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 7 and 10. 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.

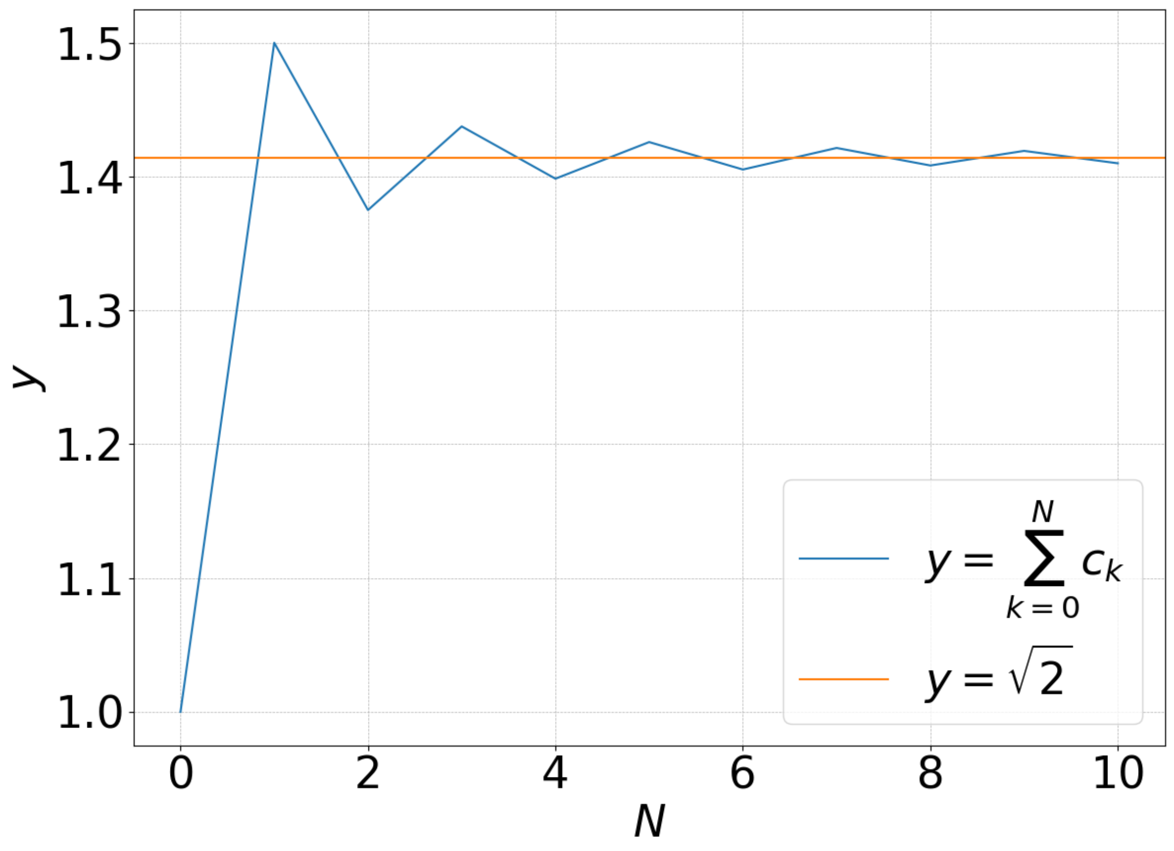


Figure 1: Line graph of partial sums

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

We use the Lagrange form of the remainder for a Taylor expansion to estimate the error of the partial sum as an estimate of $\sqrt{2}$. The remainder at $x = 1$ will be,

$$R_N(1) = \frac{f_1^{(N+1)}(\xi)}{(N+1)!}$$

for some real number ξ between 0 and 1 and following simplifications, we have,

$$|R_N(1)| = \frac{(2N-1)!}{2^{2N}(N-1)!(N+1)!}(1+\xi)^{-N+\frac{1}{2}}$$

Using the function `find_xi` in the program on page 11, we find that ξ is small and decreases as N increases. For example, for $N = 3$, $\xi = 0.230$; for $N = 10$, $\xi = 0.0681$ and for $N = 50$, $\xi = 0.0138$. Then the `print_xi_factor` also in the program on page 11 can be used to show that for $N < 80$, $0.5 \le (1+\xi)^{-N+\frac{1}{2}} \le 0.7$. Hence, we have,

