TRANSCENDENTAL EQUATIONS

1. BISECTION METHOD

AIM: TO WRITE A PROGRAM IN PYTHON TO DEMONSTARTE BISECTION METHOD

Program:

```
def f(x):
"""Define the function for which to find the root."""
return x**3 - x - 2  # Example function: f(x) = x^3 - x - 2
def bisection_method(a, b, tol, max_iter=100):
"""Implements the bisection method to find the root of function f within the interval [a, b].
Args:
a (float): The left bound of the interval.
b (float): The right bound of the interval.
tol (float): The desired tolerance (acceptable error) for the root.
max_iter (int): Maximum number of iterations to prevent infinite loops.
Returns:
float or None: The approximated root, or None if the method fails.
"""
if f(a) * f(b) >= 0:
print("Bisection method fails: f(a) and f(b) must have opposite signs to bracket a root.")
return None
c = a
iteration_counter = 0
print("\n*** BISECTION METHOD DEMONSTRATION ***")
print(f"{'Iteration':<15}{'a':<15}{'b':<15}{'c':<15}{'f(c)':<15}")
while (b - a) / 2.0 > tol and iteration_counter < max_iter:
c = (a + b) / 2.0  # Calculate the midpoint
f_c = f(c)
# Print current iteration details
print(f"{iteration_counter+1:<15}{a:<15.6f}{b:<15.6f}{c:<15.6f}{f_c:<15.6f}")
if f_c == 0.0:
break  # Found exact root
elif f(a) * f_c < 0:
b = c  # Root is in the left half
else:
```

```
a = c  # Root is in the right half
iteration_counter += 1
if iteration_counter == max_iter:
print("\nMax iterations reached. The result is an approximation.")
print(f"\nRequired Root is approximately: {c:.8f}")
return c
# --- Driver code ---
# Set initial interval [a, b] and tolerance
a_val = 1.0
b_val = 2.0
tolerance = 1e-6  # 0.000001
max_iterations_limit = 100
# Run the bisection method
root = bisection_method(a_val, b_val, tolerance, max_iterations_limit)
```

Output:

```
*** BISECTION METHOD DEMONSTRATION ***
Iteration      a              b              c              f(c)
1              1.000000       2.000000       1.500000       -0.125000
2              1.500000       2.000000       1.750000       1.609375
3              1.500000       1.750000       1.625000       0.666016
4              1.500000       1.625000       1.562500       0.252197
5              1.500000       1.562500       1.531250       0.059113
6              1.500000       1.531250       1.515625       -0.034054
7              1.515625       1.531250       1.523438       0.012250
8              1.515625       1.523438       1.519531       -0.010971
9              1.519531       1.523438       1.521484       0.000622
10             1.519531       1.521484       1.520508       -0.005179
11             1.520508       1.521484       1.520996       -0.002279
12             1.520996       1.521484       1.521240       -0.000829
13             1.521240       1.521484       1.521362       -0.000103
14             1.521362       1.521484       1.521423       0.000259
15             1.521362       1.521423       1.521393       0.000078
16             1.521362       1.521393       1.521378       -0.000013
17             1.521378       1.521393       1.521385       0.000033
18             1.521378       1.521385       1.521381       0.000010
19             1.521378       1.521381       1.521379       -0.000001

Required Root is approximately: 1.52137947
```

Conclusion: This Program has been executed successfully.

2. REGULAR FALSI

AIM: TO WRITE A PROGRAM IN PYTHON TO DEMONSTRATE NEWTON RAPHSON METHOD

PROGRAM:


```
import math
```

```
# Define the function f(x)
def f(x):
"""The function whose root we are trying to find."""
return math.cos(x) - x
# Define the derivative of the function f'(x)
def df(x):
"""The derivative of the function f(x)."""
return -math.sin(x) - 1
def newton_raphson(f_func, df_func, initial_guess, tolerance=1e-6, max_iterations=100):
"""
Implements the Newton-Raphson method to find a root of a function.
Args:
f_func: The function f(x).
df_func: The derivative of the function f'(x).
initial_guess: The starting value for the root search.
tolerance: The desired accuracy (stop when |f(x)| < tolerance).
max_iterations: Maximum number of iterations to perform.
Returns:
The approximate root if found, otherwise None.
"""
x_n = initial_guess
print(f"Starting Newton-Raphson method with initial guess x0 = {initial_guess}\n")
print(f"{'Iteration':<10} | {'x_n':<15} | {'f(x_n)':<15} | {'df(x_n)':<15}")
print("-" * 65)
for i in range(max_iterations):
f_x_n = f_func(x_n)
df_x_n = df_func(x_n)
# Check if the derivative is close to zero (division by zero risk)
if abs(df_x_n) < 1e-10:
print(f"\nError: Derivative is near zero at x = {x_n}. Cannot continue.")
return None
# Calculate the next approximation
x_n_plus_1 = x_n - f_x_n / df_x_n
print(f"{i:<10} | {x_n:<15.9f} | {f_x_n:<15.9f} | {df_x_n:<15.9f}")
# Check for convergence
if abs(f_x_n) < tolerance:
print(f"\nConvergence reached after {i+1} iterations.")
return x_n_plus_1
```

```
x_n = x_n_plus_1   4
print("\nError: Maximum iterations reached without convergence.")
return None
# --- Example Usage ---
if __name__ == "__main__":
# The equation cos(x) = x has a root around 0.7
initial_guess = 0.5
root = newton_raphson(f, df, initial_guess)
if root is not None:
print(f"\nThe approximate root is: {root:.10f}")
print(f"Value of f(root): {f(root):.10e}")
```

