SYLLABUS

SCILAB & Python Algorithm of:

- **Interpolation**

1. Newton Forward Method

2. Newton Backward Method

3. Langrange's Method

- **Solution of linear Equations**

4. Gauss-Elimination/Gauss Jordan Method

5. Gauss Jacobi/Gauss Siedel method

- **Solution of Non-linear Equation**

6. Bisection Method/Secant Method/Regula Falsi Method

7. Newton Raphson Method

- **Solution of Ordinary Differential Equations (1st order, Initial Value Problem)**

8. Eulers method

9. Modified Eulers Method

10. Runge-Kutta Method

- **Solution of Ordinary Differential Equations (2^{nd} order, Boundary Value Problem)**

11. Finite Difference Method

- **Solution of Partial Differential Equations**

12. Bender Schmidt Method

Notes for MID-SEM

Solution of Ordinary Differential Equations (ODE), First Order-Initial Value Problem (IVP)

Scilab

```
//Algorithm to compare the results of ODE function and Eulers results
clc; clear;clf()

k = 0.05; CA0 = 1.0;  t0 = 0; tf = 100; h = 5;

function dC=f(t, C)
   dC = -k * C^2
endfunction

t = t0:h:tf
n = length(t)-1
t_euler = zeros(1,n+1)
CA_euler = zeros(1,n+1)
t_euler(1) = t0
CA_euler(1) = CA0

for i = 1:n
   t_euler(i+1) = t_euler(i) + h
   CA_euler(i+1) = CA_euler(i) + h * f(t_euler(i), CA_euler(i))
end

C_num = ode(CA0, t0, t, f)

scf(0)
plot(t, C_num, 'r-', 'LineWidth', 2)
plot(t, CA_euler, 'b--', 'LineWidth', 2)
xlabel("Time (s)")
ylabel("Concentration C_A (mol/L)")
title("Second-order Batch Reactor: Numerical vs Analytical Solution")
legend("Numerical (ODE solver)", "Euler Method")
xgrid()

//Algorithm to solve the Eulers method, Modified Eulers Method, RK-4 Method
clc; clear;clf()
```

```
k=0.05
function dCa=f(t, Ca)
   dCa = -k * Ca^2
endfunction

t0 = 0; Ca0 = 1; tf = 100; h = 5
n = (tf - t0) / h

//Eulers method
t_euler = t0
Ca_euler = Ca0

for i = 1:n
   t_euler(i+1) = t_euler(i) + h
   Ca_euler(i+1) = Ca_euler(i) + h * f(t_euler(i), Ca_euler(i))
end

//Modified Eulers method
t_mod_euler = t0
Ca_mod_euler = Ca0
for i = 1:n
   t_mod_euler(i+1) = t_mod_euler(i) + h
   Ca_pred = Ca_mod_euler(i) + h * f(t_mod_euler(i), Ca_mod_euler(i))
   Ca_mod_euler(i+1) = Ca_mod_euler(i) + (h/2) * (f(t_mod_euler(i),
Ca_mod_euler(i)) + f(t_mod_euler(i+1), Ca_pred))
end

//RK-4 Method
t_rk4 = t0
Ca_rk4 = Ca0
for i = 1:n
   k1 = h * f(t_rk4(i), Ca_rk4(i))
   k2 = h * f(t_rk4(i) + h/2, Ca_rk4(i) + k1/2)
   k3 = h * f(t_rk4(i) + h/2, Ca_rk4(i) + k2/2)
   k4 = h * f(t_rk4(i) + h, Ca_rk4(i) + k3)
   Ca_rk4(i+1) = Ca_rk4(i) + (k1 + 2*k2 + 2*k3 + k4) / 6
   t_rk4(i+1) = t_rk4(i) + h
end

plot(t_euler, Ca_euler, '-ob')
plot(t_mod_euler, Ca_mod_euler, '-sg')
plot(t_rk4, Ca_rk4, '-^r')
xlabel("x")
ylabel("y")
title("Comparison of Methods")
```

```
legend(["Euler", "Modified Euler", "RK-4"])
xgrid()
```

