Name:	Manish Sharma	**Assignment No:**	Assignment 4 / Project
SR No:	05-01-00-10-42-24-1-24995	**Course Code:**	DS288/ UMC202
Email ID:	manishs1@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	November 17, 2025	**Term:**	AUG 2025

Solution 1

Convergence of the Secant Method for Multiple Roots

Let $f \in C^{m+1}(I)$ on an interval $I \subset \mathbb{R}$ and suppose $\alpha \in I$ is a root of multiplicity $m \geq 2$, i.e.,

$$f(\alpha) = f'(\alpha) = \cdots = f^{(m-1)}(\alpha) = 0, \quad f^{(m)}(\alpha) \neq 0.$$

Let $\{x_k\}$ be the sequence generated by the secant method

$$x_{n+1} \;=\; x_n \;-\; f(x_n)\,\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}, \qquad k \geq 1,$$

and define the errors $e_n := \alpha - x_k$. Under local convergence (i.e., $e_n \to 0$), we analyze the asymptotic behavior of $\{e_n\}$.

Local expansions near a multiple root. Using Taylor's theorem at α and the multiplicity-m assumption,

$$f(x_n) \;=\; \frac{f^{(m)}(\alpha)}{m!}\,e_n^m \;+\; \mathcal{O}\big(e_n^{m+1}\big), \qquad f(x_{n-1}) \;=\; \frac{f^{(m)}(\alpha)}{m!}\,e_{n-1}^m \;+\; \mathcal{O}\big(e_{n-1}^{m+1}\big).$$

Substituting into the secant update and expressing in terms of errors yields the exact identity

$$e_{n+1} = \frac{e_{n-1}\,\big(f(x_n)\big) - e_n\big(f(x_{n-1})}{f(x_n) - f(x_{n-1})}.$$

$$e_{n+1} \;=\; \frac{e_{n-1}(e_n)^m - e_n(e_{n-1})^m}{(e_n)^m - (e_{n-1})^m} \;+\; O\big(|e_{n-1}|^{\,m+1}\big) \tag{1}$$

Double roots ($m = 2$)

For double roots, neglecting the terms $O(|e_{n-1}|^3)$ and $O(|e_{n-1}|^4)$, equation becomes

$$e_{n+1} \;=\; \frac{e_{n-1}e_n}{(e_n - e_{n-1})}\,.$$

rewritten as

$$\frac{1}{e_{n+1}} \;\approx\; \frac{1}{e_n} + \frac{1}{e_{n-1}}\,.$$

Thus, the sequence $\{1/e_n\}$, $k = 0, 1, 2, \ldots$, is a Fibonacci sequence. According to the simple root convergence, the following expressions hold in the asymptotic regime:

$$e_{n+1} \;\approx\; \varphi^{-1} e_n\,,$$

where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. Recall that $\varphi^{-1} \approx 0.618$. Thus, for double roots, the secant method exhibits **linear convergence** with λ equal to 0.618.

Multiplicity larger than 2 **(**$m > 2$**)**

The superlinear convergence is lost for double roots. Then, for $m > 2$, the secant method is also assumed to be linear with AEC equal to λ. Thus, in the asymptotic range, the following equality holds:

$$e_{n+1} \;\approx\; \lambda e_n\,.$$

An equation for λ is recovered by introducing the above relation into equation (1) and neglecting higher-order terms:

$$\lambda^2 \;=\; \lambda(\lambda - 1)\big(\lambda^m + \lambda^{m-1} - 1\big)\,.$$

This polynomial equation has a unique root such that $0 < \lambda < 1$. This root is slightly lower than (and almost equal to, for large m) $(1/2)^{1/m}$. For example, when $m = 3$, an analytical expression is available:

$$\lambda \;\approx\; 0.7548776667\,.$$

Name:	Manish Sharma	**Assignment No:**	Assignment 4 / Project
SR No:	05-01-00-10-42-24-1-24995	**Course Code:**	DS288/ UMC202
Email ID:	manishs1@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	November 17, 2025	**Term:**	AUG 2025

Conclusion

- For a *simple root* ($m = 1$), the secant method is superlinear with order ≈ 1.618.
- For a *multiple root* ($m > 1$), superlinearity is lost.

(b) Muller's method for a simple root

Assumptions:

- α is a simple root: $f(\alpha) = 0$, $f'(\alpha) \neq 0$, $f \in C^2$ near α.
- The three latest iterates x_n, x_{n-1}, x_{n-2} are close to α, and the quadratic interpolant of f is well-defined.

Error model and characteristic equation: Muller's method computes the next iterate x_{n+1} as a chosen root of the quadratic that interpolates $\{(x_n, f_n), (x_{n-1}, f_{n-1}), (x_{n-2}, f_{n-2})\}$. A local error analysis around a simple root shows the leading-order error recurrence can be expressed in the homogeneous form

$$e_{n+1} = C\, e_n^{\theta_1} e_{n-1}^{\theta_2} e_{n-2}^{\theta_3} \quad \text{with} \quad \theta_1 + \theta_2 + \theta_3 = 1,$$

which, under the standard asymptotic ansatz $e_n \sim K\,\rho^n$ and matching of exponents, yields the characteristic polynomial for the order a:

$$a^3 - a^2 - a - 1 = 0.$$

This cubic has a unique real root greater than 1,

$$a = 1.839286\ldots$$

Thus Muller's method is *superlinear* with asymptotic order

$$\boxed{\text{Muller's method (simple root): } a \approx 1.839286.}$$

In particular, its order exceeds the secant method's order $\phi \approx 1.618$ for simple roots.