$$\begin{aligned} R_N(1) &\le 0.7 \cdot \frac{(2N-1)!}{2^{2N}(N-1)!(N+1)!} \\ &= 0.7 \cdot \frac{1 \cdot 3 \cdot 5 \cdot ... \cdot (2N-1)}{2 \cdot 4 \cdot 6 \cdot ... \cdot 2N} \cdot \frac{1}{2N+2} \end{aligned} \tag{1}$$

We prove the following lemma:

$$\frac{1 \cdot 3 \cdot 5 \cdot ... \cdot (2N-1)}{2 \cdot 4 \cdot 6 \cdot ... \cdot 2N} \le \frac{1}{\sqrt{2N}}$$

Proof:
$$\begin{aligned} \left(\frac{1 \cdot 3 \cdot 5 \cdot ... \cdot (2N-1)}{2 \cdot 4 \cdot 6 \cdot ... \cdot 2N}\right)^2 &= \frac{1 \cdot 1}{2 \cdot 2} \cdot \frac{3 \cdot 3}{4 \cdot 4} ... \frac{(2N-1) \cdot (2N-1)}{2N \cdot 2N} \\ &= \frac{1 \cdot 3}{2 \cdot 2} \cdot \frac{3 \cdot 5}{4 \cdot 4} ... \frac{(2N-1)}{(2N)^2} \\ &\le \frac{2N-1}{(2N)^2} \quad \text{since} \quad (n-1)(n+1) = n^2 - 1 \le n^2 \\ &\le 1/2N \qquad \square \end{aligned}$$

Applying the lemma to (1), we obtain the following overestimate for the error as N increases:

$$|R_N(1)| \approx \frac{0.7}{(2N+2)\sqrt{2N}} \tag{2}$$

Figure 2 illustrates this as an error bound. We can see that this gives an accurate approximation of the error.

