Supplemental Material for "Stabilisation of the swing pattern of an anisotropic simple pendulum"

E. McGlynn[1,2,*], C. Saracut[1], A.A. Cafolla[1,3]

[1]School of Physical Sciences, Dublin City University, Glasnevin, Dublin 9, Ireland

[2]National Centre for Plasma Science and Technology, Dublin City University, Glasnevin, Dublin 9, Ireland

[3]National Centre for Sensor Research, Dublin City University, Glasnevin, Dublin 9, Ireland

Mathematical models and computational details

a) Mathematical models of pendulums with asymmetry in the mount

Linear regime:

In the linear regime, the restoring force (F_R) and acceleration (a) components of the pendulum bob in the presence of anisotropy in the mount may be written simply in the PA frame, as follows[4], [8]:

$$\frac{F_R^x}{m} = \frac{d^2x}{dt^2} = a_x = -\omega_x^2 x \qquad \text{(SM1a)}$$

$$\frac{F_R^y}{m} = \frac{d^2y}{dt^2} = a_y = -\omega_y^2 y \qquad \text{(SM1b)}$$

We assume that the x- and y-axes coincide with the pendulum principal axes. The pendulum mass is denoted by m. The anisotropy in the mount is represented by the different angular frequencies associated with the two orthogonal PA, x and y, i.e. ω_x and ω_y, respectively.

Non-linear regime:

In order to represent the motion for larger swing angles we modify the results of the analysis of Olsson[13]. For the x- and y-components in an isotropic system (spherical pendulum) the exact results are (with some corrections of small typographical errors in Olsson's work, as discussed below):

$$\frac{F_R^x}{m} = \frac{d^2x}{dt^2} = a_x = -\omega^2 x\left(\sqrt{1-\frac{(x^2+y^2)}{l^2}}\right) - \left(\frac{x}{l^2}\{v_x^2+v_y^2\}\right) - \left(\frac{x}{l^4}\left[\{xv_x+yv_y\}^2 \Big/ 1-\frac{(x^2+y^2)}{l^2}\right]\right)$$

(SM2a)

$$\frac{F_R^y}{m} = \frac{d^2y}{dt^2} = a_y = -\omega^2 y\left(\sqrt{1-\frac{(x^2+y^2)}{l^2}}\right) - \left(\frac{y}{l^2}\{v_x^2+v_y^2\}\right) - \left(\frac{y}{l^4}\left[\{xv_x+yv_y\}^2 \Big/ 1-\frac{(x^2+y^2)}{l^2}\right]\right)$$

(SM2b)

The length of the pendulum is denoted by *l* and the x- and y-components of velocity by v_x and v_y, respectively. Retaining linear and cubic terms (i.e. in an approximation assuming small displacements and velocities) one can show analytically that an isotropic pendulum launched with an elliptical path (with major and minor axes of lengths *a* and *b*, respectively, as shown schematically in figure 1 of the main article) exhibits a natural precession with an angular frequency of magnitude $3\omega(ab)/8l^2$ [4], [13].

Unlike the linear case where there is only a single free parameter which can be adjusted to create anisotropy in the system (i.e. the angular frequencies for vibrations parallel to the PA directions), the generalisation of equations (SM2) to the case of a system with anisotropy is dependent on the specific physical origin of the anisotropy of the system, which can vary depending on the design of the mount, the cross-sectional circularity of the string and mounting holes through which the string is threaded, any bevelling of such holes, the homogeneity of the string etc.

To incorporate anisotropy into our model, we follow the criterion that the equations should reduce to (SM1) when the vibration amplitudes are small (i.e. when cubic terms are omitted). The simplest way to do this is to modify equations (SM2) above to replace ω with ω_x and ω_y in equations (2a) and (2b), respectively, as shown in equations (SM3) below (where again we assume that the x- and y-axes coincide with the pendulum PA). These are

identical to equations (1) of the main article. We believe this approach appropriately captures the essential physics of the situation by absorbing the various possible sources of system anisotropy into the parameters ω_x and ω_y.

$$a_x = -\omega_x^2 x\left(\sqrt{1-\frac{(x^2+y^2)}{l^2}}\right) - \left(\frac{x}{l^2}\left\{v_x^2+v_y^2\right\}\right) - \left(\frac{x}{l^4}\left[\left\{xv_x+yv_y\right\}^2 \Big/ 1-\frac{(x^2+y^2)}{l^2}\right]\right)$$

(SM3a)

$$a_y = -\omega_y^2 y\left(\sqrt{1-\frac{(x^2+y^2)}{l^2}}\right) - \left(\frac{y}{l^2}\left\{v_x^2+v_y^2\right\}\right) - \left(\frac{y}{l^4}\left[\left\{xv_x+yv_y\right\}^2 \Big/ 1-\frac{(x^2+y^2)}{l^2}\right]\right)$$

(SM3b)

We utilise initial conditions with relatively small displacement and velocity values for the computational results shown in section III which allow comparison with the analytical result for the natural precession which arises from an analytical solution of the approximate equations (SM3), i.e. $3\omega(ab)/8l^2$. We note also that the initial conditions we use are chosen to have an appreciable elliptical swing pattern, allowing us to demonstrate the effects clearly; this is opposite to the situation generally sought in the operation of a Foucault pendulum, where motion in a single plane is desired to eliminate the natural precession. We also follow Pippard's approach[4], assuming the anisotropy is relatively small, and define $\omega_x^2 = \omega^2 + \delta$ and $\omega_y^2 = \omega^2 - \delta$, where ω^2 is the mean of the squared frequencies for pendulum motion along the two PA directions.

b) Modelling the effect of the rotation of the mount

The crux of the computational approach is to utilise the PA frame solely for the calculation of the acceleration components using either equations (SM1) or (SM3), since the calculation of the acceleration is especially simple in the PA frame. At any instant in time the coordinates of the position and velocity vectors in the inertial laboratory frame are known, and then transformed by the rotations described in equations (SM4) and (SM5) into the PA

frame. The a_x and a_y components of the acceleration vector are then calculated using equations (SM1) or (SM3), and these are transformed back into the inertial laboratory frame, and then used in the computational differential equation solver to increment the motion and calculate the new position and velocity vectors in the laboratory frame.

In all the equations of motion shown no non-inertial (i.e. centrifugal and/or Coriolis) terms are introduced, despite the fact that we refer to a situation where the pendulum mount is rotating at a constant angular frequency. The reason is that all the calculations using the computational differential equation solver are done in the inertial frame of the laboratory. We emphasise the fact that the rotations into the PA frame of the rotating mount are only done to allow relatively easy calculation of the force components at that instant using equations (SM1) or (SM3).

The components of the position and velocity vectors in the PA frame are then calculated from those in the laboratory frame using a rotation of the type shown in equation (SM4) below, for a general vector (*P*) in the two frames:

$$\begin{pmatrix} P_x \\ P_y \end{pmatrix} = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} P_x^{lab} \\ P_y^{lab} \end{pmatrix} \qquad \text{(SM4)}$$

At each time, *t*, the angular rotation of the PA frame relative to the laboratory is calculated using $\theta = \Omega.t$, where Ω is the angular velocity of rotation of the system.

These components are then used to calculate the acceleration vector components relative to the PA frame using equations (SM1) or (SM3). These acceleration vector components are then transformed back to the laboratory frame by the inverse rotation shown in equation (SM5) below, for a general vector (*P*) in the two frames:

$$\begin{pmatrix} P_x^{lab} \\ P_y^{lab} \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} P_x \\ P_y \end{pmatrix} \qquad \text{(SM5)}$$

This is shown schematically in figure SI-1, where we equate the angle θ to $\Omega.t$, in anticipation of a constant rotation of the PA frame relative to the laboratory frame at a fixed angular velocity, Ω.

The code uses the rotations from equations (SM4) and (SM5) along with equations (SM1) or (SM3) at each time instant. One could also derive the explicit equations for the a_x and a_y components in the laboratory frame and use those in the differential equation solver. In the

non-linear regime represented by equations (SM3) these expressions are necessarily quite long, but they are considerably shorter in the linear regime and are included below for illustration. With the definition that that $\theta = \Omega.t$ the connection between the components in the PA and laboratory frames is explicitly time dependent due to the mount rotation.

These acceleration (and other vector) components in the laboratory frame are then used in the differential equation solver. The full code used for both the linear and non-linear regimes, with commenting, is provided in the supplemental material.

In our initial approach we utilised a simple 4th order Runge-Kutta (RK4) approach, which was programmed *ab-initio* and this was compared to existing Python functions to check the accuracy of the solutions. An adaptive step-size was not used, and comparison with existing Python functions such as *odeint* show that a fixed step size is sufficient and produces identical results to the existing Python functions. One advantage of the *ab-initio* code in comparison to the existing Python *odeint* function was seen when we explored the effects of random variations in the orientation of the PA, as described below. The main difference between the *ab-initio* and existing Python functions is the significantly reduced running time required by the existing Python functions, typically a reduction by a factor of around 10 or more compared to the *ab-initio* code. The codes used for all the investigations are provided below, with commenting.