references

1. Provided class lecture notes.
2. https://www.cs.toronto.edu/ rwu/csc338/2301/secant-superlinear-proof.pdf

Name:	Manish Sharma	**Assignment No:**	Assignment 4 / Project
SR No:	05-01-00-10-42-24-1-24995	**Course Code:**	DS288/ UMC202
Email ID:	manishs1@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	November 17, 2025	**Term:**	AUG 2025

Solution 2

2. Chaotic World of Fractals

(a) Mandelbrot Set

The Mandelbrot set is defined by the iterative relation:

$$z_{n+1} = z_n^2 + c, \quad z_0 = 0$$

A complex number c belongs to the Mandelbrot set if the sequence $\{z_n\}$ does not diverge, i.e., $|z_n| \leq 2$ for all n.

Python Implementation

```
def mandelbrot(c, max_iter=100):
    z = 0
    for n in range(max_iter):
        z = z*z + c
        if abs(z) > 2:
            return n
    return max_iter
```

Visualization Generate a grid of 1000×1000 points over:

$$x \in [-2.5, 1], \quad y \in [-1.5, 1.5]$$

Evaluate the iteration for each point and plot using `imshow` with appropriate colormap.

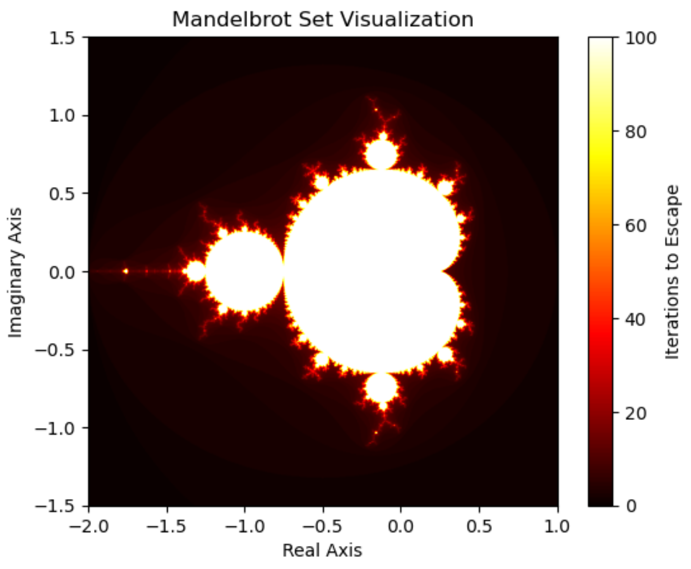

Figure 1: Mandelbrot set

Name: Manish Sharma
SR No: 05-01-00-10-42-24-1-24995
Email ID: manishs1@iisc.ac.in
Date: November 17, 2025

Assignment No: Assignment 4 / Project
Course Code: DS288/ UMC202
Course Name: Numerical Methods
Term: AUG 2025

(b) Newton's Fractal

We consider the function:

$$f(z) = z^3 + 1, \quad f'(z) = 3z^2$$

The Newton-Raphson iteration is:

$$z_{n+1} = z_n - \frac{f(z_n)}{f'(z_n)} = z_n - \frac{z_n^3 + 1}{3z_n^2}$$

Python Implementation

```
root0 = -1.0 + 0j
root1 = np.exp(1j * np.pi / 3) # 0.5 + 0.866j
root2 = np.exp(-1j * np.pi / 3) # 0.5 - 0.866j
roots = [root0, root1, root2]

def  newton_fractal(z0, tol=1e-6) :
    z,_,_,_=newton_method(f, df, z0, tol)
    root_tol=1e-3
    for i, root in enumerate(roots):
        if abs(z - root) < root_tol:
            return i
```

newton method code is used from assignment-2

Visualization Generate a grid over:N=1000*1000

$$x, y \in [-2, 2]$$

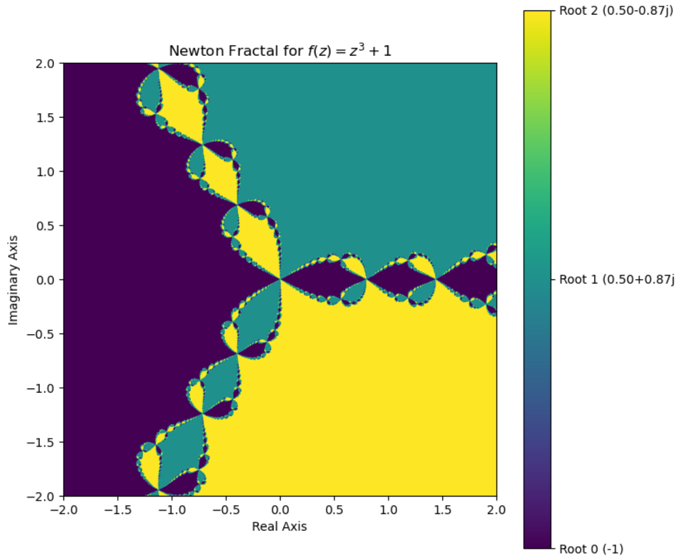

Figure 2: Newton's Fractal: displaying basins of attraction

(c) Logistic Map and Chaos

The logistic map is defined as:

$$x_{n+1} = Ax_n(1 - x_n), \quad 0 \leq x_n \leq 1$$

Python Implementation

```
def logistic_map(A, x0, n_iter):
    x = np.zeros(n_iter)
    x[0] = x0
    for i in range(1, n_iter):
        x[i] = A * x[i-1] * (1 - x[i-1])
    return x
```

Bifurcation Diagram: Vary $A \in [0.89, 3.995]$ in steps of 0.0125. For each A, iterate for 200 steps and discard the first 15. Plot x_n vs A.

