last

May 7, 2023

1 Numerical Analysis Project

1.1 Main libraries and functions

[18]:
```
# Import required libraries
from sympy import sympify, lambdify  # For symbolic mathematics
from sympy import cot  # Cotangent function from SymPy
import math  # Built-in math library
import numpy as np  # NumPy library for numerical computing
import sympy as sp  # SymPy library for symbolic mathematics
import re  # Regular expressions library
```

[19]:
```
# Function that compare signs of two numbers
def SameSign(a, b):
    if a == 0 and b == 0:
        return True
    else:
        return (a >= 0 and b >= 0) or (a < 0 and b < 0)
```

[20]:
```
e = str(math.e)
pi = str(math.pi)
```

1.2 Methods Functions

1.2.1 1. Bisection

[29]:
```
def bisection():

    # Get equation from user
    equation_str = input("Enter an equation: ")

    # Replace constents by there value
    equation_str = re.sub(r'\b[eE]\b', e, equation_str)
    equation_str = re.sub(r'\bpi\b', pi, equation_str)

    # Handle Sec, csc and cot
    equation_str = equation_str.replace('sec','1/cos')
    equation_str = equation_str.replace('csc','1/sin')
```

```python
    equation_str = equation_str.replace('cot','1/tan')

    # Convert equation string to expression
    equation_expr = sympify(equation_str)

    # Get the variable symbol used in the equation
    var = equation_expr.free_symbols.pop()

    # Convert expression to Python function ==> F(x)
    F = lambdify(var, equation_expr)

    # Check if the equation contains log or ln
    if "log" in equation_str or "ln" in equation_str:
        # If the equation contains log or ln, set a and b to 1
        a = 1
        b = 1
        step = 1
        print("Step for ln and log must be +ve")
    else:
        # If the equation doesn't contain log or ln, ask user for step
        a = 0
        b = 0
        step = int(input("Enter 1 for +ve root or -1 for -ve root: "))

    Fxa = 0
    Fxb = 0

    # Calculate initial a and b from F(x)
    while True:
        a = b
        b += step

        Fxa = F(a)
        Fxb = F(b)

        if not SameSign(Fxa, Fxb):
            break

    # Calculate c
    n = 100
    for i in range(n):
        c = (a + b) / 2
        Fxc = F(c)

        if SameSign(Fxa, Fxc):
            a, c = c, a
        else:
```

```
        b, c = c, b

    print("The root is:", c)
```

1.2.2 2. Euler

[21]:
```
def euler():
    # Get the differential equation from the user
    equation = input("Enter your differential equation (in terms of x and y): ")

    # Replace constants by their values
    equation = re.sub(r'\b[eE]\b', '2.71828', equation)
    equation = re.sub(r'\bpi\b', '3.14159', equation)

    # Handle sec, csc, and cot
    equation = equation.replace('sec', '1/cos')
    equation = equation.replace('csc', '1/sin')
    equation = equation.replace('cot', '1/tan')

    # Define symbols and convert equation string to expression
    x, y = sp.symbols('x y')
    f = sp.sympify(equation)

    # Get initial values and step size from the user
    x0 = float(input("Enter the initial value of x: "))
    y0 = float(input("Enter the initial value of y: "))
    h = float(input("Enter the step size: "))
    x_target = float(input("Enter the target x value: "))

    # Calculate the number of iterations
    n = math.ceil((x_target - x0) / h)

    # Define the differential equation
    dydx = sp.Function('y')(x).diff(x)
    diffeq = sp.Eq(dydx, f)

    # Create a lambda function for evaluating the equation
    f_eval = sp.lambdify([x, y], f)

    # Initialize the arrays for x and y values
    x_arr = np.zeros(n + 2)
    y_arr = np.zeros(n + 2)

    # Set the initial values
    x_arr[0] = x0
    y_arr[0] = y0
```

```python
    # Apply Euler's method
    for i in range(n+1):
        x_arr[i + 1] = x_arr[i] + h
        y_arr[i + 1] = y_arr[i] + h * f_eval(x_arr[i], y_arr[i])

    # Print the results
    for i in range(n + 1):
        print("x = {:.4f}, y = {:.4f}".format(x_arr[i], y_arr[i]))
```

