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 12 and 15. 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

NEED TO TAKE M = L + 1 SOMEWHERE IN THIS PROJECT. IN QUESTION 4? - WHAT IS THE DETERMINANT IN PROGRAM A HERE? - COULD IT AFFECT ERROR IN QUESTION 3?
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.

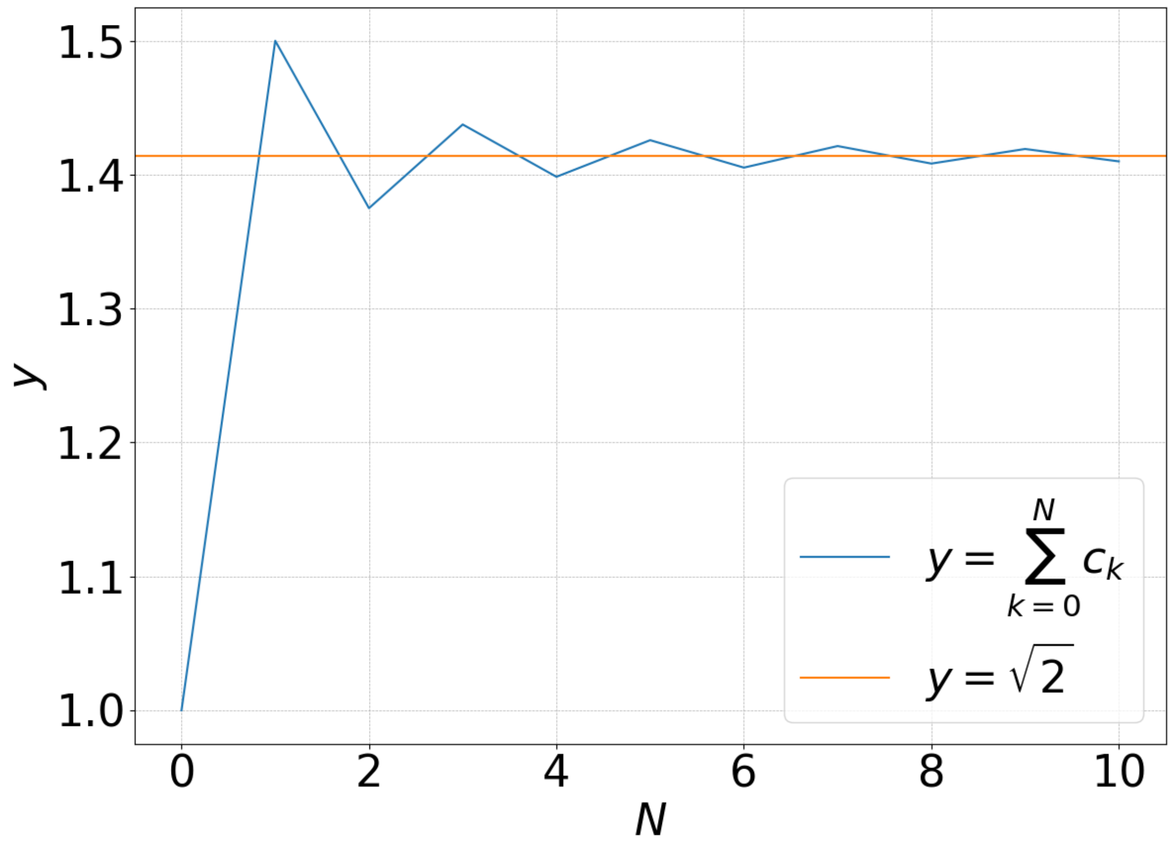


Figure 1: Line graph of partial sums

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 16, 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 16 can be used to show that for $N < 80$, $0.5 \leq (1+\xi)^{-N+\frac{1}{2}} \leq 0.7$. Hence, we have,

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

We prove the following lemma:

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

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

$$\leq 1/2N \qquad \square$$

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.

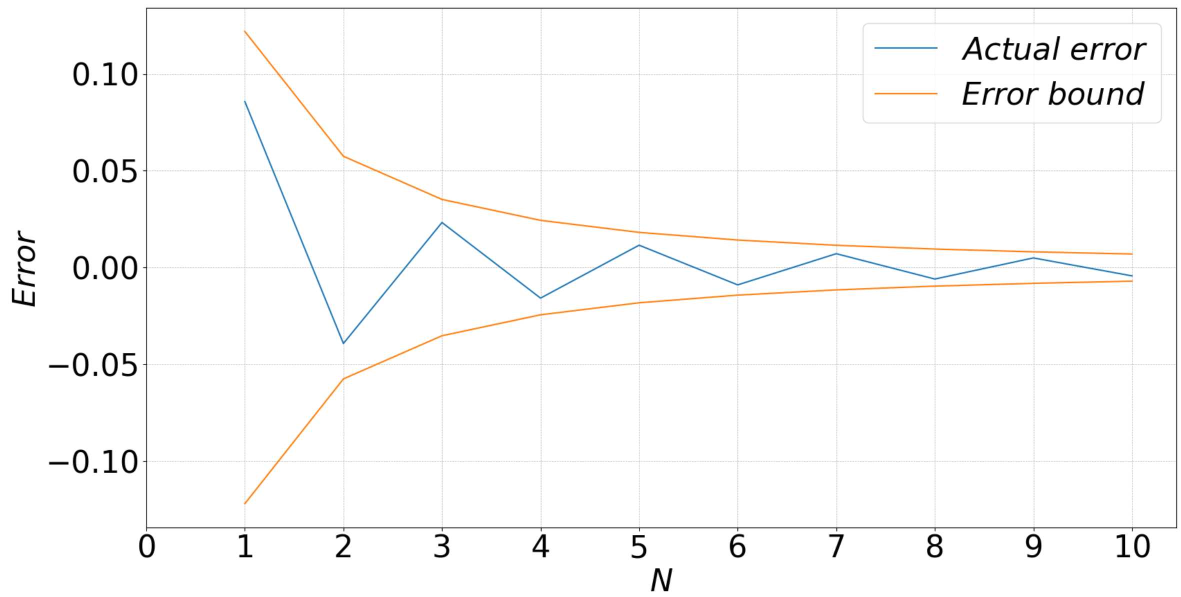


Figure 2: Actual error and estimated erro of the partial sum as an estimate of $\sqrt{2}$

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 almost exactly $\frac{2N+1}{2N+4}$. 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

