M_2 g x_2 \sin\beta + \lambda(x_1 + x_2 - R_1\theta_1 - R_2\theta_2)$$

Then, we can write down the Euler-Lagrange equations for each coordinate, for which we have 4 (since $\omega = \dot{\theta}$:

$$\begin{aligned}
\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}x_1} = \frac{\mathrm{d}}{\mathrm{d}t}\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\dot{x}_1} &\implies \lambda + M_1 g \sin\alpha = M_1\ddot{x}_1 \\
\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}x_2} = \frac{\mathrm{d}}{\mathrm{d}t}\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\dot{x}_2} &\implies \lambda + M_2 g \sin\beta = M_2\ddot{x}_2 \\
\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\theta_1} = \frac{\mathrm{d}}{\mathrm{d}t}\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\dot{\theta}} &\implies -R_1\lambda = I_1\ddot{\theta}_1 \\
\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\theta_2} = \frac{\mathrm{d}}{\mathrm{d}t}\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\dot{\theta}_2} &\implies -R_2\lambda = I_2\ddot{\theta}_2
\end{aligned}$$

Finally, we can use the second derivative of the time constraint to get a final equation:

$$\ddot{x}_1 + \ddot{x}_2 - R_1\ddot{\theta}_1 - R_2\ddot{\theta}_2 = 0$$

Using all five of these equations, it's possible to solve for $\ddot{x}_1$ and $\ddot{x}_2$ in terms of known quantities. I did this computation via a computer, which gave:

$$\begin{aligned}
\ddot{x}_1 &= \frac{3gM_1\sin\alpha + 2gM_2\sin\alpha - M_2 g\sin\beta}{3(M_1 + M_2)} \\
\ddot{x}_2 &= \frac{3gM_2\sin\beta + 2gM_2\sin\beta - M_1 g\sin\alpha}{3(M_1 + M_2)} \\
\lambda &= -g\frac{M_1 M_2}{M_1 + M_2}\frac{\sin\alpha + \sin\beta}{3}
\end{aligned}$$

□

Problem 2

A heavy, uniform rod AB moves without friction inside a cylindrical hole as shown below, remaining in the vertical plane passing through O. The initial position of the rod is as shown in the figure, with A at the cusp. Gravity acts downwards. Find the angular velocity of the rod at the moment the rod becomes horizontal.

Solution: To do this problem, we first notice that the angular velocity of the center of mass M around the point O is equal to the angular velocity of the rod about its center of mass due to geometry. Therefore, if we can find the angular velocity of the rod at the moment the rod becomes horizontal, then we have solved the problem. First, consider the bottom of the hole as 0 potential.

From geometry alone, we find that the potential energy of the rod when it is horizontal is equal to:

$$U_{\text{bottom}} = Mga\left(1 - \sqrt{a^2 - \frac{l^2}{4}}\right)$$

Likewise, we can solve for the potential energy of the rod in the initial position as:

$$U_{\text{top}} = Mgy_0$$

To calculate y_0, we use similar triangles:

$$\frac{\sqrt{a^2 - \frac{l^2}{4}}}{a} = \frac{\frac{l}{2}}{y_0} \implies y_0 = \frac{al}{2\sqrt{a^2 - \frac{l^2}{4}}}$$

The last quantity we need to calculate to solve is the moment of inertia of the rod about the point O. To do this, we use parallel axis theorem. Since we know that the moment of inertia of the rod about its center of mass is $\frac{1}{3}mL^2$, then:

$$I_{\text{center}} = \frac{1}{3}ML^2 + M\left(a^2 - \frac{l^2}{4}\right)$$

Finally, we can use conservation of energy:

$$\frac{1}{2}I_{\text{center}}\omega^2 = U_{\text{top}} - U_{\text{bottom}}$$

$$\frac{1}{2}\left(\frac{1}{3}ML^2 + M\left(a^2 - \frac{l^2}{4}\right)\right)\omega^2 = \frac{mgal}{2\sqrt{a^2 - \frac{l^2}{4}}} - mga\left(1 - \sqrt{a^2 - \frac{l^2}{4}}\right)$$

First, let the quantity $\sqrt{a^2 - \frac{l^2}{4}}$ be denoted as k. Therefore, we have the equation:

$$\frac{1}{2}\left(\frac{1}{3}ML^2 + Mk^2\right)\omega^2 = \frac{mgal}{2k^2} - mga(1-k)$$

We can now solve for ω:

$$\omega^2 = \frac{\frac{2mgal}{2k^2} - 2mga(1-k)}{\frac{1}{3}ML^2 + Mk^2}$$

$$\therefore \omega = \sqrt{\frac{\frac{mgal}{k^2} - 2mga(1-k)}{\frac{1}{3}ML^2 + Mk^2}}$$

Again, remember that $k = \sqrt{a^2 - \frac{l^2}{4}}$. Returning the substitution, I'd argue, doesn't really simplify the expression too much, so I'll leave the final expression in terms of k. □

Question 1: Rigid body rotation

Learning objectives

In this question you will:

- analytically derive Euler's equations and study the behaviour near equilibria
- numerically solve the equations and compare to theoretical expectations
- simulate the free rotation of real objects and compare to empirical evidence

Let us study the rotation of a rigid body. Recall the definitions of the angular velocity vector $\boldsymbol{\omega}$, the moment of inertia tensor $I_{ij} = \int d^3\mathbf{r}\, \rho(\mathbf{r})(r^2\delta_{ij} - r_i r_j)$, and angular momentum vector $\mathbf{L} = I \cdot \boldsymbol{\omega}$. Recall that the torque determines the rate of change of angular momentum $\boldsymbol{\tau} = \dot{\mathbf{L}}$.

1a.

Let's analyse the system in the body-frame in which the moment of inertia is diagonal, i.e.

$$I = \begin{pmatrix} I_1 & 0 & 0 \\ 0 & I_2 & 0 \\ 0 & 0 & I_3 \end{pmatrix}.$$

Let's call the basis vectors $\mathbf{e}_1$, $\mathbf{e}_2$, and $\mathbf{e}_3$. Don't forget that $\mathbf{e}_i$ change with time. In this basis, the angular momentum is $\mathbf{L} = I_1\omega_1\mathbf{e}_1 + I_2\omega_2\mathbf{e}_2 + I_3\omega_3\mathbf{e}_3$. Assuming there are no torques, derive the Euler equations,

$$I_1\dot{\omega}_1 = (I_2 - I_3)\omega_2\omega_3, \tag{1}$$
$$I_2\dot{\omega}_2 = (I_3 - I_1)\omega_1\omega_3, \tag{2}$$
$$I_3\dot{\omega}_3 = (I_1 - I_2)\omega_1\omega_2. \tag{3}$$

(Hint: for a rigid body the moment of inertia tensor is constant in the body frame. Remember than the rate of change of any vector $\mathbf{v}$ rotationg with angular velocity $\boldsymbol{\omega}$ is given by $\dot{\mathbf{v}} = \boldsymbol{\omega} \times \mathbf{v}$.)

