**Unit - 3**

**Milne method:**

def milne\_method(f, y0, t0, tn, h):

n = int((tn - t0) / h)

t = [t0 + i \* h for i in range(n + 1)]

y = [y0] \* (n + 1)

for i in range(3):

k1 = h \* f(t[i], y[i])

k2 = h \* f(t[i] + h/2, y[i] + k1/2)

k3 = h \* f(t[i] + h/2, y[i] + k2/2)

k4 = h \* f(t[i+1], y[i] + k3)

y[i+1] = y[i] + (k1 + 4\*k2 + k3 + k4) / 6

for i in range(3, n):

y[i+1] = y[i-1] + 2\*h/3 \* (f(t[i], y[i]) + 4\*f(t[i-1], y[i-1]) + f(t[i-2], y[i-2]))

return t, y

def f(t, y):

return -2 \* t \* y + 4 \* t - 3

t, y = milne\_method(f, 1, 0, 1, 0.1)

# Print the results

for i in range(len(t)):

print(f"t = {t[i]}, y = {y[i]}")

**Adam-Bashforth Method:**

def adams\_bashforth(f, x0, y0, h, n):

results = [(x0, y0)]

x = x0

y = y0

for i in range(1, n + 1):

k1 = h \* f(x, y)

k2 = h \* f(x - h, results[-1][1])

y += k1

y -= 1/2 \* k2

x += h

results.append((x, y))

return results

# Example usage

def ode\_function(x, y):

return x \* y

# Initial values

x0 = 0

y0 = 1

# Step size and number of iterations

h = 0.1

n = 10

# Solve the ODE using Adams-Bashforth method

solution = adams\_bashforth(ode\_function, x0, y0, h, n)

# Print the solution

for x, y in solution:

print(f"x = {x:.2f}, y = {y:.6f}")

**Runge-Kutta 2nd Order Method:**

import numpy as np

import matplotlib.pyplot as plt

def f(t, y):

return t - y

def runge\_kutta\_2nd\_order(f, t0, y0, h, num\_steps):

t\_values = np.zeros(num\_steps + 1)

y\_values = np.zeros(num\_steps + 1)

t\_values[0] = t0

y\_values[0] = y0

for i in range(num\_steps):

t = t\_values[i]

y = y\_values[i]

# Calculate the slope at the current point

k1 = h \* f(t, y)

# Calculate the slope at the midpoint

k2 = h \* f(t + h/2, y + k1/2)

# Update the next y value using the weighted average of the slopes

y\_next = y + k2

# Update t and y values

t\_values[i+1] = t + h

y\_values[i+1] = y\_next

return t\_values, y\_values

# Define the initial conditions and parameters

t0 = 0

y0 = 1

h = 0.1

num\_steps = 10

t\_values, y\_values = runge\_kutta\_2nd\_order(f, t0, y0, h, num\_steps)

for t, y in zip(t\_values, y\_values):

print(f"t = {t}, y = {y}")

**Runge-Kutta 4th Order Method:**

import numpy as np

import matplotlib.pyplot as plt

def f(t, y):

return t - y

def runge\_kutta\_4th\_order(f, t0, y0, h, num\_steps):

t\_values = np.zeros(num\_steps + 1)

y\_values = np.zeros(num\_steps + 1)

t\_values[0] = t0

y\_values[0] = y0

for i in range(num\_steps):

t = t\_values[i]

y = y\_values[i]

# Calculate the slopes at different points

k1 = h \* f(t, y)

k2 = h \* f(t + h/2, y + k1/2)

k3 = h \* f(t + h/2, y + k2/2)

k4 = h \* f(t + h, y + k3)

# Update the next y value using the weighted average of the slopes

y\_next = y + (k1 + 2\*k2 + 2\*k3 + k4) / 6

# Update t and y values

t\_values[i+1] = t + h

y\_values[i+1] = y\_next

return t\_values, y\_values

# Define the initial conditions and parameters

t0 = 0

y0 = 1

h = 0.1

num\_steps = 10

# Solve the ODE using Runge-Kutta 4th Order Method

t\_values, y\_values = runge\_kutta\_4th\_order(f, t0, y0, h, num\_steps)

for t, y in zip(t\_values, y\_values):

print(f"t = {t}, y = {y}")

**Unit - 4**

**Dirichlet’s method:**

from scipy.integrate import solve\_ivp

def dirichlet\_method(f, a, b, alpha, beta, n):

def ode(t, y):

return [y[1], f(t, y[0])]

def bc(ya, yb):

return [ya[0] - alpha, yb[0] - beta]

t\_span = [a, b]

t\_eval = np.linspace(a, b, n)

y\_guess = [alpha, (beta - alpha) / (b - a)]

sol = solve\_ivp(ode, t\_span, y\_guess, t\_eval=t\_eval, bc=bc)

return sol.t, sol.y[0]

import numpy as np

def f(t, y):

return y

a = 0

b = 1

alpha = 1

beta = 2

n = 10

t, y = dirichlet\_method(f, a, b, alpha, beta, n)

# Print the results

for i in range(len(t)):

print(f"t = {t[i]}, y = {y[i]}")

**Neumann’s method:**

import numpy as np

from scipy.linalg import solve

def neumann\_method(f, a, b, alpha, beta, n):

h = (b - a) / n

x = np.linspace(a, b, n+1)

A = np.zeros((n-1, n-1))

b = np.zeros(n-1)

for i in range(1, n):

A[i-1, i-1] = 2 / h\*\*2

b[i-1] = f(x[i])

A[0, 0] = 1 / h\*\*2

A[-1, -1] = 1 / h\*\*2

b[0] -= alpha / h\*\*2

b[-1] -= beta / h\*\*2

y = np.concatenate(([alpha], solve(A, b), [beta]))

return x, y

def f(x):

return -x

a = 0

b = 1

alpha = 1

beta = 2

n = 10

x, y = neumann\_method(f, a, b, alpha, beta, n)

# Print the results

for i in range(len(x)):

print(f"x = {x[i]}, y = {y[i]}")

**Robin’s/Cauchy’s method:**

def cauchy\_method(f, x0, y0, h, n):

results = [(x0, y0)]

for i in range(n):

x\_prev, y\_prev = results[-1]

x = x\_prev + h

y = y\_prev + h \* f(x\_prev, y\_prev)

results.append((x, y))

return results

# Example usage

def ode\_function(x, y):

return x \* y

# Initial values

x0 = 0

y0 = 1

# Step size and number of iterations

h = 0.1

n = 10

# Solve the ODE using Cauchy's method

solution = cauchy\_method(ode\_function, x0, y0, h, n)

# Print the solution

for x, y in solution:

print(f"x = {x:.2f}, y = {y:.6f}")