CS 3010 Programming Assignment 4 - Code Upload

CS 3010.02-1 Numerical Methods

By Nicholas Magtangob

Due : 4 / 27 / 25

Introduction

Like with every other assignment, our code is uploaded in the included zip file, but just in case it doesn't work, we will feature the code here. Another thing to note is that this project differs from the previous 3 as we switched to Python for this project to allow for the use of the Sympy library.

Additionally, the entire project is available on GitHub with the following link:
https://github.com/NiccyNet1989/CS-3010---Programming-Project-4

*If the code is not working, please check "readme.txt" included in the project submit, as it outlines the environment the program is intended to run on as well as some library dependencies.

Here is the code from the main .py file:

```python
from sympy import symbols, expand, simplify, nsimplify, Float
from sympy.parsing.sympy_parser import (
    parse_expr,
    standard_transformations,
    implicit_multiplication,
    convert_xor,
    rationalize,
)


# First create a function which reads in the contents of the txt file
def read_file(path):
    try:
        with open(path, 'r') as file:
            contents = file.read()
            contents = contents.split()
            contents = [float(element) for element in contents]
            return contents

    except FileNotFoundError:
        print("Error: File was not found")
        return []
    except Exception as e:
        print("Error: An unexpected exception occured")
        return []


def printPyramid(pyramid, pyramidName, indent):
    order = 1
    for array in pyramid:
        print(pyramidName + " for order " + str(order) + " : ", end="")
        for element in array:
            print(str(element) + "     ", end="")

        print()
        order += 1

    if (indent):
        print()


# Press the green button in the gutter to run the script.
if __name__ == '__main__':
    # Use function to read in file contents
    # Change the string in the line below to read in a different text file
    data = read_file("input2.txt")

    # for index in range(len(data)):
    #     print(data[index])

    # Now split input file into two arrays
    xValuesInitial = data[:(len(data) // 2)]
    yValuesInitial = data[(len(data) // 2):]

    # print(xValuesInitial)
    # print(yValuesInitial)

    # Begin calculating pyramid
    # First, find denominators and numerators
    denominatorPyramid = []
    order = 1
    index = 0

    while len(xValuesInitial) - order != 0:
        # Create a temporary array to add to the denominatorPyramid
        denominators = [0] * (len(xValuesInitial) - order)

        # Initial case when there is no previous array to copy from
        if order == 1:
            for i in range(len(denominators)):
                denominators[i] = xValuesInitial[index + order] -
xValuesInitial[index]
                index = index + 1
            denominatorPyramid.append(denominators)
            order = order + 1
        else:
            # Now calculate cases past initial case
            index = 0
            for i in range(len(denominators)):
                denominators[i] = (xValuesInitial[index + order] -
xValuesInitial[index])
```

```
                index = index + 1
            denominatorPyramid.append(denominators)
            order = order + 1

    # Now, find numerator pyramid
    # Divided difference pyramid must be calculated in parallel
    # This is because each next order of numerators is dependent on the
previous order of divided differences
    order = 1
    numeratorPyramid = []
    dividedDifferencePyramid = []

    while len(yValuesInitial) - order != 0:
        # Create temporary array for numerators and divided differences at
given order
        numerators = [0] * (len(yValuesInitial) - order)
        dividedDifferences = [0] * (len(yValuesInitial) - order)

        # Handle initial case when there is no previous order
        if (order == 1):
            for i in range(len(numerators)):
                numerators[i] = yValuesInitial[i + 1] - yValuesInitial[i]
            numeratorPyramid.append(numerators)

            for i in range(len(dividedDifferences)):
                dividedDifferences[i] = numeratorPyramid[0][i] /
denominatorPyramid[0][i]
            dividedDifferencePyramid.append(dividedDifferences)

            order = order + 1

        else:  # Now handle cases past initial
            for i in range(len(numerators)):
                numerators[i] = dividedDifferencePyramid[order - 2][i + 1] -
dividedDifferencePyramid[order - 2][i]
            numeratorPyramid.append(numerators)

            for i in range(len(dividedDifferences)):
                dividedDifferences[i] = numeratorPyramid[order - 1][i] /
denominatorPyramid[order - 1][i]
            dividedDifferencePyramid.append(dividedDifferences)

            order = order + 1

    for arrayIndex in range(len(dividedDifferencePyramid)):
        for elementIndex in range(len(dividedDifferencePyramid[arrayIndex])):
            dividedDifferencePyramid[arrayIndex][elementIndex] = round(
                float(dividedDifferencePyramid[arrayIndex][elementIndex]), 3)
    # printPyramid(denominatorPyramid, "Denominators", True)
    # printPyramid(numeratorPyramid, "Numerators", True)
    printPyramid(dividedDifferencePyramid, "Divided Differences", True)
    print()

    # Begin printing out expanded formula using divided difference table
    # Start by getting coefficients along diagonal of divided difference pyramid
    diagonalCoefficients = []

    # First index must be the diagonal element at order 0, which wasn't originally added to dividedDifferencesPyramid
    # Thus, it is added manually
    diagonalCoefficients.append(yValuesInitial[0])

    # Now we can add the diagonal elements, which are the first element of each array in dividedDifferencePyramid
    for array in dividedDifferencePyramid:
        diagonalCoefficients.append(array[0])

    # One last thing, we need the "x - number" values in the equation
    # These would just be "(x - xInitialValues[number])" as a string
    xTerms = []
    for xValue in xValuesInitial:
        xTerms.append("x-" + str(xValue))
    # Last element is unused, by nature of divided difference equation expansion
    del xTerms[len(xTerms) - 1]

    # print(diagonalCoefficients)
    # print(xTerms)

    polynomial = ""


    def wrapInParentheses(value):
        return "(" + value + ")"


    for index in range(len(diagonalCoefficients)):
        if index == 0:
            polynomial = polynomial + wrapInParentheses(str(diagonalCoefficients[index]))
        else:
            term = ""
            term = term + wrapInParentheses(str(diagonalCoefficients[index]))

            for integer in range(0, index):
                term = term + wrapInParentheses(str(xTerms[integer]))
            polynomial = polynomial + "+" + term

    # Polynomial can now be reasonably displayed
    print("Polynomial using Divided Difference Table:\n" + polynomial)

    # For the final part of the lab, we will be using the Python library Sympy
to parse the equation
    print("\n")


    def round_near_numbers(expr, tolerance=1e-6):
        def _round_if_close(x):
            # If x is a float and very close to an integer, round it
            if isinstance(x, Float):
                rounded = round(float(x))
                # Check if value should be rounded
                if abs(x - rounded) < tolerance:
                    return rounded
            return x

        # Apply rounding to all other numerical terms in the expression
        return expr.replace(
            lambda t: t.is_number,
            lambda t: _round_if_close(t)
        )


    def simplify_equation(equation_str, tolerance=1e-6):
        transformations = standard_transformations + (implicit_multiplication,)
        expr = parse_expr(equation_str, transformations=transformations)

        # Expand and simplify first
        expanded_expr = expand(expr)
        simplified_expr = simplify(expanded_expr)

        # Round numbers very close to integers
        rounded_expr = round_near_numbers(simplified_expr, tolerance)

        return rounded_expr


    transformations = standard_transformations + (implicit_multiplication,
convert_xor, rationalize)
    simplifiedPolynomial = simplify_equation(polynomial)

    print("Simplified Polynomial:\n" + str(simplifiedPolynomial))
```