Since there is no torque on the given system, we have:

$$\dot{\mathbf{L}} = \mathbf{L} \times \omega$$

and also the equation:

$$\dot{\mathbf{L}} = I_1\dot{\omega}_1\mathbf{e}_1 + I_2\dot{\omega}_2\mathbf{e}_2 + I_3\dot{\omega}_3\mathbf{e}_3$$

Therefore, equating the two:

$$I_1\dot{\omega}_1 = (I_2 - I_3)\omega_2\omega_3$$
$$I_2\dot{\omega}_2 = (I_1 - I_3)\omega_1\omega_3$$
$$I_3\dot{\omega}_3 = (I_1 - I_2)\omega_1\omega_2$$

where the right hand side is obtained by taking the cross product $\mathbf{L} \times \omega$.

1b.

Fill in the following function to numerically integrate the Euler equations. No need to use a sophisticated integrator; we can use small step sizes if required.

In [115…
```
import numpy as np
import matplotlib.pyplot as plt
```

In [116…
```
def euler(omega0, I1, I2, I3, times):
    """
    (All quantities in body frame, in diagonal basis of inertia tensor)
    omega0: initial angular velocity, array of shape (3,)
    I1,I2,I3: principal moments of inertia, scalars
    times: array of times at which to find omega, array of shape (N,)
    returns: omegas at times, array of shape (3,N)
    """
    omega_1 = np.array([omega0[0]])
    omega_2 = np.array([omega0[1]])
    omega_3 = np.array([omega0[2]])
    h = np.diff(times)[1]
    omegas = [[omega0[0], omega0[1], omega0[2]]]
    for i in range(len(times)-1):
        omega_1 = np.append(omega_1, (I2 - I3)/I1 * omega_2[i] * omega_3[i]
        omega_2 = np.append(omega_2, (I3 - I1)/I2 * omega_1[i] * omega_3[i]
        omega_3 = np.append(omega_3, (I1 - I2)/I3 * omega_1[i] * omega_2[i]
        omegas.append([omega_1[i], omega_2[i], omega_3[i]])
    return np.array(omegas)


# w0 = np.array([3.1,2,0])
# I1 = 1
# I2 = 2
# I3 = 1

# F = euler(w0, I1, I2, I3, np.linspace(0, 10, 1000))
# print(F[0])

# fig = plt.figure()
# ax = fig.add_subplot(111, projection='3d')
# ax.plot([F[i][0] for i in range(len(F))], [F[i][1] for i in range(len(F))]
# ax.set_xlabel('x')
# ax.set_ylabel('y')
# ax.set_zlabel('z')
# plt.show()
```

1c.

What happens when $\boldsymbol{\omega}_0$ is aligned with one of the principal moments of inertia? Check that your function `euler()` returns what you expect in this situation.

We express the vector ω_0 as:

$$\omega_0 = (\omega_{0x}, \omega_{0y}, \omega_{0z})$$

When ω_0 is aligned with one of the principal moments (let it be along $\mathbf{e}_1$), then it means that $\omega_{0x} = \omega_{0y} = 0$. In other words, we only expect ω_x to change. Furthermore, we don't expect ω_z to change since $\dot{\omega_z}$ is dependent on $\omega_{0x}\omega_{0y}$, which are both zero for all time. See below for the verification with the function.

In [117…
```
print(euler(np.array([0, 0, 1]), 2, 3, 4, np.linspace(0, 10, 1000)))
```

```
[[0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 ...
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]]
```

1d.

What happens when $I_1 = I_2 = I_3$? Again, check that your function makes sense.

When all the moments of inertia are equal, we expect that $\dot{\omega_i} = 0$, since each ω_i is dependent on the difference of the two other moments of inertia. Therefore, we expect that each ω_i is constant. This is exactly what we get with the function as well (see below)

In [118…
```
print(euler(np.array([3, 2, 1]), 1, 1, 1, np.linspace(0, 10, 1000))) #note t
```

```
[[3. 2. 1.]
 [3. 2. 1.]
 [3. 2. 1.]
 ...
 [3. 2. 1.]
 [3. 2. 1.]
 [3. 2. 1.]]
```

1e.

Show that if two of the principal moments are equal, then ω precesses around the third principal moment. (Think of a spinning coin without gravity, or an American football.) Find the angular velocity of precession $\omega_p = \frac{2\pi}{T_p}$ in terms of the parameters already defined.

Following lecture notes, if $I_1 = I_2$ for instance, then $\dot{\omega}_3 = 0$, so therefore:

$$\dot{\omega}_1 = \frac{(I_1 - I_3)\omega_3}{I_1}\omega_2 = \Omega_b\omega_2$$
$$\dot{\omega}_2 = \frac{(I_3 - I_1)\omega_3}{I_1}\omega_1 = -\Omega_b\omega_1$$

Then, let $\dot{\eta} = \omega_1 + i\omega_2 = -i\Omega_b\eta$, giving us the solution $\eta = \eta_0 e^{-i\Omega_b t}$. We can then select an axes such that $\omega_1 = \omega_0$ and $\omega_2 = 0$ at $t = 0$, giving the solution:

$$\vec{\omega} = (\omega_0 \cos(\Omega_b t), -\omega_0 \sin(\Omega_b t), \omega_3)$$

This suggests precession about the I_3 axis with angular velocitiy Ω_b. In terms of parameters alrea defined, we can see that:

$$\Omega_b = \frac{(I_1 - I_3)\omega_3}{I_1}$$

1f.

Use your solution `euler()` to estimate the precession rate ω_p of an object with $I_1 = 2$, $I = 1$, and $\boldsymbol{\omega} = (1, 1, 1)^T$ (e.g. you can find the first non-zero time when $|\boldsymbol{\omega} - \boldsymbol{\omega}_0| < \epsilon$ for some $\epsilon \ll 1$). Compare with theoretical expectations.

In [119…
```
I1 = 1
I2 = 1
I3 = 2
w0 = np.array([1, 1, 1])
F = euler(w0, I1, I2, I3, np.linspace(0, 15, 2000))
e = 0.035
for i in range(20, len(F[0])):
    diff = [F[0][i] - 1, F[1][i] - 1, F[2][i] - 1]
    if np.sqrt(diff[0]**2 + diff[1]**2 + diff[2]**2) < e:
        time = i
        break

print(np.linspace(0, 15, 2000)[time])

# fig = plt.figure()
# ax = fig.add_subplot(111, projection='3d')
# ax.plot(F[0], F[1], F[2])
# ax.set_xlabel('x')
# ax.set_ylabel('y')
# ax.set_zlabel('z')
# plt.show()
```

```
6.28064032016008
```

I set $I_3 = 2$ instead of $I_1 = 2$ so that I may use the formula derived in 1e easily. Theoretically, we expect a time:

$$T = \frac{2\pi}{\frac{(1-2)(1)}{1}} = -2\pi$$

Since this motion is periodic, we expect any multiple of 2π, so the result obtained via the `euler()` function ($6.28 \approx 2\pi$) makes sense.

1g.

Fill in the following function that analytically solves Euler's equations for $I_2 = I_3 = I$ using the precession that you calculated above.