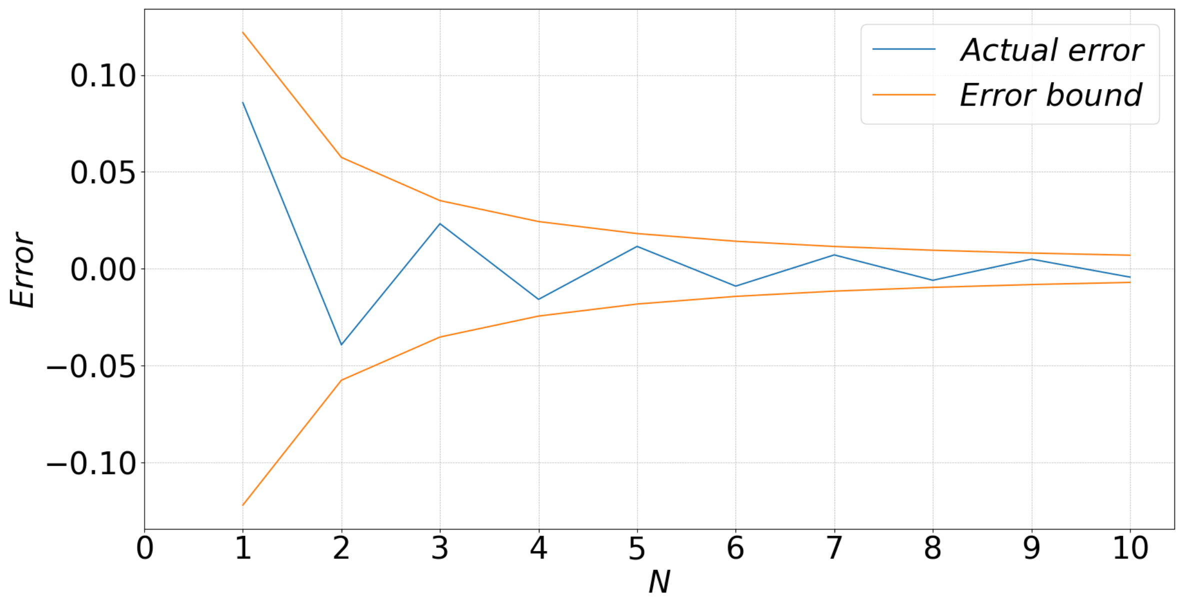


Figure 2: Actual error and estimated erro of the partial sum as an estimate of $\sqrt{2}$

We know $(1+\xi)^{-N+\frac{1}{2}}$ is relatively unchanged when N is increased by 1. Thus taking the ratio of $R_N(1)$ and $R_{N+1}(1)$ shows that incrementing N by 1 decreases the error by very close to $\frac{2N+1}{2N+4}$. Hence the larger N is, the less the percentage decrease of error is for increasing N.

Question 2

It is now much more difficult to obtain a theoretical result for the error, so we investigate $R_{L,L}(1)$ as an estimate of $\sqrt{2}$ numercially. The results in the table below show the error of the approximant for different values of L.

L	Error
0	0.41421356237309515
1	0.014213562373095234
2	0.00042045892481934466
3	1.2378941142587863e-05
4	3.644035522221145e-07
5	1.072704058913132e-08
6	3.1577518377901015e-10
7	9.29567534058151e-12
8	2.737809978725636e-13
9	7.993605777301127e-15
10	4.440892098500626e-16
11	2.220446049250313e-16
12	2.220446049250313e-16
13	6.661338147750939e-16
14	6.661338147750939e-16
15	2.220446049250313e-16

