PRACTICAL NO : 1

TRANSCENDENTAL EQUATIONS

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

PROGRAM:

```
import math
# Evaluate the user-defined function safely
def f(x, func_str):
    try:
        return eval(func_str, {"x": x, "math": math, "_builtins_": None})
    except Exception as e:
        print("Error evaluating function:", e)
        return None
# Bisection Method
def bisection(func_str, a, b, tol):
    if f(a, func_str) * f(b, func_str) >= 0:
        print("Invalid interval. f(a) and f(b) must have opposite signs.")
        return
    print("Iter\t a\t b\t Xr\t f(Xr)")
    iter = 1
    while (b - a) / 2 > tol:
        Xr = (a + b) / 2
        fx = f(Xr, func_str)
```

```
        print(f"{iter}\t{a:.3f}\t{b:.3f}\t{Xr:.3f}\t{fx:.3f}"
        if abs(fx) < tol:
break
        if f(a, func_str) * fx < 0:
            b = Xr
       else:
            a = Xr
  iter += 1
    print(f"\nApproximate root = {Xr:.3f} (correct to 3 decimal places)")
# === Main Program ===
print("=== Bisection Method ===")
func_str = input("Enter the function f(x): ")     # Example: x**3 - 4*x + 1
a = float(input("Enter the starting value a: "))  # Example: 0
b = float(input("Enter the ending value b: "))    # Example: 1
tol = 0.00003   # 3 decimal place accuracy
bisection(func_str, a, b, tol)
```

OUTPUT:

```
=== Bisection Method ===
Enter the function f(x): x*x*x -4*x +1
Enter the starting value a: 1
Enter the ending value b: 2

Iter      a       b       Xr      f(x)
  1     1.000   2.000   1.500   -1.625
  2     1.500   2.000   1.750   -0.641
  3     1.750   2.000   1.875   0.092
  4     1.750   1.875   1.812   -0.296
  5     1.812   1.875   1.844   -0.107
  6     1.844   1.875   1.859   -0.009
  7     1.859   1.875   1.867   0.041
  8     1.859   1.867   1.863   0.016
  9     1.859   1.863   1.861   0.003
 10     1.859   1.861   1.860   -0.003
 11     1.860   1.861   1.861   0.000
 12     1.860   1.861   1.861   -0.001
 13     1.861   1.861   1.861   -0.001
 14     1.861   1.861   1.861   -0.000
 15     1.861   1.861   1.861   0.000

Approximate root = 1.861 (correct to 3 decimal places)
```

CONCLUSION: The above program has been executed successfully.

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

PROGRAM:

```
import math
# Evaluate the user-defined function safely
def safe_eval(expr, x):
  try:
    return eval(expr.strip(), {"x": x, "math": math, "m": math, "_builtins_":
None})
  except (NameError, TypeError, ZeroDivisionError, SyntaxError) as e:
    print(f"Error evaluating function: {e}")
    return None
def Regula_Falsi(Func_str, a, b, tol):
  Fa = safe_eval(Func_str, a)
  Fb = safe_eval(Func_str, b)
  if Fa is None or Fb is None:
    return None
  if Fa * Fb >= 0:
    print("Invalid interval. F(a) and F(b) must have opposite signs.")
    return None
  print("\nIter.\t a\t\t b\t\t F(a)\t\t F(b)\t\t Xr\t\t F(Xr)")
    Xr_old = a # Initial guess to calculate error if needed
  for i in range(1, 101):
```

```
        # Regula Falsi Formula
        Xr = (a * Fb - b * Fa) / (Fb - Fa)
        FXr = safe_eval(Func_str, Xr)
        print(f"{i:<6}\t {a:.4f}\t {b:.4f}\t {Fa:.4f}\t {Fb:.4f}\t {Xr:.4f}\t {FXr:.4f}")
        if abs(FXr) < tol:
            return Xr
        if Fa * FXr < 0:
            b = Xr
            Fb = FXr
        else:
            a = Xr
            Fa = FXr
    print(f"\nRoot not found within 100 iterations (Current error: {abs(FXr):.6f})")
    return Xr
print("## Regula Falsi Method ##")
# Example: "x*x - 4*x - 4"
# Example: "m.cos(x) - x"
# Example: "x**3 - x - 1"
# Example: "x*x*x - 4*x - 4"
Func_str = input("Enter the function f(x): ")
a = float(input("Enter the starting value a: "))
b = float(input("Enter the starting value b: "))

tol = float(input("Enter the tolerance value: "))
```

```
root = Regula_Falsi(Func_str, a, b, tol)
if root is not None:
    print(f"\nApproximate root = {root:.3f} (correct to 3 decimal places)")
```

OUTPUT:

```
=== Regula Falsi Method ===
Enter the function f(x): x*x*x -4*x +1
Enter the starting value a: 1
Enter the ending value b: 2

Iter       a        b        f(a)     f(b)      Xr      f(Xr)
  1     1.0000   2.0000   -2.0000 1.0000   1.6667   -1.0370
  2     1.6667   2.0000   -1.0370 1.0000   1.8364   -0.1528
  3     1.8364   2.0000   -0.1528 1.0000   1.8581   -0.0175
  4     1.8581   2.0000   -0.0175 1.0000   1.8605   -0.0020
  5     1.8605   2.0000   -0.0020 1.0000   1.8608   -0.0002
  6     1.8608   2.0000   -0.0002 1.0000   1.8608   -0.0000

Approximate root = 1.8608 (correct to 3 decimal places)
```

CONCLUSION: The Above Program Has Been Executed Successfully.

18078

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

PROGRAM:

```
Import math
def safe_eval(expr, x):
    try:
        return eval(expr.strip(), {"x": x, "math": math, "_builtins_": None})
    except (NameError, TypeError, ZeroDivisionError, SyntaxError) as e:
        print(f"Error evaluating function: {e}")
        return None
def df(x, deriv_str):
    try:
        return eval(deriv_str.strip(), {"x": x, "math": math, "_builtins_": None})
    except Exception as e:
        print(f"Error evaluating derivative: {e}")
        return None
def Newton_Raphson_Method(func_str, deriv_str, x0, tol, max_iter=100):
    ai = X0
    print("\nIter.\t ai\t\t f(ai)\t\t df(ai)\t\t ai+1")
for i in range(1, max_iter + 1):
        fai = safe_eval(func_str, ai)
        dfai = df(ai, deriv_str)
        if dfai == 0:
```

```
            print("Derivative is zero. Method fails.")
            return None
        ai_p1 = ai - fai / dfai
        print(f"{i:<6}\t {ai:.4f}\t {fai:.4f}\t {dfai:.4f}\t {ai_p1:.4f}")
        if abs(ai_p1 - ai) < tol:
            print(f"\nApproximate root = {ai_p1:.3f} (correct to 3 decimal places)")
            return ai_p1
        ai = ai_p1
    print(f"\nMaximum iterations reached without convergence.")
    return ai_p1
import math
print("## Newton-Raphson Method ##")
# Example 1: "x*x - 4*x - 4"
# Example 2: "m.cos(x) - x"
# Example 3: "x**3 - x - 1"
# Example of derivative: "3*x*x - 1" for f(x)=x**3 - x - 1
func_str = input("Enter the function f(x): ")
deriv_str = input("Enter the derivative df(x): ")
x0 = float(input("Enter the initial guess x0: "))
tol = float(input("Enter the tolerance for X decimal place accuracy: "))
newton_raphson(func_str, deriv_str, x0, tol)
```

