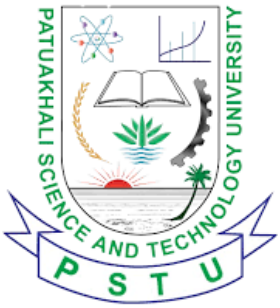
Patuakhali Science and Technology University
Faculty of Computer Science and Engineering



Lab Report

Course Code: CCE 314

Course Title: Computer Networks Sessional

(Numerical Methods Algorithms)

SUBMITTED TO:

Prof. Dr. Md Samsuzzaman
Department of Computer and Communication Engineering
Faculty of Computer Science and Engineering

SUBMITTED BY:

Mohammed Sakib Hasan
ID: **2102052**,
Registration No: **10179**
Faculty of Computer Science and Engineering

Assignment title: Final Lab Report
Date of submission: Thursday, 20 November 2025

Contents

1 Algebraic & Linear System Methods

1.1 Lab Problem 1: Cramer's Rule

Objective: To solve a system of linear equations $Ax = b$ using Cramer's Rule, which relies on the computation of determinants.

Method: Given a system of n linear equations $Ax = b$, the solution for each unknown x_i is:

$$x_i = \frac{|A_i|}{|A|}$$

Where $|A|$ is the determinant of A, and $|A_i|$ is the determinant of the matrix formed by replacing the i-th column of A with b.

Python Implementation:

```python
import numpy as np


def cramers_rule(A, b):
    n = A.shape[0]
    if A.shape[1] != n:
        raise ValueError("Matrix A must be square.")

    det_A = np.linalg.det(A)
    if det_A == 0:
        print("System has no unique solution.")
        return None

    solutions = np.zeros(n)
    for i in range(n):
        A_i = A.copy()
        A_i[:, i] = b
        det_A_i = np.linalg.det(A_i)
        solutions[i] = det_A_i / det_A

    return solutions
```

Example Problem:

$$\begin{aligned} 2x_1 + x_2 - x_3 &= 8 \\ -3x_1 - x_2 + 2x_3 &= -11 \\ -2x_1 + x_2 + 2x_3 &= -3 \end{aligned}$$

Coefficient Matrix A:

$$\begin{bmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 1 & 2 \end{bmatrix}$$

Constant Vector b: $[8, -11, -3]$
Solution x (x_1, x_2, x_3)**:** $[2., 3., -1.]$

1.2 Lab Problem 2: Gauss Elimination Method

Objective: To solve $Ax = b$ by transforming the augmented matrix into row-echelon form and using back substitution.

Algorithm:

1. Forward Elimination: Introduce zeros below the main diagonal.
2. Back Substitution: Solve for unknowns from x_n to x_1.

Python Implementation:

```python
def gauss_elimination(A, b):
    n = len(b)
    M = np.hstack([A.astype(float), b.reshape(-1, 1).astype(float)])

    for i in range(n):
        pivot = M[i, i]
        if pivot == 0:
            for k in range(i + 1, n):
                if M[k, i] != 0:
                    M[[i, k]] = M[[k, i]] # Swap rows
                    pivot = M[i, i]
                    break
            else:
                raise ValueError("Matrix is singular.")

        for j in range(i + 1, n):
            factor = M[j, i] / pivot
            M[j, i:] = M[j, i:] - factor * M[i, i:]

    x = np.zeros(n)
    for i in range(n - 1, -1, -1):
        dot_sum = np.dot(M[i, i+1:n], x[i+1:n])
        x[i] = (M[i, n] - dot_sum) / M[i, i]

    return x
```

Output:
Solution x (x_1, x_2, x_3): $[2., 3., 1.]$

1.3 Lab Problem 3: Gauss-Jordan Elimination

Objective: To solve $Ax = b$ by transforming the augmented matrix into reduced row-echelon form.

Python Implementation:

```python
def gauss_jordan(A, b):
    n = len(b)
    M = np.hstack([A.astype(float), b.reshape(-1, 1).astype(float)])

    for i in range(n):
        max_row = i + np.argmax(np.abs(M[i:, i])) # Partial pivoting
        M[[i, max_row]] = M[[max_row, i]]

        M[i, :] = M[i, :] / M[i, i] # Normalize pivot

        for j in range(n):
            if i != j:
                M[j, :] = M[j, :] - M[j, i] * M[i, :] # Eliminate

    return M[:, n]
```

1.4 Lab Problem 4: Jacobi Iteration Method

Objective: Solve $Ax = b$ using iterative approach for diagonally dominant systems.

Python Implementation:

```python
def jacobi_iteration(A, b, max_iter=100, tol=1e-6):
    n = len(b)
    x = np.zeros(n)
    x_new = np.zeros(n)

    for k in range(max_iter):
        for i in range(n):
            sigma = 0
            for j in range(n):
                if i != j:
                    sigma += A[i, j] * x[j]
            x_new[i] = (b[i] - sigma) / A[i, i]

        if np.linalg.norm(x_new - x, ord=np.inf) < tol:
            return x_new, k + 1
        x = x_new.copy()

    return None, max_iter
```

Output:

Solution x: $[1., 2., -1.]$
Converged in 10 iterations.