Explicit equations for the a_x and a_y components in the laboratory frame in the linear regime:

The acceleration components in the PA frame can be written as:

$$\begin{pmatrix} a_x \\ a_y \end{pmatrix} = \begin{pmatrix} -\omega_x^2 x \\ -\omega_y^2 y \end{pmatrix} \qquad \text{(SM6)}$$

And since

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x_{lab} \\ y_{lab} \end{pmatrix} \quad \text{(SM7)}$$

We get:

$$\begin{pmatrix} a_x \\ a_y \end{pmatrix} = \begin{pmatrix} -\omega_x^2 \cos(\theta) . x_{lab} - \omega_x^2 sin(\theta) . y_{lab} \\ +\omega_y^2 \sin(\theta) . x_{lab} - \omega_y^2 cos(\theta) . y_{lab} \end{pmatrix} \qquad \text{(SM8)}$$

In the laboratory frame the acceleration components $a^{lab}{}_x$ and $a^{lab}{}_y$ are related to those in the PA frame by the reverse of the 2D rotation, shown in equation (SM9):

$$\begin{pmatrix} a_x^{lab} \\ a_y^{lab} \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} a_x \\ a_y \end{pmatrix} \qquad \text{(SM9)}$$

Calculating the individual components of the acceleration in the laboratory frame yields:

$$\begin{pmatrix} a_x^{lab} \\ a_y^{lab} \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} -\omega_x^2 \cos(\theta) \,.\, x_{lab} - \omega_x^2 sin(\theta) . y_{lab} \\ +\omega_y^2 \sin(\theta) \,.\, x_{lab} - \omega_y^2 cos(\theta) . y_{lab} \end{pmatrix} \qquad \text{(SM10)}$$

$$\begin{pmatrix} a_x^{lab} \\ a_y^{lab} \end{pmatrix} =$$

$$\begin{pmatrix} -x_{lab}(\omega_x^2 cos^2(\theta) + \omega_y^2 sin^2(\theta)) - y_{lab}(\omega_x^2 \sin(\theta)\cos(\theta) - \omega_y^2 \sin(\theta)\cos(\theta)) \\ -x_{lab}(\omega_x^2 \sin(\theta)\cos(\theta) - \omega_y^2 \sin(\theta)\cos(\theta)) - y_{lab}(\omega_x^2 sin^2(\theta) + (\omega_y^2 cos^2(\theta)) \end{pmatrix}$$

(SM11)

And finally:

$$\begin{pmatrix} a_x^{lab} \\ a_y^{lab} \end{pmatrix} = \begin{pmatrix} f(x_{lab}, y_{lab}, t) \\ g(x_{lab}, y_{lab}, t) \end{pmatrix} \qquad \text{(SM12)}$$

With:

$$f(x, y, t) = -x_{lab}(\omega_x^2 cos^2(\theta) + \omega_y^2 sin^2(\theta)) - y_{lab}(\omega_x^2 \sin(\theta)\cos(\theta) - \omega_y^2 \sin(\theta)\cos(\theta))$$

(S13a)

$$g(x, y, t) = -x_{lab}(\omega_x^2 \sin(\theta)\cos(\theta) - \omega_y^2 \sin(\theta)\cos(\theta)) - y_{lab}(\omega_x^2 sin^2(\theta) + (\omega_y^2 cos^2(\theta))$$

(S13b)

Typographical errors in Olsson's paper:

There is a typographical error in Olsson's paper ([14]), specifically in his equation (12) for $\ddot{x}$ (written as a_x in this work), and hence also relevant to the "similar equation" he mentions for $\ddot{y}$ (written as a_y in this work). The last term should have l^4 in the denominator (and not l^2 as in the paper). This can be verified by comparison with equation (11) of Olsson's paper and following his described method of derivation, and the error is also evident from basic dimensional analysis considerations. This term is corrected in equations (1) of the main article,

and (SM2) and (SM3) in the supplemental material. If this error is not corrected the total energy checks mentioned in the Results and Analysis of the main article (section III) yield results where energy is not conserved.

We also note in passing two other typographical errors in Olsson's paper. Firstly, the second term on the left-hand side of his equation (5b) should be $-\dot{\phi}^2 sin\theta cos\theta$ (and not $-\dot{\theta}^2 sin\theta cos\theta$ as in the paper). Secondly, in the last term of his equation (10), the term a^2 in the square root should be l^2. The first of these two errors is relevant for the derivation leading to his equation (10).

Further details of the calculations for the case of random variations in PA orientation:

The random variations in the PA orientation are accomplished by giving the angle θ in equations (SM4) and (SM5) a random value between 0 and 2π at each time step (this angle is assumed to be the same at each of the intermediate "internal" stages in the Runge-Kutta process).

The Python odeint function did not function in a stable manner for this random mount reorientation, unlike the directly programmed RK4 code, which may be due to the nature of the odeint solver using the LSODA solvers from the Fortran ODEPACK. These use Adams methods (predictor-corrector) and Backward Differentiation Formula (BDF) methods, which are unsuitable for the random mount reorientation simulation, whereas the simple directly programmed RK4 code is well suited for this type of simulation. For further details see:

K. Radhakrishnan and A. C. Hindmarsh, *NASA Reference Publication 1327 Description and Use of LSODE, the Livermore Solver for Ordinary Differential Equations*,

https://computing.llnl.gov/sites/default/files/ODEPACK_pub2_u113855.pdf

Initial checks of the code:

A number of initial checks were made of the code, including setting the anisotropy and rotation of the system (parametrised by δ and Ω, respectively) equal to zero and running simulations of pendulum motion with various initial conditions and comparing the calculated frequency with that expected from the normal pendulum equation (in the linear

regime), as well as checking the conservation of total energy and z component of angular momentum over time periods of up to an hour in the non-linear regime. We have also compared the existing Python functions to the *ab-initio* RK4 code over time periods of up to 60 minutes, both with and without anisotropy introduced. All of these checks verify the physically correct operation of the various codes, the stability of the simulation over periods up to an hour (in terms of conservation of total energy and angular momentum for the isotropic mount case) and that identical results are obtained for the existing Python functions and the *ab-initio* RK4 code. A selection of the results of these checks are shown in the supplemental material (figure SI-2) which show (i) that the angular frequency of the calculated motion in the linear regime for an isotropic pendulum (δ = 0 rad^2/s^2) very closely matches the expected value for the simple pendulum ($\sqrt{g/l}$), (ii) that the energy and z-component of the angular momentum are both conserved to within 1 part in 10^{-6} using the Python *odeint* code (and to even greater accuracy using the *ab-initio* RK4 code, but with a significant increase in run time), and finally (iii) that the trajectories calculated using these two codes are indistinguishable. The trajectory data are shown for a 100 s evolution period after pendulum launch to enable easy comparison.

Supporting information figures:

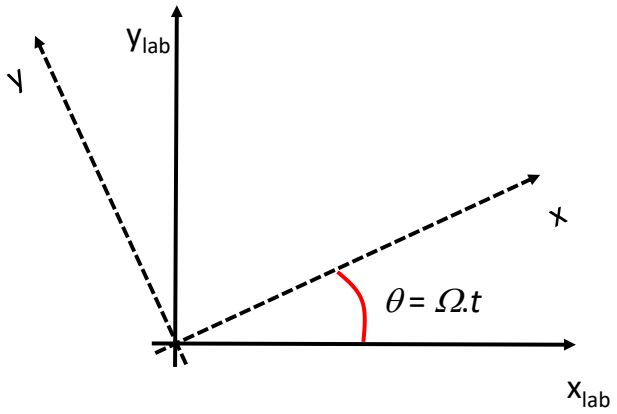

Figure SI-1: Schematic representation of the orientation of the PA frame (r = (x, y)) to the fixed laboratory axis frame (r_{lab} = (x_{lab},y_{lab})), related via a rotation through an angle θ, which is time-dependent with $\theta = \Omega.t$.