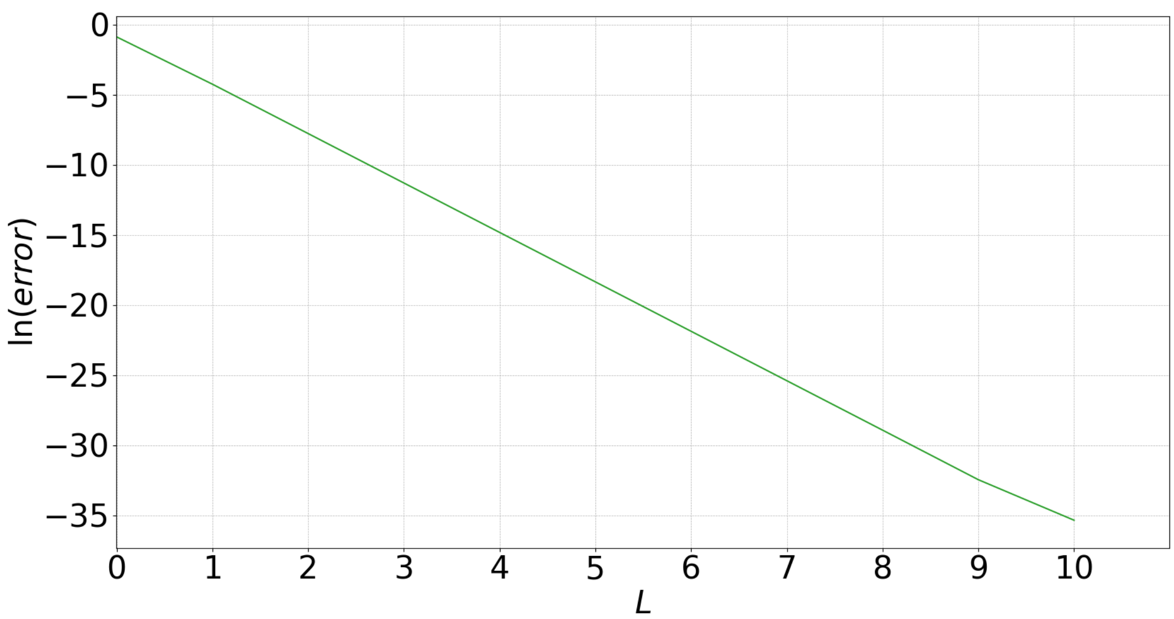


Figure 3: Plot of L against $\ln(Error)$

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 caused 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

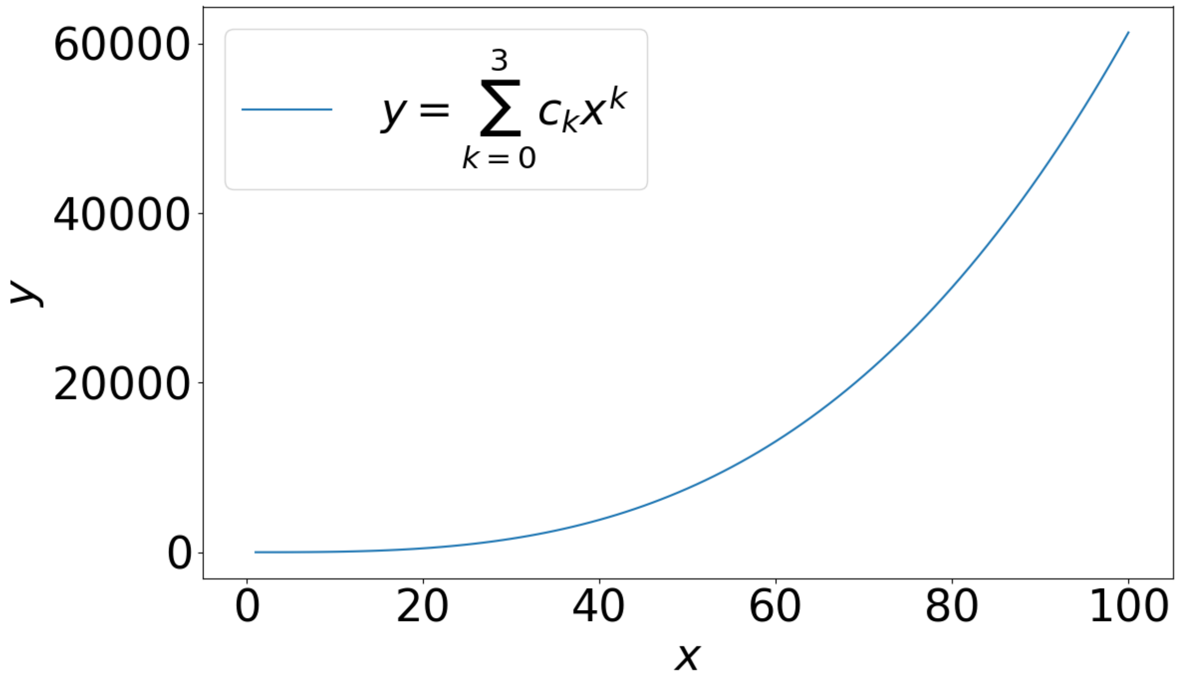
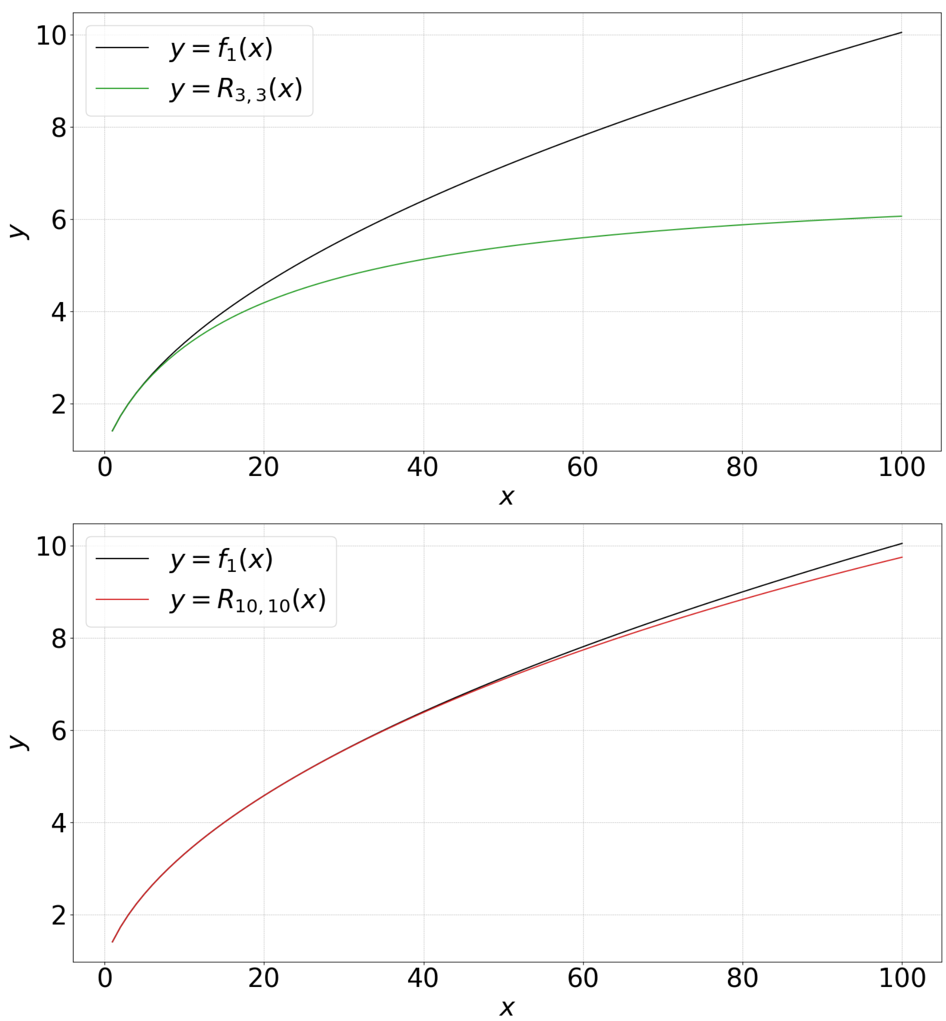