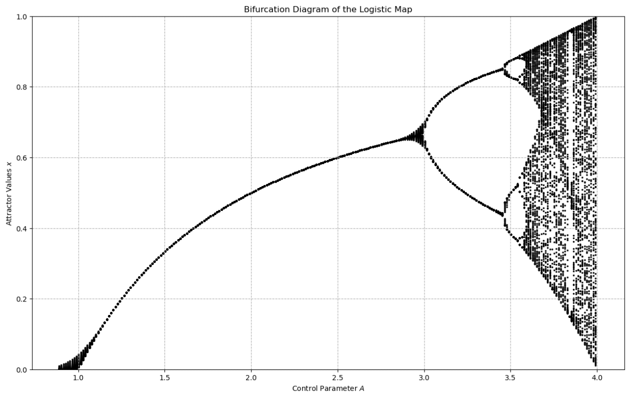

Figure 3: Bifurcation Diagram

Name:	Manish Sharma	**Assignment No:**	Assignment 4 / Project
SR No:	05-01-00-10-42-24-1-24995	**Course Code:**	DS288/ UMC202
Email ID:	manishs1@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	November 17, 2025	**Term:**	AUG 2025

Lyapunov Exponent The finite-time Lyapunov exponent is:

$$\lambda_n = \frac{1}{n}\sum_{k=0}^{n-1} \ln|f'(x_k)|, \quad f(x) = Ax(1-x), \quad f'(x) = A(1-2x)$$

```
def lyapunov_exponent(A, x0, n_iter):
    x = x0
    le_sum = 0
    for i in range(n_iter):
        x = A * x * (1 - x)
        le_sum += np.log(abs(A * (1 - 2 * x)))
    return le_sum / n_iter
```

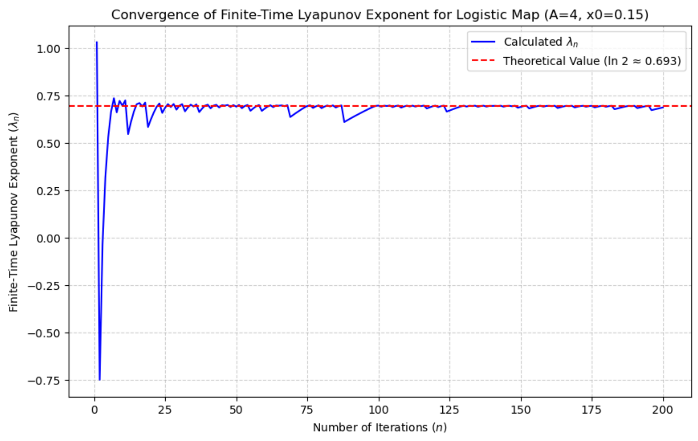

Figure 4: finite-time Lyapunov exponent as a function of n

Observation:

- A=4, $x_0 = 0.15$. Note: x_0 should be between 0 and 1
- Reported Finite-Time Lyapunov Exponent for A=4 after 200 iterations: $\lambda = 0.686236$, which approaches a constant value appearing as straight line.

Solution 3

We consider a planar robotic manipulator with three revolute joints, each of length

$$l_1 = l_2 = l_3 = 1.0 \text{ m}.$$

The joint angles are denoted by

$$\boldsymbol{\theta} = [\theta_1, \theta_2, \theta_3]^T.$$

The end-effector (end-hook) position in Cartesian coordinates is given by the forward kinematics:

$$x(\boldsymbol{\theta}) = l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2) + l_3 \cos(\theta_1 + \theta_2 + \theta_3),$$

$$y(\boldsymbol{\theta}) = l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2) + l_3 \sin(\theta_1 + \theta_2 + \theta_3).$$

The target position is

$$\mathbf{p}_t = \begin{bmatrix} -0.8 \\ 1.2 \end{bmatrix} \text{ m}.$$

Initial Conditions

$$\boldsymbol{\theta}^{(0)} = \begin{bmatrix} 0.1 \\ -0.3 \\ 0.2 \end{bmatrix} \text{ rad}, \qquad \mathbf{v}^{(0)} = \mathbf{0}.$$

(a) Derivation of Forward Kinematics

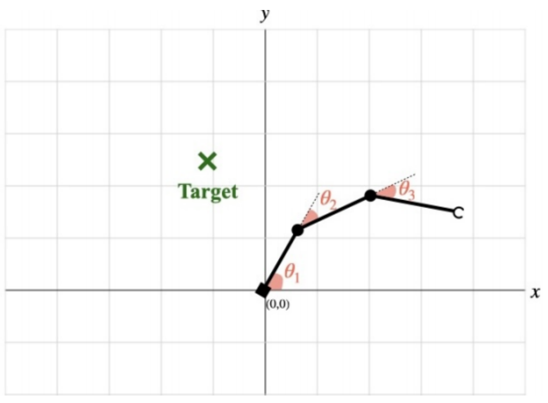

Figure 1: A 3-linked Robotic Arm fixed at origin.

From the above figure, let the length of 3 arms be l_1, l_2 and l_3 respectively.