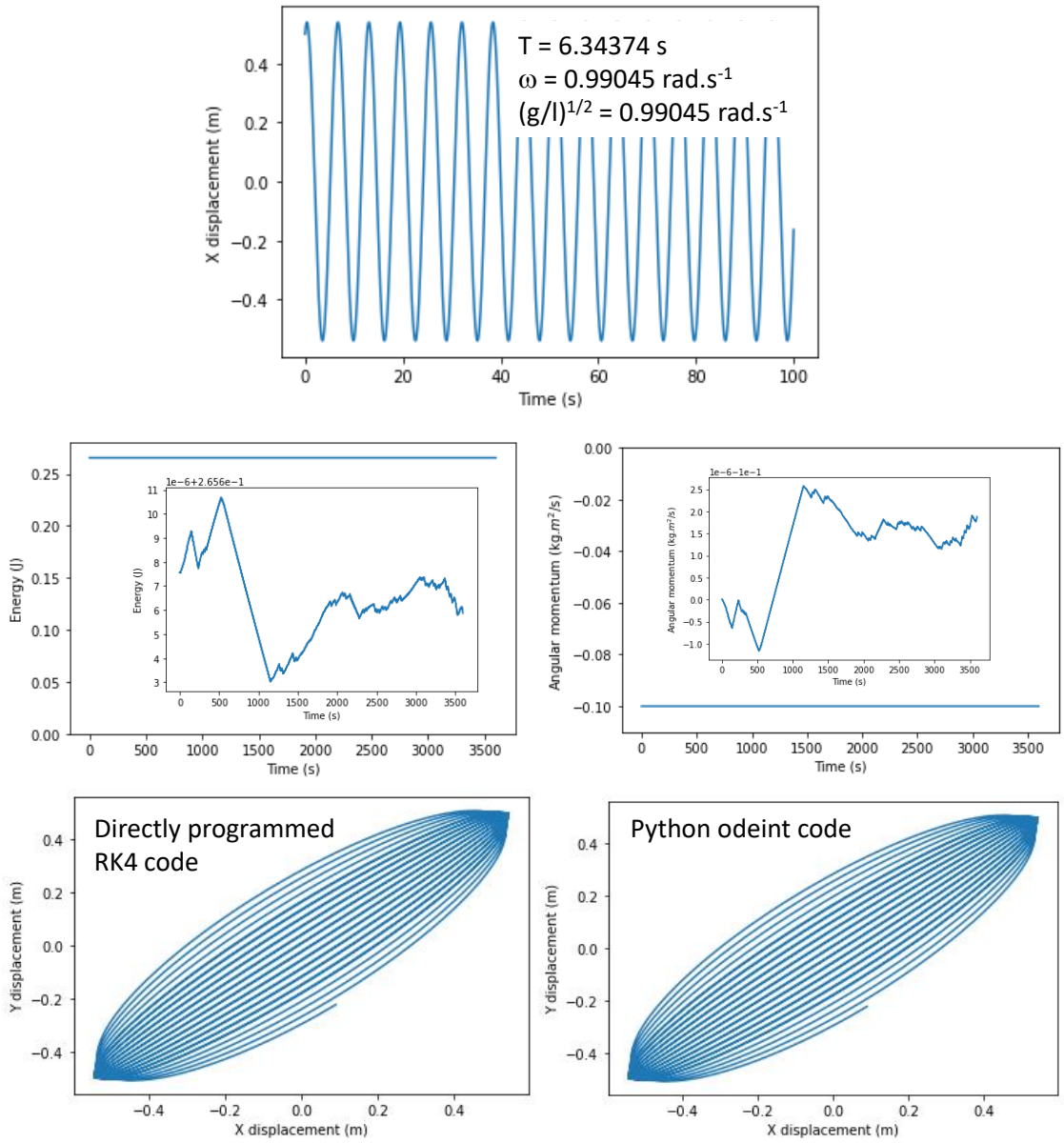

Figure SI-2: Top panel shows x displacement as a function of time in the linear regime (with $\delta = 0$) using odeint, and comparison of the value of ω extracted from this plot (via a least squares fit) and $\sqrt{g/l}$. The middle panel shows the behaviour of the total energy (left) and z-component of angular momentum (right), with mass set to 1 kg, in the non-linear regime (with $\delta = 0$) using odeint. The bottom panel shows the calculated trajectories using the directly programmed RK4 code (left) and the Python odeint function (right) for 100 s, in the non-linear regime (with $\delta = 0.01$). For all these checks the pendulum length was set to 10 m, with initial conditions of x = 0.5 m, v_x = 0.2 m/s, y = 0.5 m, and v_y = 0.0 m/s. The mount rotation frequency, Ω, was zero in all these cases.

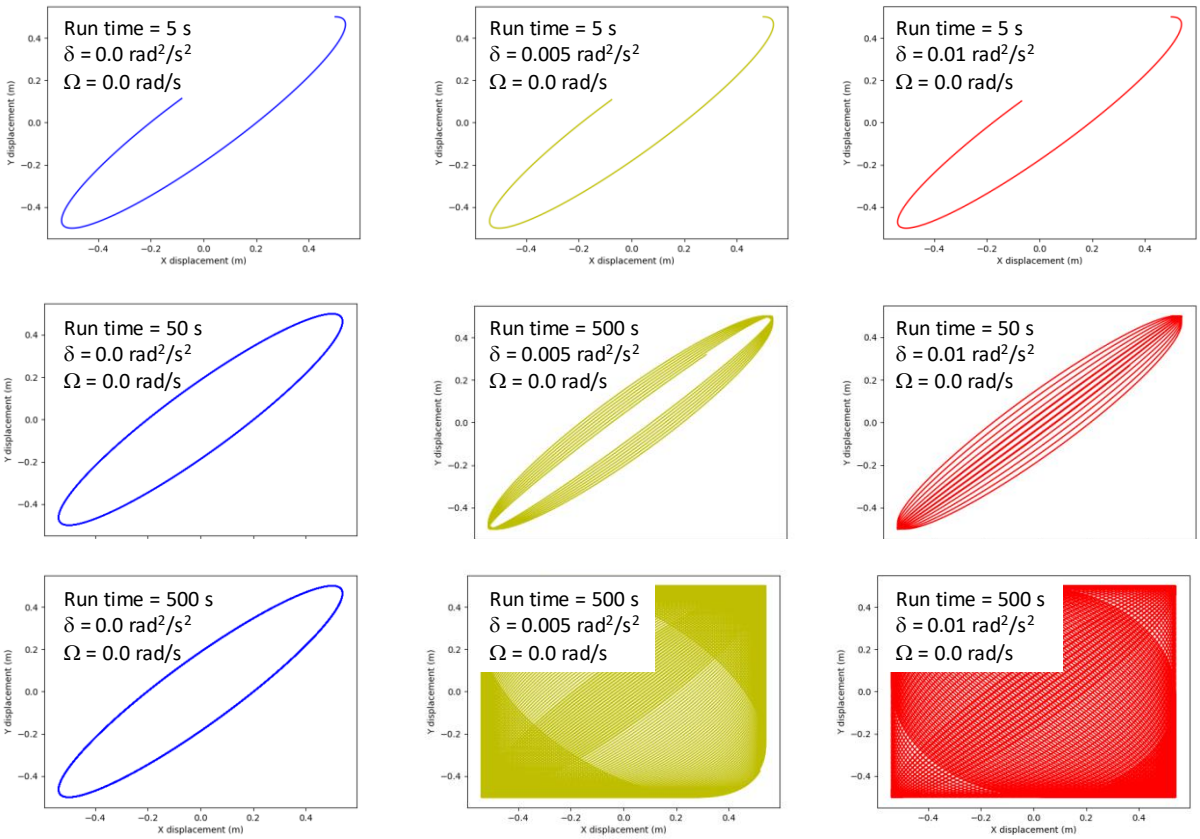

Figure SI-3: Trajectories of 2D pendulum (x displacement versus y displacement) in linear regime calculated using odeint. Left panel shows isotropic pendulum ($\delta = 0$ rad^2/s^2) with stationary mount ($\Omega = 0$ rad/s) at times 5s (top), 50 s (middle) and 500 s (bottom). The middle panel shows an anisotropic pendulum ($\delta = 0.005$ rad^2/s^2) with stationary mount ($\Omega = 0$ rad/s) at times 5s (top), 50 s (middle) and 500 s (bottom). The right panel shows an anisotropic pendulum ($\delta = 0.01$ rad^2/s^2) with stationary mount ($\Omega = 0$ rad/s) at times 5s (top), 50 s (middle) and 500 s (bottom). For all these calculations the pendulum length was set to 10 m, with initial conditions of $x = 0.5$ m, $v_x = 0.2$ m/s, $y = 0.5$ m, and $v_y = 0.0$ m/s (i.e. the same conditions for the data in figure 2 of the paper).