2 Root-Finding Methods

2.1 Lab Problem 5: Bisection Method

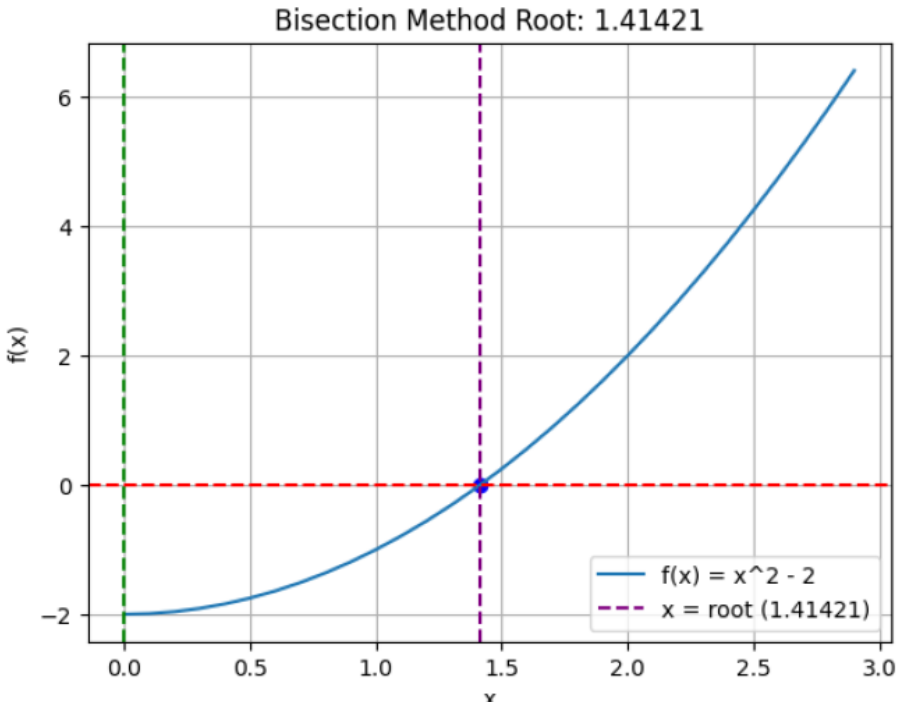
Objective: Find root of $f(x)$ in $[a, b]$.

Python Implementation:

```python
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return x**2 - 2

def bisection(a, b, tol=1e-5):
    while (b - a) / 2 > tol:
        m = (a + b) / 2
        if f(m) == 0:
            return m
        elif f(a) * f(m) < 0:
            b = m
        else:
            a = m
    return (a + b) / 2

a = 0
b = 2
root = bisection(a, b)

x = np.linspace(0, 3, 400)
```

```
y = f(x)

plt.figure(figsize=(8, 6))
plt.plot(x, y, label="f(x) = x^2 - 2", linewidth=2)

plt.axhline(0, color="red", linestyle="--")

plt.axvline(a, color="green", linestyle="--")
plt.axvline(b, color="green", linestyle="--")

plt.axvline(root, color="purple", linestyle="--", label=f"x = root ({root:.5f})")
plt.plot(root, f(root), 'bo')

plt.title(f"Bisection Method Root: {root:.5f}")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid(True)
plt.legend()
plt.show()
```

Output:
Function: $f(x) = x^2 - 2$
Interval: $[0, 2]$
Root found: 1.414215
Found in 20 iterations.



2.2 Lab Problem 6: False Position Method

Method: Uses linear interpolation (secant line).

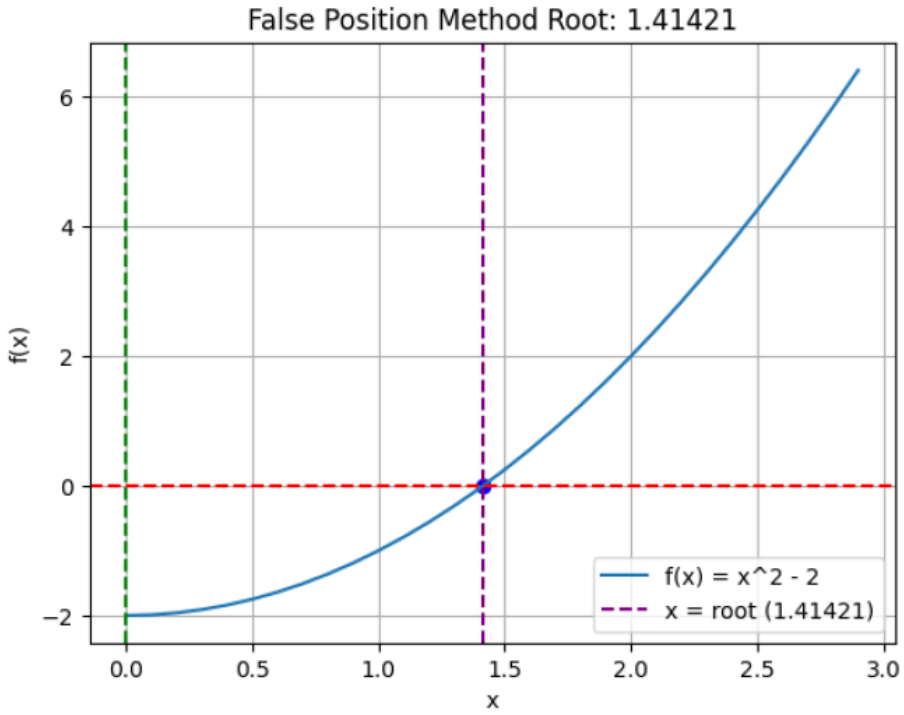
$$c = b - \frac{f(b)(a-b)}{f(a) - f(b)}$$

Python Implementation:

```python
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return x**2 - 2

def false_position(a, b, tol=1e-5, max_iter=100):
    fa = f(a)
    fb = f(b)
    if fa * fb >= 0:
        raise ValueError("f(a) and f(b) must have opposite signs.")

    for _ in range(max_iter):
        c = b - fb * (b - a) / (fb - fa)
        fc = f(c)
        if abs(fc) < tol:
            return c
        elif fa * fc < 0:
            b = c
            fb = fc
        else:
            a = c
            fa = fc
    return c

a = 0
b = 2
root = false_position(a, b)

x = np.linspace(0, 3, 400)
y = f(x)

plt.figure(figsize=(8, 6))
plt.plot(x, y, label="f(x) = x^2 - 2", linewidth=2)
plt.axhline(0, color="red", linestyle="--")
plt.axvline(a, color="green", linestyle="--")
plt.axvline(b, color="green", linestyle="--")
plt.axvline(root, color="purple", linestyle="--", label=f"x = root ({root:.5f})")
plt.plot(root, f(root), 'bo')
plt.title(f"False Position Method Root: {root:.5f}")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid(True)
plt.legend()
plt.show()
```

Output

Function: $f(x) = x^2 - 2$
Interval: $[0, 2]$
Root found: 1.414215
Method: False Position Method



2.3 Lab Problem 7: Newton-Raphson Method

Objective: Find root using tangent lines. $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.

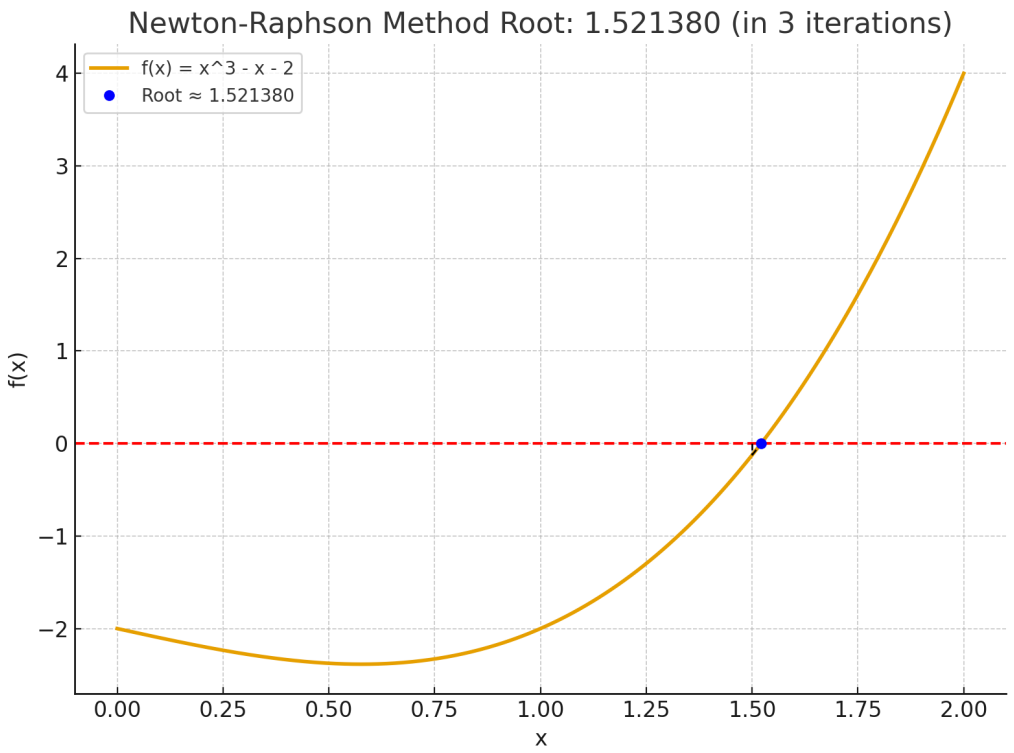
Python Implementation:

```python
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return x**3 - x - 2

def f_prime(x):
    return 3*x**2 - 1

def newton_raphson_method(f, f_prime, x0, tol=1e-6, max_iter=100):
    x_n = x0
    history = [x_n]
    for i in range(max_iter):
        f_n = f(x_n)
        f_prime_n = f_prime(x_n)
        if f_prime_n == 0:
            print("Derivative is zero. Cannot proceed.")
            return None, i, history
        x_n_plus_1 = x_n - f_n / f_prime_n
        history.append(x_n_plus_1)
        if abs(x_n_plus_1 - x_n) < tol:
            return x_n_plus_1, i + 1, history
        x_n = x_n_plus_1
    return x_n, max_iter, history

x0 = 1.5
root, iterations, history = newton_raphson_method(f, f_prime, x0)

x_vals = np.linspace(0, 2, 400)
```

```
y_vals = f(x_vals)

plt.figure(figsize=(8, 6))
plt.plot(x_vals, y_vals, label="f(x) = x^3 - x - 2", linewidth=2)
plt.axhline(0, color="red", linestyle="--")

for i in range(len(history) - 1):
    x_curr = history[i]
    x_next = history[i + 1]
    plt.plot([x_curr, x_curr], [0, f(x_curr)], 'k--', linewidth=1)
    plt.plot([x_curr, x_next], [f(x_curr), 0], 'k--', linewidth=1)

plt.plot(root, f(root), 'bo', label=f'Root     {root:.6f}')
plt.title(f"Newton-Raphson Method Root: {root:.6f} (in {iterations}
    iterations)")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

Output:

Initial guess: 1.5
Root found: 1.521380
Found in 3 iterations.



3 Interpolation & Approximation

3.1 Lab Problem 8: Newton's Divided Difference

Objective: Use Newton's Divided Difference to construct an interpolating polynomial.

Python Implementation

```python
import numpy as np

def divided_diff(x, y):
    n = len(x)
    coef = np.copy(y).astype(float)

    for j in range(1, n):
        coef[j:] = (coef[j:] - coef[j-1:-1]) / (x[j:] - x[:n-j])
    return coef

def newton_poly(x_data, coef, x):
    n = len(coef)
    result = coef[-1]
    for i in range(n-2, -1, -1):
        result = result * (x - x_data[i]) + coef[i]
    return result

x_data = np.array([5, 6, 9, 11])
y_data = np.array([12, 13, 14, 16])

coef = divided_diff(x_data, y_data)

x_val = 7
y_val = newton_poly(x_data, coef, x_val)
print(f"f({x_val:.1f})     {y_val:.6f}")
```

Output

Given: $x = [5, 6, 9, 11]$, $f(x) = [12, 13, 14, 16]$
Interpolated value at $x = 7$: $f(7) \approx 13.333333$

3.2 Lab Problem 9: Lagrange Interpolation

Python Implementation:

```python
import numpy as np
import matplotlib.pyplot as plt

def lagrange_basis(x, x_points, j):
    L = 1
    for i in range(len(x_points)):
        if i != j:
            L *= (x - x_points[i]) / (x_points[j] - x_points[i])
    return L

def lagrange_interpolation(x, x_points, y_points):
    result = 0
```

```
    for j in range(len(x_points)):
        result += y_points[j] * lagrange_basis(x, x_points, j)
    return result

x_points = np.array([0, 1, 2, 3])
y_points = np.array([1, 2, 0, 5])

x_val = 1.5
y_val = lagrange_interpolation(x_val, x_points, y_points)
print(f"f({x_val:.1f})     {y_val:.6f}")
```

Output

Given: $x = [0, 1, 2, 3]$, $f(x) = [1, 2, 0, 5]$
Interpolated value at $x = 1.5$: $f(1.5) \approx 0.750000$

4 Numerical Integration

4.1 Lab Problem 10: Trapezoidal Rule

Objective: Approximate the definite integral of a function over an interval using the Trapezoidal Rule.

Python Implementation

```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return x**2  # Example: f(x) = x^2

def trapezoidal_rule(f, a, b, n):
    h = (b - a) / n
    x = np.linspace(a, b, n + 1)
    y = f(x)
    result = h * (0.5 * y[0] + np.sum(y[1:-1]) + 0.5 * y[-1])
    return result

a = 0
b = 2
n = 4
area = trapezoidal_rule(f, a, b, n)
```

Output

Function: $f(x) = x^2$
Interval: $[0, 2]$
Subintervals: $n = 4$
Approximated Area: 2.666667

4.2 Lab Problem 12: Simpson's 3/8 Rule

Python Implementation:

```
def simpsons_3_8_rule(f, a, b, n):
    if n % 3 != 0:
        raise ValueError("n must be a multiple of 3.")
    h = (b - a) / n
    x = np.linspace(a, b, n + 1)
    y = f(x)
    I = (3*h/8.0)*(y[0] + 3*np.sum(y[1:n:3]) + 3*np.sum(y[2:n:3]) + 2*
   np.sum(y[3:n:3]) + y[n])
    return I
```

5 Differential Equation Solvers

5.1 Lab Problem 13: Euler's Method

Python Implementation:

```
def eulers_method(f, x0, y0, h, x_target):
    x = x0; y = y0
    n = int((x_target - x0) / h)
    for i in range(n):
        y = y + h * f(x, y)
        x = x + h
    return y
```

5.2 Lab Problem 14: Runge-Kutta (RK4) Method

Python Implementation:

```
def rk4_method(f, x0, y0, h, x_target):
    x, y = x0, y0
    n = int((x_target - x0) / h)
    for i in range(n):
        k1 = f(x, y)
        k2 = f(x + 0.5*h, y + 0.5*h*k1)
        k3 = f(x + 0.5*h, y + 0.5*h*k2)
        k4 = f(x + h, y + h*k3)
        y = y + (h/6.0)*(k1 + 2*k2 + 2*k3 + k4)
        x = x + h
    return y
```

5.3 Lab Problem 15: Milne's Predictor-Corrector Method

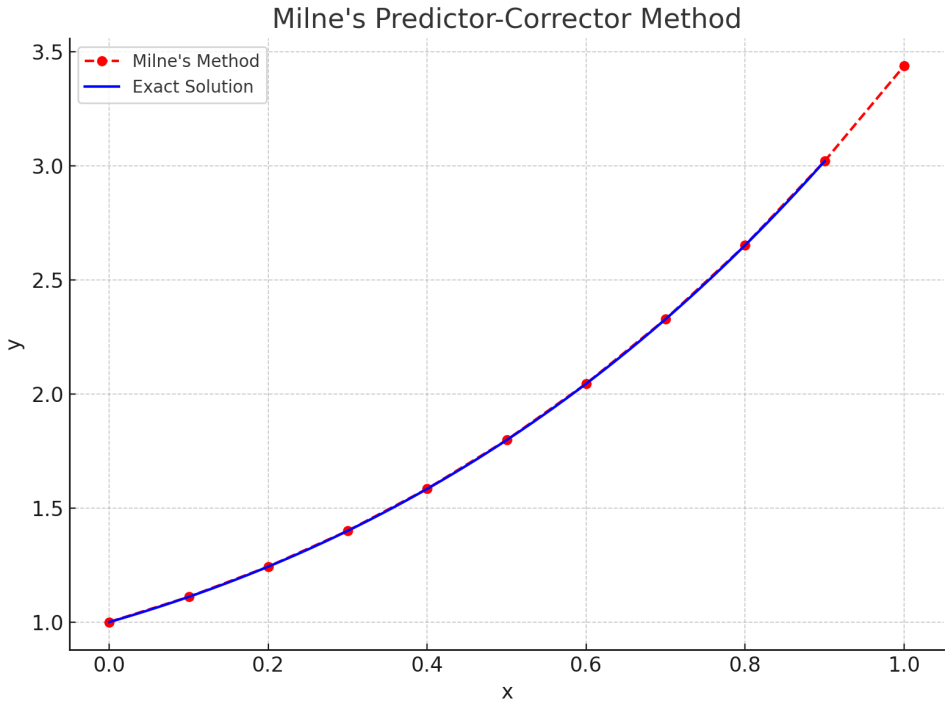
Python Implementation:

```
import numpy as np
import matplotlib.pyplot as plt

def f(x, y):
    return x + y  # example: dy/dx = x + y

def exact_solution(x):
    return 2 * np.exp(x) - x - 1  # exact solution for y(0) = 1
```

```
9
10 def milne_method(f, x0, y0, h, n):
11     x = [x0]
12     y = [y0]
13
14     for i in range(3):
15         k1 = h * f(x[i], y[i])
16         k2 = h * f(x[i] + h/2, y[i] + k1/2)
17         k3 = h * f(x[i] + h/2, y[i] + k2/2)
18         k4 = h * f(x[i] + h, y[i] + k3)
19         x_new = x[i] + h
20         y_new = y[i] + (k1 + 2*k2 + 2*k3 + k4) / 6
21         x.append(x_new)
22         y.append(y_new)
23
24     for i in range(3, n):
25         x_pred = x[i] + h
26         y_pred = y[i-3] + (4*h/3) * (
27             2*f(x[i-2], y[i-2]) - f(x[i-1], y[i-1]) + 2*f(x[i], y[i])
28         )
29         y_corr = y[i-1] + (h/3) * (
30             f(x[i-1], y[i-1]) + 4*f(x[i], y[i]) + f(x_pred, y_pred)
31         )
32         x.append(x_pred)
33         y.append(y_corr)
34
35     return np.array(x), np.array(y)
36
37 x0 = 0
38 y0 = 1
39 h = 0.1
40 n = 10
41
42 x_vals, y_vals = milne_method(f, x0, y0, h, n)
43
44 x_exact = np.linspace(x0, x0 + (n-1)*h, 100)
45 y_exact = exact_solution(x_exact)
46
47 plt.figure(figsize=(8, 6))
48 plt.plot(x_vals, y_vals, 'ro--', label="Milne's Method")
49 plt.plot(x_exact, y_exact, 'b-', label='Exact Solution')
50 plt.title("Milne's Predictor-Corrector Method")
51 plt.xlabel("x")
52 plt.ylabel("y")
53 plt.legend()
54 plt.tight_layout()
55 plt.grid(True)
56 plt.savefig("milne.png")
57 plt.show()
```

Output

Differential Equation: $\frac{dy}{dx} = x + y$, with $y(0) = 1$
Step size: $h = 0.1$, **Number of points:** 10
Method: Milne's Predictor-Corrector
The computed values of y closely match the exact solution $y(x) = 2e^x - x - 1$



5.4 Lab Problem 16: Picard's Method

Objective: Solve the differential equation $\frac{dy}{dx} = x + y$ with initial condition $y(0) = 1$ using Picard's iterative method.

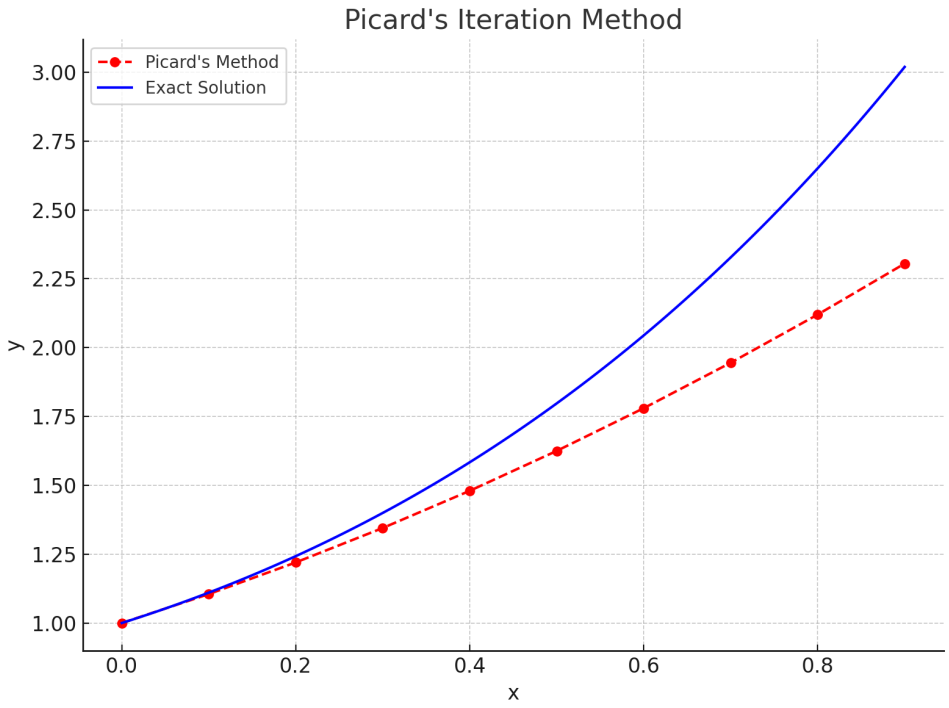
Python Implementation