- Resolve or project each arm in x and y direction, i.e
  1. $l_1 = l_1 cos(\theta_1).i_x + l_1 sin(\theta_1).i_y$
  2. $l_2 = l_2 cos(\theta_1 + \theta_2).i_x + l_1 sin(\theta_1 + \theta_2).i_y$
  3. $l_3 = l_3 cos(\theta_1 + \theta_2 + \theta_3).i_x + l_1 sin(\theta_1 + \theta_2 + \theta_3).i_y$
- Now adding all i_x values give:

$$x(\boldsymbol{\theta}) = l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2) + l_3 \cos(\theta_1 + \theta_2 + \theta_3).i_x$$

- Now adding all i_y values give:

$$y(\boldsymbol{\theta}) = l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2) + l_3 \sin(\theta_1 + \theta_2 + \theta_3).i_y$$

(b) Gradient of Loss Function

The loss function is defined as

$$L(\boldsymbol{\theta}) = \frac{1}{2}\|\mathbf{p}(\boldsymbol{\theta}) - \mathbf{p}_t\|^2.$$

We approximate the gradient using the central difference method with step size $h = 10^{-5}$:

```
def numerical_gradient(theta):
```

- Initializes a gradient vector of size 3.
- For each joint i, computes derivative via central difference:

$$\frac{\partial L}{\partial \theta_i} \approx \frac{L(\theta_i + h) - L(\theta_i - h)}{2h}$$

- Returns the gradient vector.

(c) Gradient Descent Algorithm

The iterative update rule is

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \alpha \nabla L(\boldsymbol{\theta}^{(k)}),$$

with learning rate $\alpha = 0.01$. The iteration continues until

$$\|\mathbf{p}(\boldsymbol{\theta}^{(k)}) - \mathbf{p}_t\| < 0.01 \text{ m}.$$

```
def gradient_descent(theta_init, lr=alpha):
```

- Creates local copies of θ.
- Stores the end-effector path for plotting.
- Iteratively:
  1. Computes the gradient.
  2. Updates joint angles:
  $$\theta \leftarrow \theta - \alpha \nabla L(\theta)$$
  3. Computes new end-effector location.
  4. Checks stopping condition:
  $$\|p(\theta) - p_t\| < \text{tol}$$
- Returns the full path, final angles, and iteration count.

(d) Gradient Descent with Momentum

Momentum introduces a velocity term:

$$\mathbf{v}^{(k+1)} = \beta \mathbf{v}^{(k)} + \alpha \nabla L(\boldsymbol{\theta}^{(k)}),$$
$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \mathbf{v}^{(k+1)},$$

with momentum coefficient $\beta = 0.9$.

```
def gradient_descent_momentum(theta_init, v_init, lr=alpha, beta=beta):
```

- Initializes angles and momentum vector.
- Each iteration:
  1. Computes gradient.
  2. Updates momentum:
  $$v \leftarrow \beta v + \alpha \nabla L(\theta)$$
  3. Updates angles using momentum:
  $$\theta \leftarrow \theta - v$$
  4. Computes new position and checks convergence.
- Returns path, final θ, and iterations.

(e) Comparison of Methods

- Iterations: It took **682 iterations** for gradient descent and **89 iterations** for gradient descent with momentum.
- Final Joint Angles:
  1. Standard Gradient descent, **final Joint Angles (deg): [30.08766456 102.32983005 49.56716801]**
  2. Gradient descent with momentum, **final Joint Angles (deg): [24.98348912 109.95272884 40.8386863]**
- Path Visualization:

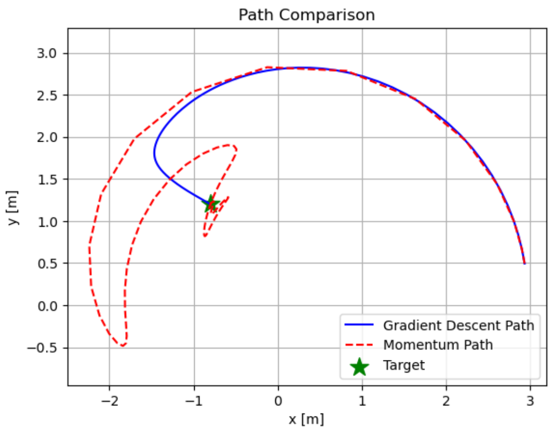

Figure 5: End-hook paths for both methods

- **Observations:** Momentum typically accelerates convergence and smooths oscillations, reducing the number of iterations compared to standard gradient descent.
  - Standard gradient descent is slow.
  - Momentum accelerates convergence by smoothing gradients.
  - Final plot clearly compares both trajectories.

Name:	Manish Sharma	**Assignment No:**	Assignment 4 / Project
SR No:	05-01-00-10-42-24-1-24995	**Course Code:**	DS288/ UMC202
Email ID:	manishs1@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	November 17, 2025	**Term:**	AUG 2025

Explanation of the Optimization Code

- `l1 = l2 = l3 = 1.0`
  Defines the lengths of the 3 robot arm links. All links are 1 meter long.
- `pt = np.array([-0.8, 1.2])`
  The target point in the plane that the end-effector should reach.
- `alpha = 0.01`
  Learning rate for standard gradient descent.
- `beta = 0.9`
  Momentum coefficient for the momentum-based optimizer.
- `h = 1e-5`
  Small step used to compute the numerical gradient by central differences.
- `tol = 0.01`
  Convergence tolerance. Optimization stops when the end-effector is within 1% Euclidean distance of the target.
- `theta0 = np.array([0.2, 0.1, -0.3])`
  Initial guess of joint angles (in radians).
- `v0 = np.zeros(3)`
  Initial momentum vector for momentum gradient descent.

Forward Kinematics

```
def forward_kinematics(theta):
```

- Extracts $\theta_1, \theta_2, \theta_3$.
- Computes the end-effector position:

$$x = l_1 \cos\theta_1 + l_2 \cos(\theta_1 + \theta_2) + l_3 \cos(\theta_1 + \theta_2 + \theta_3)$$

$$y = l_1 \sin\theta_1 + l_2 \sin(\theta_1 + \theta_2) + l_3 \sin(\theta_1 + \theta_2 + \theta_3)$$

- Returns the position as a 2D vector.

Loss Function

```
def loss(theta):
```

- Computes the forward kinematics at angles θ.
- The loss is the squared distance between the end-effector and the target:

$$L(\theta) = \frac{1}{2}\|p(\theta) - p_t\|^2$$

- The factor $1/2$ simplifies derivatives.

Running Both Methods

- `path_gd, theta_gd, iter_gd = gradient_descent(theta0)`
  Runs standard gradient descent.
- `path_mom, theta_mom, iter_mom = gradient_descent_momentum(theta0, v0)`
  Runs momentum-based optimizer.

Name:	Manish Sharma	**Assignment No:**	Assignment 4 / Project
SR No:	05-01-00-10-42-24-1-24995	**Course Code:**	DS288/ UMC202
Email ID:	manishs1@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	November 17, 2025	**Term:**	AUG 2025

Solution 4

Orthogonality of Complex Exponentials

Let the inner product be defined as

$$\langle f, g\rangle = \int_0^T f(t)\,\overline{g(t)}\,dt$$

Consider the functions

$$f(t) = e^{i\frac{2\pi k}{T}t}, \quad g(t) = e^{i\frac{2\pi m}{T}t}.$$

Then their inner product becomes

$$\langle f, g\rangle = \int_0^T e^{i\frac{2\pi k}{T}t}\cdot\overline{e^{i\frac{2\pi m}{T}t}}\,dt = \int_0^T e^{i\frac{2\pi(k-m)}{T}t}\,dt.$$

Let $n = k - m$. Then:

- If $n = 0$ (i.e., $k = m$):

$$\langle f, g\rangle = \int_0^T 1\,dt = T$$

- If $n \neq 0$:

$$\langle f, g\rangle = \int_0^T e^{i\frac{2\pi n}{T}t}\,dt = \left[\frac{T}{i2\pi n}e^{i\frac{2\pi n}{T}t}\right]_0^T = \frac{T}{i2\pi n}\left(e^{i2\pi n} - 1\right) = 0$$

  since $e^{i2\pi n} = 1$ for any integer n.

$$\langle f, g\rangle = \begin{cases} T & \text{if } k = m, \\ 0 & \text{if } k \neq m. \end{cases}$$

Thus, the complex exponentials $e^{i\frac{2\pi k}{T}t}$ and $e^{i\frac{2\pi m}{T}t}$ are orthogonal over the interval $[0, T]$ with respect to the given inner product.

Part 2: Data Loading and Preprocessing

- `pts = np.loadtxt('M.csv', delimiter=',', skiprows=1)` loads CSV rows into an $N \times 2$ array.
- `z = pts[:,0] + 1j*pts[:,1]` converts the 2D points to complex numbers $z_j = x_j + \mathrm{i}y_j$.

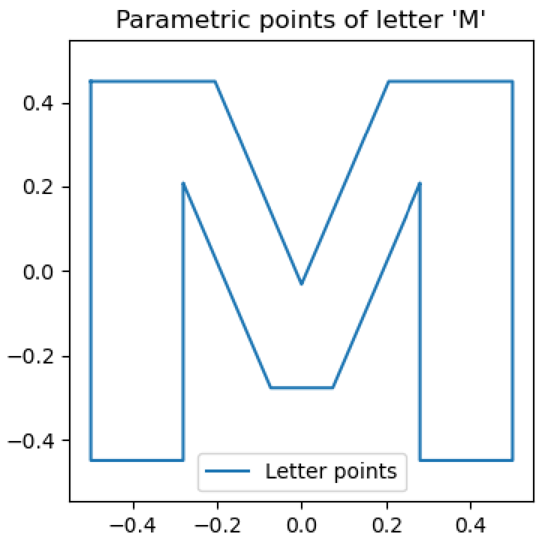

Figure 6: M.csv, Parametric points of letter 'M'

Name: Manish Sharma
SR No: 05-01-00-10-42-24-1-24995
Email ID: manishs1@iisc.ac.in
Date: November 17, 2025

Part 3: Fourier Coefficient Calculation

The parameter values are

$$t_j = \frac{j}{N}, \quad j = 0, 1, \ldots, N-1.$$

The continuous formula for Fourier coefficients is

$$c_k = \int_0^1 z(t)\, e^{-i2\pi kt}\, dt.$$

Function `ck_coeff(z_data, M_max)`

- Discretization used in code:
  - The code builds a sample grid $\{t_j\}_{j=0}^{N-1} = \text{np.linspace}(0, 1, N)$.
  - It sets $h = 1/N$ and uses the simple trapezoidal-integration formula for periodic functions, which is:

$$c_k \approx h \sum_{j=0}^{N-1} z_j\, e^{-\mathrm{i}2\pi k t_j}.$$

Why the periodic trapezoidal rule is the best method:

- **Function Periodicity:** The dataset represents a closed loop (a letter), so the function $z(t)$ is periodic, with $z(0) = z(1)$.
- **Error Cancellation:**error of the trapezoidal rule, shows that for a periodic function integrated over its full period, the endpoint error terms (and all their higher-order derivatives) are identical and perfectly cancel each other out.
- **Spectral Accuracy:** This error cancellation results in the method being **spectrally accurate**. The error typically decreases exponentially ($O(e^{-c/h})$), which is far superior to the polynomial accuracy of other methods (like Simpson's rule, which is $O(h^4)$).
- **Equivalence to the DFT:** The resulting formula is not just an approximation, but is mathematically equivalent to the **Discrete Fourier Transform (DFT)**. The DFT is the standard, most efficient, and most accurate numerical tool specifically designed to analyze the frequency components of discrete, periodic data. By using this method, you are (correctly) using the DFT.

Part 4: Curve Reconstruction

The truncated Fourier series with mode number M is

$$z_M(t) = \sum_{k=-M}^{M} c_k\, e^{i2\pi kt}.$$

Evaluating at $N = 1000$ equally spaced points $t \in [0, 1)$ gives the reconstructed coordinates

$$x(t) = \Re(z_M(t)), \quad y(t) = \Im(z_M(t)).$$

Function `reconstruct_curve(ck_coeff, M, num_points)`

Part 5: Visualization

We compare reconstructions for mode numbers

$$M \in \{2, 8, 32, 64, 128\}.$$

Each subplot displays:

- The original curve (outline points).
- The reconstructed curve using $z_M(t)$.

As M increases, finer details of the letter are captured and the approximation error decreases.

Name: Manish Sharma
SR No: 05-01-00-10-42-24-1-24995
Email ID: manishs1@iisc.ac.in
Date: November 17, 2025

Assignment No: Assignment 4 / Project
Course Code: DS288/ UMC202
Course Name: Numerical Methods
Term: AUG 2025

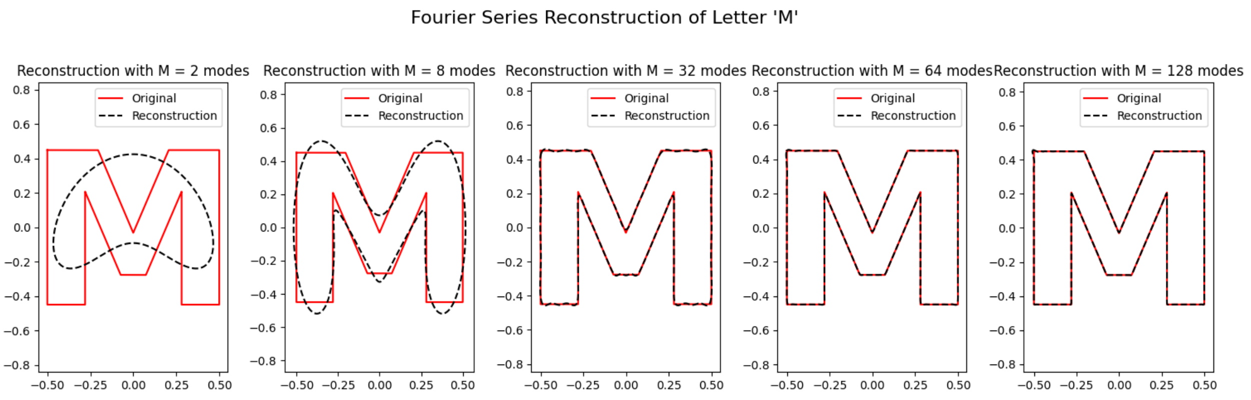

Figure 7: Curve Reconstruction for M=[2,8,32,64,128]

Part 6: Animation

To visualize the reconstruction process, animate the partial sum

$$z_n(t) = \sum_{k=-n}^{n} c_k\, e^{i2\pi kt}, \quad n = 0 \to M,$$

showing how successive frequency components contribute to the final outline. This demonstrates the synthesis of the letter curve from low-frequency to high-frequency modes.

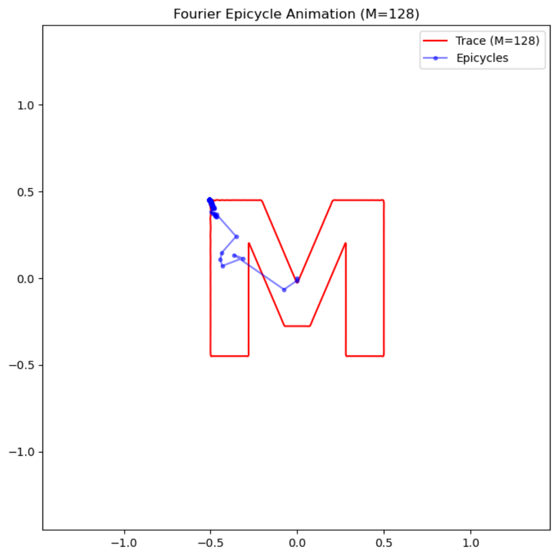

Figure 8: Animation image captured

Refer file attached with name "Q4_p6_N_1000_fps_30"

Name:	Manish Sharma	**Assignment No:**	Assignment 4 / Project
SR No:	05-01-00-10-42-24-1-24995	**Course Code:**	DS288/ UMC202
Email ID:	manishs1@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	November 17, 2025	**Term:**	AUG 2025