Python

```
#Algorithm to compare the results of ODE function and Eulers results
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

k = 0.05; CA0 = 1.0; t0 = 0; tf = 100; h = 5;

def f(C, t):
    return -k * C**2

t = np.arange(t0, tf + h, h)
n = len(t) - 1

CA_euler = np.zeros(n+1)
CA_euler[0] = CA0

for i in range(n):
    CA_euler[i+1] = CA_euler[i] + h * f(CA_euler[i], t[i])
C_num = odeint(f, CA0, t).flatten()

plt.figure(figsize=(8,6))
plt.plot(t, C_num, 'r-', linewidth=2, label="Numerical (ODE solver)")
plt.plot(t, CA_euler, 'b--', linewidth=2, label="Euler Method")
plt.xlim(0,100)
plt.ylim(0,1)
plt.xlabel("Time (s)")
plt.ylabel("Concentration C_A (mol/L)")
plt.title("Second-order Batch Reactor: Numerical vs Euler Solution")
plt.legend()
plt.grid(True)
plt.show()
```

```
#Comparison of Results of Eulers method, Modified Eulers
Method & RK-4 Method
  import numpy as np
  import matplotlib.pyplot as plt

  k = 0.05

  def f(t, Ca):
      return -k * Ca**2

  t0 = 0; Ca0 = 1; tf = 100; h = 5
  n = int((tf - t0) / h)

  t_euler = np.zeros(n+1)
  Ca_euler = np.zeros(n+1)
  t_euler[0], Ca_euler[0] = t0, Ca0

  for i in range(n):
      t_euler[i+1] = t_euler[i] + h
      Ca_euler[i+1] = Ca_euler[i] + h * f(t_euler[i],
      Ca_euler[i])

  t_mod_euler = np.zeros(n+1)
  Ca_mod_euler = np.zeros(n+1)
  t_mod_euler[0], Ca_mod_euler[0] = t0, Ca0

  for i in range(n):
      t_mod_euler[i+1] = t_mod_euler[i] + h
      Ca_pred = Ca_mod_euler[i] + h * f(t_mod_euler[i],
      Ca_mod_euler[i])
      Ca_mod_euler[i+1] = Ca_mod_euler[i] + (h/2) *
      (f(t_mod_euler[i], Ca_mod_euler[i]) +
      f(t_mod_euler[i+1], Ca_pred))

  t_rk4 = np.zeros(n+1)
  Ca_rk4 = np.zeros(n+1)
  t_rk4[0], Ca_rk4[0] = t0, Ca0

  for i in range(n):
      k1 = h * f(t_rk4[i], Ca_rk4[i])
      k2 = h * f(t_rk4[i] + h/2, Ca_rk4[i] + k1/2)
      k3 = h * f(t_rk4[i] + h/2, Ca_rk4[i] + k2/2)
      k4 = h * f(t_rk4[i] + h, Ca_rk4[i] + k3)

      Ca_rk4[i+1] = Ca_rk4[i] + (k1 + 2*k2 + 2*k3 + k4) / 6
      t_rk4[i+1] = t_rk4[i] + h

  plt.figure(figsize=(8,6))
  plt.plot(t_euler, Ca_euler, '-ob', label="Euler")
  plt.plot(t_mod_euler, Ca_mod_euler, '-sg', label="Modified
  Euler")
```

```
plt.plot(t_rk4, Ca_rk4, '-^r', label="RK-4")

plt.xlabel("Time (s)")
plt.ylabel("Concentration C_A (mol/L)")
plt.title("Comparison of Methods for 2nd-order Batch
Reactor")
plt.legend()
plt.grid(True)
plt.show()
```

Solution of Ordinary Differential Equations (ODE), 2nd Order-Boundary Value Problem (BVP)

Q. $\frac{d^2y}{dx^2} = 8x(9-x) + 2y$ BC-I: $y_{(X=0)} = 0$ BC-II $y_{(X=9)} = 0$

```
//Algorithm for Finite Difference Method
clc; clear; clf();

x0 = 0; xn = 9; y0 = 0; yn = 0
h = 0.1
x = x0:h:xn
n=(xn-x0)/h
A = zeros(n-1, n-1)
B = zeros(n-1, 1)

for i = 1:n-1
   p = 0; q = -2; r = 8*x(i+1)*(9-x(i+1))
   a = (1/h^2) - (p/(2*h))
   b = -(2/h^2) + q
   c = (1/h^2) + (p/(2*h))
   d = r

   if i > 1 then
      A(i, i-1) = a
   end
   A(i, i) = b
   if i < n-1 then
      A(i, i+1) = c
   end
   B(i) = d
end

Y_internal = A \ B
```

```
Y = [y0; Y_internal; yn]
disp(A,B)
disp('x values:'), disp(x)
disp('y values:'), disp(Y)
plot(x, Y, '-o')
xlabel('x')
ylabel('y')
title('Finite Difference Method Solution')
```

Python

```python
# Finite Difference Method
import numpy as np
import matplotlib.pyplot as plt

x0 = 0; xn = 9; y0 = 0; yn = 0
h = 1

x = np.arange(x0, xn + h, h)
n = int((xn - x0) / h)

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

for i in range(n-1):
    xi = x[i+1]

    p = 0
    q = -2
    r = 8 * xi * (9 - xi)

    a = (1/h**2) - (p/(2*h))
    b = -(2/h**2) + q
    c = (1/h**2) + (p/(2*h))
    d =r

    if i > 0:
        A[i, i-1] = a
    A[i, i] = b
    if i < n-2:
        A[i, i+1] = c
    B[i] = d
Y_internal = np.linalg.solve(A, B)

Y = np.vstack(([y0], Y_internal, [yn]))

print('x values:')
print(x)
```

```python
print('y values:')
print(Y.flatten())

plt.plot(x, Y, 'o-', label="Finite Difference Solution")
plt.xlabel('x')
plt.ylabel('y')
plt.title('Finite Difference Method Solution')
plt.grid(True)
plt.legend()
plt.show()
```

15. Solution of Partial Differential Equations (PDE)

PAGE NO.:

• Heat equation •

• Dera

$$\frac{\partial T}{\partial t} = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} \right)$$

For 1-dimension condition –

$$\frac{\partial T}{\partial t} = \alpha . \left(\frac{\partial^2 T}{\partial x^2} \right). \quad \text{— ①}$$

at $x = 0$ & $t = 0$; $T = 0$, BC I:

$x = L$ & $t = 0$; $T = 0$. BC II:

BC III: for $t = 0$; $T(x) = \sin\left(\frac{\pi x}{L}\right)$

From Forward difference,

$$\left.\frac{\partial T}{\partial t}\right|_i = \frac{T_i^{n+1} - T_i^n}{\Delta t}.$$

at time (n) $$\left.\frac{\partial^2 T}{\partial x^2}\right|_i = \frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{(\Delta x)^2} \quad \text{— central difference}$$

Put these values in eqⁿ ① –

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} = \alpha \left(\frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{(\Delta x)^2} \right)$$

Re-arranging,

$$\boxed{T_i^{n+1} = T_i^n + F_o \left(T_{i+1}^n - 2T_i^n + T_{i-1}^n \right)}$$

where, $\frac{\alpha \Delta t}{(\Delta x)^2} = F_o$ = Forier No.

Stability Condition: $F_o \leq 0.5$

Scilab

```
//Algorithm for solution of 1D Heat Equation by bender Schmidt Method

clc; clear; clf();

L = 1;  nx = 20;    dx = L/(nx-1)
alpha = 0.01; dt = 0.0005; nt = 200
Fo = alpha*dt/(dx^2)
if Fo > 0.5 then
   error("Warning: Scheme unstable. Reduce dt or increase nx.")
end

x = linspace(0, L, nx)
time = 0:dt:nt*dt

T = exp(-200*(x-0.5).^2)
T(1) = 0; T($) = 0

Tsol = zeros(nt+1, nx)
Tsol(1,:) = T

for t = 1:nt
   Told = T
   for i = 2:nx-1
      T(i) = Told(i) + Fo*(Told(i+1) - 2*Told(i) + Told(i-1))
   end
   Tsol(t+1,:) = T
end

clf()
surf(x, time, Tsol)
xlabel("Rod length x (m)")
ylabel("Time (s)")
zlabel("Temperature")
xtitle("Transient 1D Heat Conduction (Explicit FD)")

//Algorithm for 2D heat Equation by Bender Schmidt Method
clc; clear; clf();

Lx = 0.1; Ly = 0.1
nx = 21; ny = 21
dx = Lx/(nx-1); dy = Ly/(ny-1)
alpha = 0.01; dt=0.0001; tf = 1000
```

```
Fox = alpha*dt/(dx^2)
Foy = alpha*dt/(dy^2)

if Fox + Foy > 0.5 then
   error("Unstable scheme! Reduce dt or refine grid.")
end

x = linspace(0, Lx, nx)
y = linspace(0, Ly, ny)

[X,Y] = ndgrid(x,y);
T = exp(-5*((X-0.5).^2 + (Y-0.5).^2))

T(1,:) = 0; T($,:) = 0
T(:,1) = 0; T(:,$) = 0

zmax = max(T(:))

for t = 1:tf
   Told = T
   for i = 2:nx-1
      for j = 2:ny-1
         T(i,j) = Told(i,j) + ...
               Fox*(Told(i+1,j) - 2*Told(i,j) + Told(i-1,j)) + ...
               Foy*(Told(i,j+1) - 2*Told(i,j) + Told(i,j-1))
      end
   end

   if modulo(t,20)==0 then
      clf()
      surf(x,y,T')
      xlabel("x"); ylabel("y"); zlabel("Temperature")
      colorbar()
      a = gca()
      a.data_bounds = [0,0,0; Lx,Ly,zmax]
      xtitle(msprintf("2D Heat Conduction at time step %d", t))
      sleep(200)
   end
end
```

Python

```
#Algorithm for 1D Heat Equation by Bender Schmidt Method
import numpy as np
import matplotlib.pyplot as plt

L = 1.0; nx = 20; dx = L / (nx - 1); alpha = 0.01
dt = 0.0005; nt = 200

Fo = alpha * dt / (dx**2)

if Fo > 0.5:
    print("Warning: Scheme unstable. Reduce dt or increase
nx.")

x = np.linspace(0, L, nx)

T = np.sin(np.pi * x / L)
T[0] = 0.0; T[-1] = 0.0

plt.ion()
fig, ax = plt.subplots()

for t in range(1, nt + 1):
    Told = T.copy()
    for i in range(1, nx - 1):
        T[i] = Told[i] + Fo * (Told[i+1] - 2*Told[i] + Told[i-
1])

    if t % 20 == 0:
        ax.clear()
        ax.plot(x, T, '-o')
        ax.set_title(f"1D Heat Conduction (Explicit FD)\nTime
step {t}")
        ax.set_xlabel("x (rod length)")
        ax.set_ylabel("Temperature")
        ax.set_ylim(0, 1)
        plt.pause(0.1)
plt.ioff()
plt.show()
```