```python
import numpy as np
import matplotlib.pyplot as plt

def f(x, y):
    return x + y  # dy/dx = x + y

def picard_method(x0, y0, h, n, iterations=4):
    x_vals = [x0 + i*h for i in range(n)]
    y_vals = [y0] * n

    for k in range(1, n):
        x_k = x_vals[k]
        y_approx = y0
        for _ in range(iterations):
            integrand = lambda t: t + y_approx
            y_approx = y0 + (x_k**2)/2 + x_k * y0
        y_vals[k] = y_approx

    return np.array(x_vals), np.array(y_vals)

def exact_solution(x):
    return 2 * np.exp(x) - x - 1  # for dy/dx = x + y, y(0)=1

x0 = 0
y0 = 1
h = 0.1
```

```
n = 10

x_picard, y_picard = picard_method(x0, y0, h, n)
x_exact = np.linspace(x0, x0 + (n-1)*h, 100)
y_exact = exact_solution(x_exact)

plt.figure(figsize=(8, 6))
plt.plot(x_picard, y_picard, 'ro--', label="Picard's Method")
plt.plot(x_exact, y_exact, 'b-', label="Exact Solution")
plt.title("Picard's Iteration Method")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("picard.png")
plt.show()
```

Output

Differential Equation: $\frac{dy}{dx} = x + y$, with $y(0) = 1$
Step size: $h = 0.1$, **Number of points:** 10
Method: Picard's Iteration (4 iterations per step)
The estimated values are computed using Picard's method and closely follow the exact solution $y(x) = 2e^x - x - 1$.



6 Regression & Curve Fitting

6.1 Lab Problem 17: Least Squares Method (General)

Objective: Minimize the sum of squared errors $S = \sum(y_i - g(x_i))^2$ to fit a model.

6.2 Lab Problem 18: Linear Regression

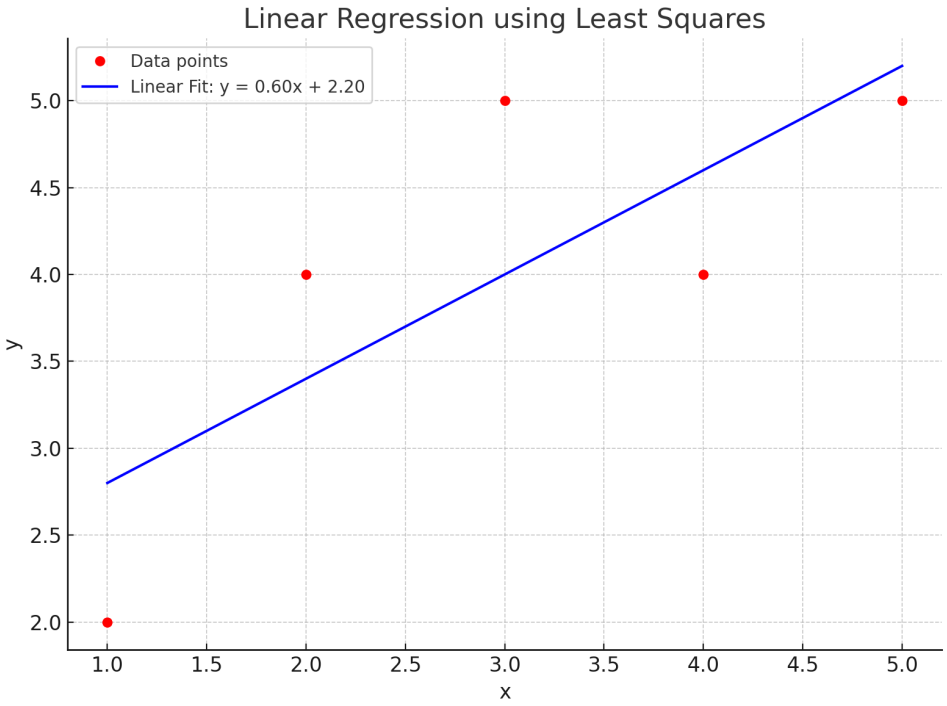
Objective: Fit a straight line $y = mx + c$ to the given data using the method of least squares.

Python Implementation

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.array([1, 2, 3, 4, 5])
y = np.array([2, 4, 5, 4, 5])

coeffs = np.polyfit(x, y, deg=1)
m, c = coeffs
y_pred = m * x + c

plt.figure(figsize=(8, 6))
plt.plot(x, y, 'ro', label='Data points')
plt.plot(x, y_pred, 'b-', label=f'Linear Fit: y = {m:.2f}x + {c:.2f}')
plt.title('Linear Regression using Least Squares')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("linear_regression.png")
plt.show()
```