Output:

```
Starting Newton-Raphson method with initial guess x0 = 0.5

Iteration  | x_n            | f(x_n)          | df(x_n)
-------------------------------------------------------------
0          | 0.500000000    | 0.377582562     | -1.479425539
1          | 0.755222417    | -0.027103312    | -1.685450632
2          | 0.739141666    | -0.000094615    | -1.673653811
3          | 0.739085134    | -0.000000001    | -1.673612030

Convergence reached after 4 iterations.

The approximate root is: 0.7390851332
Value of f(root): 0.0000000000e+00
```

Conclusion: The Program has been executed successfully.

3. NEWTON'S RAPHSON METHOD

Aim: TO WRITE A PROGRAM IN PYTHON TO DEMONSTRATE NEWTON RAPHSON METHOD

Program:

```
import math
# Define the function f(x)
def f(x):
"""The function whose root we are trying to find."""
return math.cos(x) - x
# Define the derivative of the function f'(x)
def df(x):
"""The derivative of the function f(x)."""
```

```
return -math.sin(x) - 1
def newton_raphson(f_func, df_func, initial_guess, tolerance=1e-6, max_iterations=100):
"""
Implements the Newton-Raphson method to find a root of a function.
Args:
f_func: The function f(x).
df_func: The derivative of the function f'(x).
initial_guess: The starting value for the root search.
tolerance: The desired accuracy (stop when |f(x)| < tolerance).
max_iterations: Maximum number of iterations to perform.
Returns:
The approximate root if found, otherwise None.
"""
x_n = initial_guess
print(f"Starting Newton-Raphson method with initial guess x0 = {initial_guess}\n")
print(f"{'Iteration':<10} | {'x_n':<15} | {'f(x_n)':<15} | {'df(x_n)':<15}")
print("-" * 65)
for i in range(max_iterations):
f_x_n = f_func(x_n)
df_x_n = df_func(x_n)
# Check if the derivative is close to zero (division by zero risk)
if abs(df_x_n) < 1e-10:
print(f"\nError: Derivative is near zero at x = {x_n}. Cannot continue.")
return None
# Calculate the next approximation
x_n_plus_1 = x_n - f_x_n / df_x_n
print(f"{i:<10} | {x_n:<15.9f} | {f_x_n:<15.9f} | {df_x_n:<15.9f}")
# Check for convergence
if abs(f_x_n) < tolerance:
print(f"\nConvergence reached after {i+1} iterations.")
return x_n_plus_1
x_n = x_n_plus_1
print("\nError: Maximum iterations reached without convergence.")
return None
# --- Example Usage ---
if __name__ == "__main__":
# The equation cos(x) = x has a root around 0.7
initial_guess = 0.5
```

root = newton_raphson(f, df, initial_guess)

if root is not None:

print(f"\nThe approximate root is: {root:.10f}")

print(f"Value of f(root): {f(root):.10e}")

Output:

```
Starting Newton-Raphson method with initial guess x0 = 0.5

Iteration  | x_n             | f(x_n)          | df(x_n)
-------------------------------------------------------------
0          | 0.500000000     | 0.377582562     | -1.479425539
1          | 0.755222417     | -0.027103312    | -1.685450632
2          | 0.739141666     | -0.000094615    | -1.673653811
3          | 0.739085134     | -0.000000001    | -1.673612030

Convergence reached after 4 iterations.

The approximate root is: 0.7390851332
Value of f(root): 0.0000000000e+00
```

Conclusion: The Program has been executed successfully.

Interpolation

Aim: TO WRITE A PROGRAM IN PYTHON TO DEMONSTRATE NEWTON FORWARD INTERPOLATION.

Program:

import math

def calculate_forward_difference_table(y_values):

"""

Calculates the forward difference table for a given set of y-values.

Args:

y_values (list): A list of y-values (function values).

Returns:

list: A 2D list representing the forward difference table.

"""

n = len(y_values)

table = [[0.0 for _ in range(n)] for _ in range(n)]

Initialize the first column with y_values

for i in range(n):

table[i][0] = y_values[i]