In [120…

```
def precession(omega0, I1, I, times):
    """
    (All quantities in body frame, in diagonal basis of inertia tensor)
    omega0: initial angular velocity, array of shape (3,)
    I1,I: principal moments of inertia (I2=I3=I), scalars
    times: array of times at which to find omega, array of shape (N,)
    returns: omegas at times, array of shape (N,3)
    """
    omega_1 = np.array([omega0[0]])
    omega_2 = np.array([omega0[1]])
    omega_3 = np.array([omega0[2]])
    Omega_b = (I1 - I)/I1 * omega0[0] # Omega_b = I1 - I /I1 * omega_1

    omega_0 = np.sqrt(omega0[1]**2 + omega0[2]**2)
    phase = np.arctan2(omega0[2], omega0[1])

    omegas = [[omega0[0], omega0[1], omega0[2]]]
    for i in range(len(times)-1):
        omega_1 = np.append(omega_1, omega_1[i])
        omega_2 = np.append(omega_2, omega_0*np.cos(Omega_b*times[i] + phase
        omega_3 = np.append(omega_3, omega_0*np.sin(Omega_b*times[i] + phase
        omegas.append([omega_1[i], omega_2[i], omega_3[i]])
    return np.array(omegas)

# def precession2(omega0, I1, I, times):
#     """
#     (All quantities in body frame, in diagonal basis of inertia tensor)
#     omega0: initial angular velocity, array of shape (3,)
#     I1,I: principal moments of inertia (I2=I3=I), scalars
#     times: array of times at which to find omega, array of shape (N,)
#     returns: omegas at times, array of shape (N,3)
#     """
#     omega_p = (I1/I - 1)*omega0[0]
#     A = np.linalg.norm((omega0[1], omega0[2]))
#     δ = np.arctan2(omega0[2], omega0[1])
#     omegas = np.array((np.repeat(omega0[0], len(times)),
#     A*np.cos(omega_p*times + δ),
#     A*np.sin(omega_p*times + δ))).T

#     return omegas

w0 = np.array([3.1,2,0])
```

```
I1 = 1
I = 2
F = precession(w0, I1, I, np.linspace(0, 10, 1000))
print(F[0])
```

```
[3.1 2.  0. ]
```

1h.

Recall than in torque-free motion the energy is given by the kinetic energy, $T = \frac{1}{2}\boldsymbol{\omega} \cdot I \cdot \boldsymbol{\omega}$. Fill in the following function that calculates the energy from a given $\boldsymbol{\omega}$ and I in the preferred body frame.

In [121…
```
def energy(omega,I1,I2,I3):
    """
    omega: angular velocity vector, shape (3,)
    I1,I2,I3: principal moments of inertia, scalars
    returns: energy, scalar
    """
    return  1/2 * np.dot(omega, np.dot(np.diag((I1,I2,I3)), omega))
```

1i.

In $\boldsymbol{\omega}$-space, what shape do the energy contours take? Recall that since energy is conserved, $\boldsymbol{\omega}$ is constrained to be on such surfaces.

The following function is supposed to plot multiple trajectories overlaid on the allowed energy surface. It is almost complete, except that it currently plots the energy surface as the unit sphere regardless of input. Complete the function. (Hint: you can obtain the energy surfaces by re-scaling the dimensions of a sphere.)

In [122…
```
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
%matplotlib notebook
#use %matplotlib inline if notebook doesn't work..

def plot_trajectories(trajectories,I1,I2,I3):
    """
    trajectories: array of shape (M,N,3) containing M omega vectors at N tim
                (all omega vectors assumed to be at the same energy)
    I1, I2, I3: principal moments of inertia
    returns nothing, plots trajectories as lines in 3d plot overlaid on ener
                ellipsoid (using energy of first omega vector of first traje
    """
    N = 20
    theta,phi = np.linspace(0,np.pi,N),np.linspace(0,2*np.pi,2*N)
    theta,phi = np.meshgrid(theta, phi)
    T = energy(trajectories[0, 0],I1,I2,I3)

    #Hint: rescale variables (multiply x,y,z with a factor each)
    x = np.sqrt(2*T/I1) * np.sin(theta)*np.cos(phi)
    y = np.sqrt(2*T/I2) * np.sin(theta)*np.sin(phi)
```

```
    z = np.sqrt(2*T/I3) * np.cos(theta)

    fig = plt.figure(figsize=(8,8))
    ax = Axes3D(fig)
    ax.plot_surface(x,y,z,alpha=.2,facecolors=[["w"]*N]*2*N)
    ax.set_xlabel("$\omega_1$")
    ax.set_ylabel("$\omega_2$")
    ax.set_zlabel("$\omega_3$")

    bound = np.amax([x,y,z])
    ax.set_xlim(-bound,bound)
    ax.set_ylim(-bound,bound)
    ax.set_zlim(-bound,bound)

    for omegas in trajectories:
        ax.plot(*omegas.T)
```

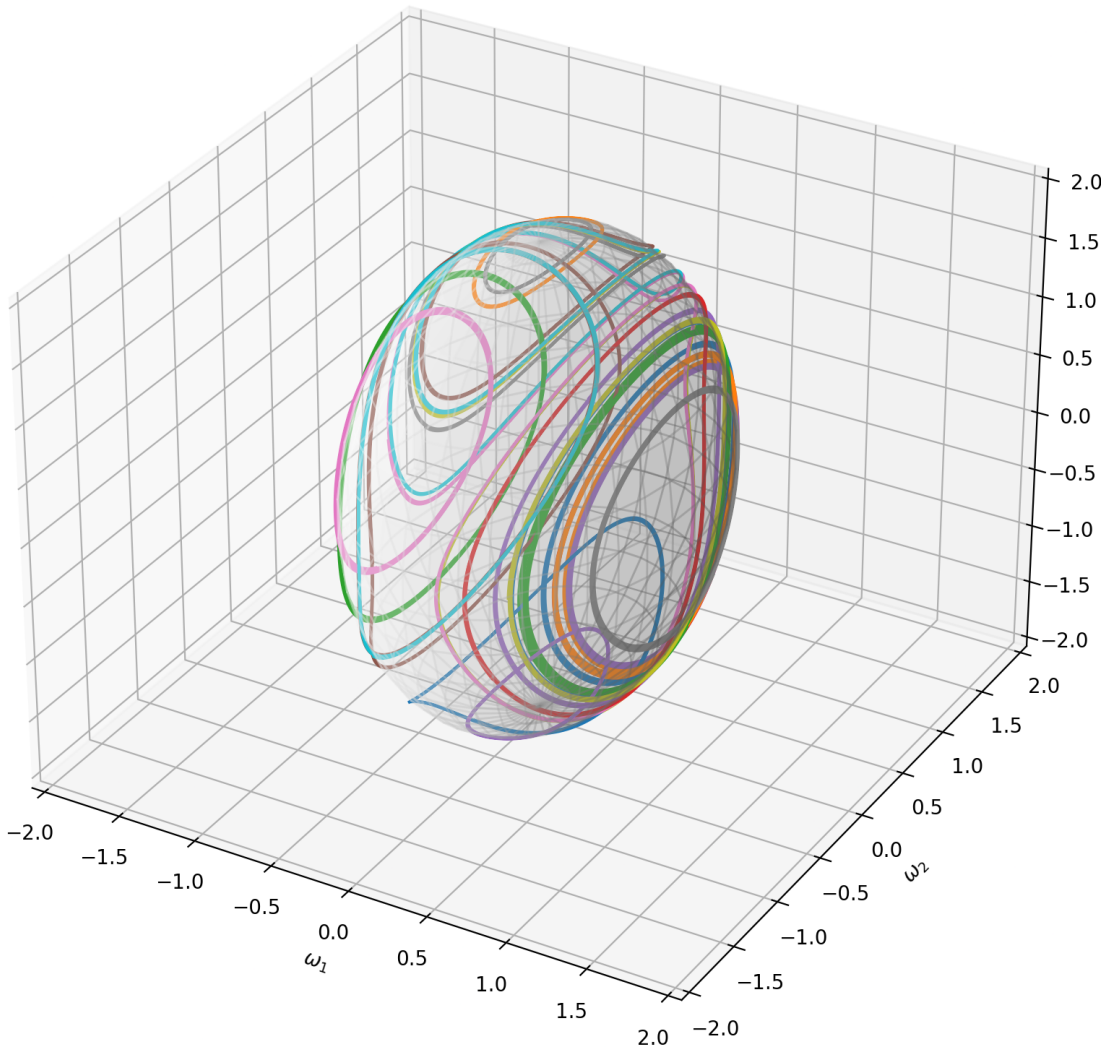
1j.