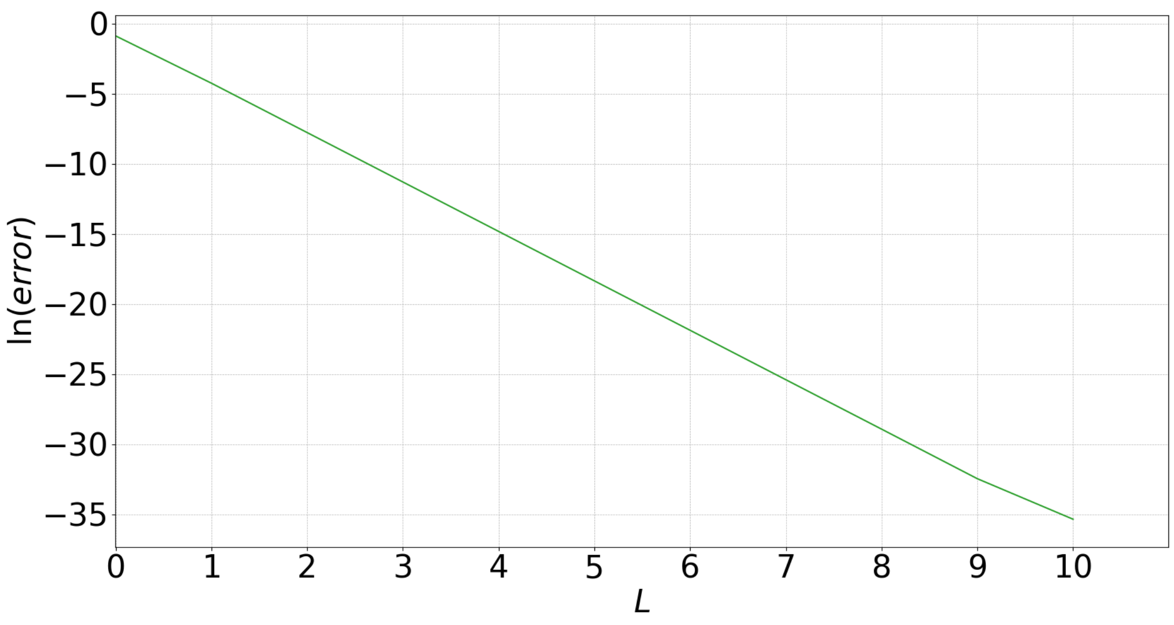


Figure 3: Plot of L against $\ln(Error)$

From figure 3, we see that for $L \leq 10$, the error decreases exponentially. We observe from the table that the minimum value of the error for $L \leq 15$ is 2.220446049250313e-16. This must be true in general since this is exactly the machine precision, i.e. it is the difference between 1.0 and the smallest 64-bit double precision floating-point value larger than 1.0. Hence, the machine precision determines the smallest error.

In cases where the matrix used to solve equation (4) is non-singular, iterative improvement will make no difference since the exact solution is found so no more improvements can be made. The determinant of the matrix in question approaches 0 as L increases. The `lstsq` function which I have used as the Python equivalent of `mldivide` finds the least-squares solution of the equation $A\mathbf{x} = \mathbf{b}$. Suppose for some L the determinant of the matrix used to solve equation (4) were 0 and let the least squares solution for the q_k be $\mathbf{y}$. Then $\mathbf{b} - A\mathbf{y}$ will be orthogonal to $A\mathbf{x}$ for any $\mathbf{x}$. Hence no more improvements can be made in this case either.

In addition, the limit on the error is cased my the machine precision, not the solution to equation (4). Thus iterative improvement would have no effect on the minimum error.

In the power series of $R_{L,L}(x)$, the first $2L+1$ terms match that of $f_1(x)$. The error of $R_{L,L}(1)$ is much less than the power series estimate of $\sqrt{2}$ for the same number of matching terms. For instance, with $L = 5$, the error of the Padé approximant is 1.07×10^{-8} while for $N = 10$ the error from the partial sum is 4.28×10^{-2}. This is surprising because using the same amount of information, a much more accurate estimate is obtained. This can be explained by the fact that as we approach the radius of convergence the error power series expansion of $f_1(x)$ diverges at a faster rate than the error term from the power series expansion of $R_{L,L}(x)$.

It is then clear that the Padé approximant should be used as an estimate of $\sqrt{2}$ to specified accuracy in all cases. The error estimation 2 shows that to have an error of 2.22×10^{-16}, which only requires $L = 11$ for the Padé approximant, you would need N to be close to 100,000. Even if you wanted more accuracy than this, that wouldn't be possible with 64-bit floats since 2.22×10^{16} is the machine precision.