Calculate subsequent columns (differences)

```
for j in range(1, n):7
for i in range(n - j):
table[i][j] = table[i + 1][j - 1] - table[i][j - 1]
return table
def print_forward_difference_table(x_values, table):
"""
Prints the forward difference table in a formatted way.
Args:
x_values (list): A list of x-values.
table (list): The forward difference table.
"""
n = len(x_values)
print("\nForward Difference Table:")
print(f"{'x':<8}{'y':<8}", end="")
for i in range(1, n):
print(f"Δ^{i}y{i-1}:<8}", end="")
print()
for i in range(n):
print(f"{x_values[i]:<8.2f}{table[i][0]:<8.4f}", end="")
for j in range(1, n - i):
print(f"{table[i][j]:<8.4f}", end="")
print()
def newton_forward_interpolation(x_values, y_values, x_target):
"""
Performs Newton's Forward Interpolation to estimate a value at x_target.
Args:
x_values (list): A list of equally spaced x-coordinates.
y_values (list): Corresponding y-values.
x_target (float): The x-value at which to interpolate.
Returns:
float: The interpolated y-value at x_target.
"""
n = len(x_values)
# Validate input for equally spaced x-values
if n < 2:
raise ValueError("At least two data points are required for interpolation.")
h = x_values[1] - x_values[0]
for i in range(1, n - 1):
```

```
if not math.isclose(x_values[i+1] - x_values[i], h):
raise ValueError("x_values must be equally spaced for Newton's Forward Interpolation.")
# Calculate the forward difference table
table = calculate_forward_difference_table(y_values)
print_forward_difference_table(x_values, table)
# Calculate 'u'
u = (x_target - x_values[0]) / h
# Initialize the result with the first y-value
result = table[0][0]
# Calculate terms iteratively
p_term = 1.0
for i in range(1, n):
p_term *= (u - (i - 1)) / i  # Calculate u(u-1)...(u-i+1) / i!
result += p_term * table[0][i] # Add the term with the leading difference
return result
# Example Usage:
if __name__ == "__main__":
x_data = [0, 10, 20, 30, 40]
y_data = [0, 0.1763, 0.3492, 0.5171, 0.6804]
x_interpolate = 25
try:
interpolated_value = newton_forward_interpolation(x_data, y_data, x_interpolate)
print(f"\nInterpolated value at x = {x_interpolate}: {interpolated_value:.4f}")
except ValueError as e:
print(f"Error: {e}")
# Another example
x_data_2 = [1, 2, 3, 4, 5]
y_data_2 = [40, 60, 65, 50, 18]
x_interpolate_2 = 1.7
try:
interpolated_value_2 = newton_forward_interpolation(x_data_2, y_data_2, x_interpolate_2)
print(f"\nInterpolated value at x = {x_interpolate_2}: {interpolated_value_2:.4f}")
except ValueError as e:
print(f"Error: {e}")
```

Output:

```
Forward Difference Table:
x       y       Δ^1y0:<8Δ^2y1:<8Δ^3y2:<8Δ^4y3:<8
0.00    0.0000  0.1763  -0.0034 -0.0016 0.0020
10.00   0.1763  0.1729  -0.0050 0.0004
20.00   0.3492  0.1679  -0.0046
30.00   0.5171  0.1633
40.00   0.6804

Interpolated value at x = 25: 0.4338
```

Conclusion: This program has been executed successfully.

Aim:TO WRITE A PROGRAM IN PYTHON TO DEMONSTRATE NEWTON BACKWARD INTERPOLATION.

Program:

```
# Python3 Program to interpolate using
# newton backward interpolation
# Calculation of u mentioned in formula
def u_cal(u, n):
temp = u
for i in range(n):
temp = temp * (u + i)
return temp
# Calculating factorial of given n
def fact(n):
f = 1
for i in range(2, n + 1):
f *= i
return f
# Driver code
# number of values given
n = 5
x = [1891, 1901, 1911, 1921, 1931]
# y is used for difference
# table and y[0] used for input
y = [[0.0 for _ in range(n)] for __ in range(n)]
y[0][0] = 46
y[1][0] = 66
y[2][0] = 81
y[3][0] = 93
```

```
y[4][0] = 101          10
# Calculating the backward difference table
for i in range(1, n):
for j in range(n - 1, i - 1, -1):
y[j][i] = y[j][i - 1] - y[j - 1][i - 1]
# Displaying the backward difference table
for i in range(n):
for j in range(i + 1):
print(y[i][j], end="\t")
print()
# Value to interpolate at
value = 1925
# Initializing u and sum
sum = y[n - 1][0]
u = (value - x[n - 1]) / (x[1] - x[0])
for i in range(1, n):
sum = sum + (u_cal(u, i) * y[n - 1][i]) / fact(i)
print("\n Value at", value,  "is", sum)
# This code is contributed by phasing17
```

Output:

```
46
66      20
81      15      -5
93      12      -3      2
101     8       -4      -1      -3

 Value at 1925 is 103.49792
```

Conclusion: The program has been executed successfully.

