

# Computational Physics Homework #1

## PHYS222

Georges Demirjian

November 2, 2025

# Contents

<b>1</b>	<b>Classical Mechanics</b>	<b>3</b>
<b>2</b>	<b>Euler Lagrange with Constraints:</b>	<b>8</b>
<b>3</b>	<b>Two-Body Problem</b>	<b>10</b>
<b>4</b>	<b>Kepler's First Law</b>	<b>12</b>
4.1	RK4 . . . . .	12
<b>5</b>	<b>Kepler's Second Law</b>	<b>14</b>
<b>6</b>	<b>Kepler's Third law</b>	<b>15</b>
<b>7</b>	<b>Quantum Mechanics</b>	<b>16</b>
<b>8</b>	<b>FEM</b>	<b>19</b>
<b>9</b>	<b>Dynamics</b>	<b>22</b>
<b>10</b>	<b>Logistic Map</b>	<b>26</b>
<b>11</b>	<b>What do 1D Maps have to do with science</b>	<b>27</b>
<b>12</b>	<b>SVD and Coherent Structures</b>	<b>30</b>
<b>A</b>	<b>Appendix: Python Codes</b>	<b>33</b>
A.1	Problem 1 . . . . .	33
A.2	Problem 2 . . . . .	36
A.3	Problem 3 . . . . .	37
A.4	Problem 4 . . . . .	40
A.5	Problem 5 . . . . .	42
A.6	Problem 6 . . . . .	44
A.7	Problem 7 . . . . .	45
A.8	Problem 8: Matlab code for FEM . . . . .	47
A.9	Problem 9 . . . . .	49
A.10	Problem 10 . . . . .	52
A.11	Problem 11 . . . . .	53
A.12	Problem 12 . . . . .	56

# 1 Classical Mechanics

We need to express the position of the pendulum in terms of generalized coordinates.

$$\mathcal{L} = T - V$$

## Euler-Lagrange Equation

$$\frac{d}{dt} \left( \frac{\partial \mathcal{L}}{\partial \dot{q}} \right) - \frac{\partial \mathcal{L}}{\partial q} = 0$$

## Expressing Coordinates

We can express  $x$  and  $y$  in terms of our generalized coordinates  $(\theta, R, \alpha)$

$$x = l \sin \theta + R \cos \alpha$$

$$y = -l \cos \theta + R \sin \alpha$$

And

$$\dot{x} = l \dot{\theta} \cos \theta - \omega R \sin \alpha$$

$$\dot{y} = -l \dot{\theta} \sin \theta + \omega R \cos \alpha$$

## Kinetic and Potential Energy

Kinetic Energy Term:

$$\begin{aligned} T &= \frac{1}{2}m(\dot{x}^2 + \dot{y}^2) \\ &= \frac{1}{2}m[(\dot{\theta}l \cos \theta - \omega R \sin \alpha)^2 + (\dot{\theta}l \sin \theta + \omega R \cos \alpha)^2] \\ &= \frac{1}{2}m[(\dot{\theta}l)^2 + (\omega R)^2 + 2\dot{\theta}\omega Rl \sin(\theta - \omega t)] \end{aligned}$$

Potential Energy term:

$$U = mgy = mg(R \sin \alpha - l \cos \theta)$$

Final form of our Lagrangian is:

$$\mathcal{L} = T - V$$

## Euler-Lagrange Equation

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \dot{\theta}} &= ml^2 + mR\omega l \sin(\theta - \omega t) \\ \frac{d}{dt} \left( \frac{\partial \mathcal{L}}{\partial \dot{\theta}} \right) &= m\ddot{\theta}l^2 + (\dot{\theta} - \omega)mR\omega l \cos(\theta - \omega t) \\ \frac{\partial \mathcal{L}}{\partial \theta} &= m\dot{\theta}R\omega l \cos(\theta - \omega t) - mgl \sin \theta \end{aligned}$$

We get:

$$-m\dot{\theta}R\omega l \cos(\theta - \omega t) + mgl \sin \theta + m\ddot{\theta}l^2 + m(\dot{\theta} - \omega)R\omega l \cos(\theta - \omega t) = 0$$

After simplifying the differential equation becomes:

$$\ddot{\theta} = \frac{g}{l} \sin \theta - \frac{R\omega^2}{l} \cos(\theta - \omega t)$$

## In Terms of Two Coupled First-Order Differential Equations

Let:

$$\begin{aligned} x_1 &= \theta, \\ x_2 &= \dot{\theta} \end{aligned}$$

Then:

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -\frac{g}{l} \sin x_1 + \frac{R\omega^2}{l} \cos(x_1 - \omega t) \end{aligned}$$

Our main goal is to find the fixed points, and initial angle  $\theta_0$  such that after one period  $T = 2\pi/\omega$ , the pendulum returns to the same position and velocity ( $\theta_{new} = \theta_0$ ,  $\dot{\theta}_{new} = \dot{\theta}_0$ ).

We first define constants:

$$g = 9.81, \quad l = 1.0, \quad R = 0.1, \quad \omega = 2.0.$$

We are using time step  $h = 0.01$ , and the bisection method tolerances are chosen as  $tol = 10^{-10}$  and  $max\_iter = 100$ .

## Numerical Method

We define  $x_1 = \theta$  and  $x_2 = \dot{\theta}$  and integrate the system

$$\dot{x}_1 = x_2, \quad \dot{x}_2 = -\frac{g}{l} \sin x_1 + \frac{R\omega^2}{l} \cos(x_1 - \omega t)$$

using the fourth-order Runge-Kutta 4. we compute the slopes

$$k_1 = f(x_1, x_2, t), \quad k_2 = f(x_1 + \frac{h}{2}k_{1,1}, x_2 + \frac{h}{2}k_{1,2}, t + \frac{h}{2}),$$

and so on, combining them as

$$x_{n+1} = x_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4), \quad t_{n+1} = t_n + h.$$

This procedure return values in froms of lists for  $\theta(t)$ ,  $\dot{\theta}(t)$ , and  $t$ .

## Detection of Fixed-Point

After integration, we detect where  $\theta(t) = 0$  through sign changes  $\theta_i * \theta_{i+1} < 0$ . For each range, we use bisection to find teh exact location of  $\theta = 0$ . The corresponding  $\dot{\theta}$  values are extracted, and we again use bisection to find the exact locations for  $\dot{\theta}(t) = 0$ . Points where both  $\theta \simeq 0$  and  $\dot{\theta} \simeq 0$  are simultaneously satisfied "fixed points":

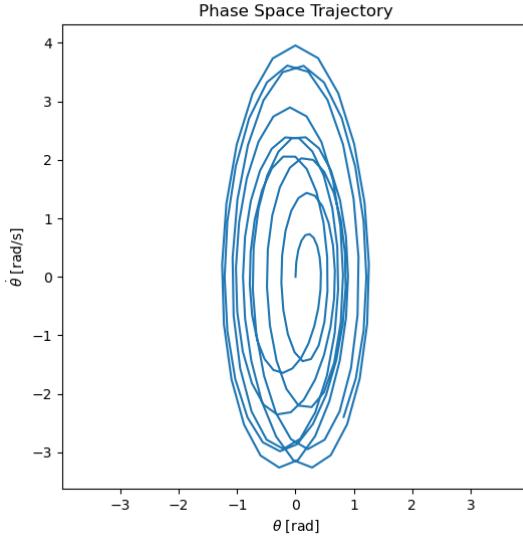


Figure 1: Phase space trajectory for  $\omega = \pi/2$

## Stability

Then for each of them we perturb  $(\theta^*, \dot{\theta}^*)$  by a small  $\epsilon$  in either variable and integrate for a short time interval. If the trajectory remains unperturbed , the point is marked stable; otherwise, unstable.

## Results and Discussion

**Case 1:**  $R = 0.1$ ,  $\omega = \pi/32$ . The pendulum is very weakly and slowly driven. The phase-space trajectory forms a narrow ellipse around the origin (Fig 2). Multiple near-zero pairs  $(\theta, \dot{\theta})$  are detected and all are numerically stable:

$$(\theta, \dot{\theta}) \sim 10^{-5}-10^{-4}.$$

These points correspond to small oscillations about the downward equilibrium—the system behaves almost as if undriven.

**Case 2:**  $R = 0.5$ ,  $\omega = \pi/16$ . With a stronger drive and higher frequency, the orbit broadens but remains regular . Two near-zero candidate points are identified, both stable. The system still exhibits approximately periodic motion about the origin, but the amplitude of  $\dot{\theta}$  grows by nearly an order of magnitude compared with Case 1.

**Case 3:**  $R = 0.5$ ,  $\omega = \pi/4$ . For faster driving, the phase trajectory enlarges further and becomes more distorted . We no longer detects zero-crossings of both  $\theta$  and  $\dot{\theta}$  simultaneously—hence, no fixed-point candidates are found. This indicates that the system no longer remains close to a single equilibrium configuration but instead oscillates around a periodically moving center.

**Case 4:**  $R = 1.0$ ,  $\omega = \pi/4$ . For a large amplitude and high frequency, the motion covers a broad region of phase space, with clear multi-loop patterns. No fixed points are

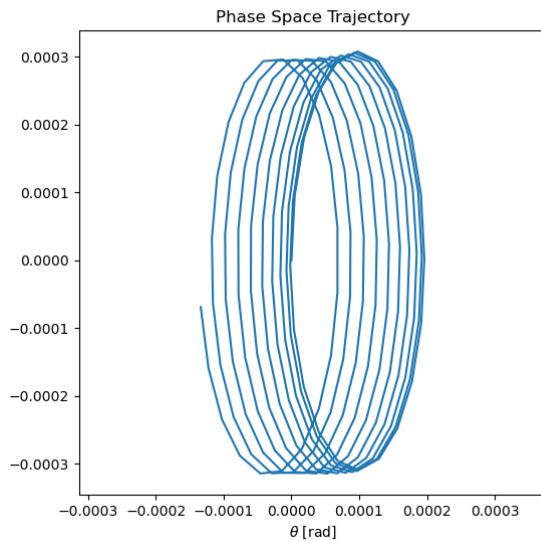


Figure 2: Phase space trajectory for  $\omega = \pi/32$  and  $R = 0.1$

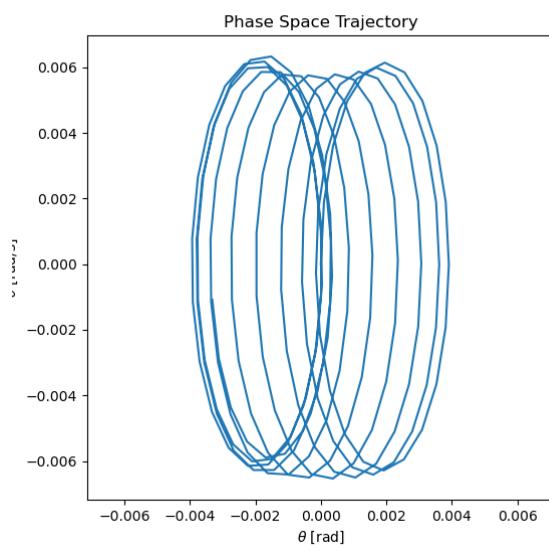


Figure 3: Phase space trajectory for  $\omega = \pi/16$  and  $R = 0.5$

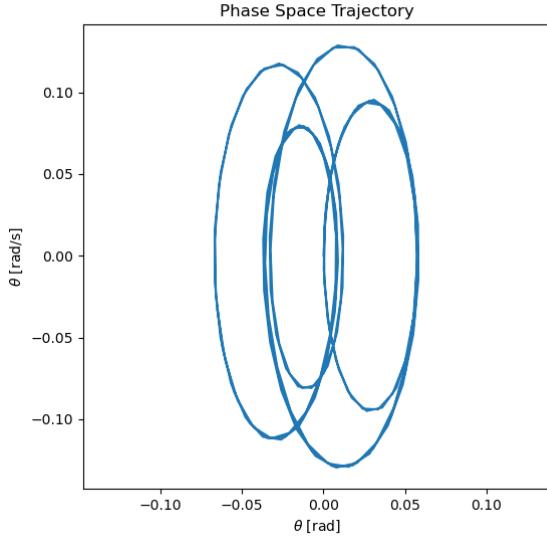


Figure 4: Phase space trajectory for  $\omega = \pi/4$  and  $R = 0.5$

detected, and the motion appears quasi-periodic. The system's effective potential varies rapidly, making the dynamics strongly time-dependent.

## Interpretation

In a periodically driven pendulum, the equations explicitly depend on time, so fixed points in the  $(\theta, \dot{\theta})$  plane do not truly exist. Instead, the system can exhibit trajectories that repeat after each driven period  $T = 2\pi/\omega$ .

For small  $R$  and  $\omega$ , the time-dependence is weak, and we get some fixed points. the pendulum behaves almost as if it were static, allowing us to identify quasi equilibriums that are almost stable. As the driving strength, frequency increases or radius enlarges, the system departs from this regime: the equilibrium-like structure dissolves, and the motion transitions into periodic or even quasi-periodic behavior.

## 2 Euler Lagrange with Constraints:

The particle of mass  $m$  moves on a circular path of fixed radius  $R$  under the influence of gravity. The constraint can be written as

$$f(r, t) = r - R = 0,$$

This equation depends only on the coordinates and not explicitly on time, meaning it is holonomic.

We eliminate the constraint by introducing a single generalized coordinate,  $\theta$ :

$$x = R \sin \theta, \quad y = R \cos \theta.$$

This automatically satisfies the constraint , reducing the system to one degree of freedom.