Solution 5

Equation of Motion

The governing second-order ODE is

$$\frac{d^2x}{dt^2} + 2\gamma\frac{dx}{dt} + \omega_0^2 x = A_0\cos(\omega t),$$

where

$$\gamma = 0.5\ \text{s}^{-1}, \quad \omega_0 = 2.7\ \text{rad/s}, \quad A_0 = 2.0\ \text{m/s}^2, \quad \omega = 2.0\ \text{rad/s}.$$

Initial conditions:

$$x(0) = 1.0\ \text{m}, \quad \dot{x}(0) = 0.0\ \text{m/s}.$$

Part 1: Numerical Implementation

We rewrite the second-order ODE as a system of first-order equations:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x \\ \dot{x_1} \end{bmatrix}$$

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \dot{x} \\ -2\gamma\dot{x_1} - \omega_0^2 x_1 + A_0\cos(\omega t). \end{bmatrix}$$

This system can be written in vector form $\frac{d\vec{y}}{dt} = f(t, \vec{y})$, where $\vec{y} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$.

```
def f(t,x):
    x1,x2 = x
    return np.array([x2,-omega_0**2*x1-2*gamma*x2-A_0*np.cos(omega*t)])
```

It takes the current time t and the state vector $\vec{y}$ (as `x`) and returns the vector of derivatives $[\frac{dx_1}{dt}, \frac{dx_2}{dt}]$.

- **Euler Method:** This function implements the forward Euler method. It iterates through time, calculating the next state using the simple formula:

  $$y_{n+1} = y_n + hf(t_n, y_n).$$

  This is a first-order method.

  ```
  def euler(f, y0, t_eval, h):
  ```

- **RK4 Method:** The fourth-order Runge-Kutta method computes the next value using a weighted average of four slopes:

  $$k_1 = hf(x_n, y_n)$$
  $$k_2 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right)$$
  $$k_3 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right)$$
  $$k_4 = hf(x_n + h, y_n + k_3)$$

  The next value y_{n+1} is then given by:

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

  Here:

  - k_1 is the slope at the beginning of the interval.
  - k_2 and k_3 are estimates of the slope at the midpoint.
  - k_4 is the slope at the end of the interval.

  It iterates from the start to the end of the time array `t_eval`, calculating the state vector $\vec{y}_{n+1}$ from $\vec{y}_n$ using the standard RK4 formulas for k_1, k_2, k_3, k_4.

  ```
  def rk4(f, y0, t_eval, h):
  ```

Tabulating $t, x(t), v(t)$ at every 500th step.

Table 1: Time t, displacement x(t) and velocity dx dt at each 500th time step.

Time (sec)	RK4-Displacement (m)	Euler-Displacement (m)	RK4-Velocity (m/sec)	Euler-Velocity (m/sec)
0	0.0	1.000000	1.000000	0.000000
5.0	0.134445	0.280675	-0.000947	-0.079707
10.0	-0.018962	0.025141	0.108940	0.090536
15.0	-0.004994	0.005493	-0.110932	-0.115318
20.0	0.035160	0.037686	0.088084	0.086169
25.0	-0.053391	-0.052932	-0.035589	-0.034587
30.0	0.054485	0.054607	-0.028222	-0.029643

Part 2: Visualization

Generate plots of displacement $x(t)$ vs. time t for both Euler and RK4 methods over $t \in [0, 30]$ s with $\Delta t = 0.01$.

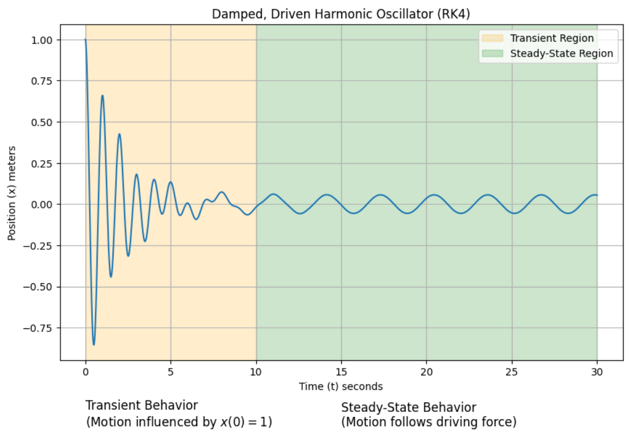

Figure 9: Oscillator's displacement x(t) vs time t for RK4

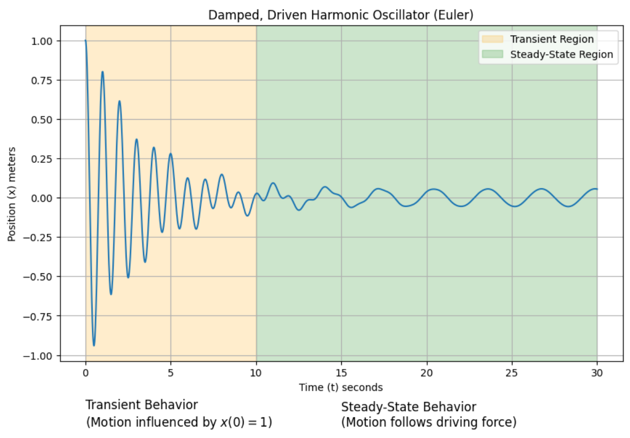

Figure 10: Oscillator's displacement x(t) vs time t for Euler

Name:	Manish Sharma	**Assignment No:**	Assignment 4 / Project
SR No:	05-01-00-10-42-24-1-24995	**Course Code:**	DS288/ UMC202
Email ID:	manishs1@iisc.ac.in	**Course Name:**	Numerical Methods
Date:	November 17, 2025	**Term:**	AUG 2025

Part 3: Analysis of Motion

It is identified that after 10 seconds the initial transient behaviour fades of to settle up due to driving force.

- **Transient behavior:**The plot shows the initial motion (transient state) is complex, as it's a mix of the system's natural, damped frequency and the driving frequency.
- **Steady-state behavior:** After transients vanish, the solution oscillates at the driving frequency ω.After 10 seconds, the motion settles into a regular, stable pattern (steady-state).
- **Responsible term:** The term responsible for the decay of the transient motion is the damping term $2\gamma\dot{x}$, which causes exponential decay of transients.

  This term (with γ) removes energy from the system (dissipation),causing the initial 'natural' part of the motion (from x(0)=1) to die out,leaving only the motion sustained by the external driving force.

Part 4: Convergence Analysis

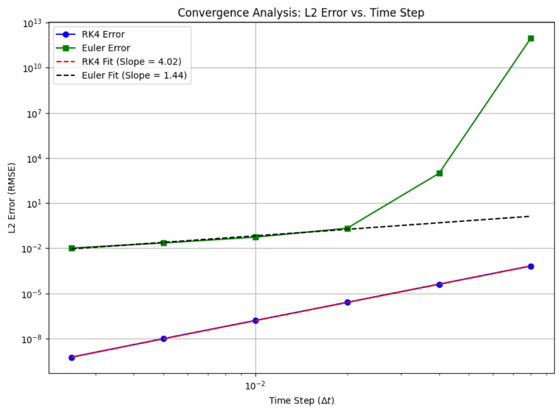

Figure 11: Convergence analysis ignoring the last two values of Δt

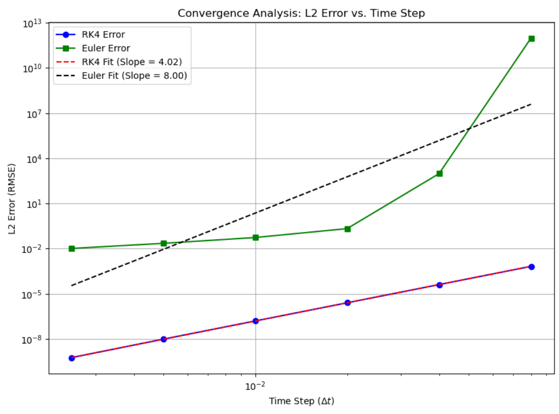

Figure 12: Convergence analysis of rk4 and euler

- Compute solutions using RK4 with $\Delta t = 0.00125$ as the true solution.
- For test step sizes $\Delta t = 0.0025, 0.005, 0.01, 0.02, 0.04, 0.08$, compute numerical solutions with Euler and RK4.
- Compute the L^2-error (root mean square error) relative to the true solution:

$$E(\Delta t) = \sqrt{\frac{1}{N}\sum_{j=1}^{N}\left(x_j^{\text{method}} - x_j^{\text{true}}\right)^2}.$$

- Ensuring we take the errors at time steps which are present in both your test is done by calculating the **true solution at the correct intervals**.

  ```
  step_ratio = int(round(dt / h_true))
  x1_true_sampled = x1_true[::step_ratio]
  ```

  This line calculates a simple step-ratio and "Go through the `x1_true` array and select every `step_ratio`-th element."

  1. For the test step $\Delta t = 0.01$:

     $$\text{step\_ratio} = \text{int(round}(0.01/0.00125)) = \text{int}(8)$$

     It measures each error of $\Delta t = 0.01$ to 8^{th} value of $\Delta t = 0.00125$

- Plot $E(\Delta t)$ vs. Δt on a log-log scale. Fit a line to estimate slope using:

- Define a **polyfit function**.This is a custom function to perform a polynomial line fit by solving the normal equations. It returns the coefficient of polynomial with given degree. For degree = 1 it returns intercept and slope of line. **Refer Assignment-3**

  ```
  def poly_least_squares(x,y,degree):
      n=degree
      A=np.zeros([n+1,n+1])
      b=np.zeros(n+1)
      x=np.dot(np.linalg.inv(A),b)
      return x
  ```

Results

- Euler's method: slope $1.44 \approx 1$ (first-order convergence), this can be seen in fig 11 if we ignore the initial values of $\Delta t = 0.08, 0.04$.

  The "order of convergence" is an asymptotic property. This means the formula $Error \approx C \cdot (\Delta t)^p$ is only truly accurate as $\Delta t \to 0$.

  For larger values of Δt it doesn't follow linear convergence.

- RK4 method: slope ≈ 4 (fourth-order convergence).

- RK4 is significantly more accurate and stable for oscillatory problems.

Name: Manish Sharma
SR No: 05-01-00-10-42-24-1-24995
Email ID: manishs1@iisc.ac.in
Date: November 17, 2025

References

[1] Faires, J. Douglas, and Richard L. Burden. Numerical methods, 4th. Cengage Learning, 2012.

[2] Lecture notes provided for the course.

[3] https://www.cs.toronto.edu/ rwu/csc338/2301/secant-superlinear-proof.pdf

[4] Assignment 2

[5] Assignment 3