Question 3

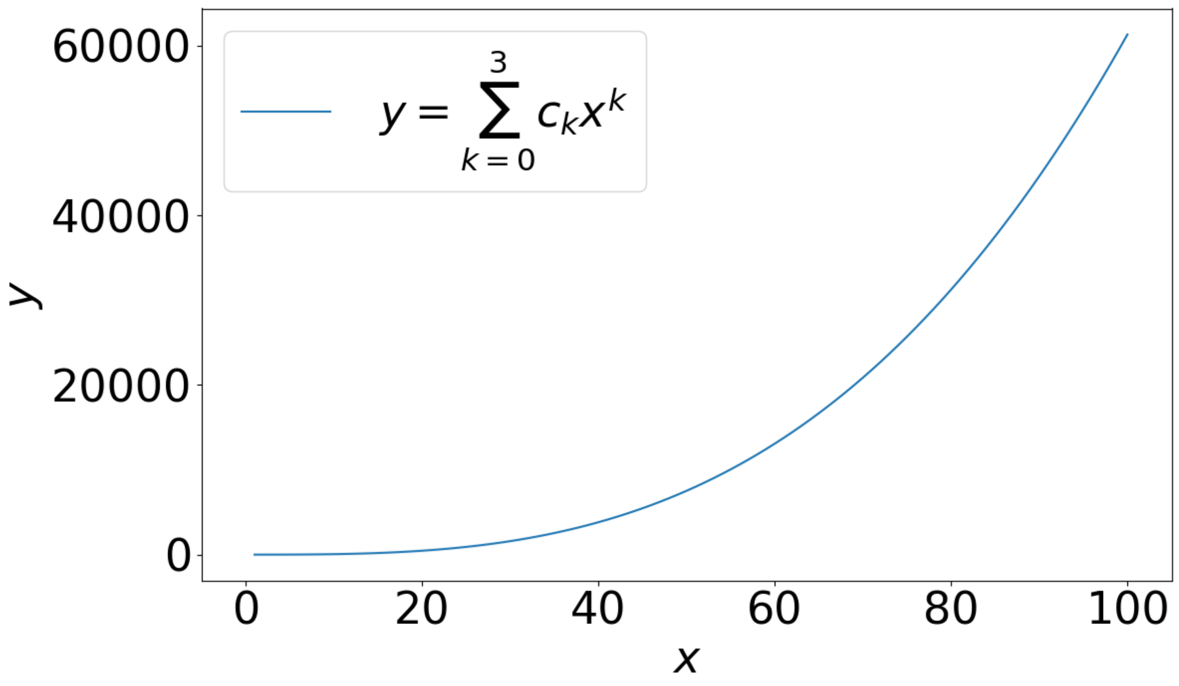
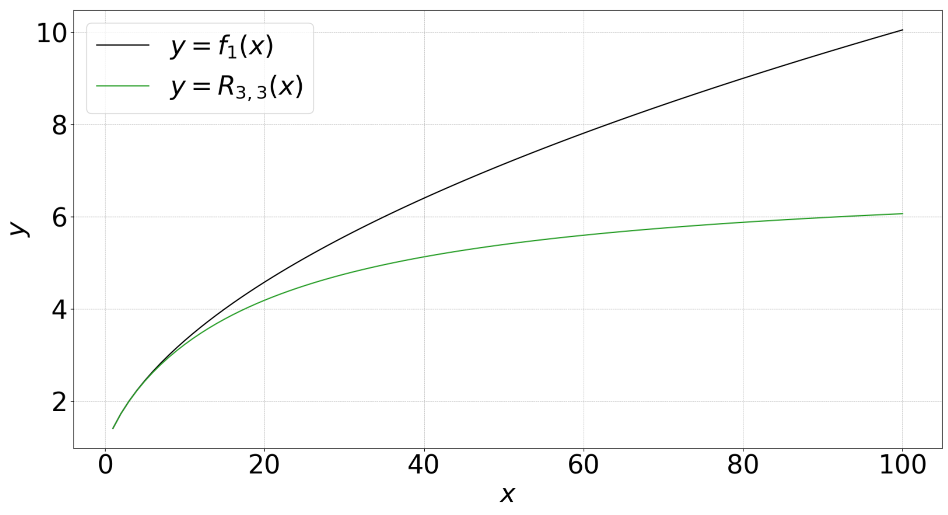