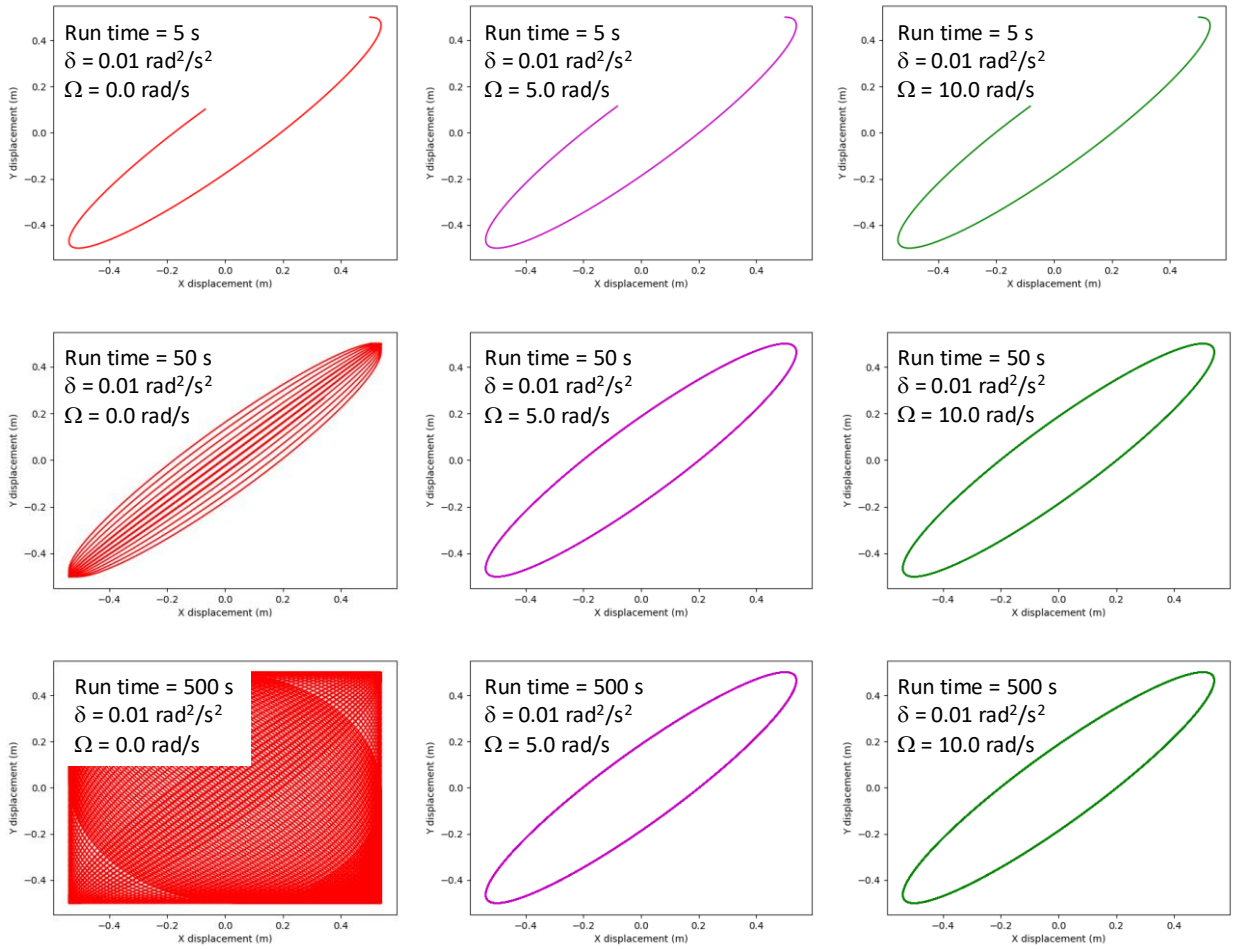

Figure SI-4: Trajectories of 2D pendulum (x displacement versus y displacement) in linear regime calculated using odeint. Left panel shows an anisotropic pendulum ($\delta = 0.01$ rad^2/s^2) with stationary mount ($\Omega = 0$ rad/s) at times 5s (top), 50 s (middle) and 500 s (bottom). The middle panel shows an anisotropic pendulum ($\delta = 0.01$ rad^2/s^2) with rotating mount ($\Omega = 5$ rad/s) at times 5s (top), 50 s (middle) and 500 s (bottom). The right panel shows an anisotropic pendulum ($\delta = 0.01$ rad^2/s^2) with rotating mount ($\Omega = 10$ rad/s) at times 5s (top), 50 s (middle) and 500 s (bottom). For all these calculations the pendulum length was set to 10 m, with initial conditions of x = 0.5 m, v_x = 0.2 m/s, y = 0.5 m, and v_y = 0.0 m/s (i.e. the same conditions for the data in figure 2 of the paper).

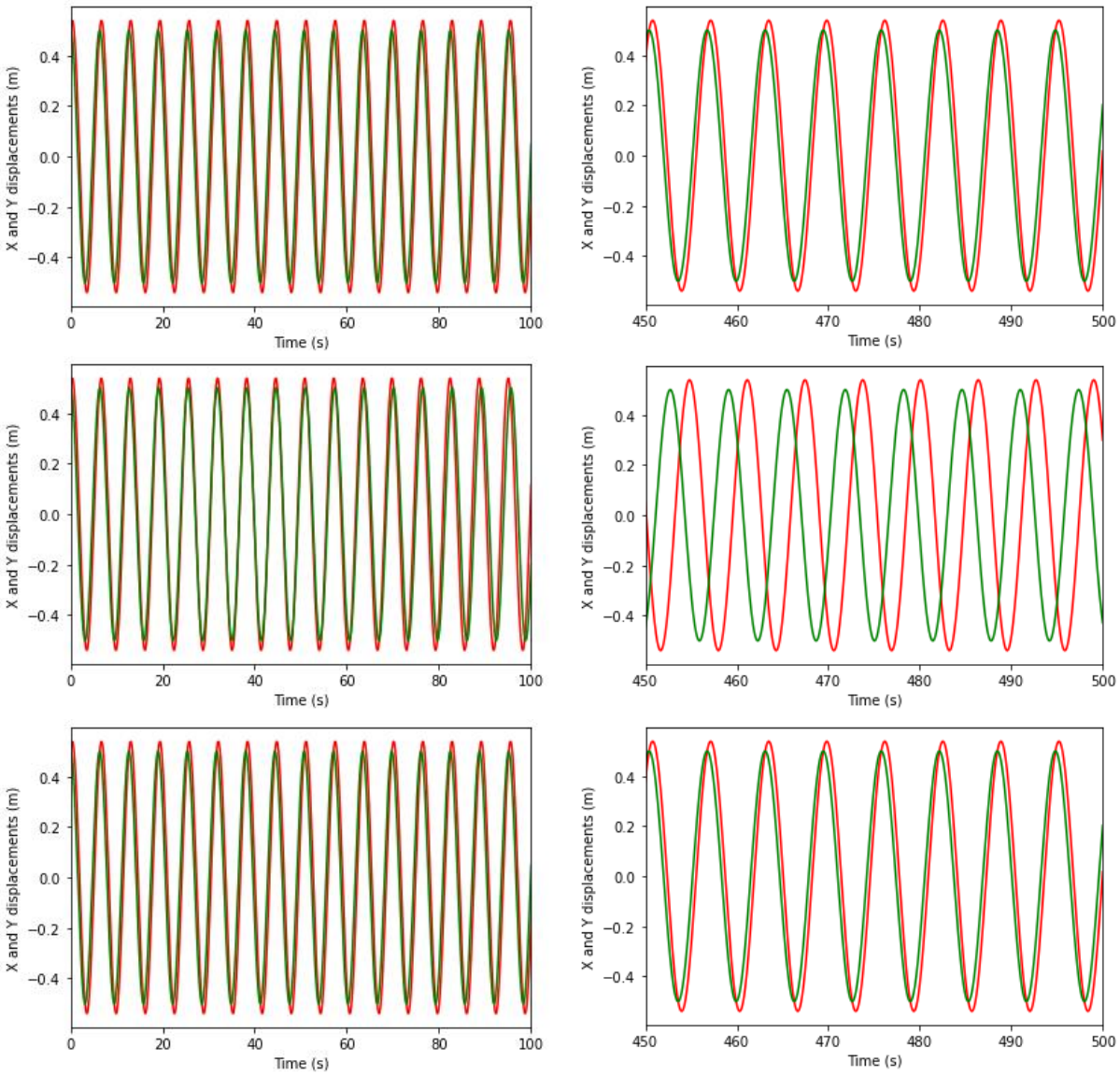

Figure SI-5: X displacement in red and Y displacement in green, versus time, of 2D pendulum from 0 s to 100s, left panel, and from 450 to 500 s, right panel) in linear regime calculated using odeint. Top panel shows isotropic pendulum ($\delta = 0$ rad^2/s^2) with stationary mount ($\Omega = 0$ rad/s), middle panel shows anisotropic pendulum ($\delta = 0.01$ rad^2/s^2) with stationary mount ($\Omega = 0$ rad/s) and bottom panel shows anisotropic pendulum ($\delta = 0.01$ rad^2/s^2) with rotating mount ($\Omega = 10$ rad/s). Pendulum length and initial conditions are the same as for data in figures 2 and 3 of the main text.

