Elementos do grupo:

* Diogo Fonseca nº 79858
* Tomás Teodoro nº 80044
* Diogo Silva nº 79828
* Tiago Granja nº 79845

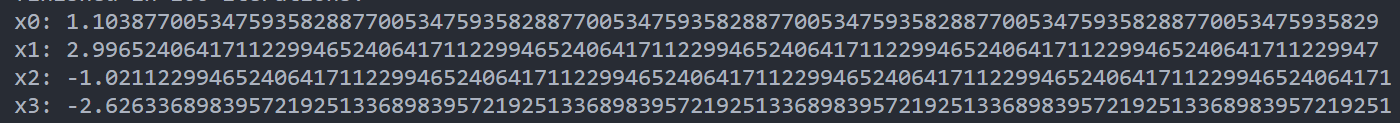
Resumo

O programa é capaz de interpolar qualquer lista de pares (x, y) com até 100 algarismos significativos de precisão através do método das diferenças divididas de Newton e do método de interpolação de neville. O programa depois de calculado o polinómio interpolador da lista de pares de pontos, passa a um processo iterativo onde pede valores de x para aproximar valores de y (no caso do método de neville, o programa calcula também o erro relativo da aproximação).

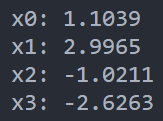
Instruções de Utilização

Nota: pressione *Ctrl+C* a qualquer momento durante a execução para parar o programa.

* **Precision:**  – O número (inteiro) de algarismos significativos a ser usado internamente pelo programa. ()

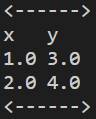
Exemplo: ()

Exemplo: ()

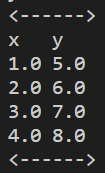


* **Number of Nodes:**  – O número de pares (x, y) a serem introduzidos. ()

Exemplo:



Exemplo:



* **X=:** O valor de x a ser interpolado pelo polinómio interpolador calculado.

Execução

Cada ficheiro aqui posto deve ser colocado no seu ficheiro particular com o mesmo nome que a sua classe, este projeto usa também o auxílio da biblioteca matemática SymPy (<https://www.sympy.org>). Para a sua execução é necessário instalar a mesma para o seu funcionamento correto. Com python instalado no computador, isto pode ser feito através do terminal com o comando “pip3 install sympy”.

Observações

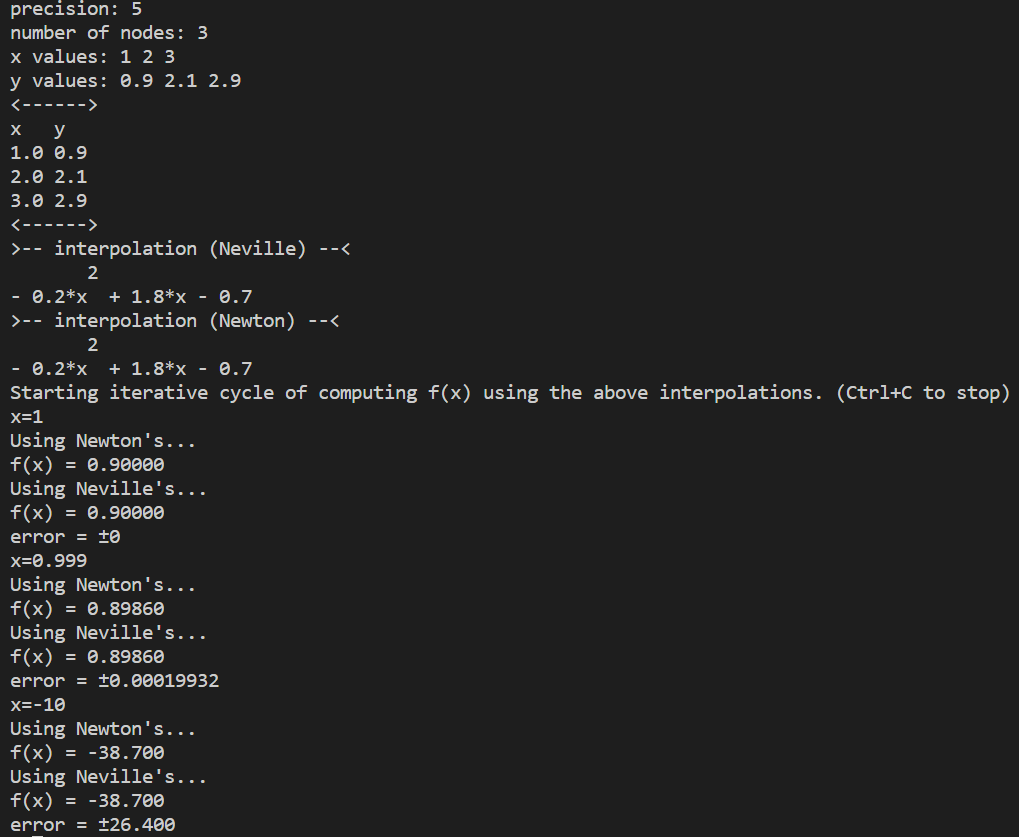
As interpolações dos métodos apresentados conseguem dar aproximações boas para qualquer valor de x perto dos valores de x dados. Isto dado que a função se porta minimamente bem, para qualquer função que tenha variações drásticas, as aproximações vão perder qualidade. Este facto pode ser demonstrado vendo o erro calculado para o método de neville conforme o x diverge dos pontos originais.