1.2.3 3. Modified_euler

[22]:
```python
def modified_euler():

    # Get the differential equation from the user
    equation = input("Enter your differential equation (in terms of x and y): ")

    # Replace constants by their values
    equation = re.sub(r'\b[eE]\b', '2.71828', equation)
    equation = re.sub(r'\bpi\b', '3.14159', equation)

    # Handle sec, csc, and cot
    equation = equation.replace('sec', '1/cos')
    equation = equation.replace('csc', '1/sin')
    equation = equation.replace('cot', '1/tan')

    x, y = sp.symbols('x y')

    # Convert equation string to SymPy expression
    f = sp.sympify(equation)

    # Prompt the user for initial values, step size, and target x value
    x0 = float(input("Enter the initial value of x: "))
    y0 = float(input("Enter the initial value of y: "))
    h = float(input("Enter the step size: "))
    x_target = float(input("Enter the target value of x: "))

    # Calculate the number of iterations needed
    n = math.ceil((x_target - x0) / h)

    # Define the derivative of y with respect to x
    dydx = sp.Function('y')(x).diff(x)

    # Define the differential equation
    diffeq = sp.Eq(dydx, f)

    # Create a Python function for f(x,y)
    f_eval = sp.lambdify([x, y], f)
```

```
    # Initialize the arrays for x and y values
    x_arr = np.zeros(n + 1)
    y_arr = np.zeros(n + 1)

    # Set the initial values
    x_arr[0] = x0
    y_arr[0] = y0

    # Modified Euler's method
    
↪print("----------------------------------------------------------------------")
    print("{:<10}|{:<12}|{:<12}|{:<12}".format("Xn", "Yn", "f(Xn,Yn)", "Yn+1"))
    
↪print("----------------------------------------------------------------------")
    for i in range(n):
        x_arr[i + 1] = x_arr[i] + h
        y_pred = y_arr[i] + h * f_eval(x_arr[i], y_arr[i])
        y_arr[i + 1] = y_arr[i] + 0.5 * h * (f_eval(x_arr[i], y_arr[i]) +
↪f_eval(x_arr[i + 1], y_pred))

        print("{:<10.6f}|{:<12.6f}|{:<12.6f}|{:<12.6f}".format(x_arr[i],
↪y_arr[i], f_eval(x_arr[i], y_arr[i]), y_arr[i + 1]))
        
↪print("----------------------------------------------------------------------")

    # Print the final result
    print("{:<10.6f}|{:<12.6f}|{:<12.6f}|{}".format(x_arr[n], y_arr[n],
↪f_eval(x_arr[n], y_arr[n]), "N/A"))
    
↪print("----------------------------------------------------------------------")
```

1.2.4 4. Secent

[23]:
```
def secant():

    # Get equation from user
    equation_str = input("Enter an equation: ")

    # Replace constents by there value
    equation_str = re.sub(r'\b[eE]\b', e, equation_str)
    equation_str = re.sub(r'\bpi\b', pi, equation_str)

    # Handle Sec, csc and cot
    equation_str = equation_str.replace('sec','1/cos')
    equation_str = equation_str.replace('csc','1/sin')
```

```python
    equation_str = equation_str.replace('cot','1/tan')

    # Convert equation string to expression
    equation_expr = sympify(equation_str)

    # Get the variable symbol used in the equation
    var = equation_expr.free_symbols.pop()

    # Convert expression to Python function ==> F(x)
    f = lambdify(var, equation_expr)

    # Check if the equation contains log or ln
    if "log" in equation_str or "ln" in equation_str:
        # If the equation contains log or ln, set a and b to 1
        x0 = 1
        x1 = 1
        step = 1
        print("Step for ln and log must be +ve")
    else:
        # If the equation doesn't contain log or ln, ask user for step
        x0 = 0
        x1 = 0
        step = int(input("Enter 1 for +ve root or -1 for -ve root: "))

    Fx0 = 0
    Fx1 = 0

    # Calculate initial a and b from F(x)
    while True:
        x0 = x1
        x1 += step

        Fx0 = f(x0)
        Fx1 = f(x1)

        if not SameSign(Fx0, Fx1):
            break

    n = 100  # Iterations
    x_next = 0  # next value in iteration

    if (f(x0) * f(x1) < 0):  # one of the two values must be negative

        for i in range(n):

            if x0 == 0 or x1 == 0:
                break
```

```
            # calculate the next value
            if not math.isclose(f(x1), f(x0)):
                x_next = ((x0 * f(x1) - x1 * f(x0)) / (f(x1) - f(x0)))
            else:
                break

            # update the value of interval
            x0 = x1
            x1 = x_next

        if math.isnan(x_next):
            print("Cannot find a root in the given interval")
        else:
            print(f"Root of the given equation = {x_next}")

    else:
        print("Can not find a root in the given interval")
```

1.2.5 5. Newton Forward

[24]:

```
def newton_forward():

    # Ask user which variable to calculate
    while True:
        variable = input("Do you want to calculate X or Y: ").lower()
        if variable == 'x' or variable == 'y':
            break
        else:
            print("Invalid input. Please enter 'x' or 'y'.")

    opposite_variable = 'y' if variable == 'x' else 'x'

    # Get input from user
    req = float(input(f"Enter the value of {opposite_variable} for which␣
 ↪{variable} is required: "))
    num_points = int(input("Enter the number of points: "))

    # Initialize arrays to store x and y values
    x_arr = np.zeros(num_points)
    y_arr = np.zeros((num_points, num_points))

    # Get input values from user
    print(f"Enter the values of {opposite_variable}: ")
    for i in range(num_points):
        x_arr[i] = float(input(f"Enter {opposite_variable}{i}: "))

    print(f"Enter the values of {variable}: ")
```

```
    for i in range(num_points):
        y_arr[i][0] = float(input(f"Enter {variable}{i}: "))

    # Calculate forward difference table
    for i in range(1, num_points):
        for j in range(num_points - i):
            y_arr[j][i] = (y_arr[j + 1][i - 1] - y_arr[j][i - 1]) / (x_arr[j +␣
↪i] - x_arr[j])

    # Print the forward difference table
    print("Forward Difference Table:")
    for i in range(num_points):
        print("{:.4f}".format(x_arr[i]), end="\t")
        for j in range(num_points - i):
            print("{:.4f}".format(y_arr[i][j]), end="\t")
        print()

    # Use forward difference table to calculate x or y at req
    result = y_arr[0][0]
    prod = 1
    for i in range(1, num_points):
        prod *= (req - x_arr[i - 1])
        result += (prod * y_arr[0][i])

    # Print the calculated value of x or y
    print("\nValue of {} at {} = {:.4f} is {:.4f}".format(variable,␣
↪opposite_variable.upper(), req, result))
```

1.2.6 6. Newton Backward

[25]:
```
def newton_backward():
    print("NEWTON METHOD:")
    num = int(input("Enter the number of points: "))
    print("The function is order of", num-1)
    arrx = np.zeros((num, num))
    arry = np.zeros((num, num))

    # Enter x values
    print("The values of X:-")
    for j in range(num):
        arrx[j][0] = float(input("Enter X{}: ".format(j)))

    # Enter y values
    print("The values of Y:-")
    for i in range(num):
        arry[i][0] = float(input("Enter Y{}: ".format(i)))
```

```python
    # Choose the point x or y
    while True:
        choice = input("Do you want to calculate X or Y: ")
        if choice.lower() == 'x':
            point = float(input("Enter the value of y: "))
            break
        elif choice.lower() == 'y':
            point = float(input("Enter the value of x: "))
            break

    # Construct the table of newton
    if choice.lower() == 'y':
        d = 1
        for x in range(1, num):
            for y in range(num-1, x-1, -1):
                arry[y][x] = (arry[y][x-1] - arry[y-1][x-1]) / (arrx[y][0] - 
 ↪arrx[y-d][0])
            d += 1

        print("\nBACKWARD DIFFERENCE TABLE:-")
        print("X \t Y")
        for i in range(num):
            print(arrx[i][0], end="\t")
            for j in range(i+1):
                print(arry[i][j], end="\t")
            print()

        sum = arry[num-1][0]
        for z in range(num-1, 0, -1):
            k = 1
            for j in range(z):
                k *= (point - arrx[num-1-j][0])
            sum += k * arry[num-1][z]

        print("\nThe value of P{}({}): {}".format(num-1, point, sum))

    else:
        d = 1
        for y in range(1, num):
            for x in range(num-1, y-1, -1):
                arrx[x][y] = (arrx[x][y-1] - arrx[x-1][y-1]) / (arry[x][0] - 
 ↪arry[x-d][0])
            d += 1

        print("\nBACKWARD DIFFERENCE TABLE:-")
        print("Y \t X")
        for i in range(num):
```

```
            print(arry[i][0], end="\t")
            for j in range(i+1):
                print(arrx[i][j], end="\t")
            print()

        sum = arrx[num-1][0]
        for z in range(num-1, 0, -1):
            k = 1
            for j in range(z):
                k *= (point - arry[num-1-j][0])
            sum += k * arrx[num-1][z]

        print("\nThe value of P{}({}): {}".format(num-1, point, sum))
```

1.2.7 7. Lagrange

[26]:
```
def lagrange():
    # Get the number of points and the variable value at which to calculate
    num = int(input("Enter number of parameters: "))
    variable = input("Do you want to calculate X or Y: ").lower()
    opposite_variable = 'y' if variable == 'x' else 'x'  # Calculate the␣
 ↪opposite variable
    value = float(input(f"You need {variable} at {opposite_variable.upper()} =␣
 ↪"))

    # Initialize arrays to hold x and y values
    Xpar = []
    Ypar = []

    # Input values of x and y
    print("Enter x and y values:")
    for i in range(num):
        Xpar.append(float(input(f"x{i+1} = ")))

    for i in range(num):
        Ypar.append(float(input(f"y{i+1} = ")))

    # Shift the x and y input arrays if needed
    if variable == 'x':
        Xpar, Ypar = Ypar, Xpar

    # Calculate the Lagrange polynomials (L)s
    L = []
    lpast = 1
    lmkam = 1
```

```
    for n in range(num):  # Loop to calculate L
        for z in range(num):  # Loop to calculate X
            if n != z:
                lpast = lpast * (value - Xpar[z])
                lmkam = lmkam * (Xpar[n] - Xpar[z])
        L.append(lpast / lmkam)
        lpast = 1
        lmkam = 1

    # Calculate the value of the opposite variable
    result = 0
    for i in range(num):
        result = result + (L[i] * Ypar[i])

    # Display the result
    print(f"{variable.upper()} = {result}")
```

1.2.8 8. Integration

[27]:
```
def integration():
    print(">>>>>>>>>>>>>>>>>(Numerical integration␣
↪application)<<<<<<<<<<<<<<<<<<\n\n")
    print("what type of data ?\n1.table + equation\n2.equation\nenter your␣
↪choice : ")
    InCh = int(input())

    if InCh == 2:
        equation = input("Enter the equation in terms of x and y: ")

        # Replace constents by there value
        equation = re.sub(r'\b[eE]\b', e, equation)
        equation = re.sub(r'\bpi\b', pi, equation)

        # Handle Sec, csc and cot
        equation = equation.replace('sec','1/cos')
        equation = equation.replace('csc','1/sin')
        equation = equation.replace('cot','1/tan')

        func = lambdify(['x', 'y'], sympify(equation))

        a = float(input("please enter the following data (a / b / n) :\na = "))
        b = float(input("b = "))
        n = int(input("n = "))
        h = (b - a) / n
        x = [0] * (n + 1)
        y = [0] * (n + 1)
```

```
        x[0] = a
        y[0] = func(a, y[0])
        for i in range(1, n + 1):
            x[i] = x[i - 1] + h
            y[i] = func(x[i], y[i])
        print("x", end="")
        for i in range(0, n + 1):
            print("  ", x[i], end="")
        print("\ny", end="")
        for i in range(0, n + 1):
            print("  ", y[i], end="")
        sum1 = y[0] + y[n]
    else:
        n = int(input("enter number of parameters :"))
        x = [0] * n
        y = [0] * n
        print("enter parameters of X & Y :")
        print("X parameters :")
        for i in range(0, n):
            x[i] = float(input("X" + str(i + 1) + "= "))
        print("Y parameters :")
        for i in range(0, n):
            y[i] = float(input("Y" + str(i + 1) + "= "))
        equation = input("Enter the equation in terms of x and y: ")
        func = lambdify(['x', 'y'], sympify(equation))
        for i in range(0, n):
            y[i] = func(x[i], y[i])
        h = x[1] - x[0]
        sum1 = y[0] + y[n-1]
        print("------------------------------------------------------")
        print("h = ", h)

    print("\nwhich rule you want :\n1. Trapezoidal\n2. Simpsons\n3. simpsons (3/8)\n")
    while True:
        ch = int(input("choose from 1 to 3 :"))
        if ch in range(1, 4):
            break

    sum2 = 0
    sum3 = 0

    if InCh == 1:
        n -= 1

    if ch == 1:
        for i in range(1, n):
```

```
            sum2 += y[i]
        result = (h / 2) * (sum1 + (2 * sum2))
    elif ch == 2:
        for i in range(1, n):
            if i % 2 == 0:
                sum2 += y[i]
            else:
                sum3 += y[i]
        result = (h / 3) * (sum1 + (2 * sum2) + (4 * sum3))
    elif ch == 3:
        for i in range(1, n):
            if i % 3 == 0:
                sum2 += y[i]
            else:
                sum3 += y[i]
        result = (3 * h / 8) * (sum1 + (2 * sum2) + (3 * sum3))

    print(f"Result: {result}")
```

1.2.9 9. Curve Fitting

[28]:
```
from scipy.optimize import curve_fit

def curve_fitting():
    equation_type = input("Enter the type of equation (polynomial/other): ")

    if equation_type == "polynomial":
        degree = int(input("Enter the degree of the polynomial: "))
        polynomial_func = create_polynomial_function(degree)

        x_str = input("Enter the x values (comma-separated): ")
        y_str = input("Enter the y values (comma-separated): ")

        # Convert the input strings to arrays
        x = np.array([float(val) for val in x_str.split(",")])
        y = np.array([float(val) for val in y_str.split(",")])

        x_sym = sp.symbols('x')
        polynomial_expr = sum(sp.sympify(coeff) * x_sym ** i for i, coeff in␣
↪enumerate(polynomial_func))
        f = sp.lambdify((x_sym, *polynomial_func), polynomial_expr)
        initial_guess = [1] * (degree + 1)  # Initial guess for the coefficients
        coeffs, _ = curve_fit(f, x, y, p0=initial_guess)

        if len(polynomial_func) == len(coeffs):
            # Update the 'polynomial_func' list with string representations of␣
↪the optimized coefficients
```

```
            polynomial_func = [str(coeff) for coeff in coeffs]

            # Print the coefficients alongside their corresponding values from␣
↪'coeffs'
            for i in range(len(polynomial_func)):
                coefficient = "{:.4f}".format(float(polynomial_func[i]))
                print(f"Coefficient for x^{i}: {coefficient}")
        else:
            print("Error: The 'polynomial_func' and 'coeffs' lists have␣
↪different lengths.")

    else:
        function_str = input("Enter the function to fit (use 'x', 'a', and 'b'␣
↪as variables): ")
        function_str = function_str.replace("^", "**")
        function_str = re.sub(r'\b[eE]\b', '2.756', function_str)

        x_str = input("Enter the x values (comma-separated): ")
        y_str = input("Enter the y values (comma-separated): ")

        # Convert the input strings to arrays
        x = np.array([float(val) for val in x_str.split(",")])
        y_str = y_str.replace("e", "2.756")
        y_str = y_str.replace("^", "**")
        y = np.array([eval(val) for val in y_str.split(",")])

        my_func = lambda x, a, b: eval(function_str)
        coeffs, _ = curve_fit(my_func, x, y)

        # Print the coefficients
        for i, coeff in enumerate(coeffs):
            print(f"Coefficient {i}: {coeff}")

# Helper function to create a polynomial function based on user input
def create_polynomial_function(degree):
    coefficients = []
    for i in range(degree + 1):
        coefficient = input(f"Enter coefficient for x^{i}: ")
        coefficients.append(coefficient)
    return coefficients
```

1.3 Ask user to choose required method

[15]:
```
print("Choose a root-finding method:")
print("1. Bisection method")
print("2. Euler method")
```

```python
print("3. Modified Euler method")
print("4. Secant")
print("5. Newton Forward")
print("6. Newton Backward")
print("7. Lagrange")
print("8. Integration")
print("9. Curve Fitting")
method = int(input("Enter method number: "))
```

```
Choose a root-finding method:
1. Bisection method
2. Euler method
3. Modified Euler method
4. Secant
5. Newton Forward
6. Newton Backward
7. Lagrange
8. Integration
9. Curve Fitting
Enter method number: 1
```

1.4 execute the chosen method by the user

[17]:
```python
if method == 1:
    bisection()
elif method == 2:
    euler()
elif method == 3:
    modified_euler()
elif method == 4:
    secant()
elif method == 5:
    newton_forward()
elif method == 6:
    newton_backward()
elif method == 7:
    lagrange()
elif method == 8:
    integration()
elif method == 9:
    curve_fitting()
else:
    print("Invalid method number.")
```

```
Enter an equation: x^2-x-3
Enter 1 for +ve root or -1 for -ve root: 1
The root is: 2.302775637731995
```

```
[ ]:
```