The following cell plots the trajectories of some randomly chosen initial ωs. You can use your visualisation of energy and your implementation of `euler()` to test each other. Once your energy visualisation is consistent with your plotted trajectories, try increasing the step size. What happens? What does this tell you about the integrator you are using?

In [123…

```
def get_random_initials(I1,I2,I3,energy=1,n=30):
    """
    I1,I2,I3: principal moments of inertia, scalars
    energy: energy of initial states
    n: number of points to sample
    returns: n randomly chosen omega vectors with given energy
    """
    randoms = np.zeros((n,3))
    for i in range(n): #sample uniformly from sphere using rejection
        x = np.random.rand(3)*2-1
        r = np.sum(x**2)
        while r > 1:
            x = np.random.rand(3)*2-1
            r = np.sum(x**2)
        randoms[i] = x/r**.5
    randoms[:,0] = randoms[:,0]*(2*energy/I1)**.5
    randoms[:,1] = randoms[:,1]*(2*energy/I2)**.5
    randoms[:,2] = randoms[:,2]*(2*energy/I3)**.5
    return randoms

I = 2,1,0.5
omega0s = get_random_initials(*I)
times = np.linspace(0,20,10000)
traj = np.array([euler(o,*I,times) for o in omega0s])
plot_trajectories(traj,*I)
```




```
/tmp/ipykernel_92/2209726821.py:25: MatplotlibDeprecationWarning: Axes3D(fig) adding itself to the figure is deprecated since 3.4. Pass the keyword argument auto_add_to_figure=False and use fig.add_axes(ax) to suppress this warning. The default value of auto_add_to_figure will change to False in mpl3.5 and True values will no longer work in 3.6.  This is consistent with other Axes classes.
  ax = Axes3D(fig)
```

For large step sizes, we get trajectories that are no longer confined to the surface of our ellipse (this would correspond to our object magically gaining energy. This implies that in order for our integrator to be good, we require small step sizes. This matches what we expect from the Euler forward method, which requires that the st side h be small.

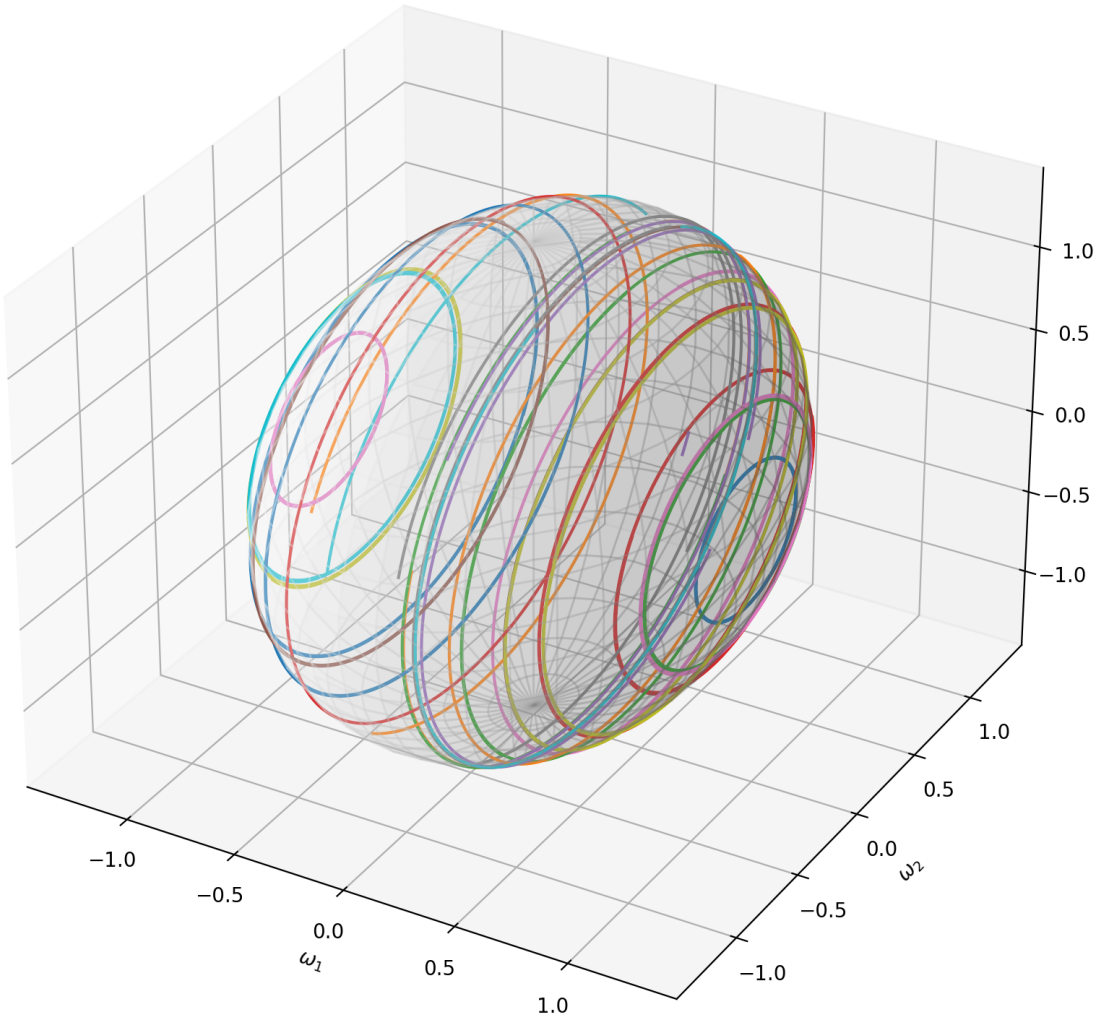
1k.

Use these visualisation techniques to compare your analytical solution `precession()` with your numerical solution `euler()` for $I_2 = I_3 = I$. Plot both $I_1 < I$ and $I_1 > I$.