O programa está preparado para obter qualquer tipo de input, mesmo que este viole as especificações previamente mencionadas, gerando uma mensagem de erro apropriada, mas continuando a execução.

Exemplo:



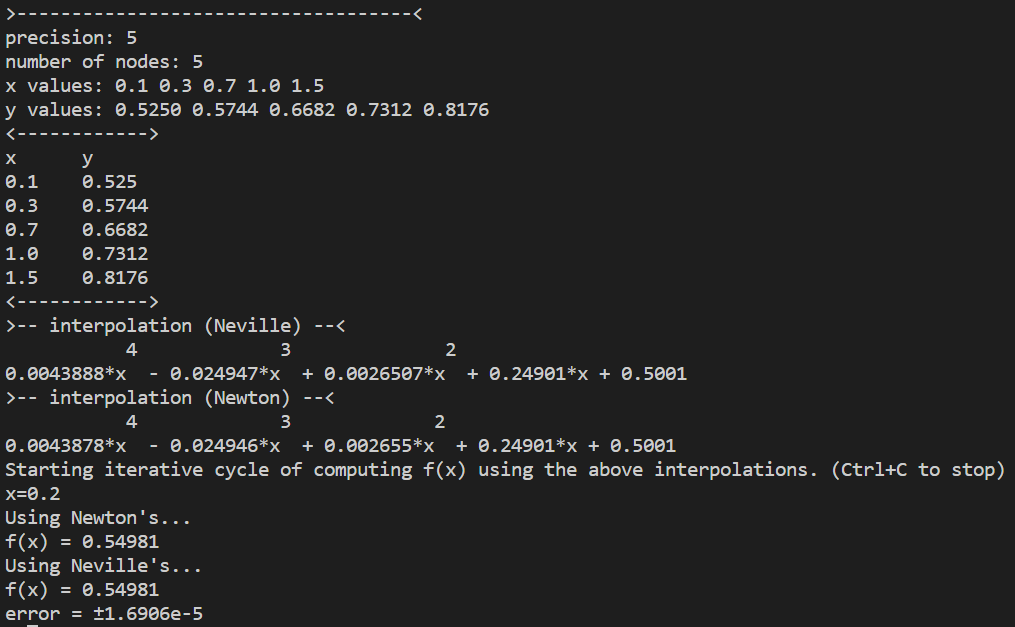
Exemplo:

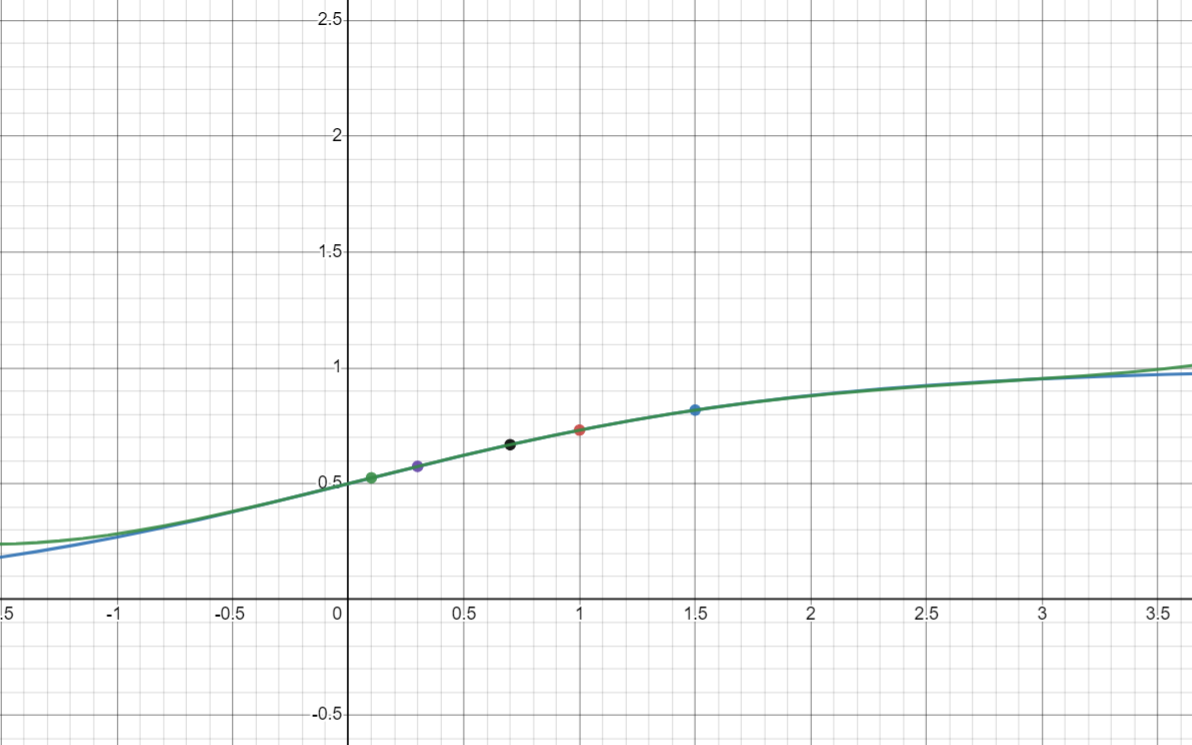


Explicação: Para um x que está nos valores originais, o erro será 0 porque se sabe o valor de y para esse ponto. Para um ponto proximo dos valores originais, o erro será muito baixo e por consequência a aproximação boa. O valor que está bastante longe dos valores originais vai ter uma incerteza enorme já que está a ser extrapolado.

Screenshots

|  |  |
| --- | --- |
|  |  |
| 0.1 | 0.525 |
| 0.3 | 0.5744 |
| 0.7 | 0.6682 |
| 1 | 0.7312 |
| 1.5 | 0.8176 |