Figure 4: Plot of power series estimate of $f_1(x)$ for $N = 3$



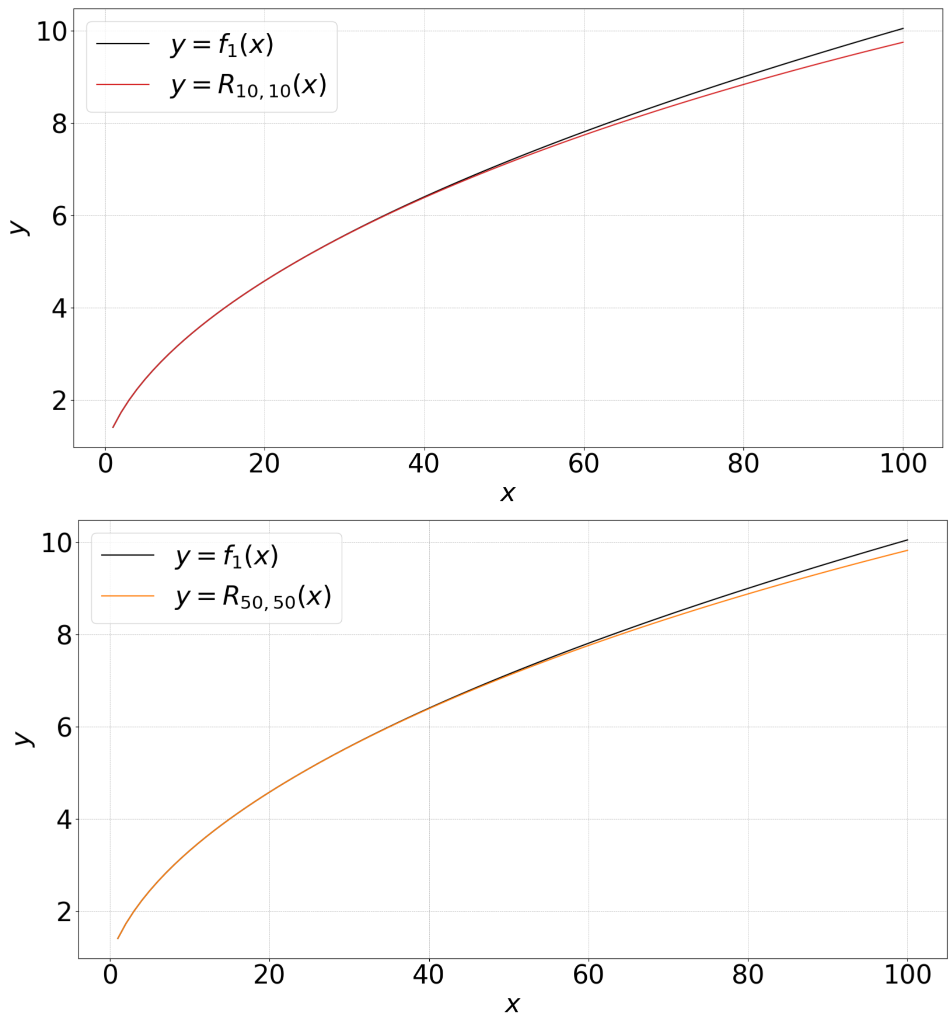


Figure 5: Plots of Padé approximant estimates of $f_1(x)$ for $L = 3, 10$ and 50

We can see from figure 4 that for $N = 3$, the estimate increases at a rate of x^N. This means it diverges from $f_1(x)$ and for larger N the estimate diverges even more quickly. This is because of the x^N term in the power series which becomes very large for $x > 1$.

In comparion, the diagonal Padé approximant with $L = 3$ stays much closer to $f_1(x)$ than the power series estimate. This is due to the fact that the Padé approximant is a fraction so its limiting behaviour as $x \to \infty$ is much more similar to $f_1(x)$ than the power series' behaviour is. However, $L = 3$ does not give a good estimate in the range $1 \leq x \leq 100$. We see that for $L = 10$ and $L = 50$, we obtain much closer estimates while the power series would only diverge further.