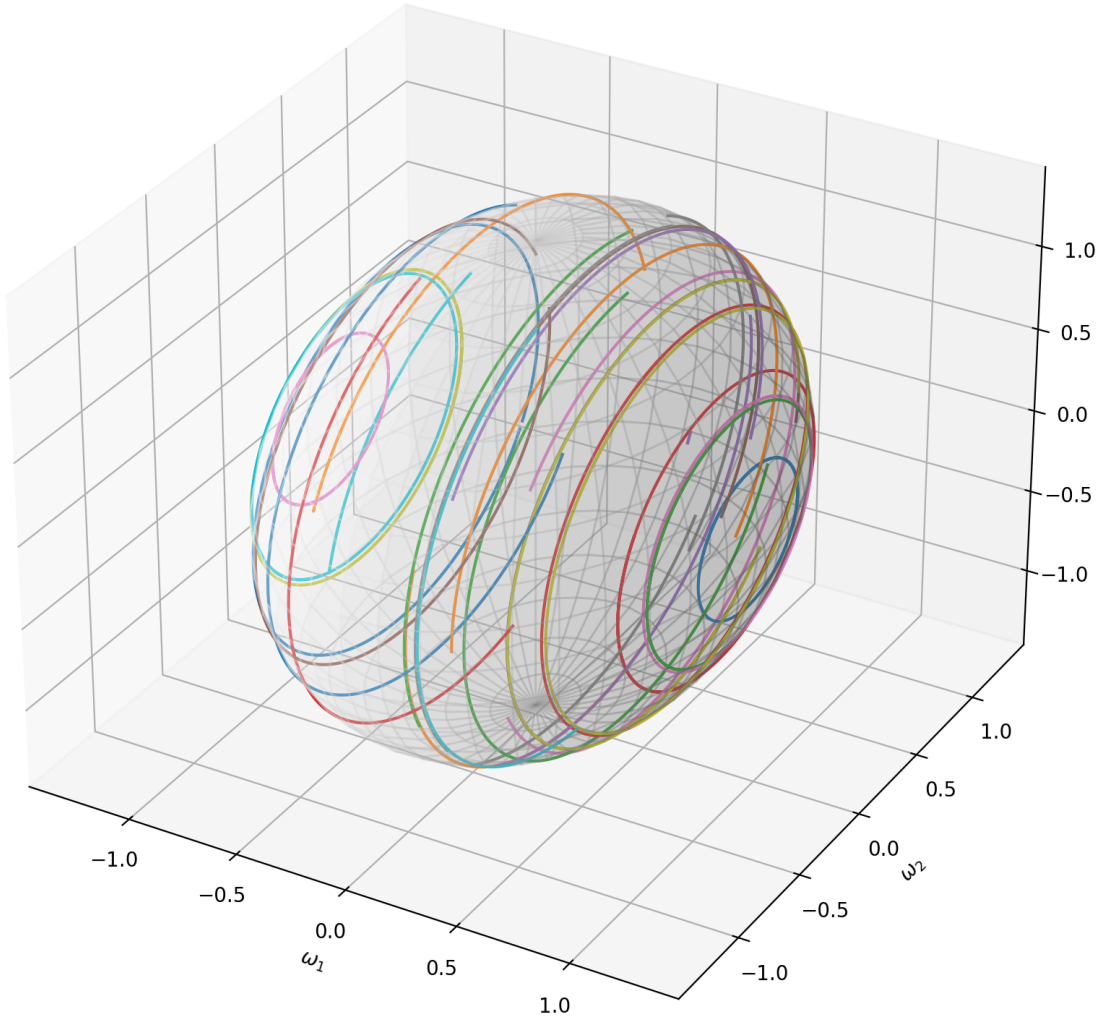
In [124…

```
times = np.linspace(0,20,10000)
```

```
# First case: I1 > I
I = [2, 1, 1]
omega0s = get_random_initials(*I)
traj_euler = np.array([euler(o,*I, times) for o in omega0s])
plot_trajectories(traj_euler, *I)
traj_precession = np.array([precession(o, I[0], I[1], times) for o in omega0
plot_trajectories(traj_precession, *I)
```

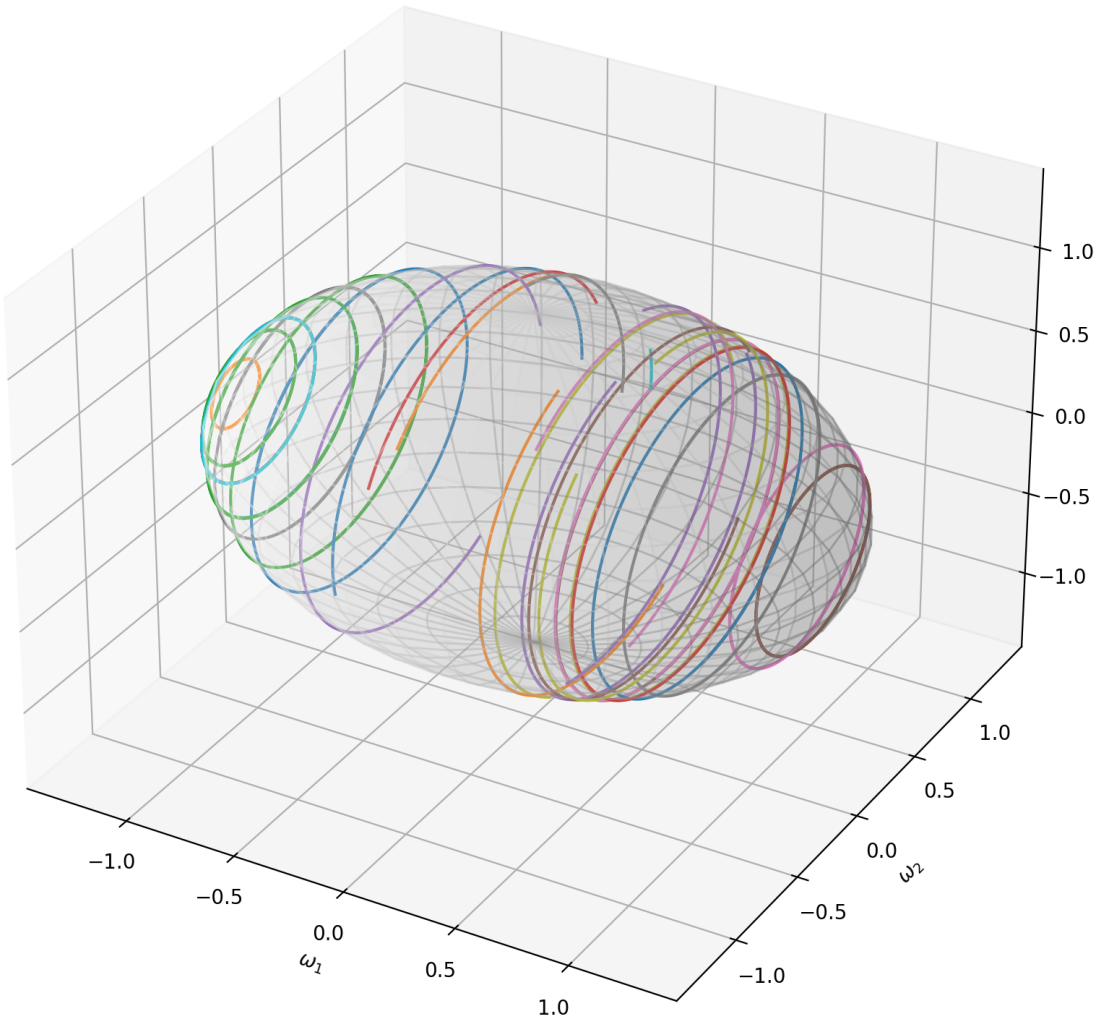



```
/tmp/ipykernel_92/2209726821.py:25: MatplotlibDeprecationWarning: Axes3D(fi
g) adding itself to the figure is deprecated since 3.4. Pass the keyword ar
gument auto_add_to_figure=False and use fig.add_axes(ax) to suppress this w
arning. The default value of auto_add_to_figure will change to False in mpl
3.5 and True values will no longer work in 3.6.  This is consistent with ot
her Axes classes.
  ax = Axes3D(fig)
```