OUTPUT:

```
=== Newton-Raphson Method ===
Enter the function f(x): x*x*x -2*x -5
Enter the derivative f'(x): 3*x*x -2
Enter the initial guess x0: 2

================ Newton-Raphson Iteration Table ================
Iter  x0          f(x0)       f(x0)       x1
---------------------------------------------------------------
1     2.000000    -1.000000   10.000000   2.100000
2     2.100000    0.061000    11.230000   2.094568
3     2.094568    0.000186    11.161647   2.094551
===============================================================

Approximate root = 2.0946 (correct to 3 decimal places)
```

CONCLUSION: The Above Program Has Been Executed Successfully.

PRACTICAL NO : 2

INTERPOLATION

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

PROGRAM:

```
def forward_difference_table(x, y):
    n = len(y)
    diff_table = [y.copy()] # First row is just y values
    for i in range(1, n):
        row = []
        for j in range(n - i):
            # Calculate the i-th difference: diff(j) = diff(j+1) - diff(j)
            value = diff_table[i-1][j+1] - diff_table[i-1][j]
            row.append(value)
        diff_table.append(row)
    return diff_table
def display_table(x, diff_table):
    n = len(x)
    print("\nForward Difference Table:")
    header = "i\t x\t\t y" + "\t\t Δy" * (n - 1)
    print(header)
```

```
    print("-" * len(header) * 2) # For visual separation

    for i in range(n):
row = [str(i), f"{x[i]:.2f}", f"{diff_table[0][i]:.2f}"]
        for j in range(1, n - i):
            row.append(f"{diff_table[j][i]:.2f}")
        print("\t".join(row))
def main():
    n = int(input("Enter the number of data points: "))
    x = []
    y = []
    print("Enter x values (equally spaced):")
    for i in range(n):
        x.append(float(input(f"x[{i}] = ")))
    print("Enter corresponding y values:")
    for i in range(n):
        y.append(float(input(f"y[{i}] = "))
    # Check equal spacing
    h_values = []
    for i in range(n - 1):
        h_values.append(x[i+1] - x[i])
    if not all(abs(h_values[i] - h_values[0]) < 1e-5 for i in range(n - 1)):
        print("\nError: X values are not equally spaced.")
        return
```

```
diff_table = forward_difference_table(x, y)
   display_table(x, diff_table
if _name_ == "_main_":
   main()
```

OUTPUT:

```
Forward Difference Table:
x       Δ^0y    Δ^1y    Δ^2y    Δ^3y    Δ^4y    Δ^5y    Δ^6y    Δ^7y    Δ^8y
-1.00   -13.00  6.00    0.00    6.00    0.00    0.00    0.00    0.00    0.00
0.00    -7.00   6.00    6.00    6.00    0.00    0.00    0.00    0.00
1.00    -1.00   12.00   12.00   6.00    0.00    0.00    0.00
2.00    11.00   24.00   18.00   6.00    0.00    0.00
3.00    35.00   42.00   24.00   6.00    0.00
4.00    77.00   66.00   30.00   6.00
5.00    143.00  96.00   36.00
6.00    239.00  132.00
7.00    371.00
```

CONCLUSION: The Above Program Has Been Executed Successfully.

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

PROGRAM:

```
import math
x = [0, 30, 60, 90]
y = [1, 0.85, 0.5, 0]
xp = 70
h = x[1] - x[0]
p = (xp - x[-1]) / h
dy1_3 = y[3] - y[2]
dy1_2 = y[2] - y[1]
dy1_1 = y[1] - y[0]
d2y2 = dy1_3 - dy1_2
d2y1 = dy1_2 - dy1_1
d3y1 = d2y2 - d2y1
print("x\t y\t ∇y\t ∇²y\t ∇³y")
print(f"{x[0]}\t {y[0]}")
print(f"{x[1]}\t {y[1]}\t {dy1_1:.4f}")
print(f"{x[2]}\t {y[2]}\t {dy1_2:.4f}\t {d2y1:.4f}")
print(f"{x[3]}\t {y[3]}\t {dy1_3:.4f}\t {d2y2:.4f}\t {d3y1:.4f}")
```

```
yp = (y[-1]
      + p * dy1_3
      + (p * (p + 1) / math.factorial(2)) * d2y2
      + (p * (p + 1) * (p + 2) / math.factorial(3)) * d3y1)
print(f"\nEstimated cos(70°) using Backward formula = {yp:.5f}")
```

OUTPUT:

```
==========================================================
x         y        ∇y       ∇2y      ∇3y
0         1
30        0.85     -0.1500
60        0.5      -0.3500 -0.2000
90        0        -0.5000 -0.1500 0.0500

Estimated cos(70°) using Backward formula = 0.34753
```

CONCLUSION: The Above Program Has Been Executed Successfully.

PRACTICAL NO : 3

CURVE FITTING

AIM: TO WRITE A PYTHON PROGRAM TO DEMONSTRATE A STRAIGHT LINE

PROGRAM:

```
def fit_line(x_values: List[float], y_values: List[float]) -> Tuple[float,
float]:
    """Return (a0, a1) for best fit line y = a0 + a1*x using least squares."""
    if len(x_values) != len(y_values) or len(x_values) == 0:
        raise ValueError("x_values and y_values must have same non-zero
length.")
    n = len(x_values)
    sum_x = sum(x_values)
    sum_y = sum(y_values)
    sum_x2 = sum(x * x for x in x_values)
    sum_xy = sum(x * y for x, y in zip(x_values, y_values))

    denom = n * sum_x2 - sum_x * sum_x
    if abs(denom) < 1e-12:
        raise ValueError("Denominator nearly zero: can't compute unique fit
(collinear x?).")

    a1 = (n * sum_xy - sum_x * sum_y) / denom
    a0 = (sum_y - a1 * sum_x) / n
    return a0, a1
def print_table(x_values: List[float], y_values: List[float]) -> None:
    """Print table of x, y, x^2, x*y and the sums."""
    n = len(x_values)
    rows = []
```

```
    for x, y in zip(x_values, y_values):
        rows.append((x, y, x*x, x*y))

    # Column widths
    w = [8, 8, 10, 10]
    header = f"{'i':<{w[0]}} {'y':<{w[1]}} {'x^2':<{w[2]}} {'x*y':<{w[3]}}"
    print(header)
    print("*" * (sum(w) + 3))

    for r in rows:
        print(f"{r[0]:<{w[0]}.4g} {r[1]:<{w[1]}.4g} {r[2]:<{w[2]}.4g} 
{r[3]:<{w[3]}.4g}")

     sum_x = sum(r[0] for r in rows)
    sum_y = sum(r[1] for r in rows)
    sum_x2 = sum(r[2] for r in rows)
    sum_xy = sum(r[3] for r in rows)

    # print the sums
    print("-" * (sum(w) + 3))
    print(f"{'SUM':<{w[0]}} {sum_y:>{w[1]}.4g} {sum_x2:>{w[2]}.4g} 
{sum_xy:>{w[3]}.4g}")
    print()

    # The image shows extra print statements for the sums:
    print(f"{'SUM_X':<{w[0]}} {sum_y:>{w[1]}.4g} {sum_x2:>{w[2]}.4g} 
{sum_xy:>{w[3]}.4g}")
    print() # extra line break from image 1000040410.jpg

    print(f"Σx = {sum_x:.4g}, Σy = {sum_y:.4g}, Σx^2 = {sum_x2:.4g}, Σxy = 
{sum_xy:.4g}")
    print()
```

```
def predict(a0: float, a1: float, x: float) -> float:
    return a0 + a1 * x

def interactive():
    print("Curve fitting (straight line) - enter data points.")
    n = int(input("How many points? "))
    x_values = []
    y_values = []

    for i in range(n):
        raw = input(f"Point {i+1} as 'x y' (e.g. 2 5): ").strip().split()
        if len(raw) < 2:
            print("Invalid input, try again.")
            return
        x_values.append(float(raw[0]))
        y_values.append(float(raw[1]))

    print()
    print_table(x_values, y_values)
    a0, a1 = fit_line(x_values, y_values)
    print(f"Best fit line: y = ({a0:.6f}) + ({a1:.6f}) x")
    choice = input("Predict y for some x? (y/n): ").strip().lower()
    if choice and choice[0] == 'y':
xv = float(input("Enter x: "))
   print(f"Predicted y = {predict(a0, a1, xv):.6f}")
if _name_ == "_main_":
    # Example usage (change values directly if you prefer):
    x_values = [0, 2, 5, 7]
    y_values = [-1, 5, 12, 20]
   # Print table and compute
    print_table(x_values, y_values)
    a0, a1 = fit_line(x_values, y_values)
    print(f"Best fit line: y = ({a0:.6f}) + ({a1:.6f}) x")
```

```
print(f"For x=0, predicted y = {predict(a0, a1, 0):.6f}")
```

OUTPUT:

```
      x           y          x^2         x*y
--------------------------------------------
      0          -1            0           0
      2           5            4          10
      5          12           25          60
      7          20           49         140
--------------------------------------------
      Σ          36           78         210

Σx = 14, Σy = 36, Σx^2 = 78, Σxy = 210

Best fit line: y = -1.137931 + 2.896552 x
For x=8, predicted y = 22.034483
```

CONCLUSION: The Above Program Has Been Executed Successfully.

AIM: TO WRITE A PYTHON PROGRAM TO DEMONSTRATE DEGREE POLYNOMIAL

PROGRAM:

```
def build_sums(x_values: List[float], y_values: List[float]) -> dict:
    """Calculates the necessary sums for the normal equations."""
    s = {
        'n': 0.0,
        'sx': 0.0,
        'sx2': 0.0,
        'sx3': 0.0,
        'sx4': 0.0,
        'sy': 0.0,
        'sxy': 0.0,
        'sx2y': 0.0
    }
    for x, y in zip(x_values, y_values):
        s['n'] += 1
        s['sx'] += x
        s['sx2'] += x**2
        s['sx3'] += x**3
        s['sx4'] += x**4
```

```
        s['sy'] += y

        s['sxy'] += x * y
        s['sx2y'] += (x**2) * y
return s
def print_table_and_sums(x_values: List[float], y_values: List[float]) -> None:
    """Prints the data points and the calculated sums in a formatted table."""
    # Header
print(f"{'x':>8}{'y':>10}{'x^2':>12}{'x^3':>12}{'x^4':>12}{'x*y':>12}{'x^2*y':>12}")
    print("-" * 78) # Separator
    # Data rows
    for x, y in zip(x_values, y_values):
print(f"{x:8.4g}{y:10.4g}{x*2:12.4g}{x3:12.4g}{x4:12.4g}{x*y:12.4g}{(x*2)*y:12.4
g}")
    # Sums
    s = build_sums(x_values, y_values)
    print("-" * 78) # Separator
    print(f"n = {s['n']:3.4g}, sx = {s['sx']:12.4g}, sx2 = {s['sx2']:12.4g}, sx3 =
{s['sx3']:12.4g}, sx4 = {s['sx4']:12.4g}")
    print(f"sy = {s['sy']:12.4g}, sxy = {s['sxy']:12.4g}, sx2y = {s['sx2y']:12.4g}")
    print()
def solve_3x3(A: List[List[float]], b: List[float]) -> List[float]:
```

```
    """
    Simple Gaussian elimination (in-place) to solve Ax = b for a 3x3 A.
    Returns the solution vector x.
    """
    # Make copies
M = [row[:] for row in A]
  rhs = b[:]
    n = 3
# Forward elimination
    for k in range(n):
        # find pivot
        pivot = M[k][k]
# Check for singularity/pivot too small (1e-14 is a common threshold)
        if abs(pivot) < 1e-14:
# try to swap with a lower row
            for i in range(k + 1, n):
                if abs(M[i][k]) > 1e-14:
                    M[k], M[i] = M[i], M[k]
                    rhs[k], rhs[i] = rhs[i], rhs[k]
                    pivot = M[k][k]
                    break
        if abs(pivot) < 1e-14:
            raise ValueError("Singular matrix in solve_3x3")
        # normalize row k
```

```
    for j in range(k, n):
      M[k][j] /= pivot
    rhs[k] /= pivot

    # eliminate
    for i in range(k + 1, n):
      factor = M[i][k]
for j in range(k, n):
        M[i][j] -= factor * M[k][j]
      rhs[i] -= factor * rhs[k]
# Back substitution
  x = [0.0] * n
  for i in range(n - 1, -1, -1):
    val = rhs[i]
    for j in range(i + 1, n):
      val -= M[i][j] * x[j]
    x[i] = val / M[i][i] if abs(M[i][i]) > 1e-14 else val
  return x
def fit_quadratic(x_values: List[float], y_values: List[float]) -> Tuple[float, float, float]:
  """Calculates the coefficients (a0, a1, a2) for the least-squares quadratic fit."""
  if len(x_values) != len(y_values) or len(x_values) == 0:
    raise ValueError("X-values and Y-values must have same non-zero length.")
```

```
   s = build_sums(x_values, y_values)
 A = [
       [s['n'],   s['sx'],  s['sx2']],
       [s['sx'],  s['sx2'], s['sx3']],
       [s['sx2'], s['sx3'], s['sx4']]
 ]
   b = [s['sy'], s['sxy'], s['sx2y']]
 # Solve for a0, a1, a2
   a0, a1, a2 = solve_3x3(A, b)
   return a0, a1, a2
def predict(a0: float, a1: float, a2: float, x: float) -> float:
   """Calculates the predicted y value for a given x using the fitted quadratic."""
   return a0 + a1*x + a2*(x**2)
if _name_ == "_main_":
# Example points from your notebook: (0, 1), (1, 6), (2, 17)
   x_values = [0.0, 1.0, 2.0]
   y_values = [1.0, 6.0, 17.0]
   print("### Input Data and Sums ###")
   print_table_and_sums(x_values, y_values)
# Fit quadratic
   a0, a1, a2 = fit_quadratic(x_values, y_values)
   print("### Fitting Results ###")
print(f"Fitted quadratic: y = {a0:.6f} + {a1:.6f} x + {a2:.6f} x^2")
   # Predictions requested in the notebook
```

```
print("\n### Predictions ###")
# Prediction for x=1.6
print(f"y(1.6) = {predict(a0, a1, a2, 1.6):.6f}")

# Prediction for x=3.0
print(f"y(3)   = {predict(a0, a1, a2, 3.0):.6f}")
```

OUTPUT:

```
       x         y        x^2        x^3        x^4        x*y      x^2*y
--------------------------------------------------------------------------
       0         1          0          0          0          0          0
       1         6          1          1          1          6          6
       2        17          4          8         16         34         68
--------------------------------------------------------------------------
       Σ        24          5          9         17         40         74

Σx = 3.0, Σy = 24.0, Σx^2 = 5.0, Σx^3 = 9.0, Σx^4 = 17.0
Σ(xy) = 40.0, Σ(x^2 y) = 74.0

Fitted quadratic: y = 1.000000 + 2.000000 x + 3.000000 x^2
y(1.6) = 11.880000
y(3)   = 34.000000
```

CONCLUSION: The Above Program Has Been Executed Successfully.

PRACTICAL NO : 4
SOLUTION OF SIMULTANEOUS ALGEBRAIC EQUATIONS

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

PROGRAM:

```
#Gaussian elimination with partial pivoting row operations
# Solve the system:
# x1 + 10 x2 - x3 = 3
# 2x1 + 3 x2 + 20 x3 = 7
# 10x1 - x2 + 2 x3 = 4
# Augmented matrix (each row: [a11, a12, a13, b])
A = [
  [1.0, 10.0, -1.0, 3.0],
  [2.0, 3.0, 20.0, 7.0],
  [10.0, -1.0, 2.0, 4.0]
]
n = len(A) # Number of equations/variables (n=3)
def print_matrix(M: List[List[float]], msg=None) -> None:
  """Prints the augmented matrix with 10.6f formatting."""
  if msg:
    print(msg)
```

```
            for r in M:
    # Join values with spaces, formatting each to 10 characters with 6 decimal
  place
   print("[" + " ".join(f"{val:10.6f}" for val in r) + "]"
     print()
def swap_rows(M: List[List[float]], i: int, j: int) -> None:
            """Swaps row i and row j in matrix M and prints the operation."""
            M[i], M[j] = M[j], M[i]
            # Print R(i+1) <-> R(j+1) to use 1-based indexing for output
            print(f"R({i+1}) <-> R({j+1})")
            print_matrix(M)
        def scale_and_add(M: List[List[float]], col: int, factor: float, row: int) ->
        None:
            """
            Performs R_dest = R_dest - k * R_src.
            In the context of elimination, row is dest (row to eliminate in), col is
        src.
            """
            n_cols = len(M[0])
            # Print R(dest+1) <- R(dest+1) - (k) * R(src+1) to use 1-based indexing
            print(f"R({row+1}) <- R({row+1}) – ({factor:.6f})*R({col})”)
         # Perform the operation: M[row] = M[row] - factor * M[col]
            for c in range(n_cols):
              M[row][c] = M[row][c] - factor * M[col][c]
            print_matrix(M)
```

```
# --- Main Solution Logic ---
# Work on a copy of the augmented matrix
M = deepcopy(A)
print_matrix(M, "Initial augmented matrix [A | b]:")
for col in range(n):
# Partial pivot: find row with max abs value in column 'col' from rows col..n-1
 # max() returns the row index 'r'
    pivot_row = max(range(col, n), key=lambda r: abs(M[r][col]))
    if pivot_row != col:
        swap_rows(M, pivot_row, col)
    pivot = M[col][col]
    if abs(pivot) < 1e-12: # Check for near-zero pivot (singularity)
        raise ValueError("Zero pivot encountered")
    for row in range(col + 1, n):
        factor = M[row][col] / pivot
        # scale_and_add(Matrix, source_row, factor, destination_row)
        scale_and_add(M, col, factor, row)
print("Upper-triangular matrix after forward elimination:")
print_matrix(M)
# Back substitution
x = [0.0] * n # Solution vector [x1, x2, x3]
```

```
# Loop backward from the last row (n-1) to the first row (0)
for i in range(n - 1, -1, -1):
   s = M[i][n]
   for j in range(i + 1, n)
      s -= M[i][j] * x[j]
   x[i] = s / M[i][i]
print("Solution vector:")
# Enumerate x starting from 1 for x1, x2, x3 display
for i, xi in enumerate(x, 1):
   print(f"x{i} = {xi:.8f}")
```

OUTPUT:

```
Initial augmented matrix [A | b]:
  [  1.000000    10.000000    -1.000000     3.000000]
  [  2.000000     3.000000    20.000000     7.000000]
  [ 10.000000    -1.000000     2.000000     4.000000]

R3 <-> R1
  [ 10.000000    -1.000000     2.000000     4.000000]
  [  2.000000     3.000000    20.000000     7.000000]
  [  1.000000    10.000000    -1.000000     3.000000]

R2 = R2 - (0.200000)*R1
  [ 10.000000    -1.000000     2.000000     4.000000]
  [  0.000000     3.200000    19.600000     6.200000]
  [  1.000000    10.000000    -1.000000     3.000000]

R3 = R3 - (0.100000)*R1
  [ 10.000000    -1.000000     2.000000     4.000000]
  [  0.000000     3.200000    19.600000     6.200000]
  [  0.000000    10.100000    -1.200000     2.600000]

R3 <-> R2
  [ 10.000000    -1.000000     2.000000     4.000000]
  [  0.000000    10.100000    -1.200000     2.600000]
  [  0.000000     3.200000    19.600000     6.200000]

R3 = R3 - (0.316832)*R2
  [ 10.000000    -1.000000     2.000000     4.000000]
  [  0.000000    10.100000    -1.200000     2.600000]
  [  0.000000     0.000000    19.980198     5.376238]

Upper-triangular matrix after forward elimination:
  [ 10.000000    -1.000000     2.000000     4.000000]
  [  0.000000    10.100000    -1.200000     2.600000]
  [  0.000000     0.000000    19.980198     5.376238]

Solution vector:
 x1 = 0.37512389
 x2 = 0.28939544
 x3 = 0.26907830
```

CONCLUSION: The Above Program Has Been Executed Successfully.

PRACTICAL NO : 5

NUMERIACAL SOLUTIONS OF FIRST AND SECOND ORDER DIFFERENTIAL EQUATIONS

AIM: TO WRITE A PROGRAM IN PYTHONTO DEMONSTRATE TAYLOR SERIES

PROGRAM:

```
# Taylor_ode_no_sympy.py
# Compute y(n0 + h) using Taylor series for ODE dy/dn = n - y^2, y(n0)=y0
# No external libraries beyond 'math'
from math import factorial
def compute_derivatives_at(n0: float, y0: float) -> List[float]:
    """
    Compute derivatives y', y'', y''', y^(4), y^(5) at (n0, y0)
    using formulas obtained by differentiating dy/dn = n - y^2.
    Returns list [y0, y1, y2, y3, y4, y5], where yk is the kth derivative.
    """
    # y^(0) = y0
    y_0 = y0
    # y' = n - y^2
    y_1 = n0 - (y_0 ** 2)
    # Second derivative: y'' = d/dn(n - y^2) = 1 - 2*y*(dy/dn) = 1 - 2*y*y'
    y_2 = 1.0 - 2.0 * y_0 * y_1
    # Third derivative: y''' = d/dn(1 - 2*y*y') = 0 - 2 * [ y'*y' + y*y'' ]
    # y''' = -2*y'^2 - 2*y*y''
```

```
    y_3 = -2.0 * (y_1 ** 2) - 2.0 * y_0 * y_2
  # Fourth derivative: y^(4) = d/dn(-2*y'^2 - 2*y*y'')
    # y^(4) = -2*(2*y'y'') - 2[ y'*y'' + y*y''' ]
    # y^(4) = -4*y'*y'' - 2*y'*y'' - 2*y*y''' = -6*y'*y'' - 2*y*y'''
    y_4 = -2.0 * y_0 * y_3 - 6.0 * y_1 * y_2
  # Fifth derivative: y^(5) = d/dn(-6*y'*y'' - 2*y*y''')
    # y^(5) = -6*[ y''y'' + y'*y''' ] - 2[ y'*y''' + y*y^(4) ]
    # y^(5) = -6*y''^2 - 6*y'*y''' - 2*y'*y''' - 2*y*y^(4)
    # y^(5) = -2*y*y^(4) - 8*y'*y''' - 6*y''^2
    y_5 = -2.0 * y_0 * y_4 - 8.0 * y_1 * y_3 - 6.0 * (y_2 ** 2)
    return [y_0, y_1, y_2, y_3, y_4, y_5]
def taylor_at(n0: float, y0: float, h: float, order: int = 5) -> Tuple[float,
List[float]]:
    """
    Evaluate Taylor polynomial of given order (<=5) for y at n0+h.
    Returns (approx_value, derivatives_list).
    ""”
    if order > 5:
        raise ValueError("This implementation supports up to 5th derivative
(order<=5).")
    derivs = compute_derivatives_at(n0, y0)
    # Build Taylor sum: y(n0+h) approx y(n0) + h*y'(n0)/1! + h^2*y''(n0)/2! +
…

    taylor_sum = 0.0
```

```
    for k in range(order + 1):
# Term = y^(k) * h^k / k!
  taylor_sum += derivs[k] * (h ** k) / factorial(k)
 return taylor_sum, derivs
if _name_ == "_main_":
    # Initial point and step
    n0 = 0.0
    y0 = 1.0
    h = 0.1
    order = 5 # use terms up to y^(5)/5!
    # Compute the approximation and the derivatives
    approx, derivs = taylor_at(n0, y0, h, order)
    print("Derivatives at n0 = {:.4g}, y0 = {:.4g}:".format(n0, y0))
    print(f"y'(0)    = {derivs[1]:.6g}")
    print(f"y''(0)   = {derivs[2]:.6g}")
    print(f"y'''(0)  = {derivs[3]:.6g}")
    print(f"y^(4)(0) = {derivs[4]:.6g}")
    print(f"y^(5)(0) = {derivs[5]:.6g}")
    print()
    print("Taylor approximation up to order {}:".format(order))
    print(f"y({n0 + h:.4g}) ≈ {approx:.10f}")
    print(f"Rounded to 4 decimal places: {approx:.4f}")
```

OUTPUT:

```
================================================================================
Derivatives at n0 = 0, y0 = 1:
 y(0)      = 1
 y'(0)     = -1
 y''(0)    = 3
 y'''(0)  = -8
 y^(4)(0)= 34
 y^(5)(0)= -186

Taylor approximation up to order 5:
 y(0.1) ≈ 0.9137928333
 Rounded to 4 decimal places: 0.9138
```

CONCLUSION: The Above Program Has Been Executed Successfully.

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

PROGRAM:

```
# Euler_method.py
# Solve dy/dx = -y with y(0)=1 using Euler's method
def f(x, y):
   """The ODE: dy/dx = -y"""
   return -y
def euler(x0, y0, h, x_target):
   """Euler's method to approximate y(x_target)"""
   steps = int((x_target - x0) / h)
   x = x0
   y = y0
   print("Step |   x    |    y   ")
   print("-----|--------|---------")
   print(f"  0  | {x:.2f} | {y:.6f}")
   for i in range(1, steps + 1):
      # Euler formula: y(i+1) = y(i) + h * f(x(i), y(i))
      y = y + h * f(x, y)
      x = x + h * f(x,y)
print(f"{i:3d}  | {x:.2f} | {y:.6f}"
   return y
if _name_ == "_main_":

   # initial values
```

```
x0 = 0.0

y0 = 1.0

h = 0.01

x_target = 0.04

result = euler(x0, y0, h, x_target)

print(f"\nApproximate value at x={x_target:.2f}:", round(result, 6))
```

OUTPUT:

```
========================================================
Step |   x    |      y
------------------------
  0  | 0.00 | 1.000000
  1  | 0.01 | 0.990000
  2  | 0.02 | 0.980100
  3  | 0.03 | 0.970299
  4  | 0.04 | 0.960596

Approximate value at x=0.04: 0.960596
```

CONCLUSION: The Above Program Has Been Executed Successfully.

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

PROGRAM:

```
# Modified_Euler's_Method.py
# Equation: dy/dx = x^2 + y, y(0) = 1
# Find y(0.2) with step size h = 0.02
def f(x, y):
   """The ODE: dy/dx = x^2 + y"""
   return x**2 + y
# Initial conditions
x0 = 0
y0 = 1
h = 0.02
x_end = 0.2
n = int((x_end - x0) / h) # Number of steps: (0.2 - 0) / 0.02 = 10
# Table header
print("-----------------------------------------------------------------------------------------")
print("i |    x(i)    |    y(i)    |   f(x(i),y(i)) (f1)  |   y'(Pred) |   f(x+h, y') (f2) |   y(i+1)")
print("-----------------------------------------------------------------------------------------")
# Print initial condition (Step 0)
print(f"{0:<2d} | {x0:10.6f} | {y0:10.6f} | {'':20} | {'':10} | {'':20} | {'':10}")


# Iterative Modified Euler Calculation
    # 1. Predictor (Standard Euler): y* = y_i + h * f(x_i, y_i)
```

```
    f1 = f(x0, y0)
  y_pred = y0 + h * f1
    # 2. Corrector (Heun's Formula): y_i+1 = y_i + (h / 2) * [ f(x_i, y_i) + f(x_i+1, y*) ]
    x_next = x0 + h
    f2 = f(x_next, y_pred)
    y_next = y0 + (h / 2) * (f1 + f2)
    # Print intermediate results for the current step (i+1)
    print(f"{i+1:<2d} | {x_next:10.6f} | {y0:10.6f} | {f1:20.6f} | {y_pred:10.6f} | {f2:20.6f} | {y_next:10.6f}")
    # Update for next iteration
    x0 = x_next
    y0 = y_next
print("-------------------------------------------------------------------------------------")
print(f"h = {h:.2f}, Number of steps = {n}")
print(f"Formula used:")
print(f"y*(i+1) = y_i + h * f(x_i, y_i) (Euler Predictor)")
print(f"y(i+1)  = y_i + (h/2) * [ f(x_i, y_i) + f(x_i + h, y*(i+1)) ] (Heun Corrector)")
print(f"Approximate value of y({x_end:.1f}) = {y0:.6f}")
print("-------------------------------------------------------------------------------------")
```

OUTPUT:

```
----------------------------------------------------------------
  i     x(i)          y(i)           f(x(i),y(i))      y* (Pred)        f(x+h, y*)       y(i+1)
----------------------------------------------------------------
  0     0.0000        1.000000       1.000000          1.020000         1.020400         1.020204
  1     0.0200        1.020204       1.020604          1.040616         1.042216         1.040832
  2     0.0400        1.040832       1.042432          1.061681         1.065281         1.061909
  3     0.0600        1.061909       1.065509          1.083220         1.089620         1.083461
  4     0.0800        1.083461       1.089861          1.105258         1.115258         1.105512
  5     0.1000        1.105512       1.115512          1.127822         1.142222         1.128089
  6     0.1200        1.128089       1.142489          1.150939         1.170539         1.151219
  7     0.1400        1.151219       1.170819          1.174636         1.200236         1.174930
  8     0.1600        1.174930       1.200530          1.198941         1.231341         1.199249
  9     0.1800        1.199249       1.231649          1.223882         1.263882         1.224204
----------------------------------------------------------------
h = 0.02, Number of steps = 10
Formula used:
y_(i+1) = y_i + (h/2) * [f(x_i, y_i) + f(x_i + h, y*)]
----------------------------------------------------------------
Approximate value of y(0.2) = 1.224204
----------------------------------------------------------------
```

CONCLUSION: The Above Program Has Been Executed Successfully.

AIM: TO WRITE A PROGRAM IN PYTHON TO DEMONSTRATE RUNGE-KUTTA 4^{th} ORDER METHOD.

PROGRAM:

```
RK4 step-by-step for y' = x + y, y(0)=1

import math

def f(x, y):

    """The ODE: y' = x + y"""

    return x + 1

def exact_solution(x):

    """The exact solution of y' - y = x with y(0)=1 is y = 2*e^x - x - 1"""

    return 2 * math.exp(x) - x - 1

# Initial values

x = 0.0

y = 1.0

h = 0.1

steps = int(0.2 / h) # Compute up to x = 0.2 (steps = 2)print("Runge-Kutta 4th order (RK4) step-by-step")

print(f"Equation: y' = x + y, y(0)={y}")

print(f"Step | x_n | y_n (before) |   k1   |   k2   |   k3   |   k4   | y_{steps * h:.1f}")

print("-" * 100)



for n in range(steps):
```

```
 # RK4 coefficients
   k1 = f(x, y) * h
k2 = f(x + h/2.0, y + (k1/2.0)) * h
   k3 = f(x + h/2.0, y + (k2/2.0)) * h
   k4 = f(x + h, y + k3) * h
   # RK4 Update Formula
   increment = (h/6.0) * (k1 + 2.0*k2 + 2.0*k3 + k4)
   y_next = y + increment
 # Print step details
 # Using 'n' for the step count (0 and 1) and 'n+1' for the y_next index (1 and 2)
   print(f"{n+1:3d}  | {x:6.3f} | {y:16.10f} | "
        f"{k1:7.6f} | {k2:7.6f} | {k3:7.6f} | {k4:7.6f} | {y_next:10.9f}")
   # Update
   x += h
   x = round(x, 10) # avoid floating accumulation
   y = y_next
print("-" * 100)
# Final output
exact_y = exact_solution(x)
absolute_error = abs(y - exact_y)
print(f"Final RK4 approximation: y({x:.3f}) = {y:.9f}")
print(f"Exact value         : y({x:.3f}) = {exact_y:.9f}")
print(f"Absolute error      : |abs({absolute_error:.12e})")
```

OUTPUT:

```
Runge-Kutta 4th order (RK4) step-by-step
Equation: y' = x + y , y(0)=1

 Step |   x_n   |    y_n (before)    |   k1        k2        k3        k4       |   y_(n+1)
-----------------------------------------------------------------------------------------------
   1  |  0.000  |     1.0000000000 | 1.000000  1.100000  1.105000  1.210500 | 1.110341667
   2  |  0.100  |     1.1103416667 | 1.210342  1.320859  1.326385  1.442980 | 1.242805142
-----------------------------------------------------------------------------------------------
Final RK4 approximation: y(0.200) = 1.242805142
Exact value            : y(0.200) = 1.242805516
Absolute error         : 3.746189507492e-07
```

CONCLUSION: The Above Program Has Been Executed Successfully.

PRACTICAL NO : 6

NUMERICAL INTEGRATION

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

PROGRAM:

```
# Trapezoidal Rule for I = integral(f(x) dx from 0 to 1) with 2 subintervals
def f(x):
    """The integrand: 1 / (1 + x^2)"""
    return 1 / (1 + x**2
# Given limits
a = 0
b = 1
n = 2 # number of subintervals
# Step size
h = (b - a) / n
# Compute x values
x = [a + i * h for i in range(n + 1)] # [0.0, 0.5, 1.0]
# Compute f(x) values
f_values = [f(xi) for xi in x] # [1.0, 0.8, 0.5]



# Display table header
print("-----------------------------------")
```

```
print("i | x(i)   | f(x(i)) = 1/(1+x^2)")
print("-----------------------------------")
# Display table values
for i in range(n + 1):
    print(f"{i:<3} | {x[i]:<8.4f} | {f_values[i]:<8.4f}")
print("-----------------------------------")
# Apply Trapezoidal Rule
# The formula in Python: I = (h / 2) * (f[0] + 2 * sum(f[1:-1]) + f[-1])
I = (h / 2) * (f_values[0] + 2 * sum(f_values[1:-1]) + f_values[-1])
# Step-by-step explanation
print(f"h = (b - a) / n = ({b} - {a}) / {n} = {h}")
print("\nUsing Trapezoidal Rule:")
print(f"I = (h / 2) * [f(x0) + 2*f(x1) + f(x2)]")
print(f"I = ({h/2}) * [{f_values[0]:.4f} + 2*{f_values[1]:.4f} +
{f_values[2]:.4f}]")

# Final result
print("\n-----------------------------------")
print(f"Approximate value of the integral I = {I:.4f}")
print("-----------------------------------")
```

OUTPUT:

```
-------------------------------------------------
  i      x(i)        f(x(i)) = 1/(1+x^2)
-------------------------------------------------
  0      0.0000        1.0000
  1      0.5000        0.8000
  2      1.0000        0.5000
-------------------------------------------------

h = (b - a) / n = (1 - 0) / 2 = 0.5

Using Trapezoidal Rule:
I = (h/2) * [f(x0) + 2*f(x1) + f(x2)]
I = (0.5/2) * [1.0000 + 2*0.8000 + 0.5000]

----------------------------------------------
Approximate value of the integral I = 0.7750
----------------------------------------------
```

CONCLUSION: The Above Program Has Been Executed Successfully.

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

PROGRAM:

```
# Simpson's 1/3 Rule for I = ∫(0 to 1) e^(-x^2) dx with n = 4
import math
# Define the function
def f(x):
 return math.exp(-x**2)
# Given values
a = 0  # lower limit
b = 1  # upper limit
n = 4  # number of subintervals (must be even)
# Step size
h = (b - a) / n
# Generate x and f(x) values
x = [a + i * h for i in range(n + 1)]
f_values = [f(xi) for xi in x]
# Display table
print("-----------------------------------------------")
print(" i   | x(i)   | f(x(i)) = e^(-x^2)")
print("-----------------------------------------------")

for i in range(n + 1):
    print(f"{i:<3} | {x[i]:<8.4f} | {f_values[i]:<10.6f}")
```

```
print("------------------------------------------------")
# Simpson's 1/3 rule computation
sum_odd = sum(f_values[i] for i in range(1, n, 2))
sum_even = sum(f_values[i] for i in range(2, n, 2))
I = (h / 3) * (f_values[0] + 4 * sum_odd + 2 * sum_even + f_values[-1])
# Show steps
print(f"\nh = (b - a) / n = ({b} - {a}) / {n} = {h}")
print("\nUsing Simpson's 1/3 Rule:")
print(f"I = (h/3) * [f(x0) + 4*(f(x1) + f(x3) + ...) + 2*(f(x2) + f(x4) + ...) + f(xn)]")
print(f"I = ({h}/3) * [{f_values[0]:.6f} + 4*{{{sum_odd:.6f}}} + 2*{{{sum_even:.6f}}} + {f_values[-1]:.6f}]")

# Display final result
print("\n------------------------------------------------")
print(f"Approximate value of the integral I = {I:.6f}")
print("------------------------------------------------")
```

OUTPUT:

```
---------------------------------------------------
  i      x(i)         f(x(i)) = e^(-x^2)
---------------------------------------------------
  0      0.0000        1.000000
  1      0.2500        0.939413
  2      0.5000        0.778801
  3      0.7500        0.569783
  4      1.0000        0.367879
---------------------------------------------------

h = (b - a) / n = (1 - 0) / 4 = 0.25

Using Simpson's 1/3 Rule:
I = (h/3) * [f(x0) + 4*(f(x1) + f(x3) + ...) + 2*(f(x2) + f(x4) + ...) + f(xn)]
I = (0.25/3) * [1.000000 + 4*(1.509196) + 2*(0.778801) + 0.367879]

------------------------------------------------
Approximate value of the integral I = 0.746855
------------------------------------------------
```


CONCLUSION : The Above Program Has Been Executed Successfully.