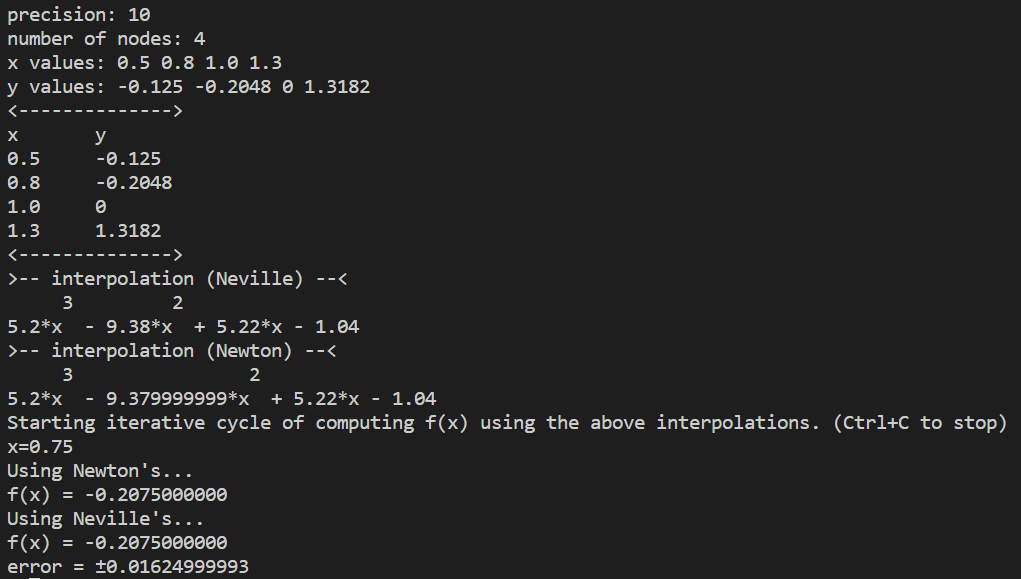


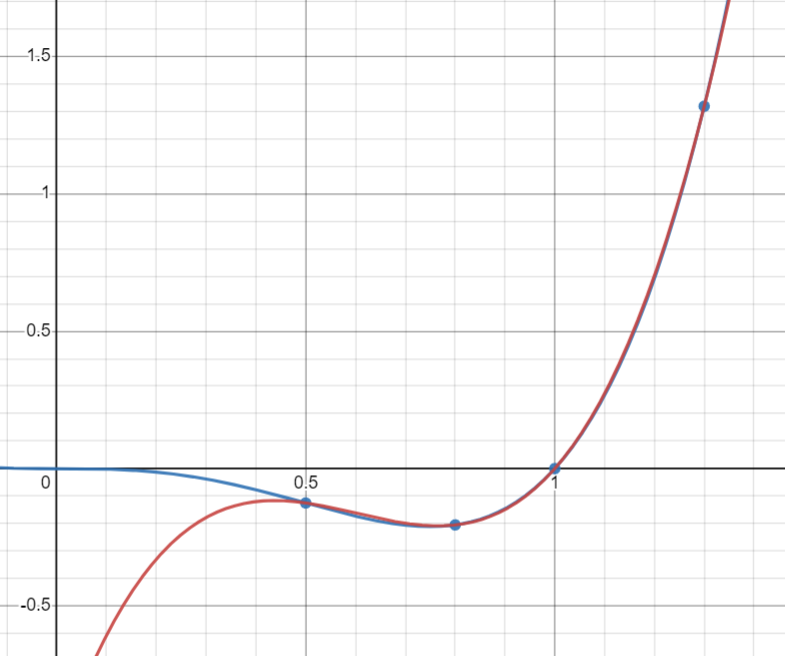


interpolado:

= 0.54983

|  |  |
| --- | --- |
|  |  |
| 0.5 | -0.125 |
| 0.8 | -0.2048 |
| 1 | 0 |
| 1.3 | 1.3182 |

