ECSE 543: Numerical Methods

Assignment 3 Report

Wenjie Wei
260685967

November 26, 2018

Contents

Introduction

This assignment explored the use of linear interpolations and other mathematical methods. The programs are programmed and compiled using Python 3.6, and the plots are generated using package matlibplot. Listing 1 shows the implementations of polynomials including their possible maneuvers. The object classes included in this file will be used for the interpolations.

1 Linear Interpolation of BH Points

1.a Lagrange Full Domain Interpolation of First Six-Point Set

Listing 2 shows the implementation of various interpolation methods. For the first six points, the Lagrange interpolation shows an interpolated polynomial

$$\begin{aligned}B(h) =& 9.275 \times 10^{-12} h^5 - 5.951 \times 10^{-9} h^4 \\ &+ 1.469 \times 10^{-6} h^3 - 1.849 \times 10^{-4} h^2 \\ &+ 1.603 \times 10^{-2} h\end{aligned}$$

whose plot is shown in Figure 1.

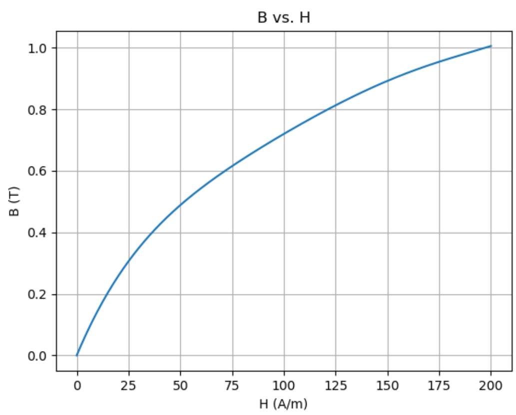

Figure 1: Interpolation of the First Six Data Points

From the figure, the interpolation has returned a plot with a **plausible** result over this range.

1.b Lagrange Full Domain Interpolation of the Second Six-Point Set

Select a second data point set, the Lagrange interpolation returned a polynomial of

$$\begin{aligned}B(h) =& 7.467 \times 10^{-19} h^5 - 3.505 \times 10^{-14} h^4 \\ &+ 5.3 \times 10^{-10} h^3 - 2.864 \times 10^{-6} h^2 \\ &+ 3.804 \times 10^{-3} h\end{aligned}$$

whose plot is shown in Figure 2.

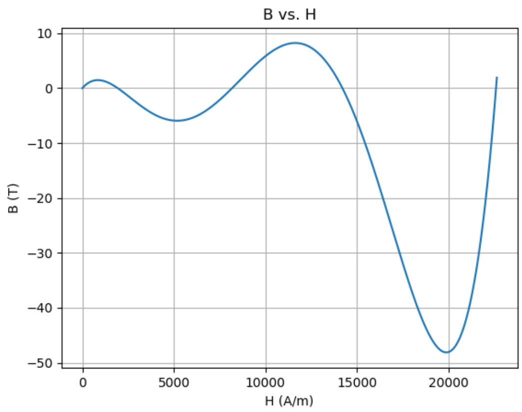

Figure 2: Interpolation of the Second Six Data Points

From this plot, we can see that the interpolation using the second set of data points is **not plausible** as the graph fluctuates violently as the value of B goes to negative at some ranges.

1.c Cubit Hermite Polynomial Interpolation

1.d Nonlinear Equation of the Magnetic Circuit

Consider the magnetic circuit shown in Figure 3.

The Magnetomotive force (MMF) can be calculated by Equation 1,

$$M = (R_g + R_c)\psi \tag{1}$$

where R_g and R_c are the reluctance of the air gap and the coil, respectively. Plug in the variables from the problem, we can transform Equation 1 to

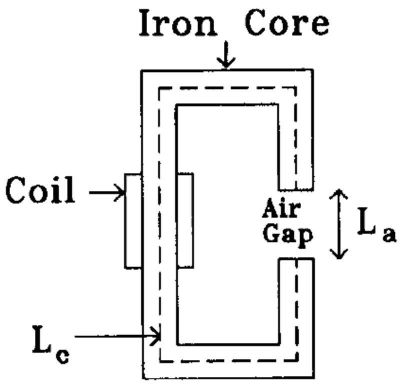

Figure 3: The Magnetic Circuit Discussed About

the equation as follows:

$$M = (\frac{l_g}{\mu_0 A} + \frac{l_c}{\mu A})\psi$$
$$NI = (\frac{l_g}{\mu_0 A} + \frac{l_c H(\psi)}{AB})\psi$$
$$NI = (\frac{l_g}{\mu_0 A} + \frac{l_c H(\psi)}{\psi})\psi$$

Simplify the equation by bringing NI to the right of the equation, and the equation will be the final formula of $f(\psi)$, as is shown in Equation

$$f(\psi) = \frac{l_g \psi}{\mu_0 A} + l_c H(\psi) - NI = 0 \tag{2}$$

Plug in the numbers, we can finalize the equation by calculating all the coefficients of the polynomial, shown in Equation 3.

$$f(\psi) = 3.979 \times 10^7 \psi + 0.3H(\psi) - 8000 \tag{3}$$

A Code Listings

Listing 1: Polynomials Implementation (`polynomial.py`).

```python
import math


class Polynomial(object):
    def __init__(self, coeff):
        self._coeff = coeff
        self._order = len(coeff) - 1

    def calculate(self, value):
        """
        This function calculates the result of the polynomial.

        :param value: value of x
        :return: value of y
        """
        result = 0
        for i in range(len(self._coeff)):
            result += self._coeff[i] * math.pow(value, i)

        return result

    def derive(self, der_order):
        result_coeff = []
        counter = 0

        for i in range(1, len(self._coeff)):
            result_coeff.append(i * self[i])
        result_poly = Polynomial(result_coeff)
        counter += 1

        if counter < der_order:
            return result_poly.derive(der_order - 1)
        else:
            return result_poly

    def __getitem__(self, item):
        return self._coeff[item]

    def __add__(self, other):
        result_coeff = []

        if isinstance(other, int):
            result_coeff = self._coeff
            result_coeff[0] += other
        else:
            self_has_higher_order = (max(self.order, other.order) == self.order)

            if self_has_higher_order:
                big_coeff = self.coefficient
                small_coeff = other.coefficient
            else:
                big_coeff = other.coefficient
                small_coeff = self.coefficient

            for i in range(len(small_coeff), len(big_coeff)):
                small_coeff.append(0)

            for i in range(len(big_coeff)):
                result_coeff.append(small_coeff[i] + big_coeff[i])

        return Polynomial(result_coeff)

    def __sub__(self, other):
        result_coeff = []
        if isinstance(other, int):
```

```
            result_coeff = self._coeff
            result_coeff[0] -= other

        else:
            self_has_higher_order = (max(self.order, other.order) == self.order)

            if self_has_higher_order:
                for i in range(len(other.coefficient), len(self.coefficient)):
                    other.coefficient.append(0)
            else:
                for i in range(len(self.coefficient), len(other.coefficient)):
                    self.coefficient.append(0)

            for i in range(len(self.coefficient)):
                result_coeff.append(self.coefficient[i] - other.coefficient[i])

        return Polynomial(result_coeff)

    def __mul__(self, other):
        result_coefficients = []

        result_order = self.order + other.order

        for i in range(result_order + 1):
            coefficient = 0
            for j in range(self.order + 1):
                for k in range(other.order + 1):
                    if j + k == i:
                        coefficient += self[j] * other[k]

            result_coefficients.append(coefficient)

        return Polynomial(result_coefficients)

    def toString(self):
        print("y = ", end="")
        for i in range(self.order, 0, -1):
            if self[i] != 1 and self[i] != -1 and self[i] != 0:
                if self[i] >= 0:
                    print("+ " + str(self[i]) + "x^" + str(i), end=" ")
                else:
                    print("- " + str(-self[i]) + "x^" + str(i), end=" ")
            elif self[i] == 1:
                print("+ x^" + str(i), end=" ")
            elif self[i] == -1:
                print("- x^" + str(i), end=" ")
            else:
                pass

        if self[0] < 0:
            print("- " + str(-self[0]))
        else:
            print("+ " + str(self[0]))

    def modify_const(self, value):
        self._coeff[0] = value

    @property
    def order(self):
        return self._order

    @property
    def coefficient(self):
        return self._coeff


class LagrangePolynomial(object):
    def __init__(self, n, xr, j, xj):
        """
        Construct a Lagrange polynomial.
```

```python

        :param n: how many points are on the x axis
        :param xr: the values of x
        :param j: the position of the current x
        :param xj: the value of x at position j
        """
        self._order = n
        self._j = j
        self._xr = []
        self._xj = xj

        self._x = 0

        for i in range(len(xr)):
            self._xr.append(-xr[i])

        self._numerator = self._create_numerator()
        self._denominator = self._create_denominator(xj)

        self._polynomial = self._create_polynomial()

    def _create_numerator(self):
        """
        This method creates the list of the parameters x_r.

        :return: no return value
        """
        i = 0
        result_numerator = Polynomial([1])

        while i < self._order:
            if i == self.j:
                i += 1

            if i >= self._order:
                break

            result_numerator *= Polynomial([self._xr[i], 1])
            i += 1

        return result_numerator

    def _create_denominator(self, x):
        """
        This method calculates the numerical result of the denominator.

        :return: the value in decimal of the denominator.
        """

        return self._numerator.calculate(x)

    def _create_polynomial(self):
        """
        This method creates the general form of the lagrange polynomial.
        :return:
        """
        denom = Polynomial([1 / self._denominator])

        return denom * self._numerator

    def set_x(self, value):
        self._x = value

    @property
    def j(self):
        return self._j

    @property
    def xj(self):
        return self._xj
```

```python

    @property
    def denominator(self):
        return self._denominator

    @property
    def numerator(self):
        return self._numerator

    @property
    def poly(self):
        return self._polynomial


if __name__ == "__main__":
    coeff1 = Polynomial([2])
    coeff2 = Polynomial([4, 5, 7, 8])

    coeff2.toString()
    (coeff2 - 3).toString()
```