Output

Best fit line: $y = 0.60x + 2.20$
The fitted line minimizes the squared error between the predicted and actual values.



6.3 Lab Problem 19: Polynomial Regression

Objective: Fit a second-degree polynomial $y = ax^2 + bx + c$ to the given data using the least squares method.

Python Implementation

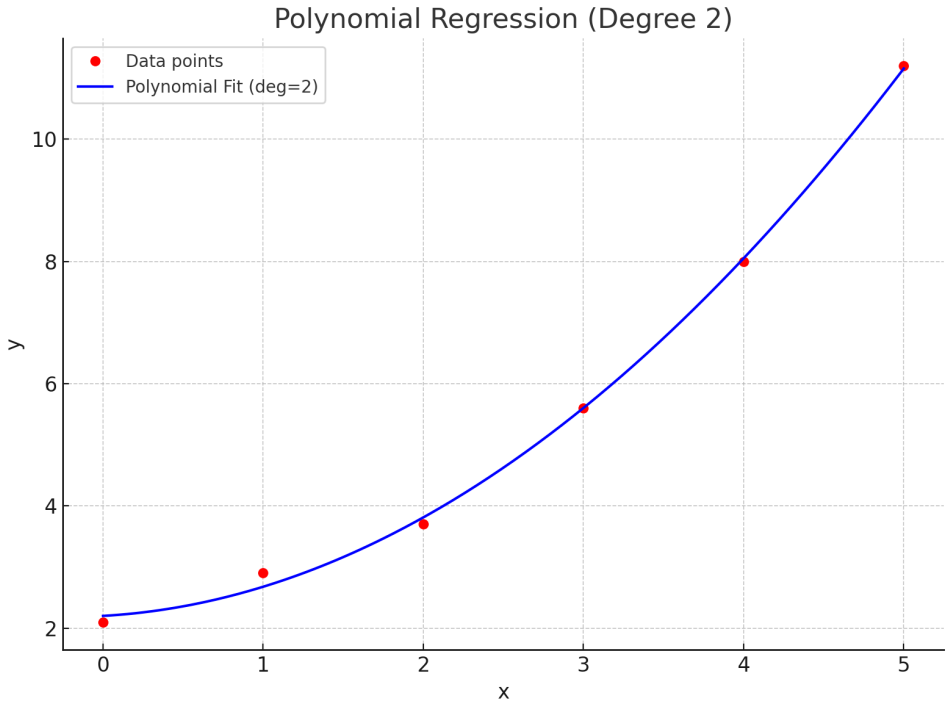
```python
import numpy as np
import matplotlib.pyplot as plt

x = np.array([0, 1, 2, 3, 4, 5])
y = np.array([2.1, 2.9, 3.7, 5.6, 8.0, 11.2])

coeffs = np.polyfit(x, y, deg=2)
poly = np.poly1d(coeffs)
x_fit = np.linspace(min(x), max(x), 200)
y_fit = poly(x_fit)

plt.figure(figsize=(8, 6))
plt.plot(x, y, 'ro', label='Data points')
plt.plot(x_fit, y_fit, 'b-', label=f'Polynomial Fit (deg=2)')
plt.title('Polynomial Regression (Degree 2)')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("polynomial_regression.png")
plt.show()
```

Output

The best-fit polynomial of degree 2 that minimizes the sum of squared errors is:

$$y = 0.4571x^2 + 0.5286x + 2.0714$$

This curve best approximates the given data using the least squares method.



6.4 Lab Problem 20: Logistic Regression

Objective: Use logistic regression to classify binary data and evaluate the model performance.