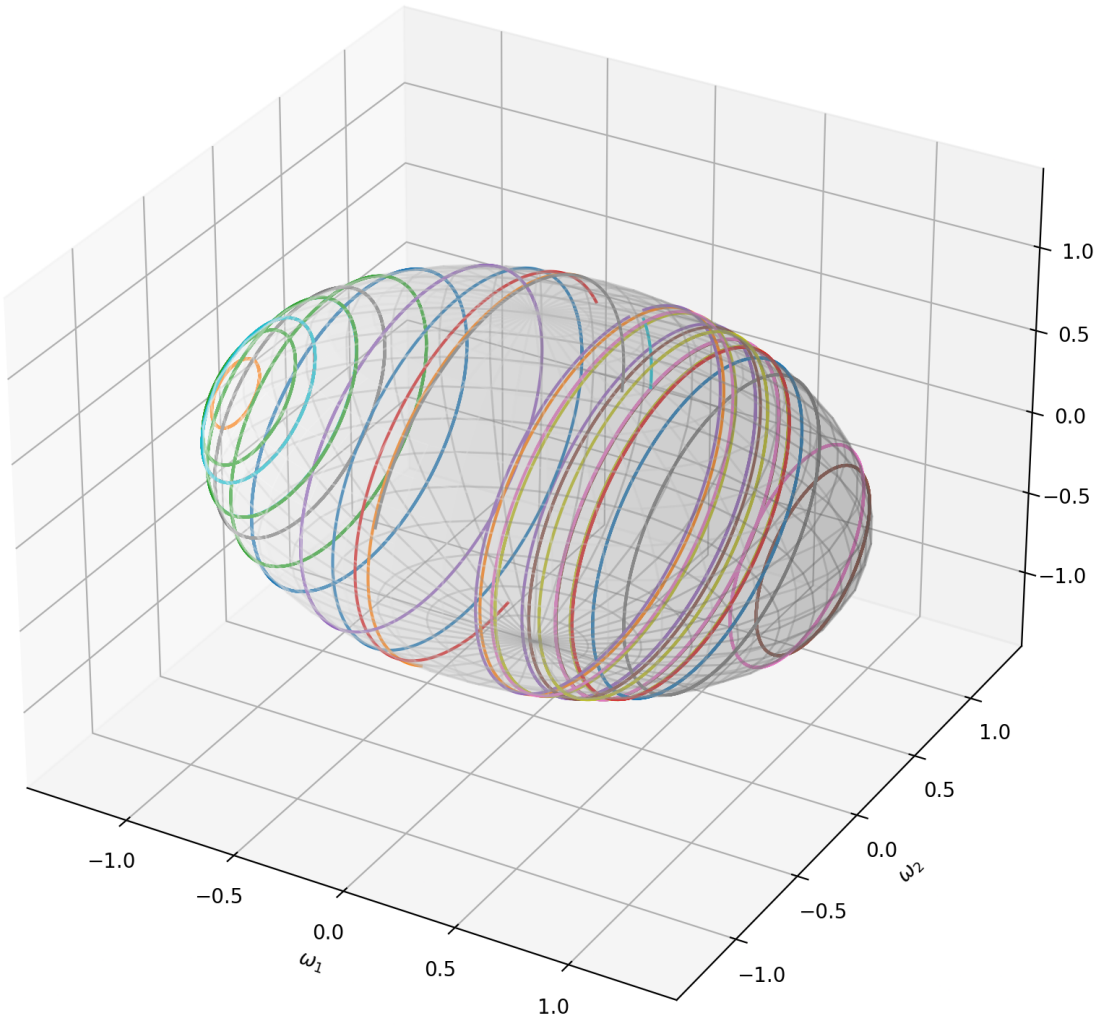


In [125…

```
# Second case: I1 < I
I = [1, 2, 2]
omega0s = get_random_initials(*I)
traj_euler = np.array([euler(o,*I, times) for o in omega0s])
plot_trajectories(traj_euler, *I)
traj_precession = np.array([precession(o, I[0], I[1], times) for o in omega0
plot_trajectories(traj_precession, *I)
```




```
/tmp/ipykernel_92/2209726821.py:25: MatplotlibDeprecationWarning: Axes3D(fi
g) adding itself to the figure is deprecated since 3.4. Pass the keyword ar
gument auto_add_to_figure=False and use fig.add_axes(ax) to suppress this w
arning. The default value of auto_add_to_figure will change to False in mpl
3.5 and True values will no longer work in 3.6.  This is consistent with ot
her Axes classes.
  ax = Axes3D(fig)
```



Overall, the plots look very similar. Of course, the `euler()` function is only an approximation to the true solution given by `precession()`, so we see that `precession()` completes more trajectories (as in we see more complete circles) when compared to `euler()`. Again, this makes sense, since `euler()` is only meant to be an approximate solution, with its accuracy getting better as we decrease the step size.

1l.

The Earth's axis of rotation precesses with a period of about 430 days (not to be confused with the precession of 26,000 years around its orbital rotation axis, which is caused mostly by tidal/gravitational forces). This precession is known as *Chandler wobble*. Assuming the Earth is a rigid oblate spherioid (i.e. $I_1 > I$), estimate the fractional asymmetry of the principal moments of inertia (i.e. $\frac{I-I_1}{I}$). From this, assuming the Earth's density is a scaled spherically symmetric distribution (like how you created the energy ellipsoid surfaces), calculate its *ellipticity* (ratio of major to minor axes).

We know that:

$$\Omega_b = \frac{I_1 - I}{I_1}\omega_3$$

Now, using the fact that the values are given as *frequencies* and not *angular velocities*, we can solve for the fractional asymmetry by taking the reciprocal Ω_b/ω_3:

$$\frac{\omega_3}{\Omega_b} = \frac{1}{430}$$

The ellipticity is then calculated as $\sqrt{\frac{I_1}{I}} = 1.001$.

1m.

Real data for the Chandler wobble is shown below.

 Source: Michael Mandeville

Does this match the prediction from Euler's equations? If not, then are Euler's euqations wrong, or do some of our assumptions (which ones?) about the Earth break down?

This doesn't match what we get from Euler's equations, which predict that we should get perfect circles, which we don't get in the figure. This is due to our assumption that the Earth is uniformly dense, which, when relaxed, will cause the moment of inertia tensor to no longer be diagonal.

1n.

Now consider the general case, $I_1 > I_2 > I_3$ (in the approriate basis). We know that the principal moments of inertia are fixed points (equilibria) for $\boldsymbol{\omega}$. However, are they stable? Analyse the motion very close to the principal moments (i.e. linearise the differential equation). Which of these equilibria are stable? Are there any other equilibria? Does this agree with the visualisation you made earlier?

Recall the solution for $\ddot{\omega}_1$ from the lecture notes:

$$\ddot{\omega}_1 = -\left[\frac{(I_3 - I_2)(I_3 - I_1)\omega_3^2}{I_1 I_2}\right]\omega_1$$

This solution implied that if I_3 were in between I_1 and I_2, then the motion would be stable around I_1. However, if I_3 is either larger than or smaller than both I_1 and I_2, then the motion is stable. In other words, motion is unstable *only* about the axis whose moment of inertia is in between the other two; this is known as the intermediate axis theorem.

