7.5

Mathematical Methods

Padé Approximants

1 Introduction

Programming Task: Writing Programs A and B

The programs written for this task can be found on pages 3 and 6. From the numpy package in Python, I use the `lstsq` function as an equivalent of `mldivide` from Matlab. I first tested Program A for basic functions such as f(x) = 0 with different values of L and M. Then I carried out testing with more complex functions such as f(x) = sin(x) and confirming the $O(x^{L+M+1})$ accuracy via polynomial division of the results.

Question 1

Using the binomial expansion, we obtain,

$$f_1(x) = 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{x^3}{16}$$

with the following formula for the coefficients,

$$c_0 = 1, \quad c_1 = \frac{1}{2}, \quad c_k = \frac{(-1)^{k-1}(2k-3)!}{2^{2k-2}k!(k-2)!} \text{ for } \ k \geq 1.$$

The radius of convergence can be found via the ratio test.

$$\begin{aligned} \text{Radius} &= \lim_{k\to\infty} \left| \frac{c_k}{c_{k+1}} \right| \\ &= \lim_{k\to\infty} \left| \frac{(2k-1)(2k-2)}{4(k+1)(k-1)} \right| \\ &= 1 \end{aligned}$$

This means that the Padé approximant will only be useful within the disk of radius 1 centred at 0 in the complex plane. Also, the further from 0 you go, the more terms of the power series are required for a precise result. Therefore, approximants with a lower value of L+M+1 become much less useful in these cases.

Taking $x = 1$ in the power series, we obtain $\sum_{k=0}^{\infty} c_k$. Since this converges, the sequence of partial sums $\sum_{k=0}^{N} c_k$ converges. This convergence is illustrated in figure 1