Figure 4: Plot of power series estimate of $f_1(x)$ for $N = 3$



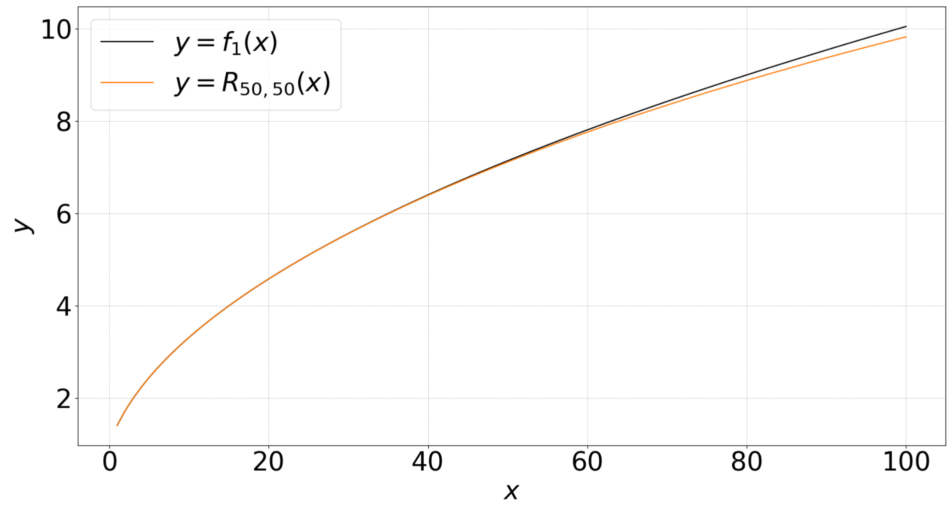


Figure 5: Plots of Padé approximant estimates of $f_1(x)$ for $L = 3, 10$ and 50

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.

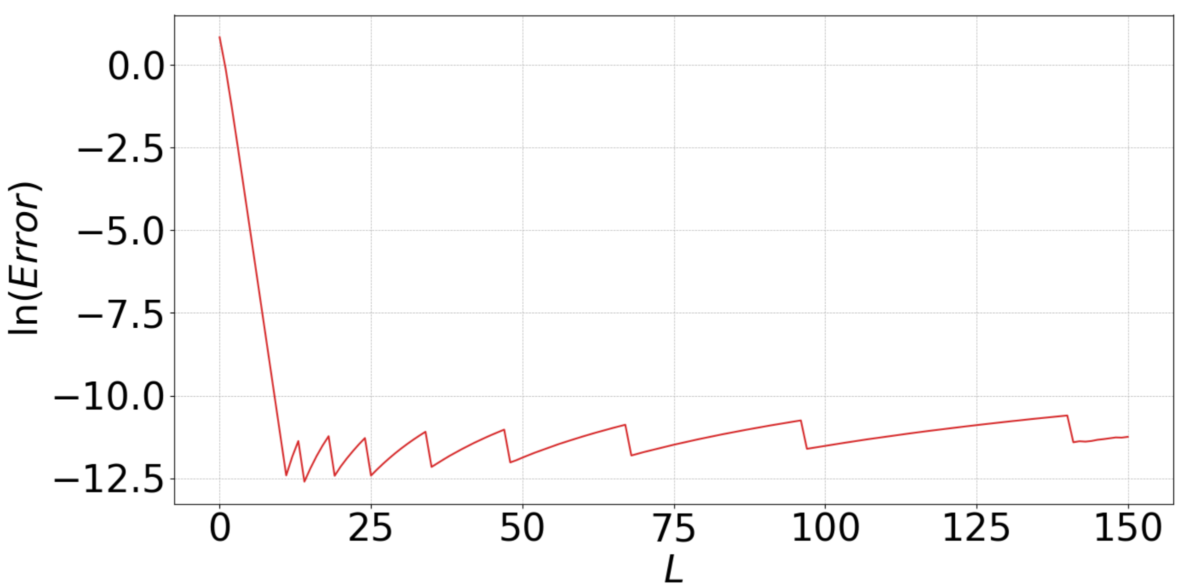


Figure 6: Plot of L against $\ln(Error)$ for $x = 10$

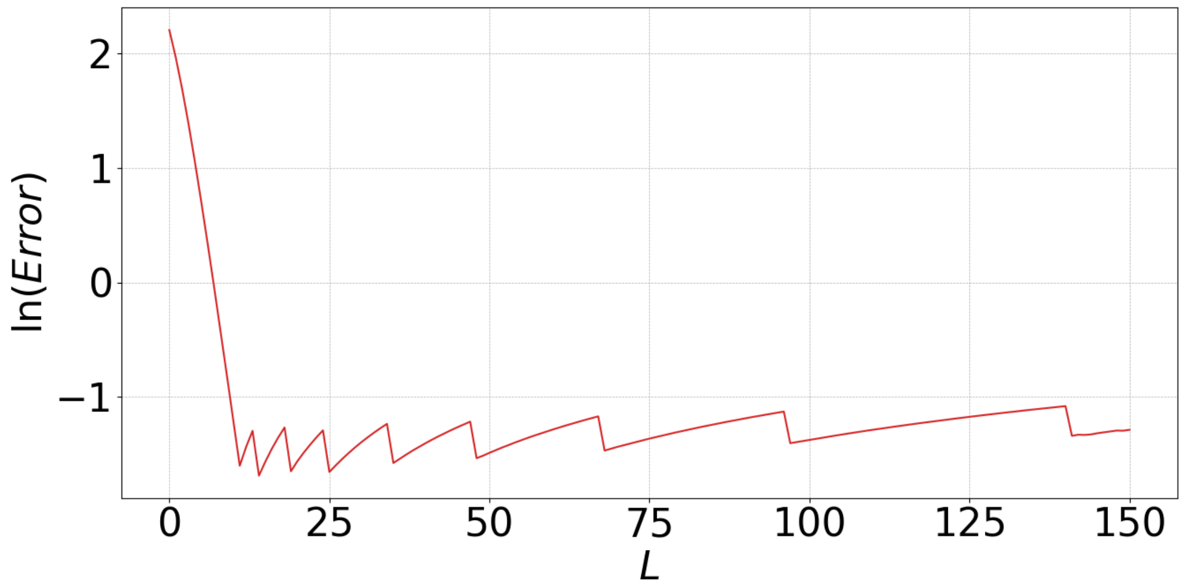


Figure 7: Plot of L against $\ln(Error)$ for $x = 100$

From Figures 6 and 7, we see that the error decreases exponentially for $L \leq 11$ since the graphs are linear for this range of L. Beyond this range, the error remains in the same area as it increases and decreases in a zig-zag pattern. The reason the error stops decreasing exponentially is that the Padé approximant calculated by Program A is not accurate for large L (i.e. the q_k and p_k calculated are slighly off). I know that the approximant is inacurate since it should give the same result as the $(2L)^{\text{th}}$ continued fraction from:

$$\sqrt{1+x} = 1 + \frac{x}{1+\sqrt{1+x}}$$
$$\implies \sqrt{1+x} = 1 + \frac{x}{2 + \frac{x}{2+\frac{x}{2+\ldots}}}$$