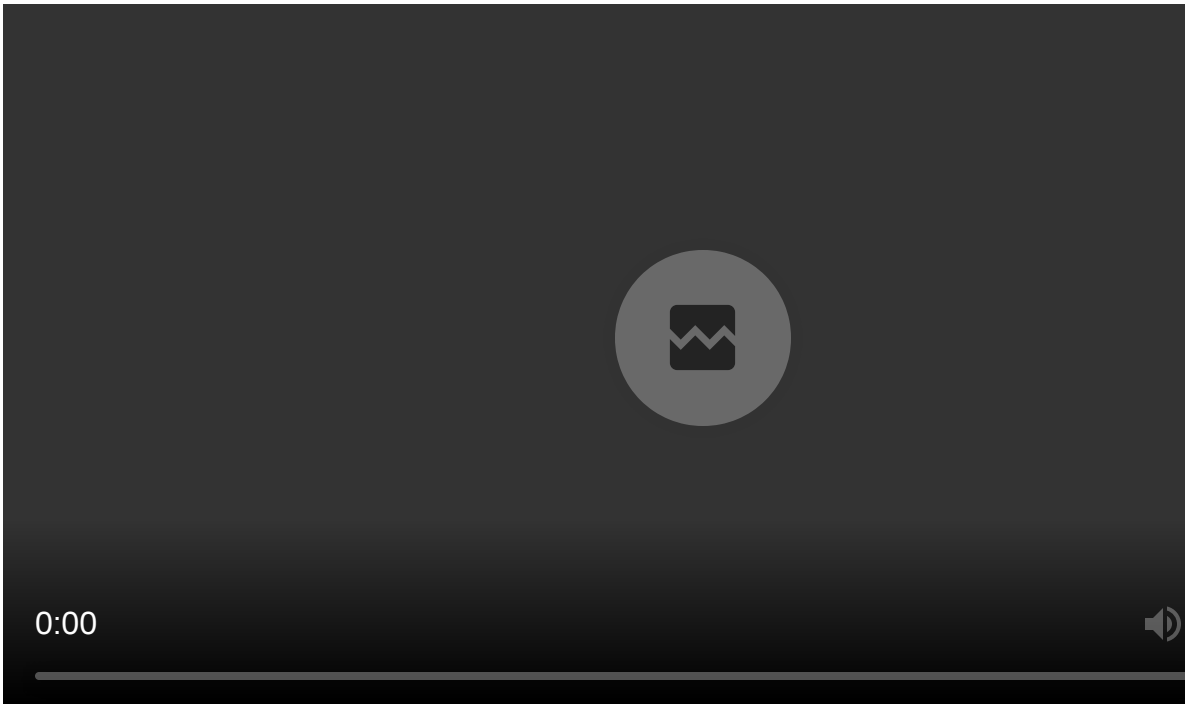
This agrees with the first visualization, since no motion about the y (looking along the axis of ω_2) axis is stable in the figure.

1o.

Consider the following video of the rotation of a free rigid body taken on the ISS. Note that the initial angular velocity vector is very close to a principal moment of inertia. (Hint: Try throwing a coin in the air. Does it display this behaviour? How about a deck of cards? Estimate the principal axes and moments of inertia of a deck of cards, and try spinning it around each of these axes while throwing it into the air. Which axes display simple rotation and which display this flipping behaviour?)

In [126…
```
from IPython.display import Video
Video("Dancing T-handle in zero-g.mp4",width=700)
```

Out[126]:



What can you conclude about the values of the principal moments of inertia, and specifically the moment that it starts off close to?

Here, the principal moments of inertia are all unequal, and the object is spinning close to its intermediate axis. This is what causes the sudden periodic flips.

1p.

Consider the following crude approximation of the T-junction:

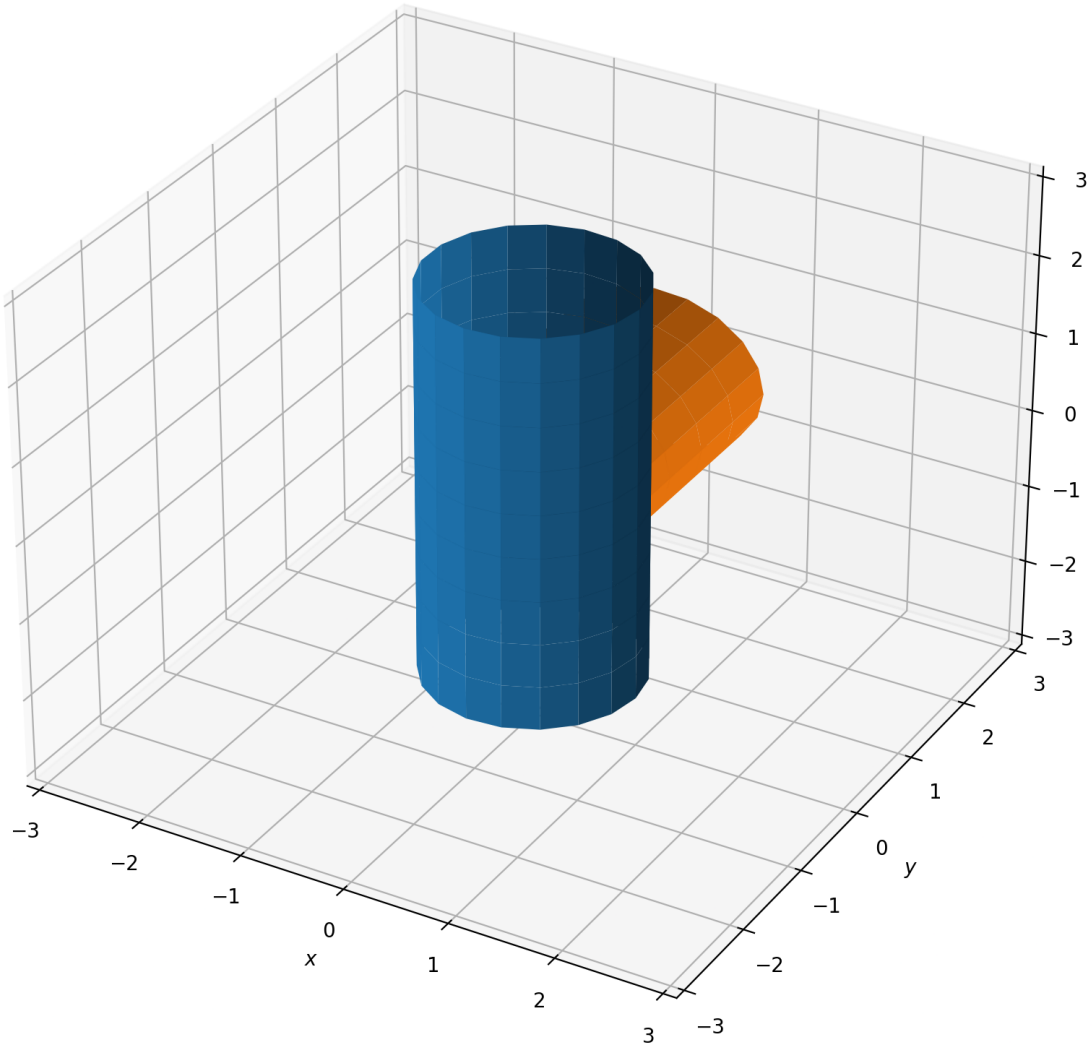
In [127…
```
theta,z = np.linspace(0,2*np.pi,20),np.linspace(0,1,10)
theta,z = np.meshgrid(theta,z)

rod = np.array([np.cos(theta),np.sin(theta),5*z])

#arrays of shape (3,M,N) that store positions of vertices
```

```
rod1 = rod-np.array([0,0,2.5]).reshape((3,1,1))
rod2 = np.array([rod[0,:5],rod[2,:5],rod[1,:5]])

fig = plt.figure(figsize=(8,8))
ax = Axes3D(fig) #create Axes3D object to display the plot
ax.plot_surface(*rod1)
ax.plot_surface(*rod2)
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")
ax.set_zlabel("$z$")
ax.set_xlim(-3,3)
ax.set_ylim(-3,3)
ax.set_zlim(-3,3)
```




```
/tmp/ipykernel_92/676182966.py:11: MatplotlibDeprecationWarning: Axes3D(fi
g) adding itself to the figure is deprecated since 3.4. Pass the keyword ar
gument auto_add_to_figure=False and use fig.add_axes(ax) to suppress this w
arning. The default value of auto_add_to_figure will change to False in mpl
3.5 and True values will no longer work in 3.6.  This is consistent with ot
her Axes classes.
  ax = Axes3D(fig) #create Axes3D object to display the plot
```