SOLUTION OF SIMULTANEOUS ALGEBRAIC EQUATIONS

AIM: TO WRITE A PROGRAM IN PYTHON TO DEMONSTRATE GUASSIAN ELIMINATION METHOD

Program:

```
import numpy as np
def gaussian_elimination(A, b):
"""
Solves a system of linear equations Ax = b using Gaussian elimination.
Args:
A (np.array): The coefficient matrix.
b (np.array): The constant vector.
Returns:
np.array: The solution vector x, or None if no unique solution exists.
"""
n = len(b)
# Create augmented matrix [A|b]
augmented_matrix = np.concatenate((A, b.reshape(n, 1)), axis=1)
# Forward elimination
for i in range(n):
# Find pivot (largest absolute value in the current column)
pivot_row = i
for k in range(i + 1, n):
if abs(augmented_matrix[k, i]) > abs(augmented_matrix[pivot_row, i]):
pivot_row = k
augmented_matrix[[i, pivot_row]] = augmented_matrix[[pivot_row, i]]
# Check for singular matrix (no unique solution)
if augmented_matrix[i, i] == 0:
print("Error: Divide by zero detected or singular matrix. No unique solution.")
return None
# Eliminate elements below the pivot
for j in range(i + 1, n):
factor = augmented_matrix[j, i] / augmented_matrix[i, i]
augmented_matrix[j, i:] -= factor * augmented_matrix[i, i:]
# Backward substitution
x = np.zeros(n)
for i in range(n - 1, -1, -1):
x[i] = (augmented_matrix[i, n] - np.dot(augmented_matrix[i, i+1:n], x[i+1:n])) / augmented_matrix[i, i]
```

```
return x                12
# Example usage:
if __name__ == "__main__":
# Define the coefficient matrix A
A = np.array([
[2, 1, -1],
[-3, -1, 2],
[-2, 1, 2]
], dtype=float)
# Define the constant vector b
b = np.array([8, -11, -3], dtype=float)
print("Coefficient Matrix A:")
print(A)
print("\nConstant Vector b:")
print(b)
solution = gaussian_elimination(A.copy(), b.copy()) # Use copies to preserve original A and b
if solution is not None:
print("\nSolution x:")
print(solution)
# Example with a singular matrix
A_singular = np.array([
[1, 2],
[2, 4]
], dtype=float)
b_singular = np.array([3, 6], dtype=float)
print("\n\nSingular Matrix A_singular:")
print(A_singular)
print("\nConstant Vector b_singular:")
print(b_singular)
solution_singular = gaussian_elimination(A_singular.copy(), b_singular.copy())
```

Output:

```
Coefficient Matrix A:
[[ 2.  1. -1.]
 [-3. -1.  2.]
 [-2.  1.  2.]]

Constant Vector b:
[  8. -11.  -3.]

Solution x:
[ 2.  3. -1.]


Singular Matrix A_singular:
[[1. 2.]
 [2. 4.]]

Constant Vector b_singular:
[3. 6.]
Error: Divide by zero detected or singular matrix. No unique solution.
```

Conclusion: The program has been executed successfully.

NUMERIACAL SOLUTIONS OF FIRST AND SECOND ORDER DIFFERENTIAL EQUATIONS

1. TAYLOR SERIES

AIM: TO WRITE A PROGRAM IN PYTHONTO DEMONSTRATE TAYLOR SERIES

PROGRAM:

import math 14

```
def taylor_series_exp(x, num_terms):
"""
Calculates the Taylor series approximation for e^x around a=0.
Args:
x (float): The value at which to evaluate e^x.
num_terms (int): The number of terms to use in the Taylor series.
Returns:
float: The approximated value of e^x.
"""
approximation = 0
for n in range(num_terms):
term = (x**n) / math.factorial(n)
approximation += term
return approximation
# Example usage:
x_value = 1.0  # Evaluate e^x at x=1
num_terms = 10 # Use 10 terms in the series
taylor_approx = taylor_series_exp(x_value, num_terms)
actual_value = math.exp(x_value)
print(f"Approximation of e^{x_value} using {num_terms} terms: {taylor_approx}")
print(f"Actual value of e^{x_value}: {actual_value}")
print(f"Difference: {abs(actual_value - taylor_approx)}")
x_value_2 = 0.5
num_terms_2 = 5
taylor_approx_2 = taylor_series_exp(x_value_2, num_terms_2)
actual_value_2 = math.exp(x_value_2)
print(f"\nApproximation of e^{x_value_2} using {num_terms_2} terms: {taylor_approx_2}")
print(f"Actual value of e^{x_value_2}: {actual_value_2}")

print(f"Difference: {abs(actual_value_2 - taylor_approx_2)}")
```

Output:

```
Approximation of e^1.0 using 10 terms: 2.7182815255731922
Actual value of e^1.0: 2.718281828459045
Difference: 3.0288585284310443e-07

Approximation of e^0.5 using 5 terms: 1.6484375
Actual value of e^0.5: 1.6487212707001282
Difference: 0.00028377070012819416
```

Conclusion: The program has been executed successfully.

2. EULER'S METHOD

AIM: TO WRITE A PROGRAM IN PYTHON TO DEMONSTARTE EULER'S METHOD.

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt
def f(x,y):
```

```
"""
Define the ordinary differential equation dy/dx = f(x, y).
For example, let's solve dy/dx = x + y.
"""
return x + y
def euler_method(f, x0, y0, h, x_final):
"""
Implements Euler's method to solve an ODE.
Args:
f: The function representing dy/dx = f(x, y).
x0: Initial value of x.
y0: Initial value of y.
h: Step size.
x_final: The value of x at which to stop the approximation.
Returns:
A tuple of lists (x_values, y_values) containing the approximated points.
"""
x_values = [x0]
y_values = [y0]
x = x0
y = y0
while x < x_final:
y = y + h * f(x, y)
x = x + h
x_values.append(x)
y_values.append(y)
return x_values, y_values
# Driver Code
if __name__ == "__main__":
x0 = 0     # Initial x value
y0 = 1     # Initial y value
h = 0.1    # Step size
x_final = 1 # Value of x at which to approximate y
x_approx, y_approx = euler_method(f, x0, y0, h, x_final)
print("Approximated solution using Euler's method:")
for i in range(len(x_approx)):
print(f"x = {x_approx[i]:.2f}, y = {y_approx[i]:.4f}")
# Plotting the results (optional)
```

```
plt.figure(figsize=(10, 6))
plt.plot(x_approx, y_approx, 'bo--', label='Euler Approximation')
plt.xlabel('x')
plt.ylabel('y')
plt.title("Euler's Method for dy/dx = x + y")
plt.grid(True)
plt.legend()
plt.show()
```

Output:

```
Approximated solution using Euler's method:
x = 0.00, y = 1.0000
x = 0.10, y = 1.1000
x = 0.20, y = 1.2200
x = 0.30, y = 1.3620
x = 0.40, y = 1.5282
x = 0.50, y = 1.7210
x = 0.60, y = 1.9431
x = 0.70, y = 2.1974
x = 0.80, y = 2.4872
x = 0.90, y = 2.8159
x = 1.00, y = 3.1875
x = 1.10, y = 3.6062
```

Conclusion: The program has been executed successfully.

3.MODIFIED EULER'S METHOD

AIM: TO WRITE A PROGRAM IN PYTHON TO DEMONSTRATE MODIFIED EULER'S METHOD

PROGRAM:

```
def modified_euler(f, x0, y0, h, num_steps):
""" Demonstrates the Modified Euler's Method (Heun's Method) for solving ODEs.
Args:
f (function): The function representing dy/dx = f(x, y).
x0 (float): The initial value of x.
y0 (float): The initial value of y.
h (float): The step size.
num_steps (int): The number of steps to take.
Returns:
tuple: A tuple containing two lists:
```

```
- x_values (list): The x-values at each step.
- y_values (list): The corresponding y-values at each step.
"""
x_values = [x0]
y_values = [y0]
for i in range(num_steps):
# Predictor step (Euler's method)
y_predictor = y_values[-1] + h * f(x_values[-1], y_values[-1])
# Corrector step (Modified Euler)
y_corrector = y_values[-1] + (h / 2) * (f(x_values[-1], y_values[-1]) + f(x_values[-1] + h, y_predictor))
x_values.append(x_values[-1] + h)
y_values.append(y_corrector)
return x_values, y_values
# Example Usage:
if __name__ == "__main__":
# Define the ODE: dy/dx = x + y
def my_ode(x, y):
return x + y

# Initial conditions
x_initial = 0.0
y_initial = 1.0
# Step size and number of steps
step_size = 0.1
number_of_steps = 10
# Solve using Modified Euler's Method
x_results, y_results = modified_euler(my_ode, x_initial, y_initial, step_size, number_of_steps)
# Print the results
print("Modified Euler's Method Results:")
print("x\t\ty")
for x, y in zip(x_results, y_results):
print(f"{x:.2f}\t\t{y:.4f}")
# Another example: dy/dx = y
def another_ode(x, y):
return y
print("\nAnother Example (dy/dx = y):")
x_results_2, y_results_2 = modified_euler(another_ode, 0.0, 1.0, 0.1, 5)
print("x\t\ty")
```

```
for x, y in zip(x_results_2, y_results_2):
print(f"{x:.2f}\t\t{y:.4f}")
```

Output:

```
Modified Euler's Method Results:
x               y
0.00            1.0000
0.10            1.1100
0.20            1.2421
0.30            1.3985
0.40            1.5818
0.50            1.7949
0.60            2.0409
0.70            2.3231
0.80            2.6456
0.90            3.0124
1.00            3.4282
```

Conclusion: the program has been executed successfully.

4. RUNGE-KUTTA 4th ORDER METHOD

AIM: TO WRITE A PROGRAM IN PYTHON TO DEMONSTRATE RUNGE KUTTA 4th ORDER METHOD

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt
# --- 1. Define the ODE Function (dy/dx = f(x, y)) ---
def f(x, y):
"""
The right-hand side of the ODE: dy/dx = (x - y) / 2
"""
return (x - y) / 2
# --- 2. Runge-Kutta 4th Order Method Implementation ---
def runge_kutta_4th_order_iterative(f, x0, y0, x_target, n_steps):
"""
Solves a first-order ODE y' = f(x, y) using the RK4 method,
printing results for each iteration.
:param f: The function f(x, y)
:param x0: Initial value of x
:param y0: Initial value of y (y(x0))
```

```
:param x_target: The x-value where the solution is desired
:param n_steps: Number of steps to take
:return: A tuple of (x_values, y_values) lists
"""
# Calculate the step size (h)
h = (x_target - x0) / n_steps
# Initialize lists to store results
x_values = [x0]
y_values = [y0]
# Set the current x and y values
x_current = x0
y_current = y0
print("\n" + "="*70)
print(f"| RK4 Method Iteration Summary | h = {h:.4f} | Target x = {x_target:.1f} |")
print("="*70)
print(f"| Step |  x_i  |  y_i  |  k1  |  k2  |  k3  |  k4  | y_i+1 |")
print("-"*70)
print(f"|  0   | {x_current:.4f} | {y_current:.4f} | {'-':^6} | {'-':^6} | {'-':^6} | {'-':^6} | {'-':^7} |")
# Iteratively apply the RK4 formula
for i in range(n_steps):
# Calculate the four increments (k1, k2, k3, k4)
k1 = h * f(x_current, y_current)
k2 = h * f(x_current + 0.5 * h, y_current + 0.5 * k1)
k3 = h * f(x_current + 0.5 * h, y_current + 0.5 * k2)
k4 = h * f(x_current + h, y_current + k3)
# Weighted average to get the new y-value
y_next = y_current + (1.0 / 6.0) * (k1 + 2 * k2 + 2 * k3 + k4)
# Print the iteration details
print(f"| {i+1:4} | {x_current:.4f} | {y_current:.4f} | {k1:.4f} | {k2:.4f} | {k3:.4f} | {k4:.4f} | {y_next:.4f} |")
# Update x and y for the next step
x_current = x_current + h
y_current = y_next
# Store the results
x_values.append(x_current)
y_values.append(y_current)
print("="*70 + "\n")
return x_values, y_values# --- 3. Example Usage ---
def main():
```

```
# Initial conditions: y(0) = 1
x_start = 0.0
y_initial = 1.0
# To keep the output manageable, we will use fewer steps for the printout.
x_end = 1.0     # Find solution up to x = 1.0
num_steps = 5    # Number of steps for detailed output
# Run the RK4 solver with iterative printout
x_rk4, y_rk4 = runge_kutta_4th_order_iterative(f, x_start, y_initial, x_end, num_steps)
# Print the final result
print(f"The final approximate solution at x = {x_end} after {num_steps} steps is y ≈ {y_rk4[-1]:.6f}")
# Comparison to the exact solution
# Analytical Solution: y(x) = 3e^(-x/2) + x - 2
y_exact_at_end = 3 * np.exp(-x_end / 2) + x_end - 2
print(f"The exact solution at x = {x_end} is y = {y_exact_at_end:.6f}")
if __name__ == "__main__":
main()
```