The kinetic and potential energies are

$$T = \frac{1}{2}mR^2\dot{\theta}^2, \quad V = mgR(1 - \cos \theta),$$

so the Lagrangian is

$$\mathcal{L} = T - V = \frac{1}{2}mR^2\dot{\theta}^2 - mgR(1 - \cos \theta).$$

Applying the Euler–Lagrange equation gives the nonlinear pendulum equation

$$\ddot{\theta} = -\frac{g}{R} \sin \theta.$$

The Euler–Lagrange equations with constraint forces take the form

$$\frac{d}{dt} \frac{\partial \mathcal{L}}{\partial \dot{q}_i} - \frac{\partial \mathcal{L}}{\partial q_i} = \sum_j \lambda_j \frac{\partial f_j}{\partial q_i},$$

where  $\lambda_j$  are the Lagrange multipliers associated with each constraint.  $f(r, t) = r - R = 0$ , so  $\partial f / \partial r = 1$ .

The total forces acting on the system on the radial direction are:

$$ma_r = mg \cos \theta - \vec{F}_{constraint}$$

$$F_{constraint} = m(R\dot{\theta}^2) + mg \cos \theta$$

## Numerical Method

We solve the equation of motion that we got from the angular Euler–Lagrange equation:

$$\ddot{\theta} = -\frac{g}{R} \sin \theta.$$

This second order differential equation integrated numerically using RK4. We set the time step  $h = 0.01$  s, and the initial conditions  $\theta(0)$  and  $\dot{\theta}(0)$ . We use two coupled equations for  $\theta$  and  $\dot{\theta}$ , defined through a function returning their derivatives:

$$f(\theta, \dot{\theta}) = (\dot{\theta}, -\frac{g}{R} \sin \theta).$$

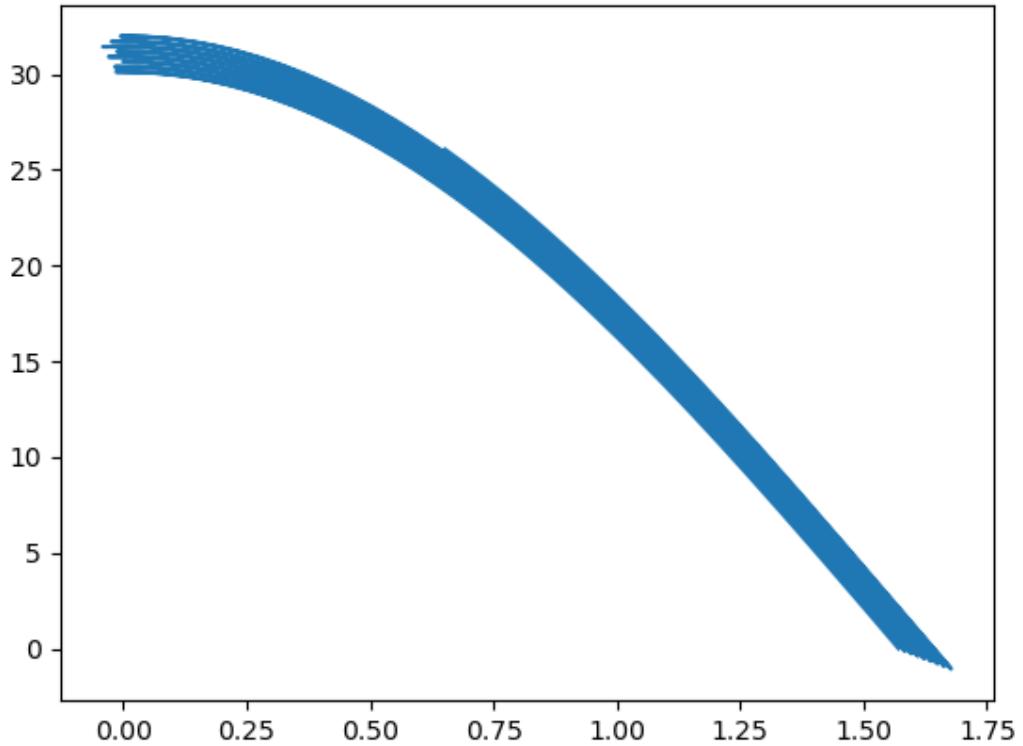


Figure 5: constraint force vs theta

At each time step, the RK4 function updates the values of  $\theta$  and  $\dot{\theta}$ . As the constraint is on a half circle, each time  $\theta < 0$  or  $\theta > \pi$ , when the object reaches  $y=0$ , it bounces back without loosing any energy, the angular speed remains the same angular velocity changes direction. We store them in lists to then compute the constraint force  $F$ .

Once  $\theta(t)$  and  $\dot{\theta}(t)$  are obtained, the constraint force is evaluated as (like a loss function)

$$F(t) = m(g \cos \theta + R\dot{\theta}^2),$$

and plotted as a function of  $\theta$ .

We observe that the constraint force varies with  $\theta$ . It reaches its maximum ,where  $\theta = 0$  and decreases for larger  $|\theta|$ , gets to a minimal value, $F_{constraint} = 0$  at  $\theta = \pi/2$ . when the object is about to bounce back on the x plane  $\theta = \pi$  and  $\theta = 0$ , the constraint force is at maximum value.

### 3 Two-Body Problem

#### Mathematical background

We have two masses  $m_1$  and  $m_2$  interacting via Newtonian gravity with  $G = 1$ . Let  $\mathbf{r}_1, \mathbf{r}_2$  be their positions. The position of the COM is:

$$\mathbf{R} = \frac{m_1 \mathbf{r}_1 + m_2 \mathbf{r}_2}{M},$$

With mass:

$$M = m_1 + m_2,$$

and the relative coordinate

$$\mathbf{r} = \mathbf{r}_1 - \mathbf{r}_2$$

Newton's third law gives equal and opposite internal forces, so the CM moves uniformly:

$$M \ddot{\mathbf{R}} = \mathbf{0} \Rightarrow \mathbf{R}(t) = \mathbf{R}_0 + \mathbf{V}_0 t.$$

The reduced mass of the system:

$$\mu = \frac{m_1 m_2}{m_1 + m_2}$$

moving in a central  $1/r$  potential:

$$\mu \ddot{\mathbf{r}} = -\frac{G m_1 m_2}{r^3} \mathbf{r}.$$

#### Integrals of motion and orbital geometry

Because the force is central, the angular momentum is conserved:

$$\mathbf{L} = \mu \mathbf{r} \times \dot{\mathbf{r}}$$

The energy of the relative motion is also conserved:

$$E = \frac{1}{2} \mu \|\dot{\mathbf{r}}\|^2 - \frac{G m_1 m_2}{r}$$

For  $E < 0$  the orbit is a Kepler ellipse. The eccentricity can be obtained from  $(E, L)$  as

$$e = \sqrt{1 + \frac{2 E L^2}{\mu (G m_1 m_2)^2}},$$

#### Numerical Part

- m1, m2, G, mu, M: problem constants, with  $\mu = \frac{m_1 m_2}{M}$  and  $M = m_1 + m_2$ .
- f(r, v, t) returns the phase-space derivatives for both bodies: we compute  $\mathbf{r}_{\text{rel}} = \mathbf{r}_1 - \mathbf{r}_2$ , its length  $r$ , and the mutual accelerations  $\mathbf{a}_1 = -(G m_1 m_2 / r^3) \mathbf{r}_{\text{rel}} / m_1$ ,  $\mathbf{a}_2 = +(G m_1 m_2 / r^3) \mathbf{r}_{\text{rel}} / m_2$ . It returns  $(\dot{\mathbf{r}}_1, \dot{\mathbf{r}}_2) = (\mathbf{v}_1, \mathbf{v}_2)$  and  $(\dot{\mathbf{v}}_1, \dot{\mathbf{v}}_2)$ .

- `rk4_step`: fourth-order Runge–Kutta update of the full two body state ( $\mathbf{r}_1, \mathbf{r}_2, \mathbf{v}_1, \mathbf{v}_2$ ) with step  $h =$ .
- At rk4 each step store  $\mathbf{r}_1, \mathbf{r}_2$ , and compute the CM.

$$\mathbf{R} = \frac{m_1 \mathbf{r}_1 + m_2 \mathbf{r}_2}{M}.$$

Given the initial data, both the initial CM position and the total momentum are zero.

- Build  $\mathbf{r}_{\text{rel}} = \mathbf{r}_1 - \mathbf{r}_2$ ,
- estimate velocities by forward differences

$$\dot{\mathbf{r}}_1(t_n) \approx \frac{\mathbf{r}_1(t_{n+1}) - \mathbf{r}_1(t_n)}{h},$$

$$\dot{\mathbf{r}}_2(t_n) \approx \frac{\mathbf{r}_2(t_{n+1}) - \mathbf{r}_2(t_n)}{h},$$

and hence  $\dot{\mathbf{r}} = \dot{\mathbf{r}}_1 - \dot{\mathbf{r}}_2$ . Then compute  $E, L, e$  analytically using the values that we found.

## results

From the simulation we obtained:

$$\mu = 0.6666667, \quad \mathbf{R}_{\text{CM}} = (0.0, 0.0)$$

which confirms that the system's center of mass remains fixed, as expected from the momentum-balanced initial conditions.

The numerical invariants evolve as:

$$L = [1.20000074, 1.20002729, 1.20006545, \dots, 1.24875802, 1.24836294, 1.24794733],$$

$$E = [-0.85333274, -0.85331615, -0.85329955, \dots, -0.84934968, -0.84931571, -0.8492823],$$

$$e = [0.27999911, 0.27995829, 0.27988562, \dots, 0.08151839, 0.08551854, 0.0895171],$$

with a mean eccentricity of  $\bar{e} = 0.1771$ .

Comparing, the values computed directly from the initial conditions are:

$$E_0 = -0.8533333, \quad L_0 = 1.2000, \quad e_0 = 0.28.$$

## Errors

$$\Delta E = 0.47\%, \quad \Delta L = 3.99\%, \quad \Delta e = 68.03\%.$$

we have small drift in energy ( $\sim 0.5\%$ ) and a larger drift in angular momentum ( $\sim 4\%$ ), because of the integration of rk4. And eventually our eccentricity has a much larger variation which stems from its nonlinear dependence on both  $E$  and  $L$ :

$$e = \sqrt{1 + \frac{2EL^2}{\mu(Gm_1m_2)^2}}.$$

Even slight changes in  $E$  or  $L$ , amplify  $e$ , resulting in 68% error.

The fact that  $E < 0$  confirms that the two-body system remains bounded. The mean eccentricity  $\bar{e} \approx 0.177$  corresponds to a mildly elliptical relative orbit, more or less consistent with the initial  $e_0 = 0.28$ .

## 4 Kepler's First Law

The motion of a particle under the gravitational attractive force between two bodies, is governed by Newton's second law:

$$\ddot{\mathbf{r}} = -\frac{\mu}{r^3} \mathbf{r},$$

where  $\mu = GM$  is the gravitational parameter of the system,  $\mathbf{r}$  is the position vector, and  $r = |\mathbf{r}|$ . Because the force is central, angular momentum is conserved:

$$L = r^2 \dot{\theta} = \text{constant}.$$

Introducing the substitution  $u = 1/r$ , and differentiating with respect to  $\theta$ , the equation of motion becomes the Binet equation:

$$\frac{d^2 u}{d\theta^2} + u = \frac{\mu}{L^2}.$$

Its analytical solution represents a conic section:

$$r(\theta) = \frac{p}{1 + e \cos \theta},$$

where  $e$  is the eccentricity of the orbit. For  $0 < e < 1$ , the trajectory is an ellipse, consistent with Kepler's 1st law.

### 4.1 RK4

We solve the Binet equation numerically using RK4.

The system of first-order equations:

$$\begin{aligned} \frac{du}{d\theta} &= u', \\ \frac{du'}{d\theta} &= \frac{\mu}{L^2} - u. \end{aligned}$$

Specify the initial conditions at  $\theta = 0$ :

$$r_0 = 1.0, \quad \dot{r}_0 = 0.0, \quad \dot{\theta}_0 = 1.2.$$

These give the specific angular momentum and total energy

$$L = r_0^2 \dot{\theta}_0 = 1.2, \quad E = \frac{1}{2} \left( \dot{r}_0^2 + r_0^2 \dot{\theta}_0^2 \right) - \frac{\mu}{r_0} = -0.28.$$

The analytical eccentricity is :

$$e = \sqrt{1 + \frac{2EL^2}{\mu^2}} = 0.44.$$

Integrate the system for  $\theta \in [0, 4\pi]$  using  $n = 4000$  steps with

$$\Delta\theta = \frac{4\pi}{4000} = 0.00314.$$

Reconstruct the radial distance  $r(\theta) = 1/u(\theta)$  at each integration step.  
Determine the minimum and maximum radii:

$$r_{\min} = \min(r), \quad r_{\max} = \max(r),$$

and from these evaluate the numerical eccentricity and semi-major axis:

$$e_{\text{num}} = \frac{r_{\max} - r_{\min}}{r_{\max} + r_{\min}}, \quad a_{\text{num}} = \frac{r_{\max} + r_{\min}}{2}.$$

Finally, compute the numerical specific energy:

$$E_{\text{num}} = -\frac{\mu}{2a_{\text{num}}}.$$

## Results

The numerical integration produced the following results:

Quantity	Value
$r_{\min}$	1.0000
$r_{\max}$	2.5714
$e_{\text{num}}$	0.4400
$a_{\text{num}}$	1.7857
$E_{\text{num}}$	-0.2800

Comparaison of Analytical and numerical findings:

$$|e_{\text{num}} - e_{\text{analytic}}| = 1.72 \times 10^{-15}, \quad |E_{\text{num}} - E_{\text{analytic}}| = 8.33 \times 10^{-16}.$$

These results confirm that the RK4 numerical integration accurately reproduced the theoretical orbit predicted using the Binet equation. The trajectory is an elliptical orbit with eccentricity  $e \approx 0.44$  and semi-major axis  $a \approx 1.79$ , consistent with the expected bound motion ( $E < 0$ ).

## 5 Kepler's Second Law

Kepler's Second Law states that the line joining a planet and the Sun sweeps out equal areas in equal times. In physical terms, this means the *areal velocity* of a planet is constant during its orbital motion around the Sun.