Out[127]: (-3.0, 3.0)

In this crude approximation, assume there is a particle of mass $m = 1$ at each vertex. Calculate the moment of inertia tensor $I_{ij} = \sum m(r^2\delta_{ij} - r_i r_j)$ (where the sum is taken over the particles of mass m at position $\mathbf{r}$). By construction, we expect it to be roughly diagonal in this basis. If it is not, diagonalise it (hint: `numpy.linalg.eig`). Note the principal moments.

In [128…
```
vertices = np.concatenate((rod1, rod2), 1) #combine all the axes
I = np.zeros((3,3))
for i in range(3):
    for j in range(3):
        if i == j:
            I[i,j] = np.sum(vertices*vertices) #r^2 delta_{ij} term
        I[i,j] -= np.sum([vertices[i,k] * vertices[j,k] for k in range(len(v

principal_moments, vectors = np.linalg.eig(I)
I = np.diag(principal_moments)
print(I)
```

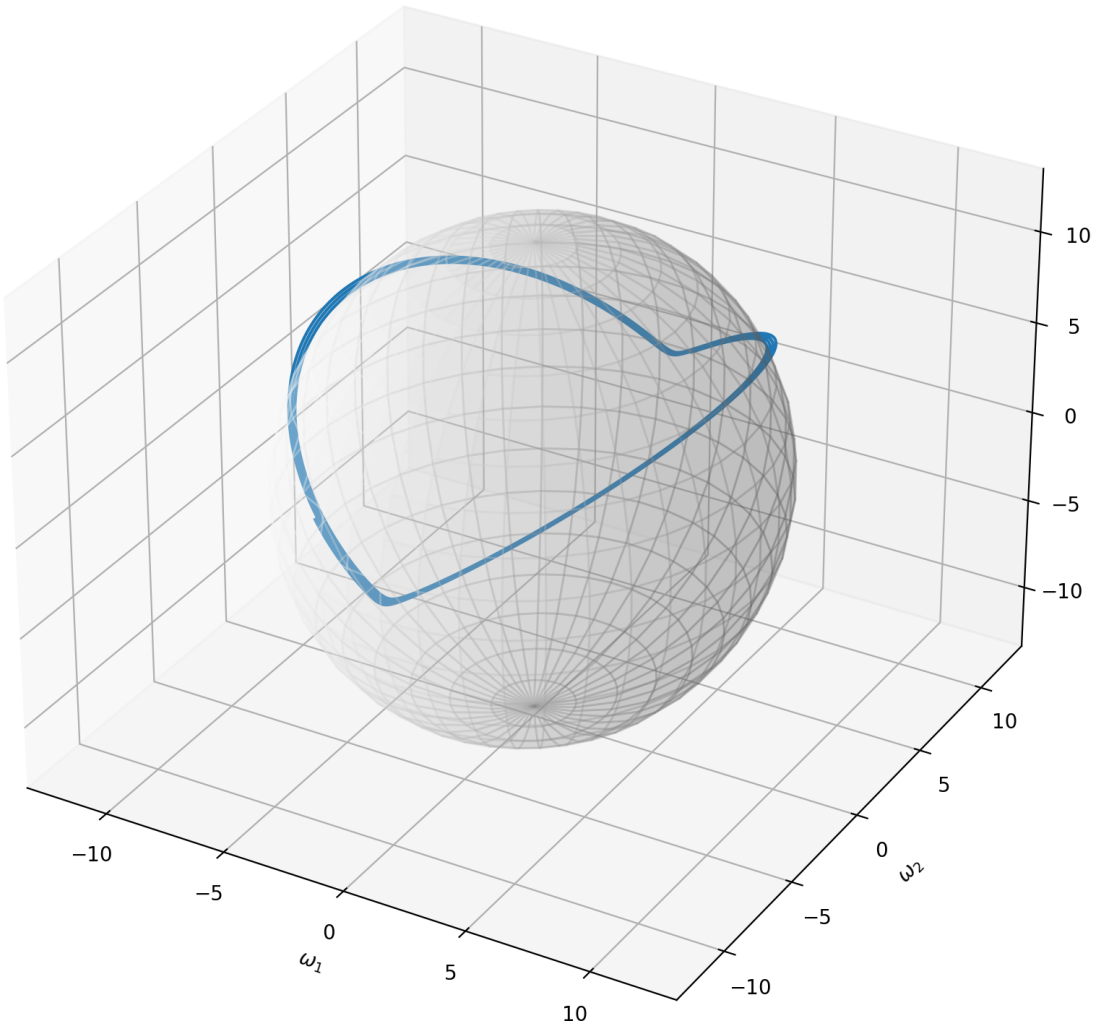
```
[[837.19550301   0.           0.        ]
 [  0.         714.00820069   0.        ]
 [  0.           0.         437.68518519]]
```

1q.

Use `plot_trajectories()` to visualise the trajectory assuming $\boldsymbol{\omega}$ starts very close to the relevant moment. (Remember you are plotting only one trajectory, so need to give it a list containing one list of angular velocity vectors.)

In [129…
```
times = np.linspace(0,20,10000)

traj = np.array([euler(np.array((1,10,1)), *principal_moments, times)])
plot_trajectories(traj, *principal_moments)
```




```
/tmp/ipykernel_92/2209726821.py:25: MatplotlibDeprecationWarning: Axes3D(fi
g) adding itself to the figure is deprecated since 3.4. Pass the keyword ar
gument auto_add_to_figure=False and use fig.add_axes(ax) to suppress this w
arning. The default value of auto_add_to_figure will change to False in mpl
3.5 and True values will no longer work in 3.6.  This is consistent with ot
her Axes classes.
  ax = Axes3D(fig)
```

1r.

In the previous questions, we assumed that objects rotate freely in zero-gravity, e.g. on the ISS. We also assumed that they rotate freely in the presence of gravity when thrown in the air. However, they don't rotate freely in the presence of gravity when, e.g. resting on a table. What exactly does it mean for something to rotate freely, and where does this difference in behaviour come from?

To rotate freely, the definition we use is that there is no net torque on the system. In space (on the ISS), we assume that there is little to no torque on the system, so we consider its

motion to be "free". Similarly when something is thrown in the air, we assume no air resistance, therefore there wouldn't be any torque on its motion either.

In [ ]: