# ECSE 543: Numerical Methods
## Assignment 3 Report

Wenjie Wei
260685967

November 24, 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

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

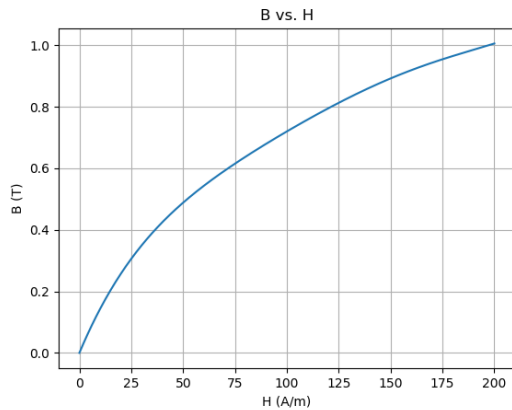whose plot is shown in Figure 1.



*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

$$B(h) = 7.467 \times 10^{-19}h^5 - 3.505 \times 10^{-14}h^4$$
$$+ 5.3 \times 10^{-10}h^3 - 2.864 \times 10^{-6}h^2$$
$$+ 3.804 \times 10^{-3}h$$
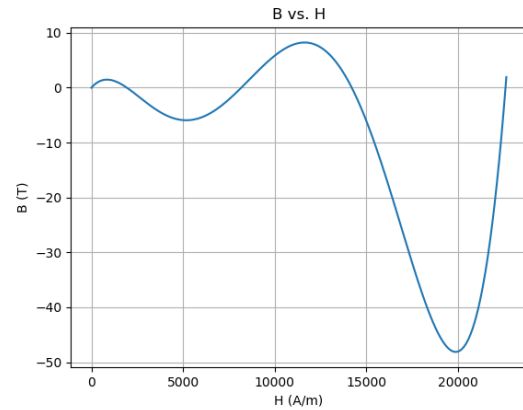
whose plot is shown in Figure 2.



*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 Hermite Polynomial Interpolation

# 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):
        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)

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

        return Polynomial(result_coeff)

    def __sub__(self, other):
        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)):
```

3

```python
66                    self.coefficient.append(0)
67
68            result_coeff = []
69            for i in range(len(self.coefficient)):
70                result_coeff.append(self.coefficient[i] - other.coefficient[i])
71
72            return Polynomial(result_coeff)
73
74        def __mul__(self, other):
75            result_coefficients = []
76
77            if isinstance(self, Polynomial) and isinstance(other, Polynomial):
78                result_order = self.order + other.order
79
80                for i in range(result_order + 1):
81                    coefficient = 0
82                    for j in range(self.order + 1):
83                        for k in range(other.order + 1):
84                            if j + k == i:
85                                coefficient += self[j] * other[k]
86
87                    result_coefficients.append(coefficient)
88            elif isinstance(self, Polynomial) and isinstance(other, int):
89                for i in range(len(self._coeff)):
90                    result_coefficients.append(other * self[i])
91            else:
92                print("The format should be polynomial * polynomial or polynomial * constant.")
93
94            return Polynomial(result_coefficients)
95
96        def toString(self):
97            print("y = ", end="")
98            for i in range(self.order, 0, -1):
99                if self[i] != 1 and self[i] != -1 and self[i] != 0:
100                    if self[i] >= 0:
101                        print("+ " + str(self[i]) + "x^" + str(i), end=" ")
102                    else:
103                        print("- " + str(-self[i]) + "x^" + str(i), end=" ")
104                elif self[i] == 1:
105                    print("+ x^" + str(i), end=" ")
106                elif self[i] == -1:
107                    print("- x^" + str(i), end=" ")
108                else:
109                    pass
110
111            if self[0] < 0:
112                print("- " + str(-self[0]))
113            else:
114                print("+ " + str(self[0]))
115
116        @property
117        def order(self):
118            return self._order
119
120        @property
121        def coefficient(self):
122            return self._coeff
123
124
125    class LagrangePolynomial(object):
126        def __init__(self, n, xr, j, xj):
127            """
128            Construct a Lagrange polynomial.
129
130            :param n: how many points are on the x axis
131            :param xr: the values of x
132            :param j: the position of the current x
133            :param xj: the value of x at position j
134            """
135            self._order = n
```

```python
136            self._j = j
137            self._xr = []
138            self._xj = xj
139
140            self._x = 0
141
142            for i in range(len(xr)):
143                self._xr.append(-xr[i])
144
145            self._numerator = self._create_numerator()
146            self._denominator = self._create_denominator(xj)
147
148        def _create_numerator(self):
149            """
150            This method creates the list of the parameters x_r.
151
152            :return: no return value
153            """
154            i = 0
155            result_numerator = Polynomial([1])
156
157            while i < self._order:
158                if i == self.j:
159                    i += 1
160
161                if i >= self._order:
162                    break
163
164                result_numerator *= Polynomial([self._xr[i], 1])
165                i += 1
166
167            return result_numerator
168
169        def _create_denominator(self, x):
170            """
171            This method calculates the numerical result of the denominator.
172
173            :return: the value in decimal of the denominator.
174            """
175
176            return self._numerator.calculate(x)
177
178        def set_x(self, value):
179            self._x = value
180
181        @property
182        def j(self):
183            return self._j
184
185        @property
186        def xj(self):
187            return self._xj
188
189        @property
190        def denominator(self):
191            return self._denominator
192
193        @property
194        def numerator(self):
195            return self._numerator
196
197
198    if __name__ == "__main__":
199        coeff1 = Polynomial([2])
200        coeff2 = Polynomial([4, 5, 7, 8])
201
202        coeff2.toString()
203        (coeff2 * 3).toString()
204        coeff2.derive(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)
            temp_poly = Polynomial([aj / temp_lagrange_poly.denominator])

            result_polynomial += temp_poly * temp_lagrange_poly.numerator

    else:
        pass

    return result_polynomial
```