For a particle of mass  $m$  orbiting under a central gravitational force, the equation of motion is:

$$\ddot{\vec{r}} = -\frac{GM}{r^3}\vec{r},$$

The areal velocity, defined as the rate of area swept by the radius vector:

$$\frac{dA}{dt} = \frac{1}{2}r^2\dot{\theta} = \frac{L}{2m}.$$

So,  $\dot{A}$  is constant

**RK4** To verify Kepler's Second Law numerically, the orbital motion of the planet is solved using rk4 just like in the previous exercise, and the swept area is computed using the Trapezoidal Rule.

So the general form of the integration of the equations is the same as the exercise before, however here in each step the instantaneous areal velocity is evaluated using the relation:

$$\frac{dA}{dt} = \frac{1}{2}(xv_y - yv_x),$$

which follows directly from  $\frac{1}{2}r^2\dot{\theta}$  in Cartesian coordinates. This quantity is stored at each time step during the integration.

**Trapezoidal Rule** The total swept area up to time  $t_n$  is obtained by numerically integrating  $\frac{dA}{dt}$  using the Trapezoidal Rule:

$$A(t_{n+1}) = A(t_n) + \frac{\Delta t}{2} \left[ \left( \frac{dA}{dt} \right)_n + \left( \frac{dA}{dt} \right)_{n+1} \right].$$

If Kepler's Second Law holds,  $A(t)$  should increase linearly with time.

**Results and Discussion** From the numerical integration using the fourth-order Runge–Kutta (RK4) method, the instantaneous areal velocity was computed at each step. The results at the *perihelion* and *aphelion* were found to be:

$$\left( \frac{dA}{dt} \right)_{\text{peri}} = 2.227899 \times 10^{15} \text{ m}^2/\text{s}, \quad \left( \frac{dA}{dt} \right)_{\text{aphelion}} = 2.227899 \times 10^{15} \text{ m}^2/\text{s}.$$

The absolute difference between these two values is approximately

$$\Delta \left( \frac{dA}{dt} \right) = 9.5 \text{ m}^2/\text{s},$$

corresponding to a relative error on the order of  $10^{-15}$ , which is negligible and attributed solely to numerical round-off and discretization errors.

This confirms that the areal velocity remains constant throughout the motion. The small error arises from finite step size in the RK4 algorithm.

## 6 Kepler's Third law

To calculate the Jupiter's mass, we use kepler's third law that states that the square of a planet's orbital period is directly proportional to the cube of the semi-major axis of its orbit. The explicit formula is

$$T^2 = \frac{4\pi^2}{GM} a^3$$

We want to solve for  $M_j$  using least squares and linear fit. We are give the Period  $T$  and the semi-major axis  $a$  of the satellites of Jupiter. We turn the equation into line equation that passes through the origin with a slope, from where we will get the mass of the planet. We set

$$y = kx$$

where  $y = T^2$  and  $x = a^3$  and so  $k = \frac{4\pi^2}{GM_j}$

$$\text{And } M_j = \frac{4\pi^2}{Gk}$$

We use least squares method to find k :

$$k = \frac{\sum_i x_i y_i}{\sum_i x_i^2}$$

from our results:

Jupiter's mass: 1.898612128134478e+27 kg

Accepted value: 1.89852e27 kg

Relative error = 4.852629125742087e-05

## 7 Quantum Mechanics

### Infinite potential well: Time independent Schrodinger Equation

$$U(x) = \begin{cases} 0 & 0 < x < L \\ \infty & \text{otherwise} \end{cases}$$

Inside the well ( $0 < x < L$ ):

$$-\frac{\hbar^2}{2m} \frac{d^2\psi}{dx^2} = E\psi$$

Rearranging:

$$\frac{d^2\psi}{dx^2} + k^2\psi = 0, \quad k^2 = \frac{2mE}{\hbar^2}$$

General Solution

$$\psi(x) = A \sin(kx) + B \cos(kx)$$

Boundary Conditions

$$\psi(0) = 0 \implies B = 0$$

$$\psi(L) = 0 \implies A \sin(kL) = 0$$

Non-trivial solution requires:

$$kL = n\pi, \quad n = 1, 2, 3, \dots$$

Quantized Energy Levels

$E_n = \frac{n^2\pi^2\hbar^2}{2mL^2}$

where  $n = 1, 2, 3, \dots$

Normalized Wavefunctions

$$\psi_n(x) = \sqrt{\frac{2}{L}} \sin\left(\frac{n\pi x}{L}\right)$$

### Numerical Implementation

we want to numerically compute the bound state energies of a particle in a 1d infinite potential well on the interval  $(0, L)$  by:

1. Converting the time-independent Schrödinger equation into a first-order system,
2. Integrating the system from  $x = 0$  to  $x = L$  using a fourth-order Runge–Kutta (RK4) method for a trial energy  $E$ ,
3. Detecting sign changes in  $\psi(L; E)$  as  $E$  is scanned, thereby bracketing eigenvalues,
4. Refining each bracket with a recursive bisection to obtain accurate numerical eigen-energies,
5. Optionally comparing the numerical energies against the known analytical values and reporting absolute/relative errors.

## Parameters and Initial Conditions

We set

$$L = 1, \quad m = 1, \quad \hbar = 1,$$

with Dirichlet boundary conditions inside the well. Numerically, it enforces

$$\psi(0) = 0, \quad \psi'(0) = \phi_0 = 1,$$

The step size for spatial integration is chosen as

$$h = 0.01,$$

and trial energies are scanned from  $E = 0$  upward in increments of 0.1.

**First-Order System (f)** Inside the well the potential  $U(x) = 0$ , so with  $m = \hbar = 1$  the Schrödinger equation reduces to

$$-\frac{1}{2}\psi''(x) = E\psi(x).$$

Introducing the auxiliary variable  $\phi = \psi'$ , the code uses

$$\psi' = \phi, \quad \phi' = -2E\psi.$$

This is implemented in `f(psi, phi, E)`, which returns  $(\psi', \phi')$ .

**RK4 rk4(f, psi, phi, E, h)** performs a single RK4 step for the first-order system at a fixed energy  $E$  and spatial step  $h$ . It the slopes

$$(k_1, k_2, k_3, k_4),$$

for both  $\psi$  and  $\phi$ , and updates  $(\psi, \phi)$

$$y_{n+1} = y_n + \frac{h}{6} (k_1 + 2k_2 + 2k_3 + k_4),$$

- **Global settings:** define  $L, m, \hbar$ , initial conditions  $(\psi_0, \phi_0)$ , and integration step  $h$ .
- **System function f:** encodes the first-order system  $(\psi', \phi')$  used by the integrator.
- **Integrator rk4:** advances  $(\psi, \phi)$  by one step  $h$  for a fixed  $E$ .
- **solveRK4:** integrates from  $x = 0$  to  $x = L$  and returns  $\psi(L; E)$  for the given  $E$ .
- **Energy scan:** loops  $E = 0, 0.1, 0.2, \dots$ ; collects intervals when  $\psi(L)$  flips sign.
- **Bisection:** refines each interval  $[a, b]$  to a tight interval around the root of  $\psi(L; E)$ .
- **Extraction of eigenvalues:** applies `bisection` to the first few brackets to get  $E_{n(\text{num})}$ .
- **Comparison and errors:** computes  $E_{n(\text{analytic})}$  and prints absolute/relative errors for each  $n$ .

**Analytical Energies and Error Reporting** For the infinite well with  $L = m = \hbar = 1$ , the analytical energies are

$$E_{n(analytic)} = \frac{n^2\pi^2}{2}.$$

We calculate

$$E_{analytic}(n) = 0.5 \cdot \pi^2 \cdot n^2$$

provides  $E_{n(analytic)}$ , and the script prints, for each  $n$ ,

$$\text{Absolute Error} = |E_{n(numerical)} - E_{n(analytic)}|, \quad \text{Relative Error} = \frac{|E_{n(numerical)} - E_{n(analytic)}|}{E_{n(analytic)}}.$$

We find the following values :

State n=1

Numerical E\_n = 4.934863281249999

Analytical E\_n = 4.934802200544679

Absolute Error = 6.10807053202933e-05

Relative Error = 1.2377538721521913e-05

State n=2

Numerical E\_n = 19.73925781250001

Analytical E\_n = 19.739208802178716

Absolute Error = 4.901032129467353e-05

Relative Error = 2.4828918821337973e-06

State n=3

Numerical E\_n = 44.41328125000037

Analytical E\_n = 44.41321980490211

Absolute Error = 6.144509825389832e-05

Relative Error = 1.3834866853566045e-06

State n=4

Numerical E\_n = 78.95722656249978

Analytical E\_n = 78.95683520871486

Absolute Error = 0.00039135378492005657

Relative Error = 4.956553588876127e-06

So we have relative errors of order  $\approx 10^{-6}$

## 8 FEM

The two-dimensional time-independent Schrodinger equation governs the stationary states of a quantum particle confined within a potential domain:

$$-\nabla^2\psi(x, y) + V(x, y)\psi(x, y) = E\psi(x, y),$$

### MATLAB

The MATLAB script defines a custom smooth, irregular geometry, generated via spline interpolation of control points. The partial differential equation is constructed in the following way :

$$-\nabla^2u + au = \lambda u,$$

which corresponds to the Schrödinger equation with an effective potential proportional to the coefficient  $a$ .

We also have the following coefficient for the system:

$$m = 0, \quad d = 1, \quad c = 1, \quad a = \frac{b^2}{4}, \quad f = 0,$$

where  $b$  is a potential scaling parameter. Eigenvalues  $\lambda$  represent the discrete energy levels  $E_n$ , and eigenvectors correspond to the spatial wavefunctions  $u_n(x, y)$ .

The eigenproblem was solved over the interval  $[0, 100]$ , and the first four eigenfunctions were plotted for both Dirichlet and Neumann boundary conditions. We control the mesh refinement by setting the maximum element size  $H_{\max} = 0.05$ , ensuring appropriate spatial resolution of the lowest energy states.

## Results

Figure 6 presents the computed eigenmodes under Dirichlet BC. At the bounds the wavefunction vanishes, just like it would for an infinite potential well. The lowest eigenstate ( $n = 1$ ) represents the ground state.

Under Neumann boundary conditions, the derivative of the wavefunction normal to the boundary is zero. Physically, this models a particle confined in a region with reflective or symmetry boundaries. As a result, the ground state no longer needs to vanish at the boundary, and the corresponding energy eigenvalue is lower compared to the Dirichlet case.

Comparing both boundary cases reveals that:

- Dirichlet conditions produce higher energy eigenvalues since the wavefunction is constrained to be zero at the boundary.
- Neumann conditions permit finite boundary amplitudes, reducing the confinement and lowering the ground state energy.

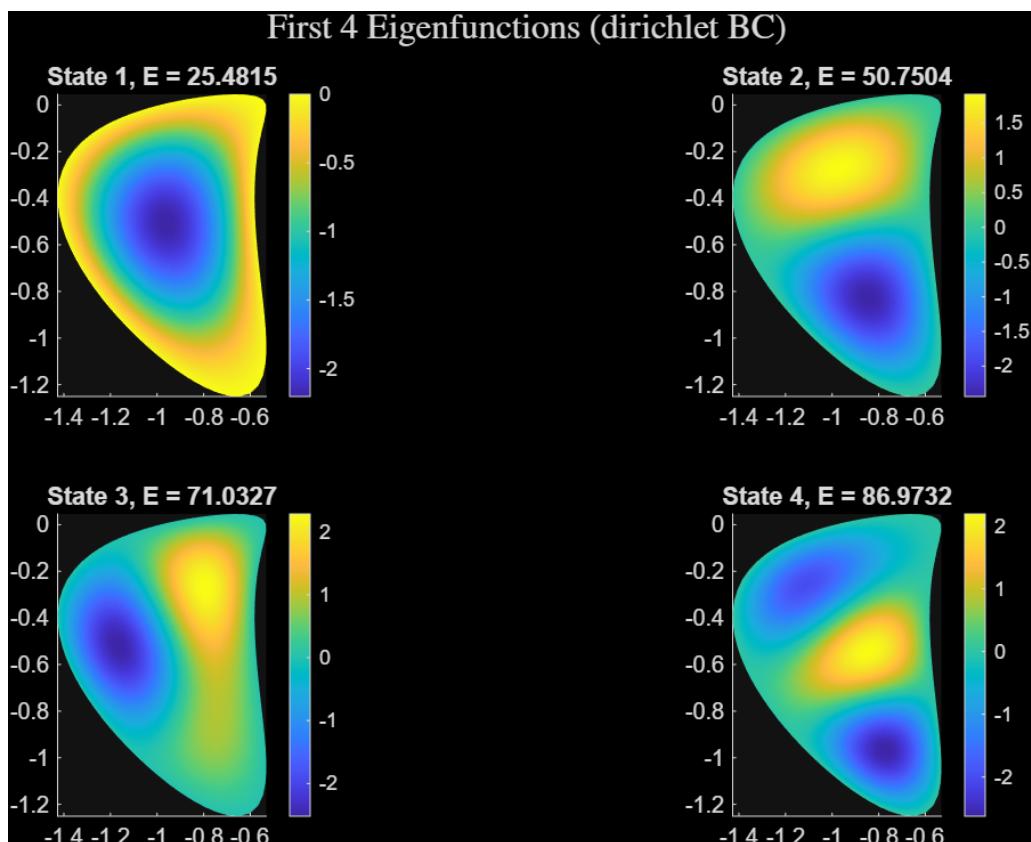


Figure 6: First four eigenfunctions computed with Dirichlet BC. The eigenenergies correspond to discrete stationary states confined by rigid boundaries.