Output:

```
=========================================================================
| RK4 Method Iteration Summary | h = 0.2000 | Target x = 1.0 |
=========================================================================
| Step |   x_i   |   y_i   |   k1   |   k2   |   k3   |   k4   |  y_i+1  |
-------------------------------------------------------------------------
|  0   | 0.0000 | 1.0000 |   -    |   -    |   -    |   -    |    -    |
|    1 | 0.0000 | 1.0000 | -0.1000 | -0.0850 | -0.0858 | -0.0714 | 0.9145 |
|    2 | 0.2000 | 0.9145 | -0.0715 | -0.0579 | -0.0586 | -0.0456 | 0.8562 |
|    3 | 0.4000 | 0.8562 | -0.0456 | -0.0333 | -0.0340 | -0.0222 | 0.8225 |
|    4 | 0.6000 | 0.8225 | -0.0222 | -0.0111 | -0.0117 | -0.0011 | 0.8110 |
|    5 | 0.8000 | 0.8110 | -0.0011 | 0.0090 | 0.0085 | 0.0181 | 0.8196 |
=========================================================================

The final approximate solution at x = 1.0 after 5 steps is y ≈ 0.819593
The exact solution at x = 1.0 is y = 0.819592
```

Conclusion: The program has been executed successfully.

1. TRAPEZOIDAL RULE

AIM: TO WRITE A PROGRAM IN PYTHON TO DEMONSTRATE TRAPEZOIDAL RULE.

PROGRAM:

```
# Import the math library for the function we want to integrate
import math
# 1. Define the function f(x)
def f(x):
"""The function to be integrated: f(x) = 1 / (1 + x^2)"""
return 1 / (1 + x**2)
# 2. Define the Trapezoidal Rule function
def trapezoidal_rule(f, a, b, n):
"""
Approximates the definite integral of f from a to b
using the Trapezoidal Rule with n subintervals.
Parameters:
f (function): The function to integrate.
a (float): The lower limit of integration.
b (float): The upper limit of integration.
n (int): The number of subintervals (trapezoids).
Returns:
float: The approximate value of the integral.
"""
# Calculate the width of each subinterval (h or delta_x)
h = (b - a) / n
# Initialize the sum for the Trapezoidal Rule
# Start with f(a) + f(b)
integral_sum = f(a) + f(b)
# Add 2 * f(x_i) for all intermediate points (i = 1 to n-1)
for i in range(1, n):
x_i = a + i * h
integral_sum += 2 * f(x_i)
# Multiply the sum by (h/2) to get the final approximation
integral_approximation = (h / 2) * integral_sum
return integral_approximation
# 3. Set the parameters for the integral
lower_limit = 0.0  # a
upper_limit = 1.0  # b
```

```
num_intervals = 6 # n
# 4. Calculate the approximation
approx_area = trapezoidal_rule(f, lower_limit, upper_limit, num_intervals)
# 5. Print the results
print(f"--- Trapezoidal Rule Demonstration ---")
print(f"Function: f(x) = 1 / (1 + x^2)")
print(f"Integration
 Limits: [{lower_limit}, {upper_limit}]")
print(f"Number of Subintervals (n): {num_intervals}")
print(f"Step Size (h): {(upper_limit - lower_limit) / num_intervals}")
print(f"\nApproximate Integral Value: {approx_area:.8f}")
# The exact integral of 1/(1+x^2) from 0 to 1 is arctan(1) - arctan(0) = pi/4
exact_value = math.pi / 4
print(f"Exact Value (for comparison): {exact_value:.8f}")
print(f"Error: {abs(exact_value - approx_area):.8f}")
```

Output:

```
--- Trapezoidal Rule Demonstration ---
Function: f(x) = 1 / (1 + x^2)
Integration Limits: [0.0, 1.0]
Number of Subintervals (n): 6
Step Size (h): 0.16666666666666666

Approximate Integral Value: 0.78424077
Exact Value (for comparison): 0.78539816
Error: 0.00115740
```

Conclusion: The program has been executed successfully.

2. SIMPSON'S 1/3 RULE

AIM: TO WRITE A PROGRAM IN PYTHON TO DEMONSTRATE SIMPSON'S 1/3 RULE.

PROGRAM:

```
import numpy as np
def simpsons_one_third_rule(f, a, b, n):
"""
Approximates the definite integral of a function f(x) from a to b
using Simpson's 1/3 Rule with n subintervals.
Args:
f (function): The function to integrate.
a (float): The lower limit of integration.
b (float): The upper limit of integration.
n (int): The number of subintervals (must be even).
Returns:
float: The approximate value of the definite integral.
Raises:
ValueError: If the number of subintervals (n) is not even.
"""
# Check if n is even, which is required for Simpson's 1/3 Rule
if n % 2 != 0:
raise ValueError("The number of subintervals (n) must be even for Simpson's 1/3 Rule.")
# Calculate the width of each subinterval (h)
h = (b - a) / n
# Generate the points (x-values) from a to b
x = np.linspace(a, b, n + 1)
# Evaluate the function at all x-points
y = f(x)
# Simpson's 1/3 Rule formula is:
# (h/3) * [y0 + 4*(y1 + y3 + ... + y(n-1)) + 2*(y2 + y4 + ... + y(n-2)) + yn]
# 1. Sum of the first and last terms (y0 + yn)
integral_sum = y[0] + y[n]
# 2. Sum of the terms with odd indices (multiplied by 4)
# y[1], y[3], y[5], ..., y[n-1]
integral_sum += 4 * np.sum(y[1:n:2])
# 3. Sum of the terms with even indices (multiplied by 2),
# excluding the first and last terms (y0 and yn)
# y[2], y[4], y[6], ..., y[n-2]
integral_sum += 2 * np.sum(y[2:n-1:2])
```

```
# Final result        25
integral = (h / 3) * integral_sum
return integral
# --- Demonstration ---
##      Example Function and Calculation
# Define the function f(x) = x^2 + 1
def my_function(x):
return x**2 + 1
# Set the parameters for integration
a = 0.0  # Lower limit
b = 2.0  # Upper limit
n = 4    # Number of subintervals (must be even, e.g., 4 or 8)
# Calculate the integral using the implemented rule
try:
approx_integral = simpsons_one_third_rule(my_function, a, b, n)
# Calculate the exact integral for comparison (analytically, integral of x^2 + 1 is x^3/3 + x)
# Definite integral from 0 to 2 is: (2^3/3 + 2) - (0^3/3 + 0) = 8/3 + 2 = 14/3 ≈ 4.666667
exact_integral = (2**3)/3 + 2
print(f"--- Simpson's 1/3 Rule Demonstration ---")
print(f"Function: f(x) = x^2 + 1")
print(f"Integration interval: [{a}, {b}]")
print(f"Number of subintervals (n): {n}")
print(f"\nApproximate Integral (Simpson's 1/3 Rule): {approx_integral:.6f}")
print(f"Exact Integral Value (for comparison): {exact_integral:.6f}")
except ValueError as e:
print(f"Error: {e}")
```

Output:

```
--- Simpson's 1/3 Rule Demonstration ---
Function: f(x) = x^2 + 1
Integration interval: [0.0, 2.0]
Number of subintervals (n): 4

Approximate Integral (Simpson's 1/3 Rule): 4.666667
Exact Integral Value (for comparison): 4.666667
```

.2. SIMPSON'S 3/8 RULE

AIM: TO WRITE A PROGRAM IN PYTHON TO DEMONSTRATE SIMPSON'S 3/8 RULE.

PROGRAM:

```
import numpy as np
# 1. Define the function to be integrated
def func(x):
"""
The function f(x) to be integrated.
Example: f(x) = 1 / (1 + x^2)
""" return 1.0 / (1.0 + x**2)
def simpsons_three_eighth_rule(a, b, n):
"""
Approximates the definite integral of func(x)
from a to b using Simpson's 3/8 Rule with n subintervals.
a: lower limit of integration
b: upper limit of integration
n: number of subintervals (must be a multiple of 3)
"""
# 2. Check if n is a multiple of 3
if n % 3 != 0:
print("Error: For Simpson's 3/8 rule, the number of subintervals (n) must be a multiple of 3.")
return None
# 3. Calculate the width of each subinterval (h)
h = (b - a) / n
# 4. Initialise the approximation sum
integral = func(a) + func(b) # Add the end points f(x0) + f(xn)
# 5. Apply the Simpson's 3/8 formula for intermediate points
# The coefficient pattern is: 3, 3, 2, 3, 3, 2, ...
for i in range(1, n):
x = a + i * h # Current x-value
if i % 3 == 0:
# Multiplier for x3, x6, x9, ... is 2
integral += 2 * func(x)
else:
# Multiplier for x1, x2, x4, x5, x7, x8, ... is 3
integral += 3 * func(x)
# 6. Final calculation: multiply the sum by 3h/8
integral = integral * (3 * h / 8)
```

```
return integral
# --- Execution Block ---
# Define the limits and number of subintervals
lower_limit = 0.0 # a
upper_limit = 1.0 # b
num_subintervals = 6 # n (must be a multiple of 3, e.g., 3, 6, 9, 12)
# Calculate the integral
result = simpsons_three_eighth_rule(lower_limit, upper_limit, num_subintervals)
# Print the result
if result is not None:
print(f"\n--- Numerical Integration using Simpson's 3/8 Rule ---")
print(f"Function: f(x) = 1 / (1 + x^2)")
print(f"Interval: [{lower_limit}, {upper_limit}]")
print(f"Subintervals (n): {num_subintervals}")
print(f"Approximate Integral Value: {result}")
# For comparison (The exact integral of 1/(1+x^2) is arctan(x))
# The exact value is arctan(1) - arctan(0) = pi/4
exact_value = np.pi / 4
error = abs(exact_value - result)
print(f"\n(For comparison, the exact value is: {exact_value})")
print(f"Absolute Error: {error}")
```