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

PROGRAM:

```
# Simpson's 3/8 Rule for I = ∫(0 to 1) e^(-x^2) dx with n = 3
import math
# Define the function
def f(x):
 return math.exp(-x**2)
# Given values
a = 0  # lower limit
b = 1  # upper limit
n = 3  # must be a multiple of 3 for Simpson's 3/8 rule
# Step size
h = (b - a) / n
# Generate x and f(x)
x = [a + i * h for i in range(n + 1)]
f_values = [f(xi) for xi in x]
# Display table
print("-----------------------------------------------")
print(" i   | x(i)   | f(x(i)) = e^(-x^2)")
print("-----------------------------------------------")


for i in range(n + 1):
    print(f"{i:<3} | {x[i]:<8.6f} | {f_values[i]:<10.6f}")
print("-----------------------------------------------")
```

```
# Apply Simpson's 3/8 rule formula (for n=3)
I = (3 * h / 8) * (f_values[0] + 3*f_values[1] + 3*f_values[2] + f_values[3])
# Step-by-step output
print(f"\nh = (b - a) / n = ({b}) / ({n}) = {h:.6f}")
print("\nUsing Simpson's 3/8 Rule:")
print(f"I = (3h/8) * [f(x0) + 3f(x1) + 3f(x2) + f(x3)]")
print(f"I = (3*{h:.6f}/8) * [{f_values[0]:.6f} + 3*{f_values[1]:.6f} +
3*{f_values[2]:.6f} + {f_values[3]:.6f}]")


# Final result
print("\n--------------------------------------------------")
print(f"Approximate value of the integral I = {I:.6f}")
print("--------------------------------------------------")
```

OUTPUT:

```
  i      x(i)         f(x(i)) = e^(-x^2)
------------------------------------------------
  0     0.000000      1.000000
  1     0.333333      0.894839
  2     0.666667      0.641180
  3     1.000000      0.367879
------------------------------------------------

h = (b - a)/n = (1 - 0)/3 = 0.333333

Using Simpson's 3/8 Rule:
I = (3h/8) * [f(x0) + 3f(x1) + 3f(x2) + f(x3)]
I = (3*0.333333/8) * [1.000000 + 3*0.894839 + 3*0.641180 + 0.367879]

---------------------------------------------
Approximate value of the integral I = 0.746992
---------------------------------------------
```

CONCLUSION: The Above Program Has Been Executed Successfully.

PRACTICAL NO : 7

TRANSPORTATION PROBLEM

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

PROGRAM:

```
# Northwest Corner Method - step-by-step
# Problem data (from your sheet)
# Costs matrix: rows = origins O1,O2 ; cols = destinations D1,D2,D3
costs = [
   [8, 6, 10],  # O1
   [10, 4, 9]   # O2
]
supply = [2000, 2500]  # supplies for O1, O2
demand = [1500, 2000, 1000] # demands for D1, D2, D3
# Make copies so we don't destroy originals if we want to reuse them
sup = supply.copy()
dem = demand.copy()
# Prepare an allocation matrix initialized to zeros
alloc = [[0 for _ in range(len(demand))] for _ in range(len(supply))]
print("Northwest Corner Method - step by step\n")
print("Initial supply:", supply)
print("Initial demand:", demand)

print()
i = 0  # origin index (row)
```

```
j = 0  # destination index (col)

step = 0

# Loop until all supplies and demands are satisfied

while i < len(sup) and j < len(dem):

   step += 1

   qty = min(sup[i], dem[j])

   alloc[i][j] = qty

   sup[i] -= qty

   dem[j] -= qty

   # Print step details

   print(f"Step {step}: Allocate {qty} units to cell O{i+1}, D{j+1}")

   print(f"    cost per unit = {costs[i][j]}")

   print(f"    Remaining supply for O{i+1} = {sup[i]}")

   print(f"    Remaining demand for D{j+1} = {dem[j]}\n")

   # Move to next row or column (if supply exhausted move down, if
demand exhausted move right)

   # If both become zero, move one and then the other: standard choice is
to advance column (j) after row

   if sup[i] == 0 and dem[j] == 0:

      # If both exhausted, advance (commonly advance row or column) -
advance column then row to avoid skipping

      # But we must ensure not to go out of bounds: handle carefully:


  # Advance column if possible, otherwise advance row.

      if j + 1 < len(dem):
```

```
  j += 1
      elif i + 1 < len(sup):
          i += 1
      else:
 break  # Finished
    elif sup[i] == 0:
      i += 1
    elif dem[j] == 0:
      j += 1
    # This else normally won't happen because qty = min(sup[i], dem[j])
forces one to zero
    else:
      pass
# Display final allocation matrix
    print("Final allocation matrix (rows = O1,O2 ; cols = D1,D2,D3):\n")
    header = ["  |"] + [f"  D{c+1}" for c in range(len(demand))] + ["| Supply"]
    print("".join(header))
    for r in range(len(alloc)):
      row_str = [f"O{r+1}|"] + [f"{alloc[r][c]:6d}" for c in range(len(alloc[r]))] +
[f"| {supply[r]:6d}"]
      print("".join(row_str))
    print()

    # Compute total cost
    total_cost = 0
```

```
for r in range(len(alloc)):
  for c in range(len(alloc[0])):
        total_cost += alloc[r][c] * costs[r][c]
# Print non-zero allocations
print("Allocations (non-zero):")
for r in range(len(alloc)):
    for c in range(len(alloc[0])):
        if alloc[r][c] != 0:
            print(f"O{r+1}, D{c+1} -> {alloc[r][c]} units at cost {costs[r][c]} => contribution = {alloc[r][c] * costs[r][c]}")


print(f"\nTotal transportation cost (initial NW-corner solution) = {total_cost}")
```

OUTPUT:

```
Northwest Corner Method - step by step

Initial supply: [2000, 2500]
Initial demand: [1500, 2000, 1000]

Step 1: Allocate 1500 units to cell (O1, D1)
        cost per unit = 8
        Remaining supply for O1 = 500
        Remaining demand for D1 = 0

Step 2: Allocate 500 units to cell (O1, D2)
        cost per unit = 6
        Remaining supply for O1 = 0
        Remaining demand for D2 = 1500

Step 3: Allocate 1500 units to cell (O2, D2)
        cost per unit = 4
        Remaining supply for O2 = 1000
        Remaining demand for D2 = 0

Step 4: Allocate 1000 units to cell (O2, D3)
        cost per unit = 9
        Remaining supply for O2 = 0
        Remaining demand for D3 = 0

Final allocation matrix (rows = O1,O2 ; cols = D1,D2,D3):

     D1  D2  D3  | Supply
O1    1500     500       0  |   2000
O2       0    1500    1000  |   2500

Allocations (non-zero):
  (O1, D1) -> 1500 units  at cost 8  =>  contribution = 12000
  (O1, D2) -> 500 units  at cost 6  =>  contribution = 3000
  (O2, D2) -> 1500 units  at cost 4  =>  contribution = 6000
  (O2, D3) -> 1000 units  at cost 9  =>  contribution = 9000

Total transportation cost (initial NW-corner solution) = 30000
```

CONCLUSION: The Above Program Has Been Executed Successfully.