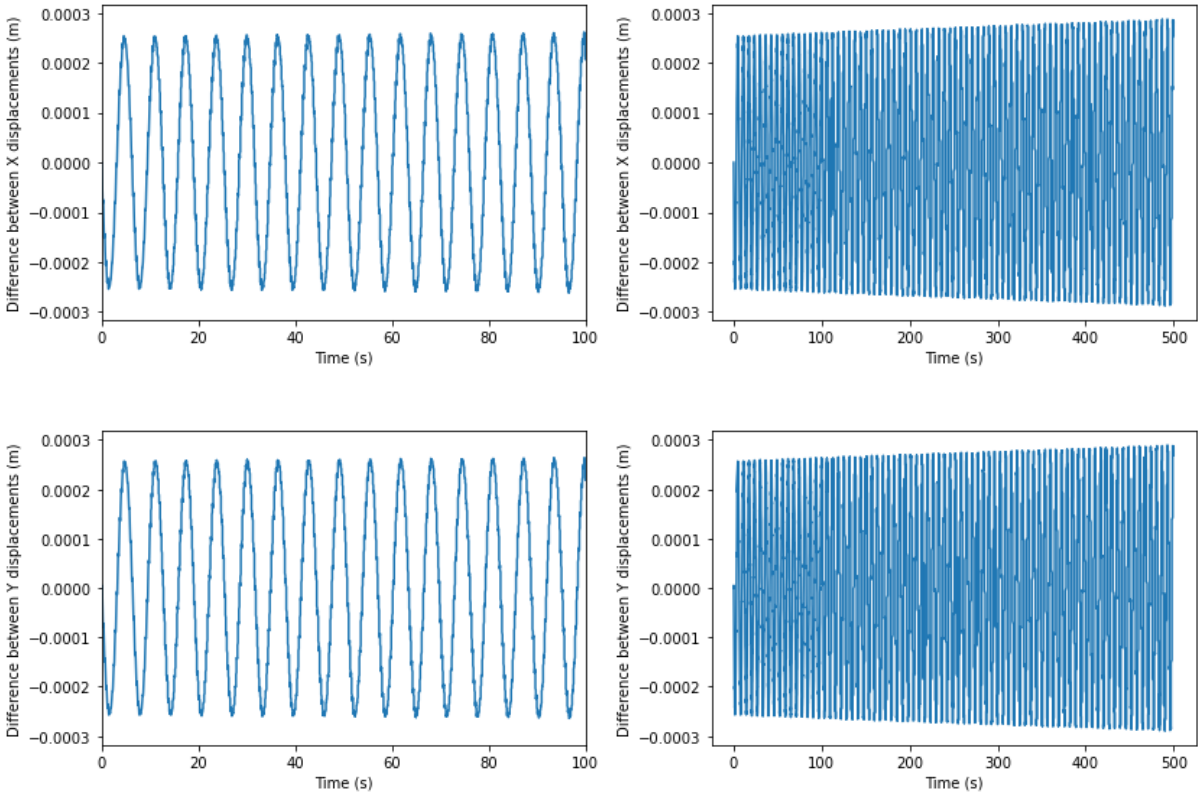

Figure SI-6: Top row: calculated difference in X displacements between isotropic pendulum (δ = 0 rad^2/s^2) with stationary mount (Ω = 0 rad/s) and anisotropic pendulum (δ = 0.01 rad^2/s^2) with rotating mount (Ω = 10 rad/s). Bottom row: calculated difference in Y displacements between isotropic pendulum (δ = 0 rad^2/s^2) with stationary mount (Ω = 0 rad/s) and anisotropic pendulum (δ = 0.01 rad^2/s^2) with rotating mount (Ω = 10 rad/s). All calculations are for a 2D pendulum in linear regime using odeint. Left panels show behaviour from 0 s to 100s, and right panels from 0 s to 500s. Pendulum length and initial conditions are the same as for data in figures 2 and 3 of the main text.

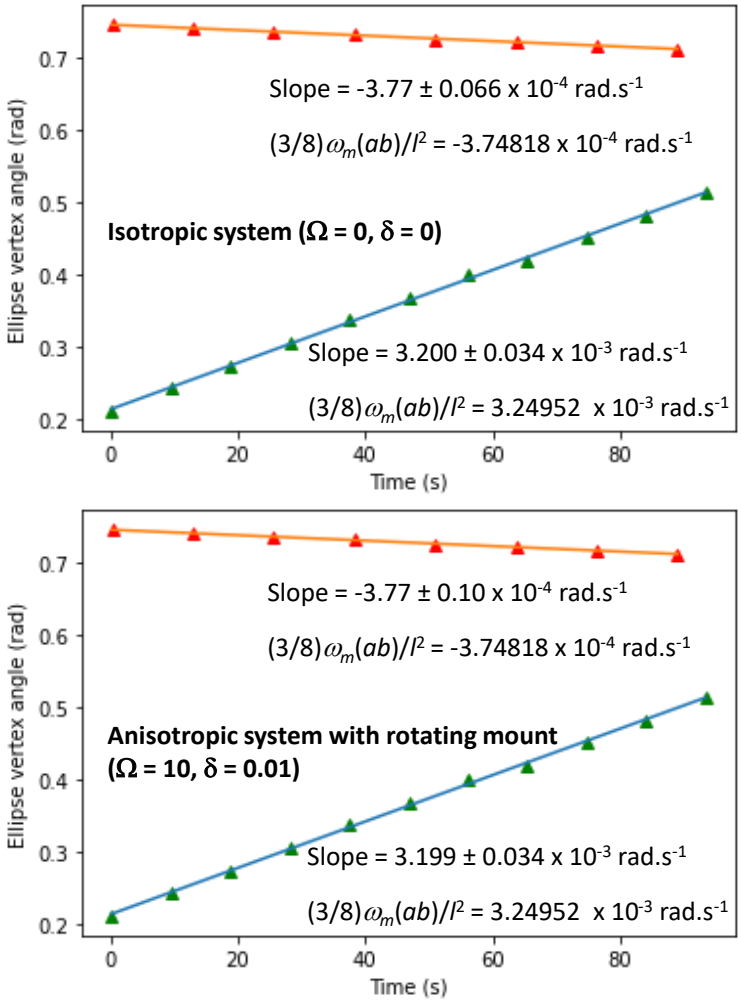

Figure SI-7: Precession of the vertices of the motion as a function of time for both the isotopic mount (top) and rotating anisotropic mount (bottom) pendulums. The data shown with red triangles is for a system with initial conditions of x = 0.5 m, v_x = 0.2 m/s, y = 0.5 m, and v_y = 0.0 m/s, and l = 10.0 m, while the data shown with green triangles is for a system with initial conditions of x = 0.2 m, v_x = 0.1 m/s, y = 0.0 m, and v_y = 0.25 m/s, and l = 2.4 m. The solid lines are least squares fits and the slopes and fit uncertainties are shown in the legends.

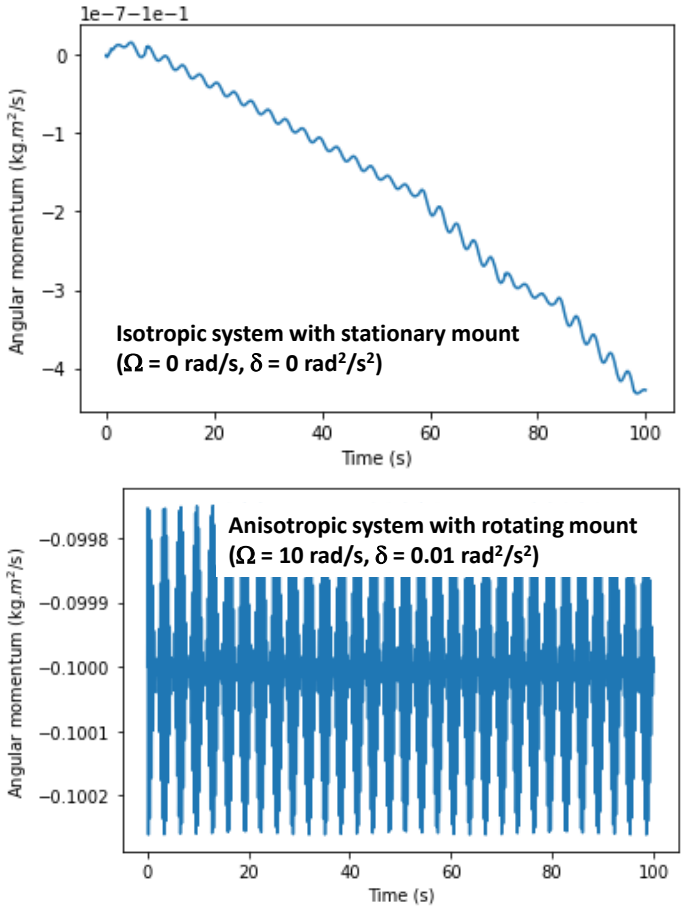

Figure SI-8: The z-component of angular momentum, with mass set to 1 kg, in the non-linear regime for (top) a stationary isotropic mount (with $\delta = \Omega = 0$) and (bottom) a rotating anisotropic mount (with $\delta = 0.01$ rad²/s² and $\Omega = 10$ rad/s). For these calculations the pendulum length was set to 10 m, with initial conditions of x = 0.5 m, v_x = 0.2 m/s, y = 0.5 m, and v_y = 0.0 m/s.

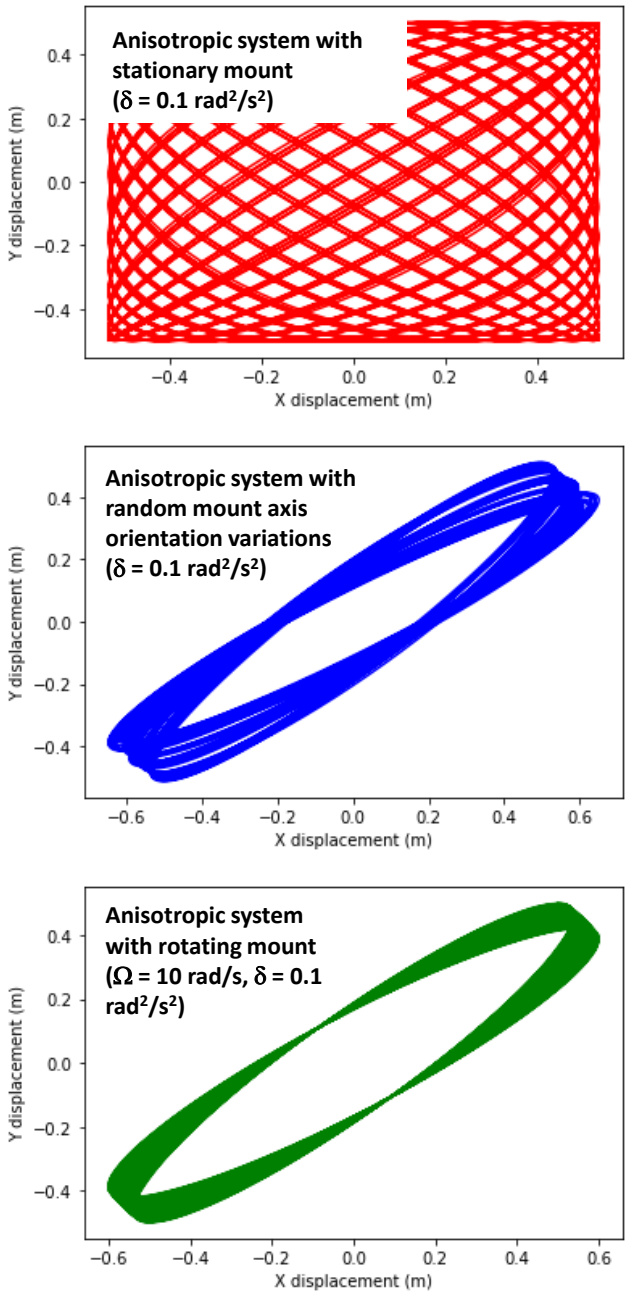

Figure SI-9: Trajectories of 2D pendulum in non-linear regime calculated over 400 s using directly programmed RK4 code. Top image shows a stationary anisotropic mount ($\delta = 0.1$ rad^2/s^2), the middle image shows an anisotropic mount with random mount axis orientation variations ($\delta = 0.1$ rad^2/s^2), and the bottom image shows a steadily rotating anisotropic mount ($\delta = 0.1$ rad^2/s^2 and $\Omega = 10$ rad/s). For all these calculations the pendulum length was set to 10 m, with initial conditions of x = 0.5 m, v_x = 0.2 m/s, y = 0.5 m, and v_y = 0.0 m/s.

Python codes:

1. HM_RK4_rot (directly programmed 4^{th} order Runge-Kutta with steady rotation of mount)
2. HM_RK4_ran (directly programmed 4^{th} order Runge-Kutta with random random mount axis orientation variations)
3. Py_odeint_rot (in-built Python function odeint ordinary differential equation solver)

HM_RK4_rot

```
import time
import numpy as np
import scipy as sp
import random as rd
import matplotlib.pyplot as plt

start_t = time.time()

# Define parameters
le = 10.0 #pendulum length

omega = 10.0 # mount spin angular velocity
h=0.01 # time step
t = np.arange(0,100.0,h) #This is the time range # total time of simulation in seconds

x0 = [0.5,0.2] #Initial x and vx values in laboratory frame
xdata=np.zeros((len(t),2)) #Setting up array for x and vx data

y0 = [0.5,0.0] #Initial y and vy values in laboratory frame
ydata=np.zeros((len(t),2)) #Setting up array for x and vx data

#Other parameters
g = 9.81 # acceleration due to gravity
w0m=np.sqrt(g/le) # pendulum mean angular frequency
w02m=w0m**2 # square of pendulum mean angular frequency
delta = 0.01 # deviation from symmetry of w02m

#-----------------------------------------------------------------------------------------------------
```

```
#Defining restoring force in laboratory frame depending on angle of pendulum mount frame with respect to laboratory frame (theta = omega.t)
def acc(xf,yf,vxf,vyf,w02mf,lef,deltaf,omegaf,tf):

    r=np.matrix([[xf],[yf]])
    v=np.matrix([[vxf],[vyf]])

    rot1=np.matrix([[np.cos(omegaf*tf),np.sin(omegaf*tf)],[-np.sin(omegaf*tf),np.cos(omegaf*tf)]])

    rdash=rot1*r
    xm=float(rdash[0])
    ym=float(rdash[1])

    vdash=rot1*v
    vxm=float(vdash[0])
    vym=float(vdash[1])

    #Defining acceleration in frame of pendulum mount, either in small angle approximation or larger angle approximation (remove and replace "#" symbols as required)
    #axm=-((w02mf+deltaf)*xm)
    axm=-((w02mf+deltaf)*xm*((1-(((xm**2)+(ym**2))/(lef**2)))**(0.5)))-(xm/(lef**2))*((vxm**2)+(vym**2))-(xm/(lef**4))*(((xm*vxm)+(ym*vym))**2)/(1-(((xm**2)+(ym**2))/(lef**2)))

    #aym=-((w02mf-deltaf)*ym)
    aym=-((w02mf-deltaf)*ym*((1-(((xm**2)+(ym**2))/(lef**2)))**(0.5)))-(ym/(lef**2))*((vxm**2)+(vym**2))-(ym/(lef**4))*(((xm*vxm)+(ym*vym))**2)/(1-(((xm**2)+(ym**2))/(lef**2)))

    adash=np.matrix([[axm],[aym]])

    rot2=np.matrix([[np.cos(omegaf*tf),-np.sin(omegaf*tf)],[np.sin(omegaf*tf),np.cos(omegaf*tf)]])

    a=rot2*adash
```

```
    return a

#Defining x acceleration in laboratory frame depending on angle of pendulum mount frame with respect to laboratory frame (theta = omega.t)
def xacc(xf,yf,vxf,vyf,w02mf,lef,deltaf,omegaf,tf):

    restacc=acc(xf,yf,vxf,vyf,w02mf,lef,deltaf,omegaf,tf)

    return float(restacc[0])

#Defining y acceleration in laboratory frame depending on angle of pendulum mount frame with respect to laboratory frame (theta = omega.t)
def yacc(xf,yf,vxf, vyf,w02mf,lef,deltaf,omegaf,tf):

    restacc=acc(xf,yf,vxf,vyf,w02mf,lef,deltaf,omegaf,tf)

    return float(restacc[1])

count = 0
xdata[0,:]=x0 #Setting initial x and vx conditions into x and vx data array
ydata[0,:]=y0 #Setting initial y and vy conditions into y and vy data array

n = len(t)
looparray = np.arange(1,n,1)

#4th order Runge-Kutta for 2D pendulum, assuming the restoring forces depend on both x and y in laboratory frame (but not in pendulum mount frame)
and this varies with time. The x and y variables are treated the same in the intermediate Runge-Kutta steps.
for count in looparray:
    Rx1=xdata[count-1,1]*h
    Sx1=(xacc(xdata[count-1,0],ydata[count-1,0],xdata[count-1,1],ydata[count-1,1],w02m,le,delta,omega,t[count-1]))*h

    Ry1=ydata[count-1,1]*h
```

```
Sy1=(yacc(xdata[count-1,0],ydata[count-1,0],xdata[count-1,1],ydata[count-1,1],w02m,le,delta,omega,t[count-1]))*h

Rx2=(xdata[count-1,1]+(0.5*Sx1))*h
Sx2=(xacc(xdata[count-1,0]+(0.5*Rx1),ydata[count-1,0]+(0.5*Ry1),xdata[count-1,1]+(0.5*Sx1),ydata[count-
1,1]+(0.5*Sy1),w02m,le,delta,omega,t[count-1]+(0.5*h)))*h

Ry2=(ydata[count-1,1]+(0.5*Sy1))*h
Sy2=(yacc(xdata[count-1,0]+(0.5*Rx1),ydata[count-1,0]+(0.5*Ry1),xdata[count-1,1]+(0.5*Sx1),ydata[count-
1,1]+(0.5*Sy1),w02m,le,delta,omega,t[count-1]+(0.5*h)))*h

Rx3=(xdata[count-1,1]+(0.5*Sx2))*h
Sx3=(xacc(xdata[count-1,0]+(0.5*Rx2),ydata[count-1,0]+(0.5*Ry2),xdata[count-1,1]+(0.5*Sx2),ydata[count-
1,1]+(0.5*Sy2),w02m,le,delta,omega,t[count-1]+(0.5*h)))*h

Ry3=(ydata[count-1,1]+(0.5*Sy2))*h
Sy3=(yacc(xdata[count-1,0]+(0.5*Rx2),ydata[count-1,0]+(0.5*Ry2),xdata[count-1,1]+(0.5*Sx2),ydata[count-
1,1]+(0.5*Sy2),w02m,le,delta,omega,t[count-1]+(0.5*h)))*h

Rx4=(xdata[count-1,1]+(Sx3))*h
Sx4=(xacc(xdata[count-1,0]+(Rx3),ydata[count-1,0]+(Ry3),xdata[count-1,1]+(Sx3),ydata[count-1,1]+(Sy3),w02m,le,delta,omega,t[count-1]+(h)))*h

Ry4=(ydata[count-1,1]+(Sy3))*h
Sy4=(yacc(xdata[count-1,0]+(Rx3),ydata[count-1,0]+(Ry3),xdata[count-1,1]+(Sx3),ydata[count-1,1]+(Sy3),w02m,le,delta,omega,t[count-1]+(h)))*h

delx=(Rx1+(2*Rx2)+(2*Rx3)+Rx4)/6
delvx=(Sx1+(2*Sx2)+(2*Sx3)+Sx4)/6

dely=(Ry1+(2*Ry2)+(2*Ry3)+Ry4)/6
delvy=(Sy1+(2*Sy2)+(2*Sy3)+Sy4)/6

delxa=[delx,delvx]
```

```
    delya=[dely,delvy]

    xdata[count,:]=xdata[count-1,:]+delxa
    ydata[count,:]=ydata[count-1,:]+delya

#-----------------------------------------------------------------------------------------------------

#Define relevant quantities in solution

#Perpendicular distance from z-axis
rd=((xdata[:,0]**2)+(ydata[:,0]**2))**(1/2)

#Height
z=(((le**2)-(rd**2))**(1/2))

#z velocity
vz=(-1.0/z)*((xdata[:,0]*xdata[:,1])+(ydata[:,0]*ydata[:,1]))

#Energy (assuming mass = 1 kg)
PE=g*(le-z)
KE=((xdata[:,1]**2)+(ydata[:,1]**2)+(vz**2))/2
TE=PE+KE

#Angular momentum
ang_mtm=(xdata[:,0]*ydata[:,1])-(ydata[:,0]*xdata[:,1])

#-----------------------------------------------------------------------------------------------------

# Plot the solution
plt.close('all')
```

```
fig, ax = plt.subplots()
ax.plot(xdata[:,0],ydata[:,0],'g')
#ax.plot(t,xdata)
#ax.plot(t,ang_mtm)
#plt.ylim((-0.11,0.0))
#plt.xlim((0,100))
plt.xlabel('X displacement (m)')
plt.ylabel('Y displacement (m)')
#ax.set_aspect('equal')
plt.show()

end_t = time.time()
print("Run Time = ", f"{end_t-start_t:.3f}")

#np.savetxt('t.txt', t)
#np.savetxt('x.txt', xdata)
#np.savetxt('y.txt', ydata)
#np.savetxt('mod.txt', rd)
```

HM_RK4_ran

```
import time
import numpy as np
import scipy as sp
import random as rd
import matplotlib.pyplot as plt

start_t = time.time()

# Define parameters
le = 10.0 #pendulum length

h=0.01 # time step
t = np.arange(0,100.0,h) #This is the time range # total time of simulation in seconds

x0 = [0.5,0.2] #Initial x and vx values in laboratory frame
xdata=np.zeros((len(t),2)) #Setting up array for x and vx data

y0 = [0.5,0.0] #Initial y and vy values in laboratory frame
ydata=np.zeros((len(t),2)) #Setting up array for x and vx data

#Other parameters
g = 9.81 # acceleration due to gravity
w0m=np.sqrt(g/le) # pendulum mean angular frequency
w02m=w0m**2 # square of pendulum mean angular frequency
delta = 0.01 # deviation from symmetry of w02m

#-----------------------------------------------------------------------------------------------------

#Defining restoring force in laboratory frame depending on angle of pendulum mount frame with respect to laboratory frame (theta is random)
```

```
def acc(xf,yf,vxf,vyf,w02mf,lef,deltaf,angf,tf):

    r=np.matrix([[xf],[yf]])
    v=np.matrix([[vxf],[vyf]])

    rot1=np.matrix([[np.cos(angf),np.sin(angf)],[-np.sin(angf),np.cos(angf)]])

    rdash=rot1*r
    xm=float(rdash[0])
    ym=float(rdash[1])

    vdash=rot1*v
    vxm=float(vdash[0])
    vym=float(vdash[1])

    #Defining acceleration in frame of pendulum mount, either in small angle approximation or larger angle approximation (remove and replace "#" symbols as required)
    #axm=-((w02mf+deltaf)*xm)
    axm=-((w02mf+deltaf)*xm*((1-(((xm**2)+(ym**2))/(lef**2)))**(0.5)))-(xm/(lef**2))*((vxm**2)+(vym**2))-(xm/(lef**4))*(((xm*vxm)+(ym*vym))**2)/(1-(((xm**2)+(ym**2))/(lef**2)))

    #aym=-((w02mf-deltaf)*ym)
    aym=-((w02mf-deltaf)*ym*((1-(((xm**2)+(ym**2))/(lef**2)))**(0.5)))-(ym/(lef**2))*((vxm**2)+(vym**2))-(ym/(lef**4))*(((xm*vxm)+(ym*vym))**2)/(1-(((xm**2)+(ym**2))/(lef**2)))

    adash=np.matrix([[axm],[aym]])

    rot2=np.matrix([[np.cos(angf),-np.sin(angf)],[np.sin(angf),np.cos(angf)]])

    a=rot2*adash
```

```
    return a

#Defining x acceleration in laboratory frame depending on angle of pendulum mount frame with respect to laboratory frame (theta is random)
def xacc(xf,yf,vxf,vyf,w02mf,lef,deltaf,angf,tf):

    restacc=acc(xf,yf,vxf,vyf,w02mf,lef,deltaf,angf,tf)

    return float(restacc[0])

#Defining y acceleration in laboratory frame depending on angle of pendulum mount frame with respect to laboratory frame (theta is random)
def yacc(xf,yf,vxf, vyf,w02mf,lef,deltaf,angf,tf):

    restacc=acc(xf,yf,vxf,vyf,w02mf,lef,deltaf,angf,tf)

    return float(restacc[1])

count = 0
xdata[0,:]=x0 #Setting initial x and vx conditions into x and vx data array
ydata[0,:]=y0 #Setting initial y and vy conditions into y and vy data array

n = len(t)
looparray = np.arange(1,n,1)

#4th order Runge-Kutta for 2D pendulum, assuming the restoring forces depend on both x and y in laboratory frame (but not in pendulum mount frame) and this varies with time. The x and y variables are treated the same in the intermediate Runge-Kutta steps.
for count in looparray:

    #Defining random mount angle; either with same angle value in all Runge-Kutta sub-steps, or different values at each Runge-Kutta sub-step, or set to a constant value of zero
    ang1 = ang2 = ang3 = ang4 = rd.uniform(0,2*np.pi)
```

```
#ang1 = ang2 = ang3 = ang4 = 0.0

Rx1=xdata[count-1,1]*h
Sx1=(xacc(xdata[count-1,0],ydata[count-1,0],xdata[count-1,1],ydata[count-1,1],w02m,le,delta,ang1,t[count-1]))*h

Ry1=ydata[count-1,1]*h
Sy1=(yacc(xdata[count-1,0],ydata[count-1,0],xdata[count-1,1],ydata[count-1,1],w02m,le,delta,ang1,t[count-1]))*h

Rx2=(xdata[count-1,1]+(0.5*Sx1))*h
Sx2=(xacc(xdata[count-1,0]+(0.5*Rx1),ydata[count-1,0]+(0.5*Ry1),xdata[count-1,1]+(0.5*Sx1),ydata[count-
1,1]+(0.5*Sy1),w02m,le,delta,ang2,t[count-1]+(0.5*h)))*h

Ry2=(ydata[count-1,1]+(0.5*Sy1))*h
Sy2=(yacc(xdata[count-1,0]+(0.5*Rx1),ydata[count-1,0]+(0.5*Ry1),xdata[count-1,1]+(0.5*Sx1),ydata[count-
1,1]+(0.5*Sy1),w02m,le,delta,ang2,t[count-1]+(0.5*h)))*h

Rx3=(xdata[count-1,1]+(0.5*Sx2))*h
Sx3=(xacc(xdata[count-1,0]+(0.5*Rx2),ydata[count-1,0]+(0.5*Ry2),xdata[count-1,1]+(0.5*Sx2),ydata[count-
1,1]+(0.5*Sy2),w02m,le,delta,ang3,t[count-1]+(0.5*h)))*h

Ry3=(ydata[count-1,1]+(0.5*Sy2))*h
Sy3=(yacc(xdata[count-1,0]+(0.5*Rx2),ydata[count-1,0]+(0.5*Ry2),xdata[count-1,1]+(0.5*Sx2),ydata[count-
1,1]+(0.5*Sy2),w02m,le,delta,ang3,t[count-1]+(0.5*h)))*h

Rx4=(xdata[count-1,1]+(Sx3))*h
Sx4=(xacc(xdata[count-1,0]+(Rx3),ydata[count-1,0]+(Ry3),xdata[count-1,1]+(Sx3),ydata[count-1,1]+(Sy3),w02m,le,delta,ang4,t[count-1]+(h)))*h

Ry4=(ydata[count-1,1]+(Sy3))*h
Sy4=(yacc(xdata[count-1,0]+(Rx3),ydata[count-1,0]+(Ry3),xdata[count-1,1]+(Sx3),ydata[count-1,1]+(Sy3),w02m,le,delta,ang4,t[count-1]+(h)))*h

delx=(Rx1+(2*Rx2)+(2*Rx3)+Rx4)/6
```

```
    delvx=(Sx1+(2*Sx2)+(2*Sx3)+Sx4)/6

    dely=(Ry1+(2*Ry2)+(2*Ry3)+Ry4)/6
    delvy=(Sy1+(2*Sy2)+(2*Sy3)+Sy4)/6

    delxa=[delx,delvx]
    delya=[dely,delvy]

    xdata[count,:]=xdata[count-1,:]+delxa
    ydata[count,:]=ydata[count-1,:]+delya

#-------------------------------------------------------------------------------------------------------

#Define relevant quantities in solution

#Perpendicular distance from z-axis
rd=((xdata[:,0]**2)+(ydata[:,0]**2))**(1/2)

#Height
z=(((le**2)-(rd**2))**(1/2))

#z velocity
vz=(-1.0/z)*((xdata[:,0]*xdata[:,1])+(ydata[:,0]*ydata[:,1]))

#Energy (assuming mass = 1 kg)
PE=g*(le-z)
KE=((xdata[:,1]**2)+(ydata[:,1]**2)+(vz**2))/2
TE=PE+KE

#Angular momentum
ang_mtm=(xdata[:,0]*ydata[:,1])-(ydata[:,0]*xdata[:,1])
```

```
#-----------------------------------------------------------------------------------------------------

# Plot the solution
plt.close('all')

fig, ax = plt.subplots()
ax.plot(xdata[:,0],ydata[:,0])
#ax.plot(t,xdata)
#ax.plot(t,ang_mtm)
#plt.ylim((-0.11,0.0))
#plt.xlim((0,100))
plt.xlabel('X displacement (m)')
plt.ylabel('Y displacement (m)')
#ax.set_aspect('equal')
plt.show()

end_t = time.time()
print("Run Time = ", f"{end_t-start_t:.3f}")

#np.savetxt('t.txt', t)
#np.savetxt('x.txt', xdata)
#np.savetxt('y.txt', ydata)
#np.savetxt('mod.txt', rd)
```