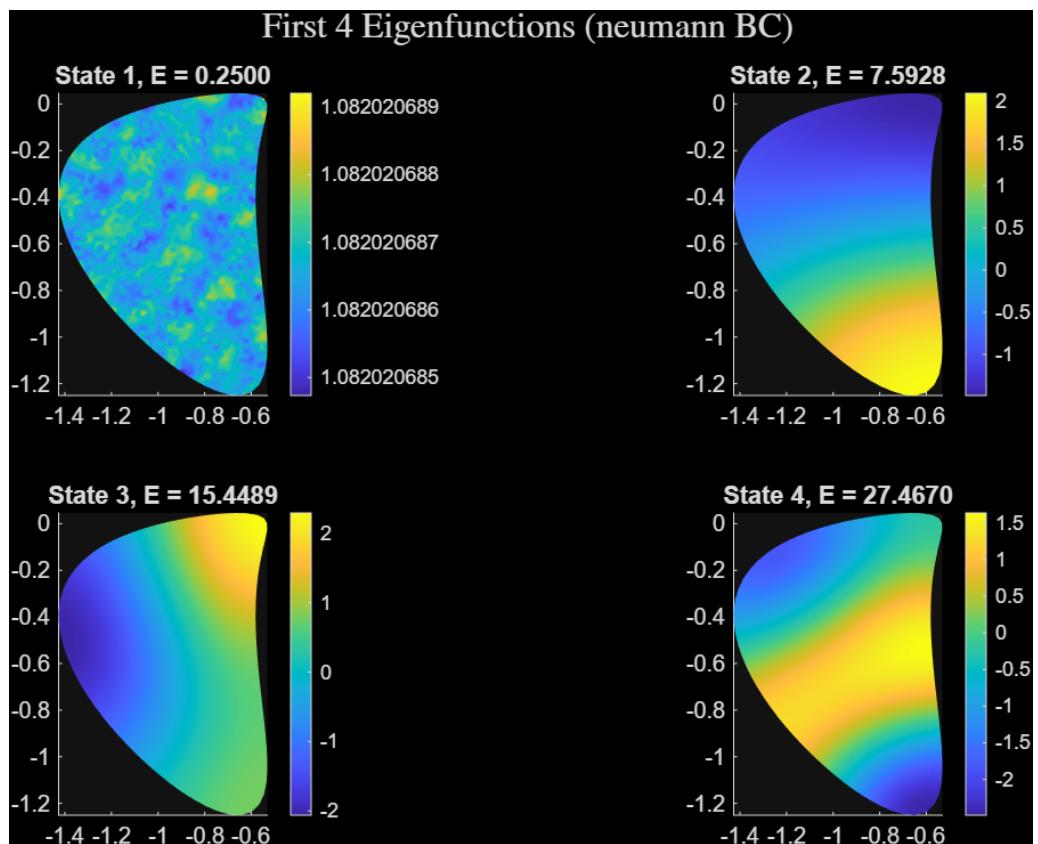


Figure 7: First four eigenfunctions computed with Neumann boundary conditions. The boundary allows zero-flux conditions, resulting in lower energies and smoother eigenmodes near the edges.

## 9 Dynamics

We are now studying the Lorenz System

$$\begin{aligned}\dot{x} &= \sigma(y - x), \\ \dot{y} &= rx - y - xz, \\ \dot{z} &= xy - bz,\end{aligned}\tag{1}$$

where  $\sigma$ ,  $r$ , and  $b$  are system parameters. For  $\sigma = 10$ ,  $b = 8/3$ , and  $r = 28$ , the system exhibits deterministic chaos. We start the problem by numerically integrating the Lorenz equations, then we analyze the stability of fixed points through Jacobian eigenvalues, estimate the Lyapunov exponent, and construct the Lorenz return map  $z_{n+1} = f(z_n)$  to verify the instability of limit cycles.

### Numerical Integration (RK4)

The integration is carried out using the classical fourth-order Runge–Kutta (RK4) method:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4),\tag{2}$$

where  $k_i$  are intermediate slope estimates computed from the Lorenz vector field. The function `rk4` in the code performs one RK4 step, while `solveRk4` integrates the system for a chosen number of iterations  $n$ , returning arrays of  $(x, y, z, t)$ .

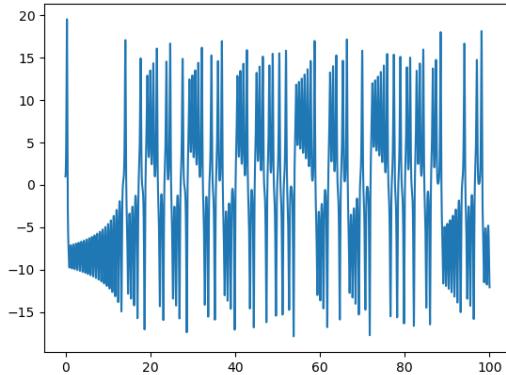


Figure 8:  $x(t)$

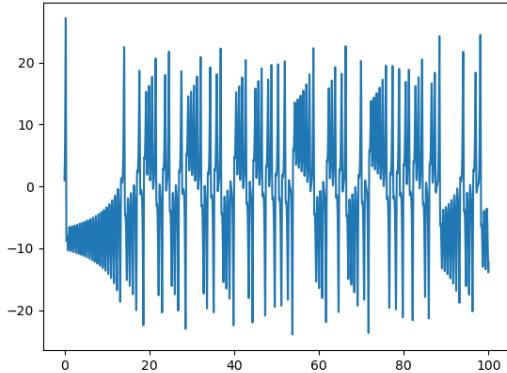


Figure 9:  $y(t)$

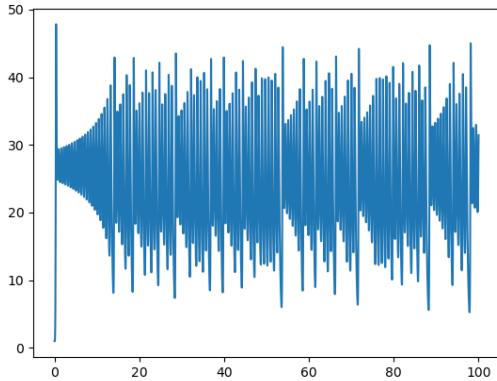


Figure 10:  $z(t)$

$x(t), y(t), z(t)$  plots

## Fixed Points and Jacobian Stability

We solve for the fixed points, using the bisection method, where we defined  $g(x) = r - 1 - x^2/b$ , trying to find the roots for that function for  $r = 28$  my fixed points are  $(-8.640987597877146, -8.640987597877146, 28)$  For  $r < 1$ , only the origin exists and is stable. For  $r > 1$ , two symmetric fixed points emerge via a pitchfork bifurcation, and the all the fixed points found in from  $r \in [2, 31]$  are unstable. The local stability of each fixed point is determined by the Jacobian matrix

$$J(x, y, z, r) = \begin{pmatrix} -\sigma & \sigma & 0 \\ r - z & -1 & -x \\ y & x & -b \end{pmatrix}. \quad (3)$$

We evaluate the eigenvalues of  $J$  at each fixed point. If all real parts are negative, the fixed point is stable; if any are positive, it is unstable. For the canonical parameters ( $r = 28$ ), all equilibrium are unstable, consistent with the presence of a chaotic attractor.

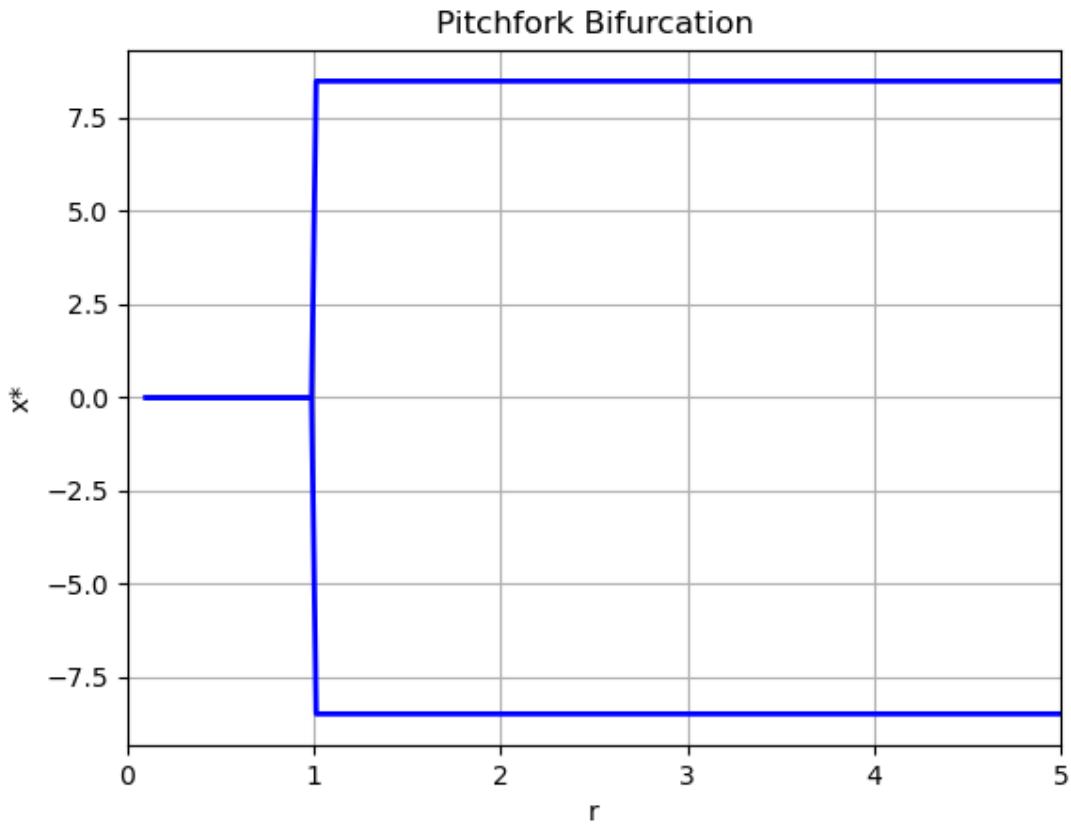


Figure 11: Bifurcation diagram

## Lyapunov Exponent Estimate

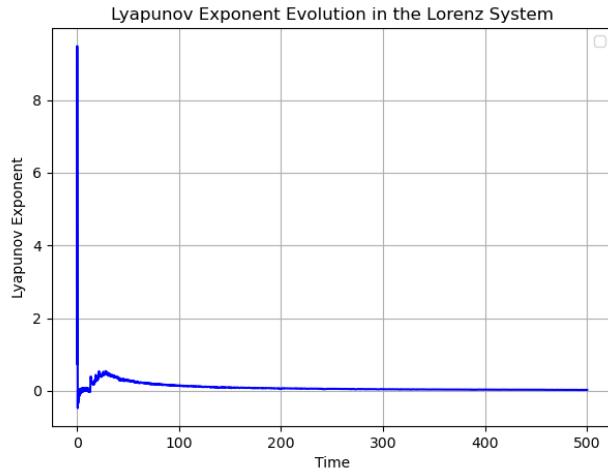
We compute an approximate Lyapunov exponent by evolving two trajectories with nearly identical initial conditions:

$$\delta(t) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2},$$

and evaluating

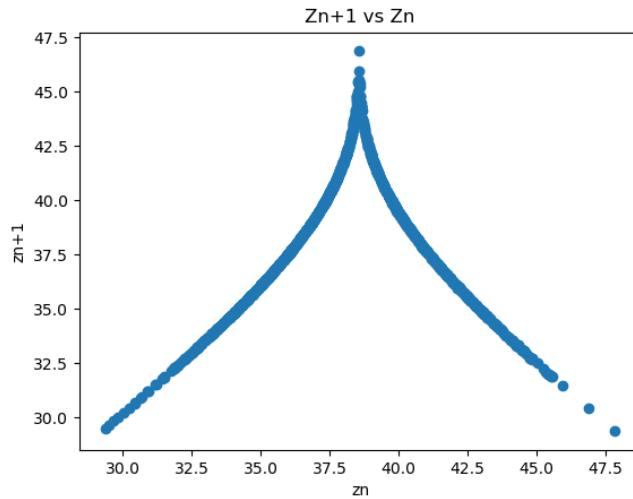
$$\lambda(t) = \frac{1}{t} \ln \frac{\delta(t)}{\delta(0)}.$$

Initially,  $\lambda(t)$  is positive and fluctuates strongly, reflecting exponential divergence of nearby trajectories. As time increases,  $\lambda(t)$  gradually converges toward zero.



## Lorenz Return Map and Limit Cycle Stability

The sequence of local maxima of  $z(t)$ , denoted  $z_n$ , forms a one-dimensional map  $z_{n+1} = f(z_n)$ . We get all the maxima. Then we plot  $z_{n+1}$  versus  $z_n$  (these are the consecutive maximum values).



## 10 Logistic Map

The logistic map is the one-parameter iterated map

$$x_{n+1} = f_\mu(x_n) = \mu x_n(1 - x_n), \quad x_n \in [0, 1], \quad \mu \in [0, 4].$$

As  $\mu$  increases period doublings start to appear and then at a specific  $\mu$ , the system becomes chaotic and exhibits a pseudo-random behavior.

$$\delta = \lim_{n \rightarrow \infty} \frac{\Delta_n}{\Delta_{n+1}} = 4.669201609\dots,$$

**Code.** Trajectory generator.

- `x_np1(x, mu)` returns one logistic iteration  $f_\mu(x)$ .
- `logistic_map(x0, mu)` first advances an internal state `xn` by `trans` steps to reduce transient effects and store the converging values, then builds a list `xar` by repeated calls to `x_np1`.