Output:

```
--- Numerical Integration using Simpson's 3/8 Rule ---
Function: f(x) = 1 / (1 + x^2)
Interval: [0.0, 1.0]
Subintervals (n): 6
Approximate Integral Value: 0.7853958624450428

(For comparison, the exact value is: 0.7853981633974483)
Absolute Error: 2.3009524054984354e-06
```

Conclusion :The program has been executed successfully.

TRANSPORTATION PROBLEM

AIM: TO WRITE A PROGRAM IN PYTHON TO DEMONSTRATE TRANSPORTATION PROBLEM USING NORTHWEST METHOD.

PROGRAM:

```
import numpy as np
def northwest_corner_method(supply, demand, cost_matrix):
"""
Finds an initial basic feasible solution for a transportation problem
using the Northwest Corner Method.
Args:
supply (list): A list of supply amounts for each source (row).
demand (list): A list of demand amounts for each destination (column).
cost_matrix (list of lists): The matrix of transportation costs.
Returns:
tuple: A tuple containing the initial allocation matrix and the
total transportation cost.
"""
# Convert lists to numpy arrays for easier manipulation
supply = np.array(supply, dtype=float)
demand = np.array(demand, dtype=float)
cost = np.array(cost_matrix, dtype=float)
# Initialize the allocation matrix with zeros
num_sources = len(supply)
num_destinations = len(demand)
allocation = np.zeros((num_sources, num_destinations))
# Initialize row (i) and column (j) indices
i, j = 0, 0
total_cost = 0
print("---  Allocation Steps ---")
# The NWC method continues as long as there's unfulfilled supply or demand
while i < num_sources and j < num_destinations:
# 1. Determine the allocation amount at the current (i, j) corner
# The amount is the minimum of the remaining supply or remaining demand
allocate_amount = min(supply[i], demand[j])
# 2. Record the allocation
allocation[i, j] = allocate_amount
# 3. Calculate the cost for this allocation
```

```
cost_at_cell = cost[i, j] * allocate_amount
total_cost += cost_at_cell
# Print the step details
print(f"Allocating {allocate_amount:.0f} units to cell ({i+1}, {j+1}) (Cost: {cost[i, j]:.0f}).")
# 4. Update the remaining supply and demand
supply[i] -= allocate_amount
demand[j] -= allocate_amount
# 5. Move to the next cell
if supply[i] == 0:
# If supply is exhausted, move to the next row (source)
i += 1
elif demand[j] == 0:
# If demand is fulfilled, move to the next column (destination)
j += 1
print("------------------------")
return allocation, total_cost
# ---      Example Data ---
# Note: For the NWC method to work simply, the problem must be balanced
# (Total Supply = Total Demand).
# Example: 3 Sources (S1, S2, S3) and 4 Destinations (D1, D2, D3, D4)
# Total Supply: 30 + 50 + 20 = 100
# Total Demand: 20 + 40 + 30 + 10 = 100
supply_quantities = [30, 50, 20]
demand_quantities = [20, 40, 30, 10]
cost_matrix = [
[10, 2, 20, 11],  # Costs from Source 1
[12, 7, 9, 20],   # Costs from Source 2
[4, 14, 16, 18]   # Costs from Source 3]
# ---      Execution ---
if sum(supply_quantities) != sum(demand_quantities):
print("     Error: The transportation problem is unbalanced. Total Supply must equal Total Demand.")
else:
initial_allocation, total_transport_cost = northwest_corner_method(
supply_quantities,
demand_quantities,
cost_matrix
)
## Initial Basic Feasible Solution
```

```
print("\n##  Initial Basic Feasible Solution (Northwest Corner Method)")
print("\n### Allocation Matrix")
# Use pandas or a formatted print for a nice matrix display (using basic print here)
# The [::-1] is to convert from float to int for cleaner display
print(np.array(initial_allocation, dtype=int))
print("\n### Total Transportation Cost")
print(f"The total initial cost is: **${total_transport_cost:.2f}**")
```

Output:

```
---  Allocation Steps ---
Allocating 20 units to cell (1, 1) (Cost: 10).
Allocating 10 units to cell (1, 2) (Cost: 2).
Allocating 30 units to cell (2, 2) (Cost: 7).
Allocating 20 units to cell (2, 3) (Cost: 9).
Allocating 10 units to cell (3, 3) (Cost: 16).
Allocating 10 units to cell (3, 4) (Cost: 18).
-------------------------

##  Initial Basic Feasible Solution (Northwest Corner Method)

### Allocation Matrix
[[20 10  0  0]
 [ 0 30 20  0]
 [ 0  0 10 10]]

### Total Transportation Cost
The total initial cost is: **$950.00**
```

Conclusion: The Program has been executed successfully.