Py_odeint_rot

```
import time
import numpy as np
import scipy as sp
from scipy.integrate import odeint
import matplotlib.pyplot as plt

start_t = time.time()

# Define parameters
le = 10.0 #pendulum length

omega = 10.0 # mount spin angular velocity
h=0.01 # time step
t = np.arange(0,100.0,h) #This is the time range # total time of simulation in seconds

[x0,vx0] = [0.5,0.2] #Initial x and vx values in laboratory frame

[y0,vy0] = [0.5,0.0] #Initial y and vy values in laboratory frame

init_state = [x0, y0, vx0, vy0]

#Other parameters
g = 9.81 # acceleration due to gravity
w0m=np.sqrt(g/le) # pendulum mean angular frequency
w02m=w0m**2 # square of pendulum mean angular frequency
delta = 0.01 # deviation from symmetry of w02m

#-----------------------------------------------------------------------------------------------------
```

```
#Defining restoring force in laboratory frame depending on angle of pendulum mount frame with respect to laboratory frame (theta = omega.t)
def acc(state,tf,w02mf,lef,deltaf,omegaf):

    xf, yf, vxf, vyf = state

    r=np.matrix([[xf],[yf]])
    v=np.matrix([[vxf],[vyf]])

    rot1=np.matrix([[np.cos(omegaf*tf),np.sin(omegaf*tf)],[-np.sin(omegaf*tf),np.cos(omegaf*tf)]])

    rdash=rot1*r
    xm=float(rdash[0])
    ym=float(rdash[1])

    vdash=rot1*v
    vxm=float(vdash[0])
    vym=float(vdash[1])

    #Defining acceleration in frame of pendulum mount, either in small angle approximation or larger angle approximation (remove and replace "#" symbols as required)
    #axm=-((w02mf+deltaf)*xm)
    axm=-((w02mf+deltaf)*xm*((1-(((xm**2)+(ym**2))/(lef**2)))**(0.5)))-(xm/(lef**2))*((vxm**2)+(vym**2))-(xm/(lef**4))*(((xm*vxm)+(ym*vym))**2)/(1-(((xm**2)+(ym**2))/(lef**2)))

    #aym=-((w02mf-deltaf)*ym)
    aym=-((w02mf-deltaf)*ym*((1-(((xm**2)+(ym**2))/(lef**2)))**(0.5)))-(ym/(lef**2))*((vxm**2)+(vym**2))-(ym/(lef**4))*(((xm*vxm)+(ym*vym))**2)/(1-(((xm**2)+(ym**2))/(lef**2)))

    adash=np.matrix([[axm],[aym]])

    rot2=np.matrix([[np.cos(omegaf*tf),-np.sin(omegaf*tf)],[np.sin(omegaf*tf),np.cos(omegaf*tf)]])
```

```
    a=rot2*adash

    return [vxf, vyf, float(a[0]), float(a[1])]

#Using odeint for 2D pendulum

sol = odeint(acc, init_state, t, args = (w02m, le, delta, omega))

# Extract the solution

xdata = sol[:, 0]
ydata = sol[:, 1]
vxdata = sol[:, 2]
vydata = sol[:, 3]

#-------------------------------------------------------------------------------------------------------------

#Define relevant quantities in solution

#Perpendicular distance from z-axis
rd=((xdata**2)+(ydata**2))**(1/2)

#Height
z=(((le**2)-(rd**2))**(1/2))

#z velocity
vz=(-1.0/z)*((xdata*vxdata)+(ydata*vydata))

#Energy (assuming mass = 1 kg)
PE=g*(le-z)
```

```
KE=((vxdata**2)+(vydata**2)+(vz**2))/2
TE=PE+KE

#Angular momentum
ang_mtm=(xdata*vydata)-(ydata*vxdata)

#-----------------------------------------------------------------------------------------------------------

# Plot the solution
plt.close('all')

fig, ax = plt.subplots()
ax.plot(xdata,ydata)
#ax.plot(t,xdata)
#ax.plot(t,ang_mtm)
#plt.ylim((-0.11,0.0))
#plt.xlim((0,100))
plt.xlabel('X displacement (m)')
plt.ylabel('Y displacement (m)')
#ax.set_aspect('equal')
plt.show()

end_t = time.time()
print("Run Time = ", f"{end_t-start_t:.3f}")

#np.savetxt('t.txt', t)
#np.savetxt('x.txt', xdata)
#np.savetxt('y.txt', ydata)
#np.savetxt('mod.txt', rd)
```