*Listing 2: Lagrange Interpolation Implementation (**interpolation.py**).*

```python
from polynomial import Polynomial, LagrangePolynomial


def lagrange_full_domain(xr, y, points=None):
    """
    This is the method for the lagrange full domain interpolation.
    X is the variable that varies.
    Y is the variable that varies with respect to X.

    :param X: X vector of type Matrix
    :param Y: Y vector of type Matrix
    :param points: select the range of data to be interpolated if needed.
    :return: Polynomial expression for y(x)
    """
    result_polynomial = Polynomial([0])

    if points is None:
        for j in range(len(xr)):
            xj = xr[j]
            aj = y[j]

            temp_lagrange_poly = LagrangePolynomial(len(xr), xr, j, xj)

            result_polynomial += Polynomial([aj]) * temp_lagrange_poly.poly

    else:
        pass

    return result_polynomial


def cubit_hermite(xr, y, slopes):
    result = Polynomial([0])

    for j in range(len(xr)):
        xj = xr[j]
        aj = y[j]
        bj = slopes[j]

        temp = LagrangePolynomial(len(xr), xr, j, xj).poly
        lagrange_backup = LagrangePolynomial(len(xr), xr, j, xj).poly

        # Calculate the polynomial u(x)
        temp = (temp.derive(1) * Polynomial([-xj, 1])) * Polynomial([-2])
        temp = temp + 1

```

```
        square = lagrange_backup * lagrange_backup
        uj = temp * square

        # Calculate the polynomial v(x)
        vj = Polynomial([-xj, 1]) * square

        aj_poly = Polynomial([aj])
        bj_poly = Polynomial([bj])

        result += uj * aj_poly + vj * bj_poly

    return result
```