**Period detection.** we test periods  $T \in \{1, 2, 4, 8, 16, 32, 64\}$  by searching the converged part of the trajectory and checking where recurrences almost happen separated by  $T$  samples:

- The outer index `i` loops over the values of the trajectory, the list for `x`.
  - For a given  $T$ , a necessary condition for period  $T$  is  $|x[i] - x[i + T]| < \varepsilon$  with a tolerance  $\varepsilon = 10^{-12}$ .
  - The inner loop over `k` verifies that all points repeat with same  $T$ .
  - The function returns the first detected  $T$ , else it returns none:
- bifurcation*
- create  $\mu$  `r_vals = linspace(3, 3.57, 10000)`.
  - For each  $\mu$  run the period detection. Whenever the detected period jumps from the current value to a strictly larger one, store a bifurcation at that  $\mu$ .

**Results** We get the following detected bifurcation points (period  $\rightarrow$  parameter):

2	3.001824
4	3.445614
8	3.542523
16	3.563843
32	3.568518
64	3.569601

Forming  $\Delta_n = \mu_n - \mu_{n-1}$  and the ratios  $\Delta_n / \Delta_{n+1}$  for  $n = 2, 3, 4$  yield

$$\frac{\Delta_2}{\Delta_3} \approx 4.5455, \quad \frac{\Delta_3}{\Delta_4} \approx 4.5610, \quad \frac{\Delta_4}{\Delta_5} \approx 4.3158,$$

comparing it with the Feigenbaum constant  $\delta \approx 4.66920$ , we see that we are a little bit off.

A similar framework can be extended to estimate  $\alpha$  from the vertical scalings in the bifurcation diagram.

## 11 What do 1D Maps have to do with science

This system is a set of three coupled nonlinear differential equations. It is defined as

$$\begin{aligned}\dot{x} &= -y - z, \\ \dot{y} &= x + ay, \\ \dot{z} &= b + z(x - c),\end{aligned}$$

where  $a$ ,  $b$ , and  $c$  are system parameters. We choose,  $a = b = 0.2$  and vary  $c$ , which serves as the control parameter that determines the system's dynamic regime. For low values of  $c$ , the system exhibits periodic motion, while for higher values, it undergoes a period-doubling that eventually lead to chaos, which resembles logistic map, where we start by period doubling then chaos and pseudo-random behavior.

### Numerical implementation

The equations were solved using the fourth-order Runge–Kutta (RK4) method with a time step  $h = 0.01$ . The numerical steps can be summarized as follows:

1. Define the system of equations through functions  $f_x(y, z)$ ,  $f_y(x, y)$ , and  $f_z(x, z)$ .
2. Apply the RK4 update rule:

$$x_{n+1} = x_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4),$$

with similar updates for  $y$  and  $z$ , where each  $k_i$  is computed using intermediate evaluations of the system.

3. Integrate for a large number of iterations ( $n = 10^5$ ) to ensure the transient behavior has decayed, then analyze the steady-state oscillations.

To study the transition to chaos:

- The  $(x, y)$  phase portraits were plotted for different  $c$  values ( $c = 2.5, 3.5, 4.5$ ).
- Successive maxima of  $x(t)$  were extracted ( $x_{n+1}$  vs.  $x_n$ ) for  $c = 5$ .
- A bifurcation-like diagram was constructed by plotting the maxima of  $x$  as a function of  $c$  in the range  $2.5 \leq c \leq 3.5$  (and also for  $2.5 \leq c \leq 6$  ).

### Results and Discussion

Figure 12 shows the phase portrait for  $c = 2.5$ , which corresponds to a single closed trajectory in phase space. When  $c$  increases to 3.5 (Figure 13), the system completes two loops before returning to its starting point, signifying a period-2 orbit. At  $c = 4.5$  (Figure 14), the motion becomes more complex, suggesting further period-doubling and some chaos.

The Lorenz-style map ( $x_{n+1}$  versus  $x_n$ ) for  $c = 5$  shows a non-linear structure similar to the logistic map, confirming the chaotic behavior where deterministic dynamics generate unpredictable outcomes.

Finally, the bifurcation diagram (Figure 16) reveals a tree-like pattern: as  $c$  increases, single points split into two, then four, and so on, until it gets chaotic. .

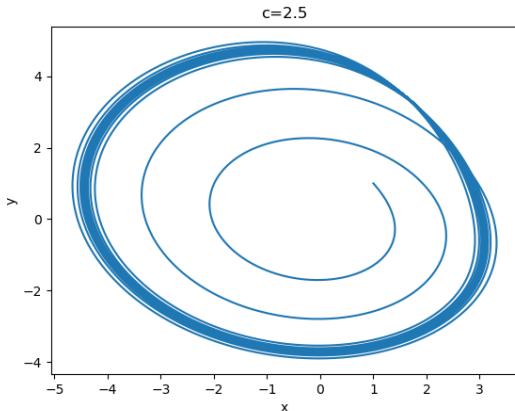


Figure 12: Phase portrait for  $c = 2.5$ : simple limit cycle (period-1).

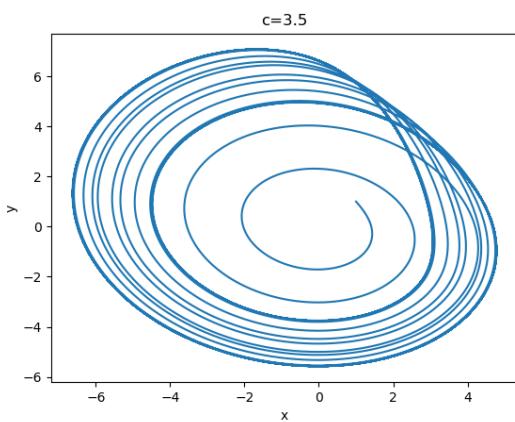


Figure 13: Phase portrait for  $c = 3.5$ : period-2 limit cycle.

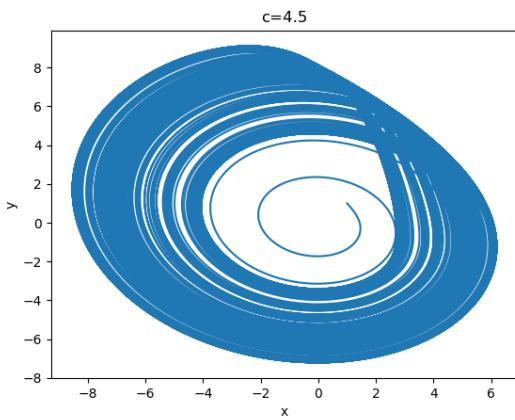


Figure 14: Phase portrait for  $c = 4.5$ : onset of chaotic oscillations.

## Conclusion

The numerical experiments confirm that this system undergoes a bifurcations as the control parameter  $c$  increases, leading eventually to chaotic motion. I tried plotting the bifurcation diagram for  $r$  in  $[2.5, 6]$ , where we see a clearer tree like figure, a mixture of

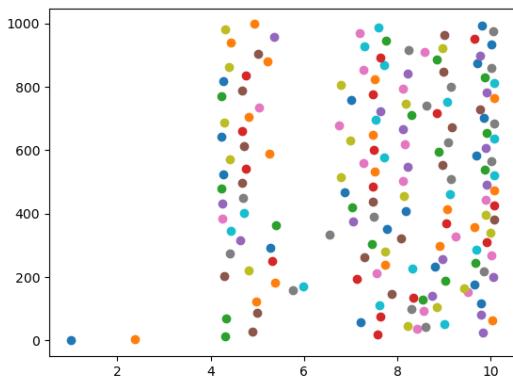


Figure 15: Successive maxima map ( $x_{n+1}$  vs  $x_n$ ) for  $c = 5$ .

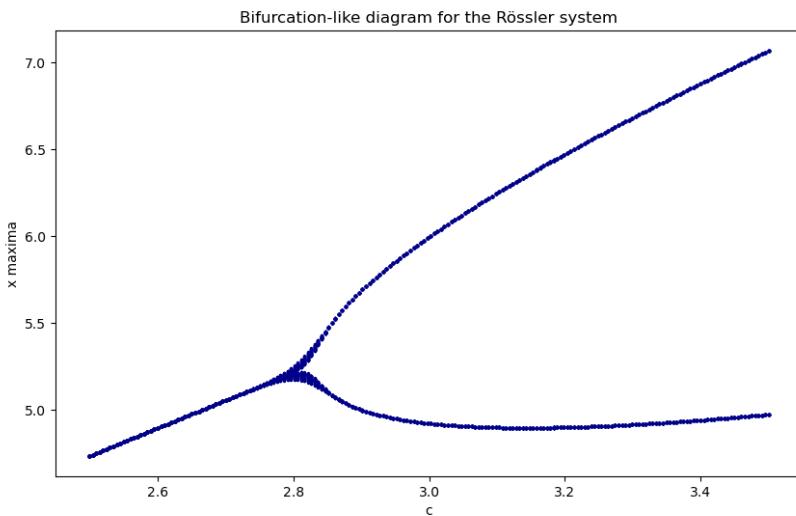


Figure 16: Bifurcation-like diagram for the system in the interval  $[2.5, 3.5]$  for  $r$ . The period-doubling route to chaos resembles that of the logistic map.

period doublings and some chaos within some  $c$  values, and as  $c$  increases the system undergoes clear chaotic motion.

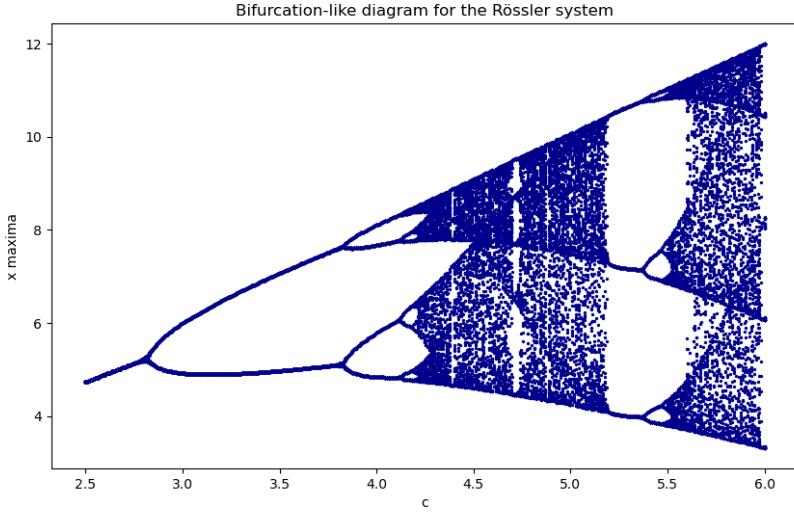


Figure 17: Bifurcation diagram for the system until  $r$  between 2,5 and 6. The period-doubling route to chaos resembles that of the logistic map.

## 12 SVD and Coherent Structures

We have a stream-function

$$\psi(x, y, t) = A \sin(\pi f(x, t)) \sin(\pi y), \quad f(x, t) = a(t)x^2 + b(t)x,$$

with

$$a(t) = \varepsilon \sin(\omega t), \quad b(t) = 1 - 2\varepsilon \sin(\omega t).$$

Because in 2D,  $u = -\partial_y \psi$  and  $v = \partial_x \psi$ , the velocity field is

$$u(x, y, t) = -A\pi \cos(\pi y) \sin(\pi f(x, t)),$$

$$v(x, y, t) = A\pi \sin(\pi y) \cos(\pi f(x, t)) \partial_x f(x, t) = A\pi \sin(\pi y) \cos(\pi f(x, t)) (2a(t)x + b(t)).$$

The flow is on  $[0, 2] \times [0, 1]$  grid.

**Helper functions** We define the following function:  $a(t)$ ,  $b(t)$ ,  $f(x, t)$ ,  $dpsix(x, y, t)$  and  $dpsi(y, x, t)$

**Create grid and Generate u and v for 200 frames** We create the grid with points spaced at a distance 0.1 from one another. We then generate U and V containing velocity component at each time, we have a total of 200 snapshots.

**D matrix** We then create the matrix D, that contains the stacked components of U and V, each column representing a snapshot and each row corresponding to a spatial component of both velocities, so it has a dimension of  $2 \times 21 \times 11 \times 200$ .

we then remove the temporal mean to isolate coherent fluctuations:

$$\tilde{D} = D - \bar{D}, \quad \tilde{D}_{ij} = D_{ij} \frac{1}{T} \sum_{k=1}^T D_{ik}$$

## SVD

For the singular value decomposition step, we start by computing the temporal covariance

$$C_t = \tilde{D}^\top \tilde{D},$$

then its eigendecomposition  $C_t = V \Lambda V^\top$  with  $\Lambda = \text{diag}(\lambda_1 \geq \dots \geq \lambda_T \geq 0)$ .

The singular values and spatial modes follow from

$$\sigma_i = \sqrt{\lambda_i}, \quad U = \tilde{D} V \Sigma^{-1} \quad (\Sigma = \text{diag}(\sigma_i)).$$

$U$  is a spatial coherent structure and  $V$  is its temporal coefficient. Modes with the largest  $\sigma_i$  maximize variance.

## Modeling the temporal basis

Assume most energetic mode are the first 3.

$$D_r = \tilde{D}_r + \bar{D}.$$

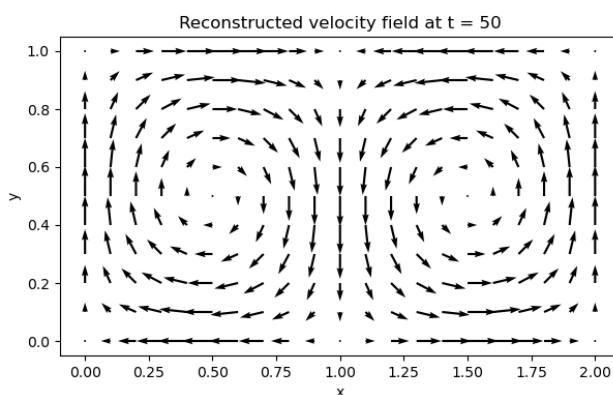
Each column of  $D_r$  is a time-dependent approximation of the full state. Split each column back into  $u_r$  and  $v_r$  by unstacking the first  $NM$  entries (for  $u$ ) and the next  $NM$  (for  $v$ ), then reshape to the grid.

we keep the first 3, we get:  $U_r$ : has dimension  $(2 \times 21 \times 11 \times 3)$   $V_r$ : has dimension  $(200 \times 3)$   $\Sigma_3$ : has dimension  $(3 \times 3)$

Then

$$D_{rec} = U_r \Sigma_r V_r^\top + D_{\text{mean}},$$

which gives the dataset of shape  $(2 \times 21 \times 11, 200)$ . Each column represents a reconstructed snapshot of the velocity field.



**Snapshot at  $t = 50$  Quiver plot.** Two counter-rotating gyres

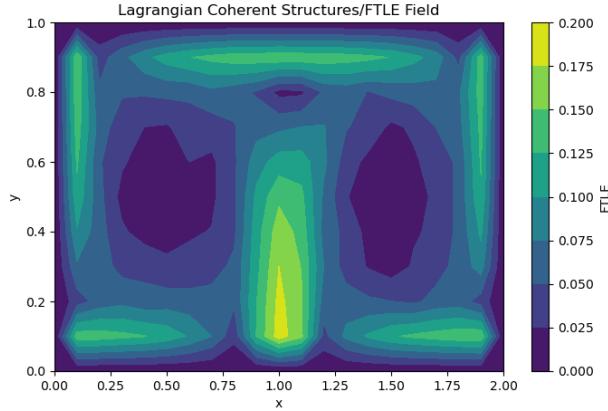
## Computing Finite Time Lyapunov Exponent

We start by solving for the particles from  $(X_0, Y_0)$  to  $(X_f, Y_f)$ , we use Rk4. The function velocity to find the nearest points on the grid for x and y, and closest time index, and returns the velocities at that exact time and position (from the spatial and temporal

matrices U and V). we use this velocity function at each step in rk4, to find the slopes k1,k2,k3,k4... Then we integrate the rk4 step for every point on the grid, for a total of time = 10. We store the final position in Xf and Yf, so that we can eventually see how they changed. We know that

$$FTLE = \frac{\ln \sqrt{\lambda_{max}}}{T}$$

where T = 10 and  $\lambda_{max}$  is the largest eigenvalue of the cauchy stress tensor  $C = F^T F$ , F is the jacobian of the flow map, we calculate the components of F using central differences. After finding all of the above and computing the finite time lyapunov exponent. We plot it using `plt.contourf()` (creates a filled contour plot), where the colors represent the FTLE value at specific locations.



# A Appendix: Python Codes

## A.1 Problem 1

```
#Classical Mechanics
import numpy as np
import matplotlib.pyplot as plt

g = 9.81
l=1
R=1
w=np.pi/4
h=0.1
T=2 *np.pi/w
teta0=0
tetadot0 =0

def f(x1,x2,t):
    v1=x2
    v2 = -g/l *np.sin(x1) + R*w**2/l *np.cos(x1-w*t)
    return v1,v2
#rk4 step
def rk4(x1,x2,t):
    k1=f(x1,x2,t)
    k2 = f( x1+h*k1[0]/2, x2+h*k1[1]/2 ,t+h/2)
    k3 = f( x1+h*k2[0]/2, x2+h*k2[1]/2 ,t+h/2)
    k4 = f( x1+h*k3[0] , x2+h*k3[1] ,t +h)
    x1np1 = x1 + h*(k1[0]+2*k2[0]+2*k3[0]+k4[0])/6
    x2np1 = x2 + h*(k1[1]+2*k2[1]+2*k3[1]+k4[1])/6
    tnp1 = t + h
    return x1np1,x2np1,tnp1
# rk4 integration for time t
def solverk4(t_end):
    t0=0
    t = [t0]
    x1= [teta0]
    x2= [tetadot0]
    while t[-1]<t_end:
        tc = t[-1]
        x1c = x1[-1]
        x2c = x2[-1]
        x1_new, x2_new, t_new = rk4(x1c, x2c, tc)
        x1.append(x1_new)
        x2.append(x2_new)
        t.append(t_new)
    return x1,x2,t
theta,thetadot,t = solverk4(20)
#bisection
def bisection(vals, indices):
    ai, bi = indices
    while abs(ai - bi) > 1:
        midi = (ai + bi) // 2
        if vals[ai] * vals[midi] < 0:
            bi = midi
        else:
            ai = midi
    return ai
```

```

#ranges for phase space points (theta,thetadot)=(0,0)
def findRanges(theta_vals):
    ranges = []
    for i in range(len(theta_vals)-1):
        if theta_vals[i] * theta_vals[i+1] < 0:
            ranges.append((i, i+1))
    return ranges
#we find ranges in which theta=0, then use bisection to find the
#corresponding exact point with the corresponding index
# then we go over the indices that we found for which theta = 0, and
# check if thetadot is also zero, if yes, then that is our fixed point
ranges = findRanges(theta)
theta0_indices = [] #stores the indices for which theta = 0
for r in ranges:
    idx = bisection(theta, r)
    theta0_indices.append(idx)

thetadot_at0 = []
for i in theta0_indices:
    thetadot_value = thetadot[i]
    thetadot_at0.append(thetadot_value)
ranges_thetadot = findRanges(thetadot_at0)

common_indices = []
for r in ranges_thetadot:
    idx = bisection(thetadot_at0, r)
    if abs(thetadot_at0[idx]) < 1e-3:
        common_indices.append(theta0_indices[idx])
fixed_pts = []
for i in common_indices:
    theta_value = theta[i]
    thetadot_value = thetadot[i]
    pt = (theta_value, thetadot_value)
    fixed_pts.append(pt)

print("Fixed points:", fixed_pts)
eps = 0.1
for i, fp in enumerate(fixed_pts):
    th0, thd0 = fp
    theta1, thetadot1, t1 = solverk4(5)
    teta0 = th0 + eps
    tetadot0 = thd0
    theta2, thetadot2, t2 = solverk4(5)
    teta0 = th0
    tetadot0 = thd0 + eps
    theta3, thetadot3, t3 = solverk4(5)
    final_th1 = abs(theta1[-1])
    final_thd1 = abs(thetadot1[-1])
    final_th2 = abs(theta2[-1])
    final_thd2 = abs(thetadot2[-1])
    final_th3 = abs(theta3[-1])
    final_thd3 = abs(thetadot3[-1])
    if final_th2 > 0.1 or final_thd2 > 0.1 or final_th3 > 0.1 or
       final_thd3 > 0.1:
        print(f"Fixed point {i+1}: UNSTABLE")
    else:
        print(f"Fixed point {i+1}: STABLE")

```

```
#plot phase space
teta0=0
tetadot0=0
theta,thetadot,t = solverk4(20)
plt.figure(figsize=(6, 6))
plt.plot(theta , thetadot)
plt.xlabel(r'\theta [rad]')
plt.ylabel(r'\dot{\theta} [rad/s]')
plt.title('Phase Space Trajectory')
plt.axis('equal')
plt.show()
```

## A.2 Problem 2

```
import numpy as np
import matplotlib.pyplot as plt
g= 10
R =1
h=0.01
theta0 = np.pi/2
thetadot0 = 0
m=1
def f(theta,thetadot):
    return [thetadot, -g/R*np.sin(theta)]
def rk4(theta,thetadot):
    k1 = f(theta,thetadot)
    k2 = f(theta + h/2 * k1[0],thetadot + h/2 * k1[1] )
    k3 = f(theta + h/2 * k2[0], thetadot + h/2 * k2[1])
    k4 = f(theta + h/2 * k3[0],thetadot + h/2 * k3[1])
    thetanp1 = theta + h/6 * ( k1[0] + 2*k2[0]+2*k3[0]+k4[0])
    thetadnp1 = thetadot + h/6 * ( k1[1] + 2*k2[1]+2*k3[1]+k4[1])
    return thetanp1,thetadnp1
t_values = np.linspace(0,10,1000)
def solveRk4(theta0,thetadot0):
    theta,thetadot = [theta0],[thetadot0]
    for t in t_values[1:]:
        tet = theta[-1]
        tetdot = thetadot[-1]
        if tet < 0:
            tet = 0
            tetdot = -tetdot
        if tet > np.pi:
            tet = np.pi
            tetdot = -tetdot
        currVals = rk4(tet,tetdot)
        theta.append(currVals[0])
        thetadot.append(currVals[1])
    return theta,thetadot

theta,thetadot = solveRk4(theta0,thetadot0)
theta = np.array(theta)
thetadot = np.array(thetadot)
#find constraint force F = mR \dot{\theta}^2 + mgcos\theta
F = m * R * thetadot ** 2 + m * g * np.cos(theta)
plt.figure()
plt.plot(theta,F)
plt.show()
```

### A.3 Problem 3

```
#Two Body Problem
import numpy as np
m1 = 1
m2 = 2
G = 1
mu = (m1 * m2) / (m1 + m2)
M = m1 + m2
def f(r,v,t):
    r_rel = [r[0][0]-r[1][0],r[0][1] - r[1][1]]
    rd = np.sqrt(r_rel[0]**2 + r_rel[1]**2)
    k = G * m1 * m2 / (rd**3)
    a1 = [-k* r_rel[0] / m1, -k * r_rel[1] / m1]
    a2 = [k * r_rel[0] / m2, k * r_rel[1] / m2]
    a = [a1,a2]
    return v , a

empty = [[0,0],[0,0]]
def rk4_step(r, v, h):
    k1_r, k1_v = f(r, v, 0)
    r2 = [[0,0],[0,0]]
    v2 = [[0,0],[0,0]]
    for i in range(2):
        for j in range(2):
            r2[i][j] = r[i][j] + h/2* k1_r[i][j]
            v2[i][j]=v[i][j]+ h / 2 * k1_v [i] [j]
    k2_r, k2_v = f(r2, v2, 0)
    r3 = [[0,0],[0,0]]
    v3 = [[0,0],[0,0]]
    for i in range(2):
        for j in range(2):
            r3 [i][j] = r[i][j] + h/2 * k2_r [i][j]
            v3[i][j]= v[i][j] + h/2 * k2_v[i][j]
    k3_r, k3_v = f(r3, v3, 0)
    r4 = [[0,0],[0,0]]
    v4 = [[0,0],[0,0]]
    for i in range(2):
        for j in range(2):
            r4[i][j]= r[i][j] + h* k3_r[i][j]
            v4[i][j]= v[i][j]+h*k3_v[i][j]
    k4_r, k4_v = f(r4, v4, 0)
    rnp1 = [[0,0],[0,0]]
    vnp1 = [[0,0],[0,0]]
    for i in range(2):
        for j in range(2):
            rnp1[i][j] = r[i][j] + h/6 * (k1_r[i][j] + 2*k2_r[i][j] +
                2*k3_r[i][j] + k4_r[i][j])
            vnp1[i][j]= v[i][j] + h/6* (k1_v[i][j]+ 2*k2_v[i][j]+ 2*
                k3_v[i][j] + k4_v[i][j])
    return rnp1, vnp1

r10= [1.0, 0]
r20=[-0.5, 0]
v10=[0, 0.8]
v20=[0, -0.4]
h= 0.01
t_end= 20
```

```

n_steps= int(t_end / h)
r1_traj=[]
r2_traj=[]
cm_traj=[]
t=0
t=0
r=[r10 , r20]
v=[v10 , v20]

for i in range(n_steps):

    r1x, r1y = r[0][0], r[0][1]
    r2x, r2y = r[1][0], r[1][1]
    cm_x =(m1*r1x+ m2*r2x)/M
    cm_y = (m1*r1y + m2*r2y)/ M
    r1_traj.append([r1x, r1y])
    r2_traj.append([r2x, r2y])
    cm_traj.append([cm_x, cm_y])
    r, v = rk4_step(r,v, h)
    t += h

r1_traj = np.array(r1_traj)
r2_traj = np.array(r2_traj)
cm_traj = np.array(cm_traj)

print("Reduced mass = ", mu )
cm_pos = [cm_traj[-1, 0], cm_traj[-1, 1]]
print("CM position:", cm_pos)

r1_cm = r1_traj - cm_traj
r2_cm = r2_traj - cm_traj
r_rel = r1_traj - r2_traj
v1_rel = (r1_traj[1:] - r1_traj[:-1]) / h # velocity of body 1 (
    forward finite differences)
v2_rel = (r2_traj[1:] - r2_traj[:-1]) / h # velocity of body 2
v_rel = v1_rel - v2_rel
r = np.linalg.norm(r_rel[:-1], axis=1) #magnitudes
v = np.linalg.norm(v_rel, axis=1)
E = 0.5 * mu * v**2 - G * m1 * m2 / r
L = mu * r * v
print("values of L=",L)
print("values of E=",E)
ecc = np.sqrt(1 + (2 * E * L**2) / (mu * (G * m1 * m2)**2))
print("Values of e",ecc)
print(f"eccentricity: e={np.mean(ecc)}")

# E,L and e from ics
r_rel_ic = [r10[0] - r20[0], r10[1] - r20[1]]
v_rel_ic = [v10[0] - v20[0], v10[1] - v20[1]]
r_ic = np.sqrt(r_rel_ic[0]**2 + r_rel_ic[1]**2)
v_ic = np.sqrt(v_rel_ic[0]**2 + v_rel_ic[1]**2)
Eic = 0.5 * mu * v_ic**2 - G * m1 * m2 / r_ic
Lic = mu * abs(r_rel_ic[0] * v_rel_ic[1] - r_rel_ic[1] * v_rel_ic[0])
e = np.sqrt(1 + (2 * Eic * Lic**2) / (mu * (G * m1 * m2)**2))

print("Energy from ic: E=", Eic)
print("Energy error =", abs(E[-1] - Eic)/abs(Eic)*100, "%")
print("Angular Momentum from ic: L=", Lic)