Truncating this fraction after the $(2L)^{\text{th}}$ two and simplifying, we obain the diagonal Padé approximant. The program `continued_fraction.py` on page 23 demonstrates that the results of Program A and what the approximant should be (using the continued fraction) are different; using $L = 15$ as an example, the error from the Padé approximant is 5.03×10^{-6} while the error should be 2.77×10^{-8}. This must be due to the limitations of the 64-bit float arithmetic used for Program A and hence explains why the error stops decreasing as L is increased.

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.

Question 4

I first present some graphs which give the error for different L using a diagonal approximant and the error for different orders using the power series. This is so that the optimal such L and order can be found for calculation across the whole range $0 \leq x \leq 20$.

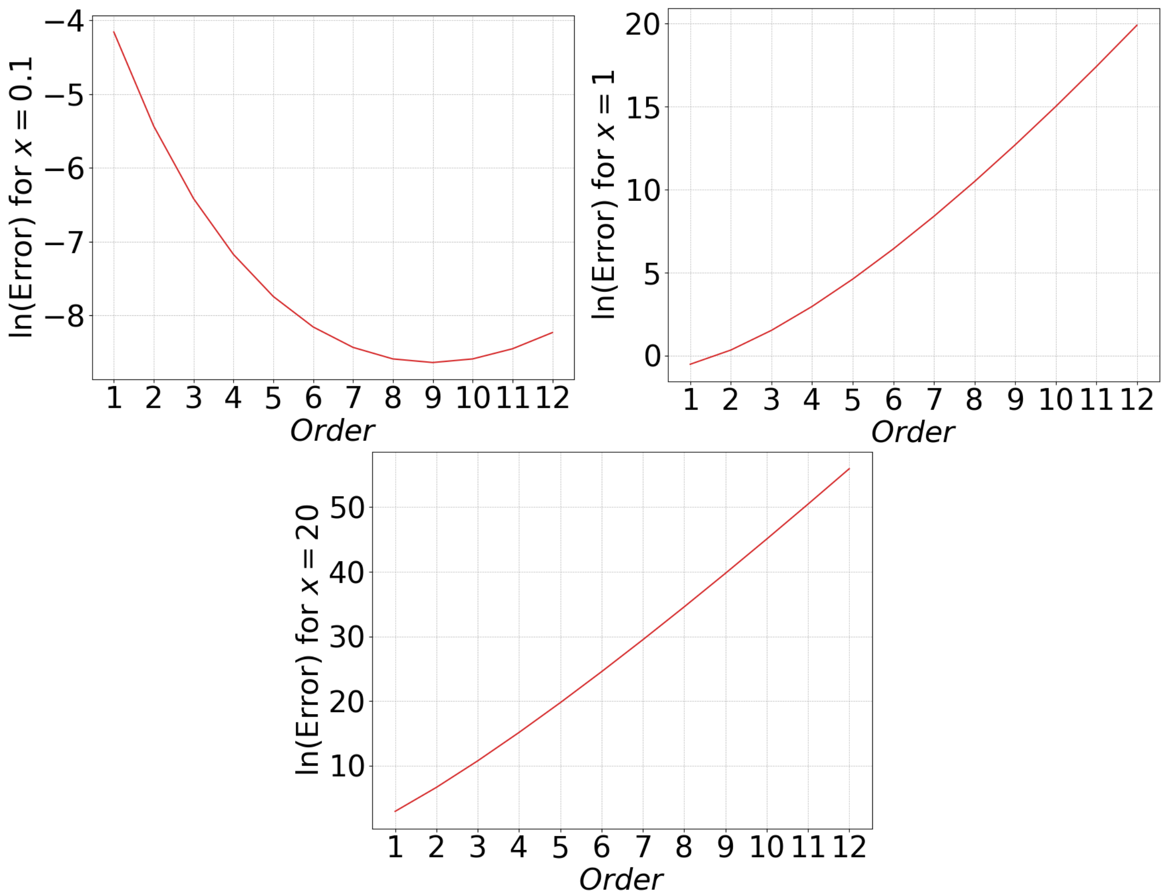


Figure 8: Plots of power series order against $\ln(Error)$ for $x = 0.1, 1$ and 20