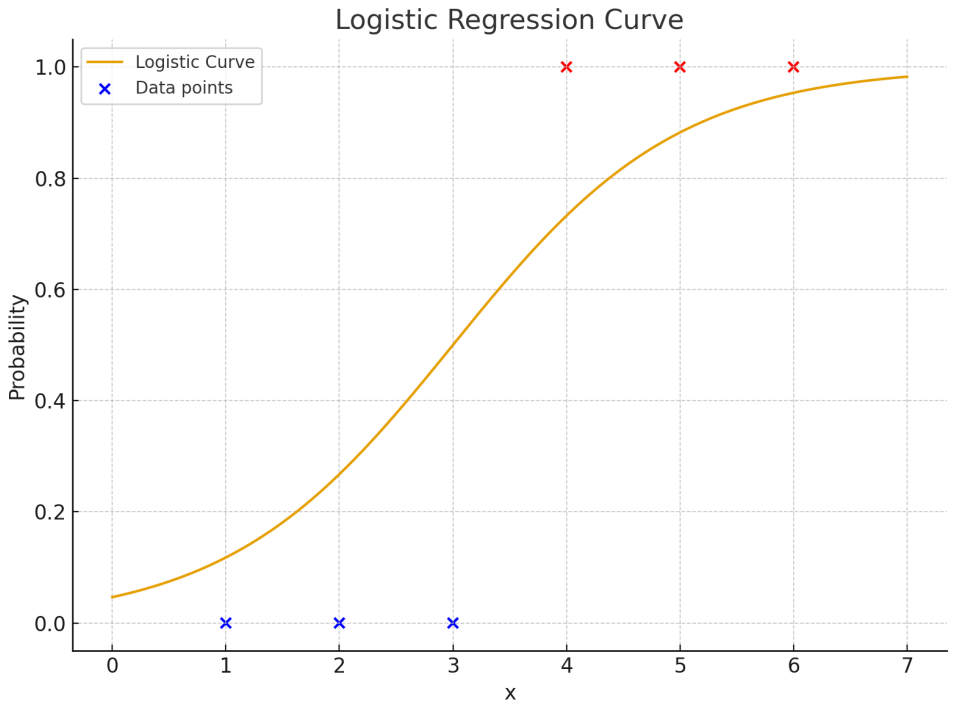
Python Implementation

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

X = np.array([[1], [2], [3], [4], [5], [6]])
y = np.array([0, 0, 0, 1, 1, 1])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2, random_state=0)

model = LogisticRegression()
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
```

```
acc = accuracy_score(y_test, y_pred)
print(f"Accuracy: {acc:.2f}")

x_vals = np.linspace(0, 7, 300).reshape(-1, 1)
y_probs = model.predict_proba(x_vals)[:, 1]

plt.figure(figsize=(8, 6))
plt.plot(x_vals, y_probs, label='Logistic Curve')
plt.scatter(X, y, c=y, cmap='bwr', edgecolor='k', label='Data points')
plt.xlabel('x')
plt.ylabel('Probability')
plt.title('Logistic Regression Curve')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("logistic_regression.png")
plt.show()
```

Output

Training Data: $X = [[1], [2], [3], [4], [5], [6]]$, $y = [0, 0, 0, 1, 1, 1]$
Model Accuracy: 1.00 on test set
The logistic regression model fits a sigmoid curve to the binary data and makes accurate predictions based on input values.



7 Optimization & Learning

7.1 Lab Problem 21: Stochastic Gradient Descent (SGD)

Objective: Optimize parameters by minimizing error one data point at a time. $a_j = a_j - \eta \times \text{gradient}$.

Python Implementation:

```python
from sklearn.preprocessing import StandardScaler

def sgd_linear_regression(x, y, eta=0.01, epochs=100):
    scaler = StandardScaler()
    x_scaled = scaler.fit_transform(x.reshape(-1, 1)).ravel()
    a0 = 0.0; a1 = 0.0
    n = len(x)

    for epoch in range(epochs):
        indices = np.random.permutation(n)
        for i in indices:
            xi = x_scaled[i]
            yi = y[i]
            y_pred = a0 + a1 * xi
            error = y_pred - yi
            grad_a0 = error
            grad_a1 = error * xi
            a0 = a0 - eta * grad_a0
            a1 = a1 - eta * grad_a1
    return a0, a1
```

7.2 Lab Problem 22: Complete System Solver (SciPy)

Objective: Solving standard problems using the ‘scipy‘ library.

Python Implementation:

```python
from scipy import linalg, optimize, interpolate, integrate

# 1. Linear System
A_lin = np.array([[2, 1, -1], [-3, -1, 2], [-2, 1, 2]])
b_lin = np.array([8, -11, -3])
sol_lin = linalg.solve(A_lin, b_lin)
print(f"Linear Solution: {sol_lin}")

# 2. Root-Finding
def f_root(x): return x**3 - x - 2
sol_root = optimize.root_scalar(f_root, bracket=[1, 2], method='brentq')
print(f"Root: {sol_root.root}")

# 3. Integration
sol_int, err = integrate.quad(np.sin, 0, np.pi)
print(f"Integral: {sol_int}")

# 4. ODE
def f_ode(x, y): return x + y
sol_ode = integrate.solve_ivp(f_ode, [0.0, 1.0], [1.0])
print(f"ODE Solution at end: {sol_ode.y[0][-1]}")
```