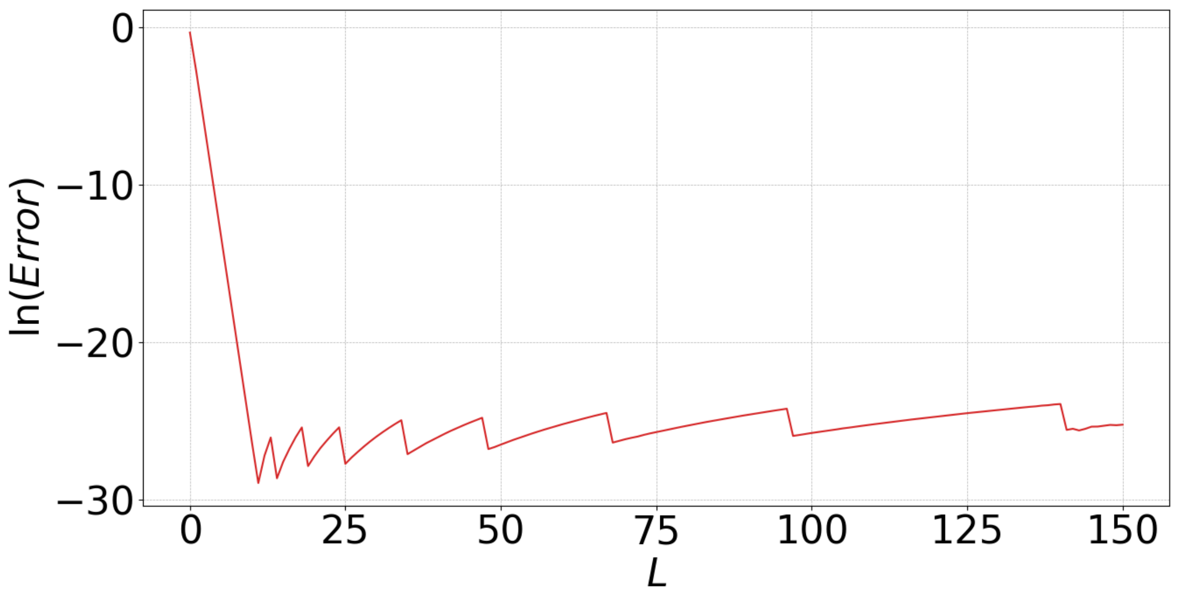


Figure 6: Plot L against $\ln(error)$ for $x = 10$

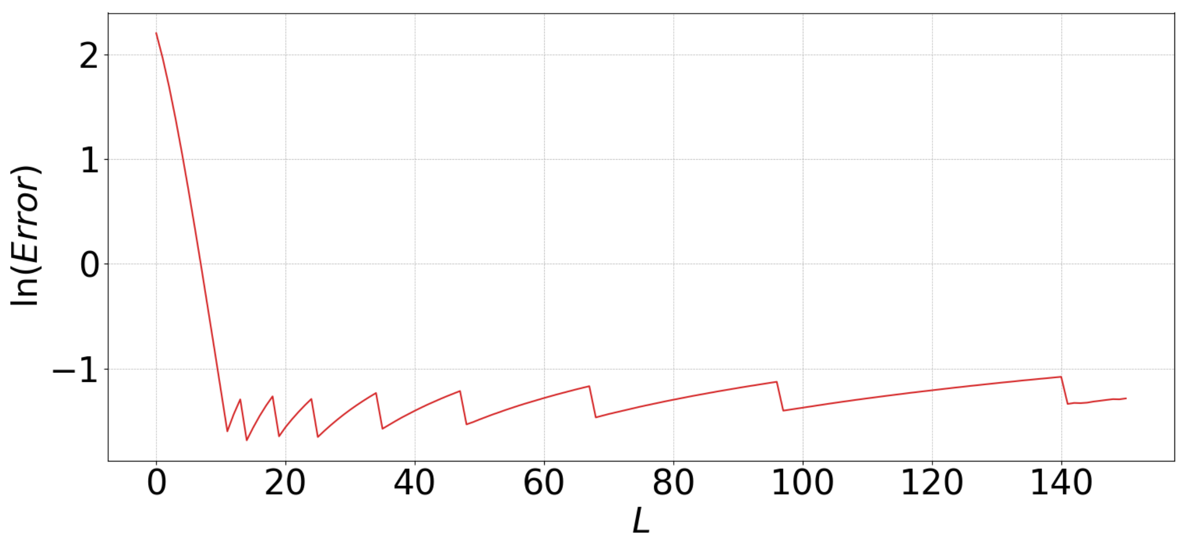


Figure 7: Plot L against $\ln(error)$ for $x = 100$

- Exponential decrease up to L = 10
- Same value of L as for x = 1. WHY?
- Apprears to be (can't be certain since haven't/can't test higher L) limit to minimum error. WHY?
- Around this minimimum value we get distinctive zig-zaggin. WHY ZIG-ZAGGING?