```

```
print("Angular Momentum error = ",abs(L[-1] - Lic)/Lic*100,"%")
print("Eccentricity from ic: e = ",e)
print("eccentricity error = ",abs(ecc[-1] - e)/e*100,"%")
```

## A.4 Problem 4

```

#Kepler's first law
import numpy as np
# constants
mu = 1.0
r0 = 1.0
rdot0 = 0.0
thetadot0 = 1.2
# angular momentum and E from ICs
L = r0**2 * thetadot0
v2 = rdot0**2 + (r0 * thetadot0)**2
E = 0.5 * v2 - mu / r0
e_an = np.sqrt(1 + (2 * E * L**2) / mu**2)
# Binet equation
u0 = 1 / r0
up0 = -rdot0 / L
n=4000
theta0 = 0
theta1 = 4 * np.pi
dtheta = (theta1 - theta0) / n
theta = np.zeros(n + 1)
u = np.zeros(n + 1)
up = np.zeros(n + 1)
theta[0] = theta0
u[0] = u0
up[0] = up0
# DE
def f1(u, up):
    return up

def f2(u, up):
    return mu / L**2 - u
# RK4 integration
for i in range(n):
    k1u = dtheta * f1(u[i], up[i])
    k1up = dtheta * f2(u[i], up[i])
    k2u = dtheta * f1(u[i] + 0.5 * k1u, up[i] + 0.5 * k1up)
    k2up = dtheta * f2(u[i] + 0.5 * k1u, up[i] + 0.5 * k1up)
    k3u = dtheta * f1(u[i] + 0.5 * k2u, up[i] + 0.5 * k2up)
    k3up = dtheta * f2(u[i] + 0.5 * k2u, up[i] + 0.5 * k2up)
    k4u = dtheta * f1(u[i] + k3u, up[i] + k3up)
    k4up = dtheta * f2(u[i] + k3u, up[i] + k3up)
    u[i + 1] = u[i] + (k1u + 2*k2u + 2*k3u + k4u) / 6
    up[i + 1] = up[i] + (k1up + 2*k2up + 2*k3up + k4up) / 6
    theta[i + 1] = theta[i] + dtheta
r = 1 / u
#numerical results
r_min = np.min(r)
r_max = np.max(r)
e_num = (r_max - r_min) / (r_max + r_min)
a_num = 0.5 * (r_max + r_min)
E_num = -mu / (2 * a_num)
# compare
print("r_min=", r_min)
print("r_max=", r_max)
print("e_ee=", e_num)
print("a_ee=", a_num)

```

```
print("E_num=", E_num)
print("abs_error_for_e=", abs(e_num - e_an))
print("abs_error_for_E=", abs(E_num - E))
```

## A.5 Problem 5

```
#Kepler 2nd
import numpy as np
#parameters
G = 6.67430e-11
M = 1.989e30
m = 5.972e24
e = 0.0167
a = 1.496e11
r_p = a * (1 - e)
r_a = a * (1 + e)
v_p = np.sqrt(G * M * (1 + e) / (a * (1 - e)))
#ics
x0 = r_p
y0 = 0.0
vx0 = 0.0
vy0 = v_p
T = 2 * np.pi * np.sqrt(a**3 / (G * M)) #period
h = T / 10000 # time steps
num_steps = 10000
state = np.array([x0, y0, vx0, vy0])
tarr = [] # time arra
traj = [] # trajectory
r = [] # just r values
av = [] # areal velocity valeus

def derivatives(state):
    x, y, vx, vy = state
    r = np.sqrt(x**2 + y**2)
    ax = -G * M * x / r**3
    ay = -G * M * y / r**3
    return np.array([vx, vy, ax, ay])

def rk4_step(state, h):
    k1 = derivatives(state)
    k2 = derivatives(state + 0.5 * h * k1)
    k3 = derivatives(state + 0.5 * h * k2)
    k4 = derivatives(state + h * k3)
    return state + (h / 6.0) * (k1 + 2*k2 + 2*k3 + k4)

for i in range(num_steps):
    x, y, vx, vy = state
    rvals = np.sqrt(x**2 + y**2)
    dAdt = 0.5 * (x * vy - y * vx)
    tarr.append(i * h)
    traj.append([x, y])
    r.append(rvals)
    av.append(dAdt)
    state = rk4_step(state, h)

tarr = np.array(tarr)
traj = np.array(traj)
r = np.array(r)
av = np.array(av)
swept_area = np.zeros(num_steps)

for i in range(1, num_steps):
```

```

swept_area[i] = swept_area[i-1] + 0.5 * h * (av[i-1] + av[i])

i_peri = np.argmin(r) #index of perihelion
i_ap= np.argmax(r) # index of aphelion
dAdt_p = av[i_peri] #Areal velocity at perihelion
dAdt_a = av[i_ap] #Areal velocity at aphelion

print(f"Areal velocity at perihelion:{dAdt_p:.6e} m /s")
print(f"Areal velocity at aphelion:{dAdt_a:.6e} m /s")
print(f"Difference:{abs(dAdt_p-dAdt_a):.6e} m /s")
print(f"Relative difference:{abs(dAdt_p-dAdt_a)/dAdt_p*100:.10f}%" )

```

## A.6 Problem 6

```
import numpy as np
periods_days = np.array([1.7691, 3.5512, 7.1541, 16.689])
semi_major = np.array([421700, 670900, 1070412, 1882709])
G = 6.674e-11
T = periods_days * 86400 # seconds
a = semi_major * 1e3 # meters

x = a**3
y = T**2

# Least squares slope
k = np.sum(x *y)/np.sum(x**2)
M = (4*np.pi**2) /(G* k)

print(f"Jupiter's mass:{M:.3e}kg")
print(f"Accepted value: 1.898e27kg")
```

## A.7 Problem 7

```
#Quantum mechanics
import numpy as np
L=1
m=1
hb=1
E = 4
psi0=0
psiL=0
phi0 =1 #initial guess
x = 0
# here i need to solve rk4 while looping for different values for
# Energy eigenvalue
# then i try to find where it lande at x = L on the boundary
def f(psi,phi,E):
    dpsi = phi
    dphi = - 2 * E * psi
    return dpsi,dphi

def rk4(f,psi,phi,E,h):
    k1psi, k1phi = f(psi,phi,E)
    k2psi, k2phi= f(psi + h/2 *k1psi, phi + h/2 * k1phi,E)
    k3psi, k3phi= f(psi + h/2 *k2psi, phi + h/2 * k2phi,E)
    k4psi, k4phi= f(psi + h *k3psi, phi + h * k3phi,E)
    phinew = phi + h/6 * ( k1phi + 2*k2phi + 2*k3phi + k4phi)
    psinew = psi + h/6 * (k1psi + 2*k2psi + 2*k3psi + k4psi)
    return psinew,phinew

def solveRK4(E,h):
    x=0
    psiL = psi0
    phi = phi0
    while x < L:
        psiL,phi = rk4(f,psiL,phi,E,h)
        x+=h
    return psiL

psi = psi0
phi = phi0
h = 0.01
E=0 #initila guess for energy
#in this case we will work with shooting method for values of E
#lets star with
psiLold=0
rootRanges = []
i=0
while E < 100: #loop through different values of E
    x = 0
    psi = psi0
    phi = phi0
    psiL = solveRK4(E,h)
    if psiLold * psiL < 0:
        rootRanges.append([E-0.1,E])
        i += 1
    psiLold = psiL
    E += 0.1
# now we use bisection to find exact number in each
```

```

def bisection(a,b):
    mid = (a+b)/2
    if abs(a-b)< 10e-5:
        return max(a,b)
    psi_a = solveRK4(a,h)
    psi_mid = solveRK4(mid,h)
    if psi_a * psi_mid < 0:
        return bisection (a,mid)
    else:
        return bisection (b,mid)

def E_analytic(n):
    return 0.5 * (np.pi ** 2) * (n ** 2)
# Store the numerical values found earlier
E_num = E_num = [bisection(a, b) for a, b in rootRanges[:5]]
# Compute analytical and error values
for n, E_calc in enumerate(E_num, start=1):
    E_theo = E_analytic(n)
    abs_err = abs(E_calc - E_theo)
    rel_err = abs_err / E_theo
    print(f"State{n}={n}")
    print(f"Numerical\u20d7E_n\u20d7={E_calc}")
    print(f"Analytical\u20d7E_n\u20d7={E_theo}")
    print(f"Absolute\u20d7Error\u20d7={abs_err}")
    print(f"Relative\u20d7Error\u20d7={rel_err}")

```

## A.8 Problem 8: Matlab code for FEM

```
%% --- 2D Schr dinger Equation Solver (Finite Element) ---
% Author: [Your Name]
% Date: [Date]
% Description: Solves the 2D stationary Schr dinger equation
%               under Dirichlet or Neumann boundary conditions.

clear; clc; close all;

%% --- Parameters ---
b = 1; % potential scaling constant
rRange = [0, 100]; % eigenvalue range to search
nStates = 4; % number of eigenfunctions to plot
bcType = "dirichlet"; % change to "neumann" for Neumann BCs and "dirichlet"

%% --- Geometry Definition ---
% Using a custom smooth irregular shape
points = 0.95 * [0 -2.5 0 -1; -2.2 0 0.1 0];
sp = spmak(-4:8, [points points]);
f = fnplt(sp, [0 4]);
pg2 = polyshape(f(1,:), f(2,:), 'Simplify', true);
tr = triangulation(pg2);

% Plot geometry
figure;
plot(pg2);
axis equal;
title('Domain\u222a Geometry');

%% --- PDE Model Setup ---
model = createpde();
geometryFromMesh(model, tr.Points', tr.ConnectivityList');

% Choose Boundary Condition Type
if bcType == "dirichlet"
    applyBoundaryCondition(model, 'dirichlet', ...
        'Edge', 1:model.Geometry.NumEdges, 'u', 0);
elseif bcType == "neumann"
    applyBoundaryCondition(model, 'neumann', ...
        'Edge', 1:model.Geometry.NumEdges, 'g', 0, 'q', 0);
end

%% --- Mesh Generation ---
generateMesh(model, "Hmax", 0.05);
figure;
pdemesh(model);
title("Generated Mesh");

%% --- PDE Coefficients (Schr dinger-type) ---
% - u + a*u = *u -> specifyCoefficients(m,d,c,a,f)
specifyCoefficients(model, 'm', 0, 'd', 1, 'c', 1, ...
    'a', b^2/4, 'f', 0);

%% --- Solve Eigenvalue Problem ---
results = solvepdeeig(model, rRange);
```

```

%% --- Display Eigenvalues ---
disp('First few eigenvalues:');
disp(results.Eigenvalues(1:nStates));

%% --- Plot First nStates Eigenfunctions ---
cm = parula(128);
figure;
for k = 1:nStates
    subplot(2, 2, k);
    pdeplot(model, 'XYData', real(results.Eigenvectors(:,k)), ...
        'ColorMap', cm);
    axis equal tight;
    title(sprintf('State %d, E = %.4f', k, results.Eigenvalues(k)));
end

sgtitle(sprintf('First %d Eigenfunctions (%s BC)', nStates, bcType),
...
'Interpreter', 'latex');

```

## A.9 Problem 9

```
# Dynamics
import numpy as np
import matplotlib.pyplot as plt
sig = 10
r = 28
b= 8/3

def fx(x,y):
    return sig*(y-x)

def fy(x,y,z):
    return r*x - y - x*z

def fz(x,y,z):
    return x*y - b*z

def kak(x,y,z):
    return [fx(x,y), fy(x,y,z), fz(x,y,z)]

def rk4(x,y,z,h):
    k1 = kak(x,y,z)
    k2 = kak(x + h/2 * k1[0],y + h/2 * k1[1],z + h/2 * k1[2])
    k3 = kak(x + h/2 * k2[0],y + h/2 * k2[1],z + h/2 * k2[2])
    k4 = kak (x + h *k3[0],y + h*k3[1],z + h*k3[2] )
    xnp1= x + h/6 * (k1[0] + 2* k2[0] + 2*k3[0] + k4[0])
    ynp1 = y + h/6 * (k1[1] + 2* k2[1] + 2*k3[1] + k4[1])
    znp1 = z + h/6 * (k1[2] + 2* k2[2] + 2*k3[2] + k4[2])
    return xnp1,ynp1,znp1

h=0.01
def solveRk4(x0,y0,z0,n=1000):
    x = x0
    y = y0
    z = z0
    t0 =0
    x_list = [x0]
    y_list=[y0]
    z_list =[z0]
    t = [t0]
    for i in range(n):
        x,y,z = rk4(x,y,z,h)
        x_list.append(x)
        y_list.append(y)
        z_list.append(z)
        t.append((i+1) * h)
    return x_list,y_list,z_list,t

x,y,z,t = solveRk4(1,1,1,n=10000)

plt.figure()
plt.plot(t,x)
plt.show()
plt.figure()
plt.plot(t,y)
plt.show()
plt.figure()
```

```

plt.plot(t,z)
plt.show()

# Pitchfork bifurcation
x_star_pos = []
x_star_neg = []
r_vals = []
rootRanges = []
def g(x):
    return r - 1 - x*x/b

for i in range (len(x)-1):
    if g(x[i]) * g(x[i+1]) < 0:
        rootRanges.append([x[i],x[i+1]])

def bisection(a,b,r):
    if abs(g(a)-g(b)) < 1e-7:
        return a
    mid = (a + b)/2
    if g(a) * g(mid) < 0:
        return bisection(mid,a,r)
    else:
        return bisection(mid,b,r)

r_vals = np.linspace(0.1, 5, 200)
x_star_pos, x_star_neg = [], []

for rp in r_vals:
    if rp < 1:
        x_star_pos.append(0)
        x_star_neg.append(0)
    else:
        pos_root = bisection(0,20,rp)
        x_star_pos.append(pos_root)
        x_star_neg.append(-pos_root)

#Plot Bifurcation
plt.figure()
plt.plot(r_vals, x_star_pos, 'b-', linewidth=2)
plt.plot(r_vals, x_star_neg, 'b-', linewidth=2)
plt.xlabel('r')
plt.ylabel('x*')
plt.title('Pitchfork Bifurcation')
plt.grid(True)
plt.xlim(0, 5)
plt.show()

# Jacobian and eigenvalue analysis
def J(x, y, z, r):
    return np.array([
        [-sig, sig, 0],
        [r - z, -1, -x],
        [y, x, -b]
    ])

fixed_points = [(0,0,0)]
J_eigv = []