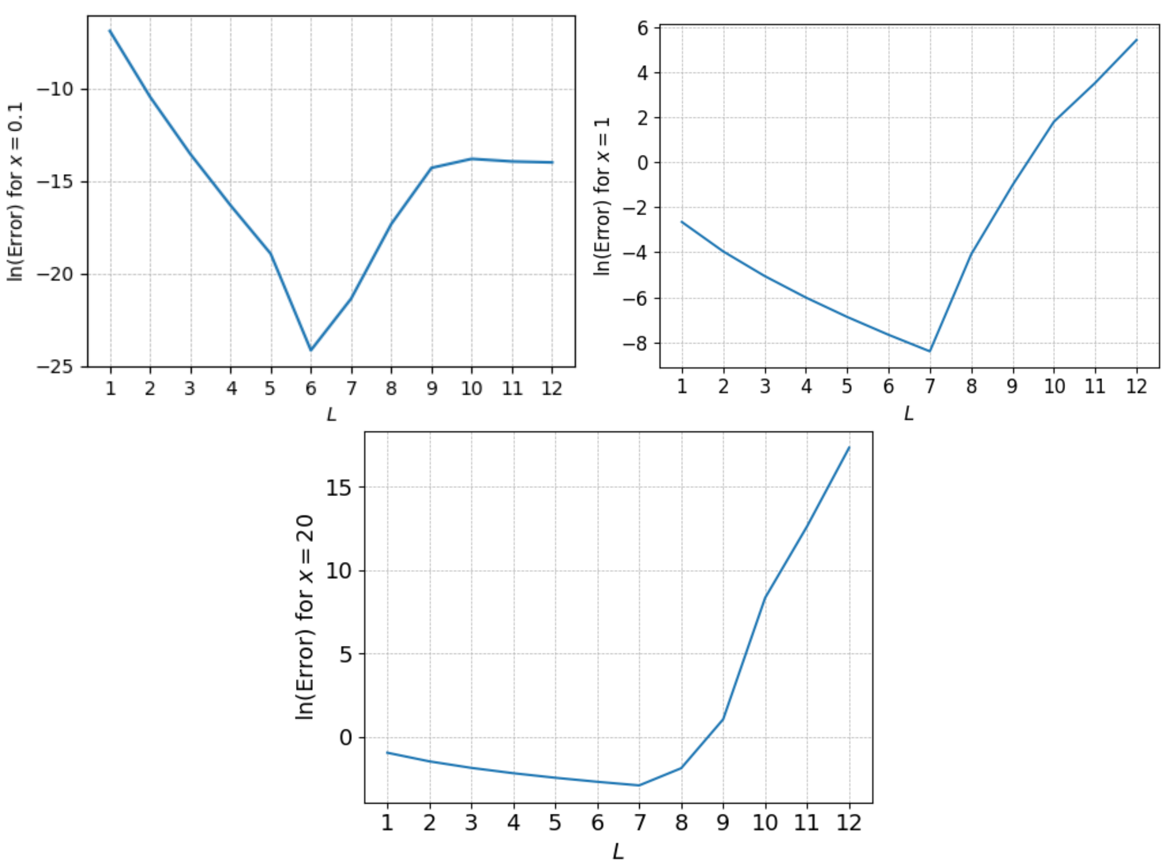


Figure 9: Plots of L against $\ln(Error)$ for $x = 0.1, 1$ and 20

From Figure 8 we can see that for large $x \geq 1$ the error of the power series estimates diverges exponentially so the power series is not useful for any order here. On the other hand, the graph with $x = 0.1$ demonstrates that the series gives a good estimate for much smaller x as the x^n terms do not diverge. In particular, the order of 9 gives the minimum error.

Figure 9 shows that the Padé approximant can give a relatively small error for different values of x in the range $[0, 20]$. The minimum error is given by either $L = 6$ or $L = 7$ and taking the diagonal approximant.

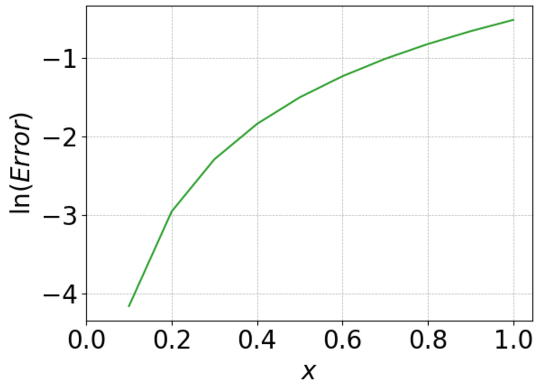


Figure 10: Error from the order 1 power series

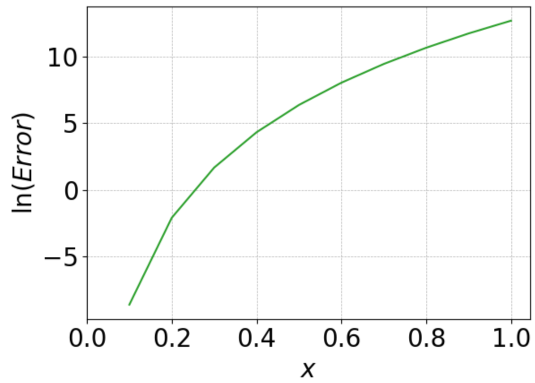


Figure 11: Error from the order 9 power series

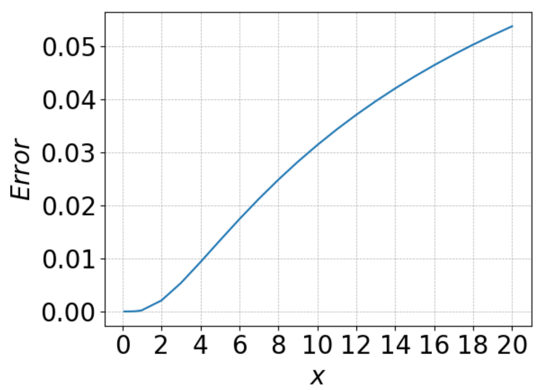


Figure 12: Error of Padé approximant with $L = 7$ in range the [0,20]

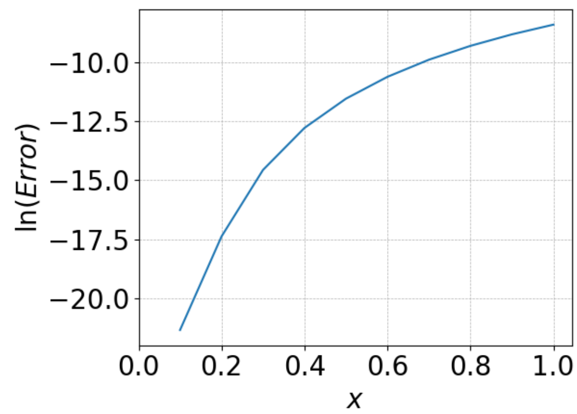


Figure 13: Error of Padé approximant with $L = 7$ zoomed in on the range [0,1]

Since the power series diverges for larger x, only the Padé approximant is a good basis for calculating $f_2(x)$ in the range $0 \leq x \leq 20$. Figure 12 shows that the error remains small up to $x = 20$. In the range $0 \leq x \leq 1$, the Padé approximant approximant also gives far more accurate calculations than the power series comparing Figures 10, 11 and 13. In addition, while the order 9 power series gives a small error for $x = 0.1$, this quickly diverges.

The reason that the Padé approximant is more accurate than the truncated power series is similar to the reasoning for $f_1(x)$; the limiting behaviour is better because the approximant is a fracion and for small x the power series expansion of the Padé approximant is better than the truncated power series with the matching first $L+M+1$ terms.

Question 5

I will display the values of x at the poles and zeros of the Padé approximant of $f_1(x)$ for selected L.

<u>Poles</u>

$L = 1: \quad x \approx -4$

$L = 3: \quad x \approx -20.196, -2.572, -1.232$

$L = 10: \quad x \approx -179.079, -20.197, \ \ldots\ldots, -1.095, -1.023$

<u>Zeros</u>

$L = 1: \quad x \approx -1.333$

$L = 3: \quad x \approx -5.312, -1.636, -1.052$

$L = 10: \quad x \approx -45.021, -11.511, \ \ldots\ldots, -1.052, -1.006$

All poles and zeros are real. Additionally, there are exactly L poles and zeros for

$R_{L,L}(x)$ meaning that there is one more pole and zero when L is increased by one. All poles and zeros are negative, with the least negative values approaching 1 as L is increased. Meanwhile, the largest negative increases as L is increased and the remaining poles/zeros lie between this value and 1.
ALSO ANOMALOUS POLES and ZEROS NON-REAL FOR LARGE L

The Padé approximant of $f_3(x)$ is the inverse of the Padé approximant of $f_1(x)$. Therefore, the poles of this approximant are the zeros of the approximant of $f_1(x)$ and vice versa. Hence the results above can be used to describe their behaviour.

We have the following results for $f_4(x)$.