Overall, we can see that the Padé approximant almost converges to $f_1(x)$ for large x despite the fact that this is outside the radius of convergence of the power series. However, there is a limitation on the accuracy of the estimate you can get where increasing the value of L will not improve the result.

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,
                                       rcond = None)[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
    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
    print(approximant.evaluate_approximant([1, 2, 3]))
```

Program_B.py for Programming Task

```
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)
```

Question_1.py for Question 1

```
import numpy as np
import math
import matplotlib.pyplot as plt


def find_coefficient(k):
    if k == 0:
        return 1
    elif k == 1:
        return 1/2
    numerator = (-1)**(k-1) * math.factorial(2*k - 3)
    denominator = 2**(2*k - 2) * math.factorial(k) * \
                        math.factorial(k-2)
    return numerator / denominator


def find_partial_sum(N):
    sum = 1
    # c_0 = 1 and is always included in the sum
    for i in range(1, N + 1):
        sum += find_coefficient(i)
    return sum


def plot_partial_sum(N):
    current_sum = 1
    y_vector = [1]
    for i in range(1, N + 1):
        current_sum += find_coefficient(i)
        y_vector.append(current_sum)

    plt.rc('font', size = 32)
    plt.grid(linestyle = '--', linewidth = 0.5)
    plt.plot(y_vector, color = 'C0',
        label = '$y = \sum_{k=0}^{N}c_{k}$')
    plt.axhline(math.sqrt(2), color = 'C1',
        label = '$y = \sqrt{2}$')
    plt.legend(loc = 'best')
    plt.xlabel('$N$')
    plt.ylabel('$y$')
    plt.show()


def error_bound(N):
    return 0.69 * 1/math.sqrt(2*N) * 1/(2*N + 2)
    return 0.69 * 2**(-2*N) * (math.factorial(2*N - 1)) / \
        (math.factorial(N - 1) * math.factorial(N + 1))


def plot_error(N):
    current_sum = 1
    y_vector = []
    bound_vector = []
    x_axis = np.arange(1, N + 1, 1)
    for m in range(1, N + 1):
        current_sum += find_coefficient(m)
        y_vector.append(current_sum)
        bound_vector.append(error_bound(m))
    error_vector = np.array(y_vector) - math.sqrt(2)

    plt.rc('font', size = 32)
    plt.grid(linestyle = '--', linewidth = 0.5)
    plt.plot(x_axis, error_vector, color = 'C0',
                    label = '$Actual$ $error$')
    plt.plot(x_axis, bound_vector, color = 'C1',
                    label = '$Error$ $bound$')
    plt.plot(x_axis, np.negative(bound_vector), color = 'C1')
    plt.legend(loc = 'best')
    plt.xticks(np.append(0, x_axis))
    plt.xlabel('$N$')
    plt.ylabel('$Error$')
    plt.show()


def tabulate_error(N):
    partial_sums = []
    current_sum = 0
    for m in range(N + 1):
        current_sum += find_coefficient(m)
        partial_sums.append(current_sum)

    error_vector = np.absolute(np.array(partial_sums) \
                                        - math.sqrt(2))

    print('N          Error')
    for k in range(N + 1):
        print(k, '\t', error_vector[k])


def find_xi(N):
    power = 1/2 - N
    partial_sum = find_partial_sum(N)
    error = abs(math.sqrt(2) - partial_sum)
    coefficient = 2**(-2*N) * (math.factorial(2*N - 1)) / \
        (math.factorial(N - 1) * math.factorial(N + 1))
    xi = (error / coefficient)**(1 / power) - 1
    return xi


def print_xi_factor(N):
    for i in range(1, N + 1):
        print( (1+find_xi(i))**(1/2 - i))


if __name__ == '__main__':
    tabulate_error(11)
    #plot_error(10)
    #print( find_xi(50) )
    #print_xi_factor(90)
```