```

```

for r_val in range(1, 31):
    xstar = np.sqrt(b * (r_val - 1))
    fixed_points.append((xstar, xstar, r_val - 1))
    fixed_points.append((-xstar, -xstar, r_val - 1))
stability = False
for (xfp, yfp, zfp) in fixed_points:
    eigvals = np.linalg.eigvals(J(xfp, yfp, zfp, r_val))
    J_eigv.append(eigvals)
    if np.all(np.real(eigvals) < 0):
        print("Stable\u2022fixed\u2022point")
        stability = True
if stability == False:
    print("No\u2022stable\u2022Points")

for i in range(len(fixed_points)):
    if fixed_points[2] == 28:
        print(fixed_points[i])
# Lyapunov exponent calculation
x10, y10, z10 = 1.2, 1.2, 1.2
x1, y1, z1, t1 = solveRk4(x10, y10, z10, n=50000)
x20, y20, z20 = 1.200000001, 1.20001, 1.2
x2, y2, z2, t2 = solveRk4(x20, y20, z20, n=50000)

delta = np.sqrt((np.array(x1) - np.array(x2))**2 +
                 (np.array(y1) - np.array(y2))**2 +
                 (np.array(z1) - np.array(z2))**2)

delta0 = delta[0]
lyapunov_exp = []
for i in range(1, len(delta)):
    if delta[i] > 0 and t1[i] > 0:
        lyap = (1/t1[i]) * np.log(delta[i] / delta0)
        lyapunov_exp.append(lyap)
plt.figure(figsize=(7,5))
plt.plot(t1[1:len(lyapunov_exp)+1], lyapunov_exp, 'b-', linewidth=1.5)
plt.xlabel('Time')
plt.ylabel('Lyapunov\u2022Exponent')
plt.title('Lyapunov\u2022Exponent\u2022Evolution\u2022in\u2022the\u2022Lorenz\u2022System')
plt.legend()
plt.grid(True)
plt.show()

# Plot zn+1 vs zn -> z = f(z)
z_maxima = []
for i in range(1, len(z)-1):
    if z[i] > z[i-1] and z[i] > z[i+1]:
        z_maxima.append(z[i])

zn = z_maxima[:-1] #all elements except last two
znp1 = z_maxima[1:] #all except first two
plt.figure()
plt.plot(zn, znp1, 'o')
plt.xlabel('zn')
plt.ylabel('zn+1')
plt.title('Zn+1\u2022vs\u2022Zn')
plt.show()

```

## A.10 Problem 10

```

# Logistic map
import numpy as np
import matplotlib.pyplot as plt
#logistic map formula
x0 = 0.6
def x_npi(x,mu):
    return mu * x * (1-x)
n = 5000 #nb of data points
trans =4000 # transient, to ignore starting values until it converges.
# solving for logistic map's x_n values.
def logistic_map(x0,mu):
    xn = x0
    for j in range(trans):
        xn = x_npi(xn,mu)
    xar = [x0]
    for i in range (1,n):
        xar.append(x_npi(xar[i-1],mu) )
    return xar
x = logistic_map(x0,3.5697)
periods= [1 ,2, 4, 8, 16, 32, 64]
def detect_period(x):
    n = len(x)
    for i in range(n//2,n-64): #n//2 kills transient
        for T in periods:
            if i+T < n and abs(x[i] - x[i+T]) < 1e-12:
                period_exists= True
                for k in range(1, min(T, n-i-64)):
                    if abs(x[i+k] - x[i+k+T]) >= 1e-12:
                        period_exists = False
                        break
                if period_exists:
                    return T
    return None

r_vals = np.linspace(3,3.57,10000)
bifurcations = []
current_period = 1
for r in r_vals:
    xr = logistic_map(x0,r)
    period = detect_period(xr)
    if period:
        if current_period != period:
            bifurcations.append((period ,r))
        current_period = period
#bifurcations = bifurcations[:-1]
print(bifurcations)
deltas = []
for i in range (2,len(bifurcations)-1):
    deltan = bifurcations[i][1] - bifurcations[i-1][1]
    deltanp1 = bifurcations[i+1][1] - bifurcations[i][1]
    delta = deltan / deltanp1
    deltas.append(delta)

print("The Feignebaum constant is = 4.66920")
print (deltas)
# find alpha

```

## A.11 Problem 11

```
# 11. What do 1D maps have to do with Science?
import numpy as np
import matplotlib.pyplot as plt
a = 0.2
b = 0.2
t0= 0

n = 100000
def fx(y,z):
    return -y - z
def fy(x,y):
    return x + a*y

def fz (x,z):
    return b + z*(x - c)

def f(x,y,z):
    return fx(y,z),fy(x,y),fz(x,z)
def rk4 (x,y,z,h):
    k1 = f(x,y,z)
    k2 = f(x+h/2 * k1[0],y+h/2*k1[1], z + h/2 * k1[2])
    k3 =f(x+h/2 * k2[0],y+h/2*k2[1], z + h/2 * k2[2])
    k4 = f(x+h * k3[0],y+h*k3[1], z + h * k3[2])
    xnp1= x + h/6 * (k1[0] + 2* k2[0] + 2*k3[0] + k4[0])
    ynp1 = y + h/6 * (k1[1] + 2* k2[1] + 2*k3[1] + k4[1])
    znp1 = z + h/6 * (k1[2] + 2* k2[2] + 2*k3[2] + k4[2])

    return xnp1,ynp1,znp1

h=0.01
x0, y0,z0 = 1,1,1

def solveDE(x,y,z,h):
    x_list = [x]
    y_list = [y]
    z_list = [z]
    t_list = [t0]

    for i in range(n):
        x,y,z = rk4(x,y,z,h)
        x_list.append(x)
        y_list.append(y)
        z_list.append(z)
        t_list.append(t0 + (i+1) * h)
    return x_list,y_list,z_list,t_list

c = 2.5
x2,y2,z2,t2 = solveDE(x0,y0,z0,h)
plt.figure()
plt.plot(y2,x2)
plt.xlabel('x')
plt.ylabel('y')
plt.title('c=2.5')
plt.show()

c = 3.5
```

```

x3,y3,z3,t3 = solveDE(x0,y0,z0,h)
plt.figure()
plt.plot(y3,x3)
plt.xlabel('x')
plt.ylabel('y')
plt.title('c=3.5')
plt.show()

c = 4.5
x4,y4,z4,t4 = solveDE(x0,y0,z0,h)
plt.figure()
plt.plot(y4,x4)
plt.xlabel('x')
plt.ylabel('y')
plt.title('c=4.5')
plt.show()
#part 2
c=5
x5,y5,z5,t5 = solveDE(x0,y0,z0,h)

def find_max(x,t):
    x_max = []
    t_max = []
    for i in range (n):
        if x[i] > x[i-1] and x[i] > x [i+1]:
            x_max.append(x[i])
            t_max.append(t[i])
    return x_max,t_max

x_max5 = find_max(x5,t5)
x5n = x_max5[:-1]
x5np1 = x_max5 [1:]

plt.figure()
plt.plot(x5n,x5np1,'o')
plt.show()

#part 3
c_values = np.linspace(2.5, 3.5,400)
all_maxima = []

for c in c_values:
    x, y, z, t = solveDE(x0, y0, z0, h)
    x_max, t_max = find_max(x, t)
    all_maxima.append(x_max[-100:]) #skip transient

c_list = []
xmax_list = []
for i, c in enumerate(c_values):
    for xm in all_maxima[i]:
        c_list.append(c)
        xmax_list.append(xm)

plt.figure(figsize=(10, 6))
plt.scatter(c_list, xmax_list, s=2, color='darkblue')
plt.xlabel("c")
plt.ylabel("x_maxima")
plt.title("Bifurcation-like diagram for the Rossler system")

```

```
plt.show()
```

## A.12 Problem 12

```

# SVD and Coherent Structures
import numpy as np
import matplotlib.pyplot as plt
# Parameters
A = 0.25
e = 0.1
w = 2*np.pi/10
# Helper functions
def a(t):
    return e * np.sin(w * t)

def b(t):
    return 1 - 2 * e * np.sin(w*t)

def f(x,t):
    return a(t) * x**2 + b(t) * x

def dpsix(x,y,t):
    return -A*np.pi*np.cos(np.pi*y)*np.sin(np.pi*f(x,t))

def dpsiy(x,y,t):
    return A*np.pi*np.cos(np.pi*f(x,t))*(2*a(t)*x + b(t))*np.sin(np.pi*y)

# Generate u and v for 200 time frames
x_grid = np.linspace(0, 2, 21)
y_grid = np.linspace(0, 1, 11)
X, Y = np.meshgrid(x_grid, y_grid, indexing='ij')

U = []
V = []
t_grid = np.linspace(0, 200, 201)

for t in t_grid:
    u = dpsix(X, Y, t)
    v = dpsiy(X, Y, t)
    U.append(u)
    V.append(v)
# Create D matrix
D = []
for k in range(len(U)):
    u = np.ravel(U[k])
    v = np.ravel(V[k])
    col = np.concatenate([u, v])
    D.append(col)

D = np.array(D).T
D_mean = D.mean(axis=1, keepdims=True)
D_tilde = D - D_mean
Ct = D_tilde.T @ D_tilde
eigvals, V_mat = np.linalg.eigh(Ct)
# Sort eigenvalues
idx = np.argsort(eigvals)[::-1]
eigvals = eigvals[idx]
V_mat = V_mat[:, idx]
sigma = np.sqrt(eigvals)

```

```

U_mat = (D_tilde @ V_mat) / sigma
# we want to keep the 3 dominant modes
U_r = U_mat[:, :3]
sigma_r = sigma[:3]
V_r = V_mat[:, :3]
plt.figure(figsize=(8,5))
for i in range(3):
    plt.plot(t_grid, V_r[:, i], label=f'Mode_{i+1}')
plt.xlabel("Time")
plt.ylabel("Temp_Coef")
plt.title("Temporal_Evolution_of_dominant_modes")
plt.legend()
plt.tight_layout()
plt.show()

D_rec = (U_r * sigma_r) @ V_r.T + D_mean # reconstruction only with
    leading modes
# at t =50 snapshot
UV50 = D_rec[:, 50]
pts = X.size
U_plt = UV50[:pts].reshape(X.shape) # just returning from 1d column to
    21x11 matrix
V_plt = UV50[pts:].reshape(X.shape)
plt.figure()
plt.quiver(X, Y, U_plt, V_plt)
plt.title("Reconstructed_flow_at_t=50")
plt.show()
U = []
V = []

for k in range(201):
    uv = D_rec[:, k]
    U.append(uv[:pts].reshape(X.shape)) # (21,11)
    V.append(uv[pts:].reshape(X.shape))
U = np.array(U)
V = np.array(V)

def velocity(x,y,t): #velocity at nearest indices
    ix = np.argmin(np.abs(x_grid - x))
    iy = np.argmin(np.abs(y_grid - y))
    it = np.argmin(np.abs(t_grid - t))
    u = U[it,ix,iy]
    v = V[it,ix,iy]
    return u,v

def rk4(x,y,t,dt):
    k1 = velocity(x, y, t)
    k2 = velocity(x + dt / 2* k1[0], y + dt/2* k1[1], t + dt/2)
    k3 = velocity(x + dt/2 *k2[0],y + dt/2* k2[1],t + dt/2)
    k4 = velocity(x + dt/2 * k3[0], y+ dt/2* k3[1],t + dt/2)
    xnp1 = x + dt/6 * (k1[0] + 2 * k2[0] + 2* k3[0] + k4[0] )
    ynp1 = y + dt/6* (k1[1] + 2 * k2[1] + 2* k3[1] + k4[1] )
    return xnp1,ynp1
#Solve rk4
t0 = 0
tf = 10
dt = 0.01
Xf = np.zeros_like(X)

```

```

Yf = np.zeros_like(Y)
for i in range (21):
    for j in range (11):
        x,y = X[i,j],Y[i,j]
        t=t0 #reset time
        for k in range(100):
            x,y = rk4(x,y,t,dt)
            t += dt
        Xf [i,j] = x
        Yf [i,j] = y
ftle = np.zeros_like(X)
for i in range (1,20):
    for j in range (1,10):
        dxx0= (Xf[i+1,j]- Xf[i-1,j]) /(X[i+1,j] - X[i-1,j]) #central
            differences derivatives fr F matrix
        dxy0= (Xf[i,j+1]-Xf[i,j-1])/(Y[i,j+1] - Y[i,j-1])
        dyx0= (Yf[i+1,j]- Yf[i-1,j])/(X[i+1,j] - X[i-1,j])
        dyy0= (Yf[i,j+1]- Yf[i,j-1])/(Y[i,j+1] - Y[i,j-1])
        F = np.array([[dxx0,dxy0],[dyx0,dyy0]])
        C = F.T @ F #caushy stress tensor
        eigv = np.linalg.eigvals(C)
        lmax = np.max(eigv)
        ftle[i,j] = 1/10 * np.log(np.sqrt(lmax)) # FTLE = ln(\lamda_max)
            /T (T = 10)

plt.figure(figsize=(8, 5))
plt.contourf(X, Y, ftle)
plt.colorbar(label='FTLE')
plt.title('Lagrangian-Coherent-Structures/FTLE-Field')
plt.xlabel('x')
plt.ylabel('y')
plt.show()

```