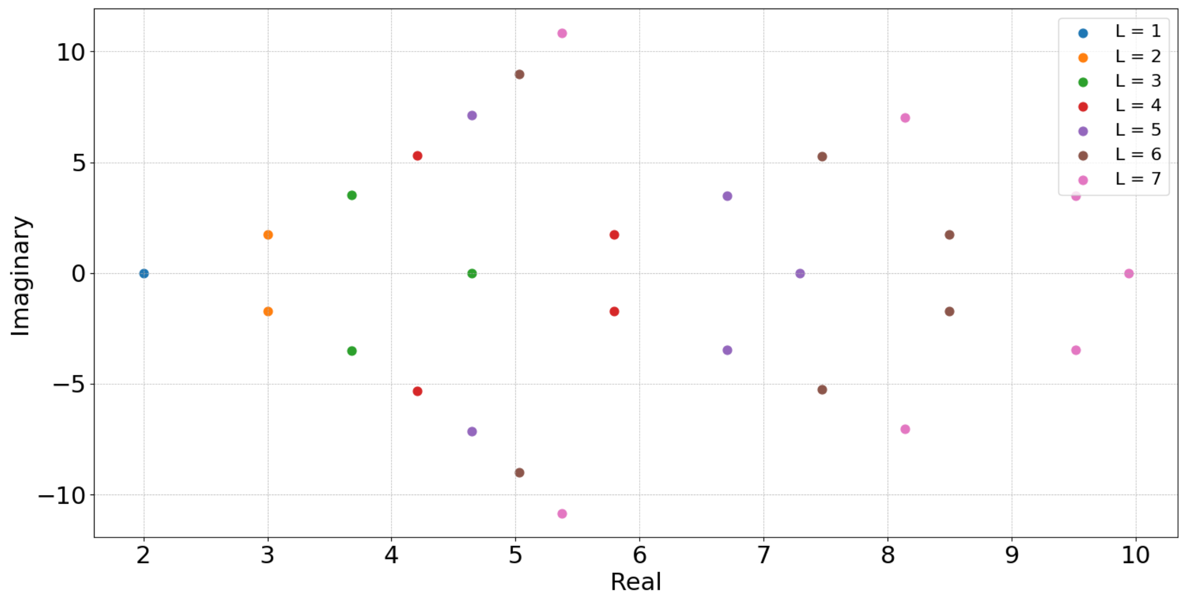


Figure 14: Scatter graph showing poles of $R_{L,L}(x)$

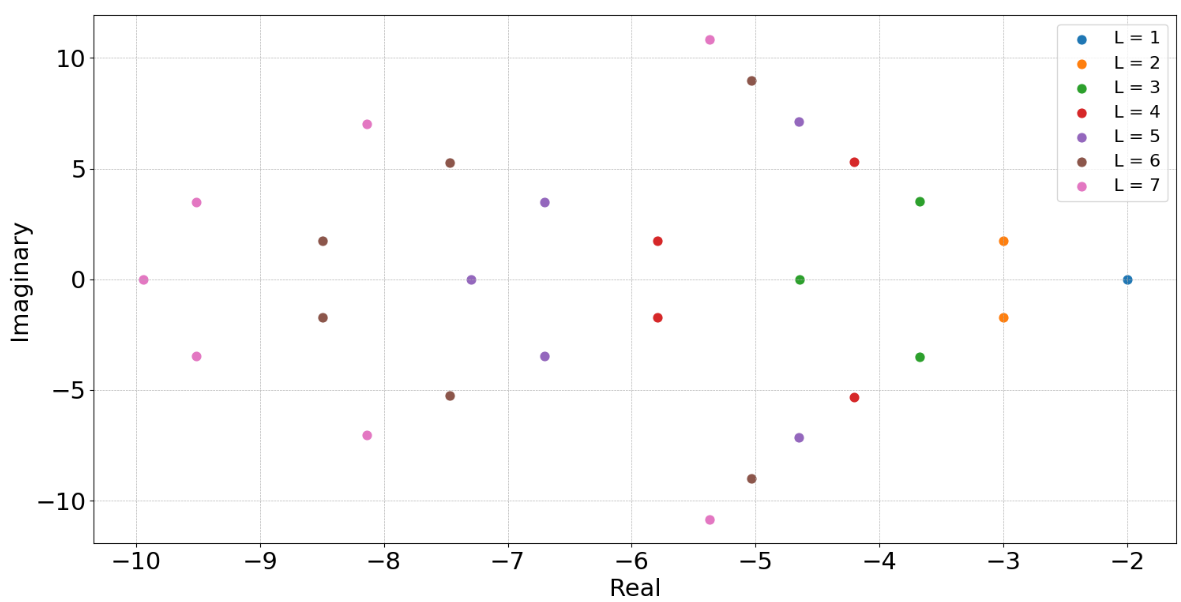


Figure 15: Scatter graph showing zeros of $R_{L,L}(x)$

From Figures 14 and 15, we notice that there are mostly complex roots in conjugate pairs with the magnitude increasing as L increases. There are no real poles or zeros for $L = 2, 4$ or 6 however there is exactly one for all odd L excluding 1, and exactly two for the remaining even $L \geq 8$.

We can obtain similar graphs for $f_5(x)$. The number of real poles and zeros remains the same as above apart from the fact that there are two real poles for $L = 4$ and $L = 6$. When there are real poles, there is always one very close to -1 and for even values of L there is an additional positive real pole. For instance, for $L = 14$ there are poles -1.00 and 1.48. As L increases above 10, additional poles and zeros are within a distance of 2 to the origin.

For $f_6(x)$, there is a real pole for $L \geq 2$ with increasingly large magnitute. For example, for $L = 9$ there is a pole at $x = 7390.11$ and for $L = 10$ there is one at $x = -22133$. This large real pole alternated between positive and negative values for odd and even values of L respectively and seems to increase in magnitude by a factor of approximately 3 each time L is increased. There is a second real pole for even L which is just less than -2. The remaining poles tend towards $-\frac{1}{2} \pm \frac{\sqrt{3}}{2}$ as outline in Figure 16.

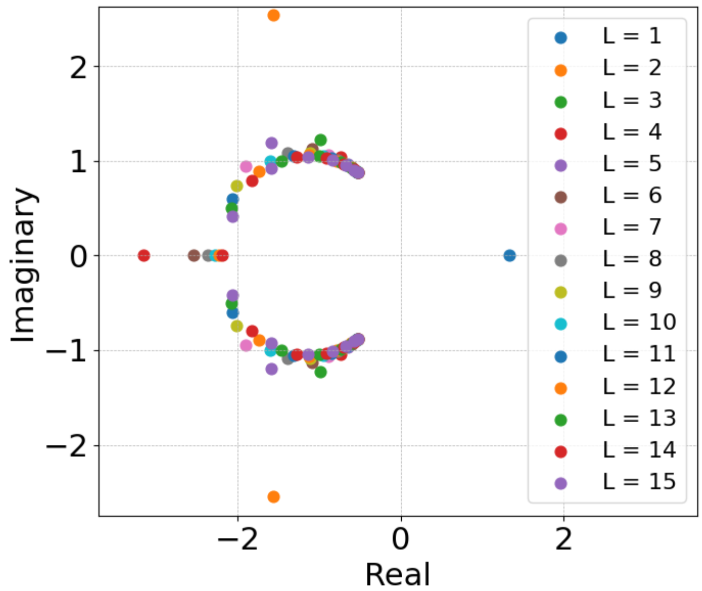


Figure 16: Graph showing poles of f_6

The zeros of $f_6(x)$ are positioned in an identical fashion to Figure 16 with an increasing number of roots approaching $-\frac{1}{2} \pm \frac{\sqrt{3}}{2}$.

The least negative poles and zeros of the approximants of $f_1(x)$ and $f_3(x)$ tending to -1 corresponds to a branch point of both functions at $x = -1$. The remaining poles and zeros can be considered to be lying on a branch cut along the negative real axis. PLOT or table

$f_4(x)$ has no zeros, poles or branch points so there is no correspondence with the approximants. $f_5(x)$ has a pole at $z = -1$ which corresponds to the real poles we find very close to -1 in the approximants. Table **??** demonstrates this in more detail. TABLE

	_	R	RE	REA	REAC	REACT
_	0	1	2	3	4	5
C	1	1	2	3	3	4
CA	2	2	2	2	3	4
CAT	3	3	3	3	3	3

We find that $-\frac{1}{2} \pm \frac{\sqrt{3}}{2}$ are zeros and branch points of $f_6(x)$ which explains why the zeros approach these values. ∞ is the third and final branch point so any branch cut requires two segments. Thus the poles lying along the real axis can be considered to be lying along a segment of a branch cut which goes from one of the finite branch points and to ∞ along the real axis.

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])
        c_target = np.array(c_target, dtype=np.float64)
        # This is the product of the vector and matrix
        # We make sure the dtype is float64

        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.array(np.vstack(rows),
                        dtype=np.float64)
        # 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 odd_product(last_int):
    product = 1
    next_int = 1
    while next_int <= last_int:
        product *= next_int
        next_int += 2
    return product


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

    # numerator = (-1)**(k-1) * odd_product(2*k - 3)
    # denominator = 2**k * math.factorial(k)
    # 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()
```

```python
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)
```

Question_1.py for Question 2

```python
import numpy as np
import math
import matplotlib.pyplot as plt
from Program_A import approximator
import Question_1


def tabulate_error(L):
    c_vector = np.empty([0], dtype = np.double)
    for i in range(2*L + 1):
        c_vector = np.append(c_vector,
            Question_1.find_coefficient(i))
    pade_approximants = []
    for m in range(L + 1):
        approximant = approximator(c_vector, m, m)
        pade_approximants.append(
            approximant.evaluate_approximant(1))

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

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


def plot_log_error(L):
    c_vector = np.empty([0], dtype = np.double)
    for i in range(2*L + 1):
        c_vector = np.append(c_vector,
            Question_1.find_coefficient(i))
    pade_approximants = []
    for m in range(0, L + 1):
        approximant = approximator(c_vector, m, m)
        pade_approximants.append(
            approximant.evaluate_approximant(1))
    error_vector = np.absolute(np.array(pade_approximants) \
                                        - math.sqrt(2))

    log_vector = np.log(error_vector)
    x_vector = np.arange(len(log_vector))

    plt.rc('font', size = 32)
    plt.grid(linestyle = '--', linewidth = 0.5)
    plt.plot(log_vector, color = 'C2')
    plt.xlabel('$L$')
    plt.ylabel('$\ln(error)$')
    plt.xticks(x_vector)
    plt.show()


if __name__ == '__main__':
    L = 2
    x = 1

    c_vector = np.empty([0], dtype = np.double)
```

```
    for i in range(2*L + 1):
        c_vector = np.append(c_vector,
            Question_1.find_coefficient(i))
    approximant = approximator(c_vector, L, L)
    estimate = approximant.evaluate_approximant(1)

    tabulate_error(L)
    # print('Machine precision: ', np.finfo(np.float64).eps)

    #plot_log_error(10)

# ITERATIVE IMPROVEMENT ?
```

Question_2.py for Question 3

```
import math
import numpy as np
import matplotlib.pyplot as plt
from Program_A import approximator
import Question_1


def f_1(x):
    return math.sqrt(1 + x)


def series_estimate(N, x):
    sum = 0
    for i in range(N + 1):
        sum += Question_1.find_coefficient(i) * x**i
    return sum


def diagonal_approximant(L, x):
    c_vector = np.empty([0], dtype = np.double)
    for i in range(2*L + 1):
        c_vector = np.append(c_vector,
            Question_1.find_coefficient(i))
    approximant = approximator(c_vector, L, L)
    return approximant.evaluate_approximant(x)


def plot_comparison(N, L):
    plt.rc('font', size = 32)
    plt.grid(linestyle = '--', linewidth = 0.5)

    x_vector = np.arange(1,101, 1)
    f_1_vector = np.vectorize(f_1)(x_vector)
    series_vector = np.vectorize(series_estimate)(N,
                                            x_vector)
    approximant_vector = np.vectorize(diagonal_approximant)\
                                    (L, x_vector)

    plt.figure(1)
    plt.plot(x_vector, f_1_vector, label = '$y=f_{1}(x)$',
        color = 'black')
    label_string = str(L) + ',' + str(L)
    plt.plot(x_vector, approximant_vector, color = 'C1',
        label = '$y=R_{{{}}}(x)$'.format(label_string))
    plt.xlabel('$x$')
    plt.ylabel('$y$')
    plt.legend(loc = 'best')
    plt.show()

    plt.plot(2)
    plt.plot(x_vector, series_vector, color = 'C0', label =
        '$y=\sum_{{k=0}}^{{{}}}c_{{k}}x^{{k}}$'.format(N))
    plt.xlabel('$x$')
    plt.ylabel('$y$')
    plt.legend(loc = 'best')
    plt.tight_layout()
```

```
    # Or just plot the error
    plt.show()


class approximant_investigation:
    def __init__(self, chosen_val1, chosen_val2):
        self.chosen_val1 = chosen_val1
        self.chosen_val2 = chosen_val2


    def create_graph(self, x):
        # plot log of error against L
        plt.rc('font', size = 32)
        plt.grid(linestyle = '--', linewidth = 0.5)

        L_vector = np.arange(0, 21, 1)
        # Can't have L too large or error with x**k at some
        # point
        approximant_vector = []
        for i in range(len(L_vector)):
            approximant_vector.append(
                            diagonal_approximant(i, x))
        approximant_vector = np.array(approximant_vector)
        error_vector = f_1(x) - approximant_vector
        print(error_vector)
        log_error = np.log(np.absolute(error_vector))
        plt.plot(L_vector, log_error, color = 'C3')

        plt.xlabel('$L$')
        plt.ylabel('$\ln(Error)$')
        plt.tight_layout()
        plt.show()
        # Also plot something to do with error


    def display_graphs(self):
        self.create_graph(self.chosen_val1)
        self.create_graph(self.chosen_val2)



if __name__ == '__main__':
    # plot_comparison(3, 50) # N, L, fig_index

    pade_test = approximant_investigation(10, 100)
    pade_test.display_graphs()
```

continued_fraction.py for Question 3