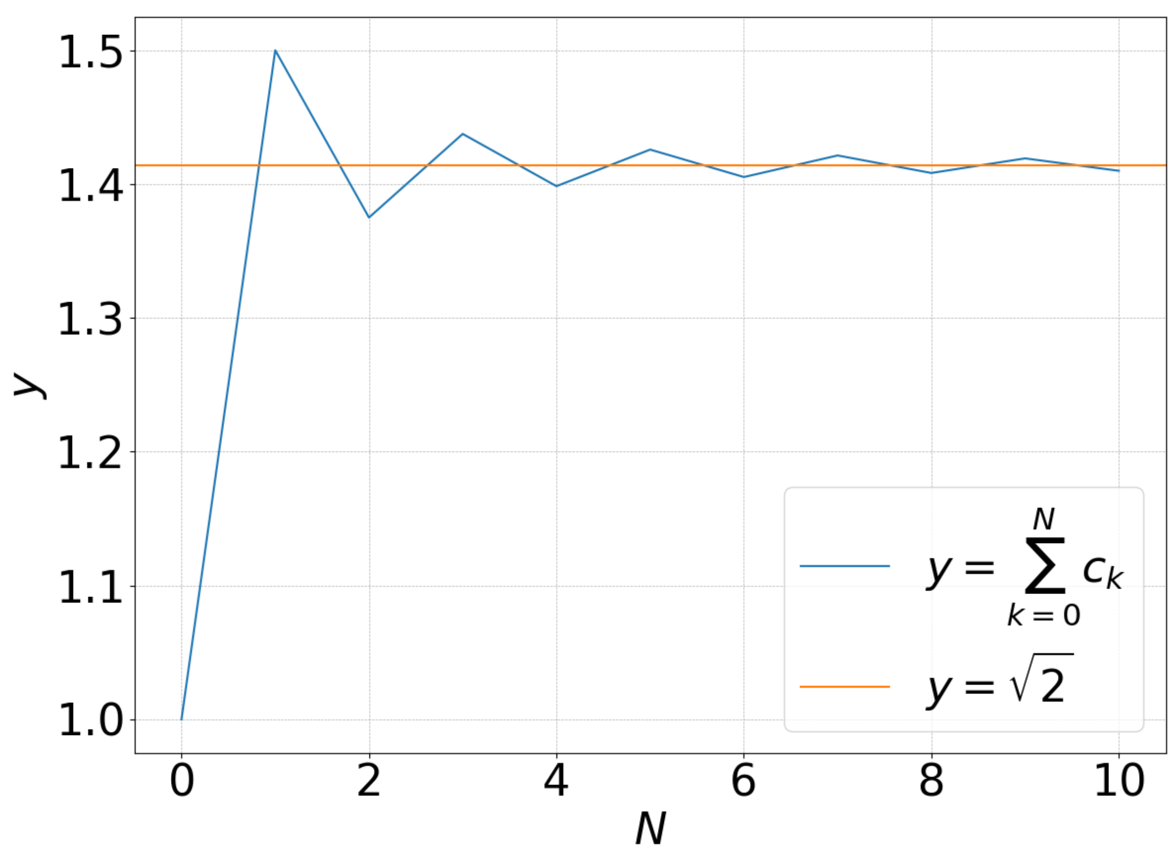

Figure 1: Line graph of partial sums

We notice that the partial sums oscillate above and below $\sqrt{2}$, with the error growing smaller as k increases.

$c_k = (1 - \frac{3}{2k})c_{k-1}$.

Question 2

Program_A.py for Programming Task

```
import numpy as np
import math


class approximator:
    def __init__(self, c_vector, L, M, x = None):
        self.c_vector = c_vector
        self.L = L
        self.M = M
        self.x = x

        self.solve5()
        # Calling this function calls self.solve4()

    def solve4(self):
        # 4 refers to name of exquation in project
        # description

        if self.M == 0:
            self.q_vector = []
            # Will be empty if M = 0
            return

        # This vector is multiplied by the matrix
        c_target = np.negative(self.c_vector[self.L + 1 :
                                              self.L + self.M + 1])
        # This is the product of the vector and matrix

        rows = []

        for i in range(self.M):
            # indexing matrix rows
            row = np.flip(self.c_vector[0 : 1 + self.L + i])
            # up to c_L
            if len(row) >= self.M:
                row = row[:self.M]
            else:
                additional_zeros = np.zeros(self.M - \
                                            len(row))
                # fill rest of row with zeros if space
                # (matrix has width M)
                row = np.append(row, additional_zeros)
            rows.append(row)

        c_matrix = np.vstack(rows)
        print(c_matrix)
        print(np.linalg.det(c_matrix))

        q_vector = np.linalg.lstsq(c_matrix, c_target)[0]
        # Need 0 index of result from lstsq function

        # NOTE: q_vector starts from q_1 unlike p_vector
        # which starts from p_1
        self.q_vector = q_vector
```

```python
    def solve5(self):
        self.solve4()
        # Need to get q_k's first to solve (5)

        p_vector = np.empty([0])
        for k in range(self.L + 1):
            sum = 0
            for s in range(1, 1 + min(k, self.M)):
                sum += self.q_vector[s - 1] * \
                    self.c_vector[k - s]
                # s - 1 since q_vector starts from q_1
            p_k = self.c_vector[k] + sum

            p_vector = np.append(p_vector, p_k)

        self.p_vector = p_vector


    def R_LM(self, x):

        numerator = 0
        for k in range(self.L + 1):
            numerator += self.p_vector[k] * x ** k

        denominator = 1
        for k in range(1, self.M + 1):
            denominator += self.q_vector[k - 1] * x ** k

        return numerator / denominator



    def evaluate_approximant(self, x_vector):
        vfunc = np.vectorize(self.R_LM)
        # vectorise function so that it can be applied to
        # a set x

        return(vfunc(x_vector))



if __name__ == '__main__':
    c_vector = np.empty([0], dtype = np.double)
    # Can then append the coefficients to this list

    c_vector = np.append(c_vector, [0, 1, 0, 1/3, 0, 2/15, 0, 17/315, 0])
    L = 3
    M = 4

    approximant = approximator(c_vector, L, M)

    # For testing:
    print('c_vector: ', c_vector)
    print('p_vector: ', approximant.p_vector)
    print('q_vector: ', approximant.q_vector)
    # Index 0 entry of q_k meaningless but want to keep
    # other indexing consistent
```

```python
print(approximant.evaluate_approximant([1, 2, 3]))
```

Program_B.py for Programming Task

```python
import numpy as np


def find_roots(polynomial):
    # polynomial should be a 1D array of coefficients
    # starting with the leading coefficient
    return np.roots(polynomial)
```