```python
import numpy as np
import matplotlib.pyplot as plt
import math
import Question_1
from Program_A import approximator


def continued_fraction(L, x):
    fraction_term = 0
    for _ in range(2*L):
        fraction_term = x / (2 + fraction_term)
    return 1 + fraction_term

def plot_cont_fraction():
    plt.rc('font', size = 32)
    plt.grid(linestyle = '--', linewidth = 0.5)

    L_vector = np.arange(0, 51, 1)

    # vfunc = np.vectorize(continued_fraction)
    # estimate_vector = vfunc(L_vector, 10)

    estimate_vector = []
    for i in range(len(L_vector)):
        estimate_vector.append(continued_fraction(i, 10))
    estimate_vector = np.array(estimate_vector)
    error_vector = estimate_vector - math.sqrt(11)
    print(error_vector)
    log_error = np.log(np.absolute(error_vector))
    plt.plot(L_vector, log_error, color = 'C3')

    plt.xlabel('$L$')
    plt.ylabel('$\ln(Error)$')
    plt.tight_layout()
    plt.show()

def diagonal_approximant(L):
    c_vector = np.empty([0], dtype = np.double)
    for i in range(2*L + 1):
        c_vector = np.append(c_vector,
            Question_1.find_coefficient(i))
    approximant = approximator(c_vector, L, L)
    print(math.sqrt(11) - approximant.evaluate_approximant(10))

diagonal_approximant(15)
print( math.sqrt(11) - continued_fraction(15, 10))
```

Question_4.py for Question 4

```
import numpy as np
import math
import matplotlib.pyplot as plt
from Program_A import approximator


def asymptotic_series(order, x):
    # order is the highest power of x
    sum = 0
    for i in range(order + 1):
        sum += (-1)**i * math.factorial(i) * x**i
    return sum


def find_expansion_coefficient(k):
    return (-1)**k * math.factorial(k)


def generate_approximant(L, M):
    c_vector = np.empty([0], dtype = np.double)
    for i in range(L + M + 1):
        c_vector = np.append(c_vector,
                find_expansion_coefficient(i))
    approximant = approximator(c_vector, L, M)
    return approximant


def get_error(series_order, L, M):
    x_vector = np.append( np.arange(1, 10) / 10,
                        np.arange(1, 21) )
    numerical_results = [0.91563334, 0.85211088,
0.80118628, 0.75881459, 0.72265723, 0.69122594, 0.66351027,
0.63879110, 0.61653779, 0.59634736, 0.46145532, 0.38560201,
0.33522136, 0.29866975, 0.27063301, 0.24828135, 0.22994778,
0.21457710, 0.20146425, 0.19011779, 0.18018332, 0.17139800,
0.16356229, 0.15652164, 0.15015426, 0.14436271, 0.13906806,
0.13420555, 0.12972152]

    series_results = np.empty([0])
    for i in range(len(x_vector)):
        series_results = np.append(series_results,
                asymptotic_series(series_order, x_vector[i]))

    approximant = generate_approximant(L, M)
    approximant_results = \
        approximant.evaluate_approximant(x_vector)

    series_error = np.absolute(series_results -
                                    numerical_results)
    approximant_error = np.absolute(approximant_results -
                                    numerical_results)
    return series_error, approximant_error


def plot_series_error(y):
    # Plots the error given x
```

```
    x_vector = np.arange(1, 11) / 10
    plt.rc('font', size = 20)
    plt.grid(linestyle = '--', linewidth = 0.5)
    plt.plot(x_vector, np.log(y[:10]), color = 'C2')
    plt.xlabel('$x$')
    plt.ylabel('$\ln(Error)$')
    plt.tight_layout()
    x_ticks = np.arange(0, 1.2, 0.2)
    plt.xticks(x_ticks)
    plt.show()


def plot_approximant_error(y):
    # Plots the error given x
    #x_vector = np.append( np.arange(1, 10) / 10,
    #                      np.arange(1, 21) )
    x_vector = np.arange(1, 11) / 10
    # switch x_vector depending on graph desired
    plt.rc('font', size = 20)
    plt.grid(linestyle = '--', linewidth = 0.5)
    plt.plot(x_vector, np.log(y[:len(x_vector)]), color = 'C0')
    # plt.plot(x_vector, y[:len(x_vector)], color = 'C0')
    # plot without log for [0, 20] range
    plt.xlabel('$x$')
    plt.ylabel('$\ln(Error)$')
    plt.tight_layout()
    # x_ticks = np.append([0, 0.5], np.arange(1, 21))
    # x_ticks = list(range(0, 21, 2))
    x_ticks = np.arange(0, 1.2, 0.2)
    plt.xticks(x_ticks)
    plt.show()


def get_series_error_change(x, actual_value, order_vector):
    series_error = np.empty([0])
    for N in range(len(order_vector)):
        series_error = np.append( series_error,
                abs(asymptotic_series(N + 1, x) -
                                          actual_value) )
    return series_error


def get_approximant_error_change(x, actual_value, L_vector):
    approximant_error = np.empty([0])
    for L in range(len(L_vector)):
        approximant = generate_approximant(L + 1, L + 1)
        approximant_result = approximant.\
            evaluate_approximant(x)
        approximant_error = np.append( approximant_error,
            abs(approximant_result - actual_value) )
    return approximant_error


def plot_series_error_change():
    # Plots the error from power series varying order
    order_vector = np.arange(1, 13, 1)
    x_list = [0.1, 1, 20]
```

```python
    numerical_results = [0.91563334, 0.59634736, 0.12972152]
    for index in range(len(x_list)):
        plt.rc('font', size = 32)
        plt.grid(linestyle = '--', linewidth = 0.5)
        y = get_series_error_change(x_list[index],
                                    numerical_results[index],
                                    order_vector)
        plt.plot(order_vector, np.log(y), color = 'C3')
        plt.ylabel('ln(Error) for ' '$x = {}$'.\
                   format(x_list[index]))
        plt.xlabel('$Order$')
        plt.tight_layout()
        plt.xticks(order_vector)
        plt.show()


def plot_approximant_error_change():
    L_vector = np.arange(1, 13, 1)
    # Plots the error varying L or order
    x_list = [0.1, 1, 20]
    numerical_results = [0.91563334, 0.59634736, 0.12972152]
    for index in range(len(x_list)):
        plt.grid(linestyle = '--', linewidth = 0.5)
        y = get_approximant_error_change(x_list[index],
                                    numerical_results[index],
                                    L_vector)
        plt.rc('font', size = 12)
        plt.plot(L_vector, np.log(y))
        plt.xlabel('$L$')
        plt.ylabel('ln(Error) for ' '$x = {}$'.\
                   format(x_list[index]))
        plt.tight_layout()
        plt.xticks(L_vector)
        plt.show()


if __name__ == '__main__':
    #plot_series_error_change()
    #plot_approximant_error_change()

    series_error, approximant_error = get_error(1, 7, 7)
    # change order from 1 to 9
    #plot_series_error(series_error)
    plot_approximant_error(approximant_error)
```