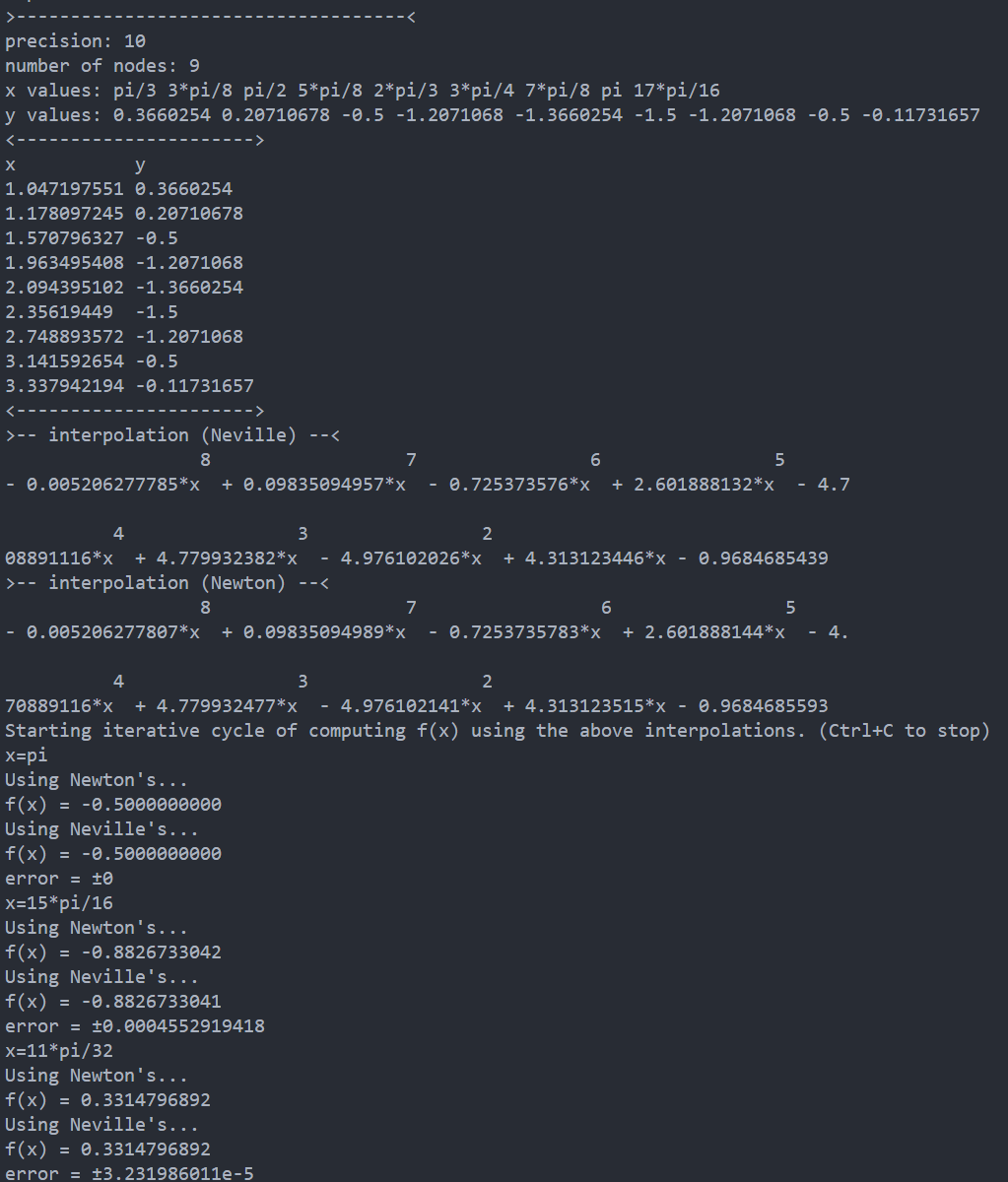


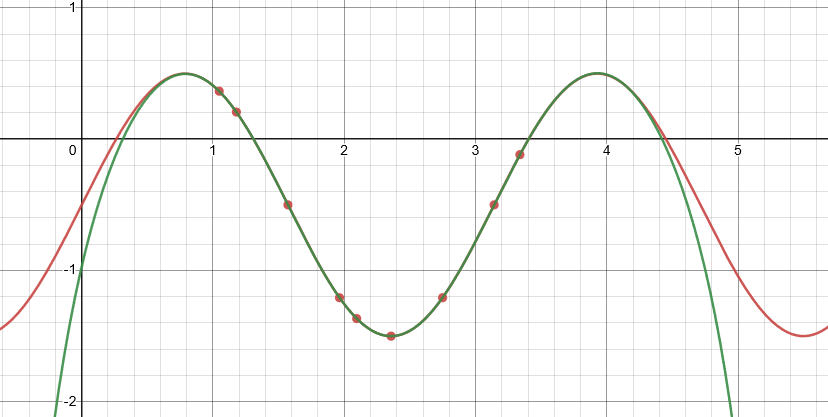


interpolado:

= -0.2109375

|  |  |
| --- | --- |
|  |  |
|  | 0.3660254 |
|  | 0.20710678 |
|  | -0.5 |
|  | -1.2071068 |
|  | -1.3660254 |
|  | -1.5 |
|  | -1.2071068 |
|  | -0.5 |
|  | -0.11731657 |





interpolado:

= -0.5

interpolado:

= -0.8826834324

interpolado:

= 0.3314696123

Código

main.py

import math\_input

import neville

import newton

from sympy import \*

def get\_max\_len(*values*):

    max\_len = 0

    for value in *values*:

        pretty\_str = pretty(value, *full\_prec*=False)

        if len(pretty\_str) > max\_len:

            max\_len = len(pretty\_str)

    return max\_len

def print\_values(*x\_values*, *y\_values*):

    max\_len = max(get\_max\_len(*x\_values*), get\_max\_len(*y\_values*))

    print("<" + "-"\*max\_len\*2 + ">")

    formatter = "{:<" + str(max\_len) + "}"

    formatter += " {:<" + str(max\_len) + "}"

    print(formatter.format('x', 'y'))

    for i in range(len(*x\_values*)):

        print(formatter.format(pretty(*x\_values*[i], *full\_prec*=False), pretty(*y\_values*[i], *full\_prec*=False)))

    print("<" + "-"\*max\_len\*2 + ">")

exit = false

print("tip: Ctrl+C to exit!")

while exit == false:

    print(">------------------------------------<")

    try:

        precision = math\_input.math\_input.get\_precision()

        num\_nodes = math\_input.math\_input.get\_num\_nodes()

        x\_values = math\_input.math\_input.get\_values(num\_nodes, precision, "x")

        y\_values = math\_input.math\_input.get\_values(num\_nodes, precision, "y")

        print\_values(x\_values, y\_values)

        neville\_success = False

        newton\_success = False

        exit\_x\_input = False

        result\_neville = 0

        result\_newton = 0

        error\_neville = -1

        error\_newton = -1

        try:

            print(">-- interpolation (Neville) --<")

            result = neville.neville.solve(x\_values, y\_values)

            result\_neville = result[0]

            error\_neville = result[1]

            neville\_success = True

            pprint(simplify(result\_neville), *use\_unicode*=False)

        except Exception as e:

            print("Neville's method failed.")

            print(str(e))

        try:

            print(">-- interpolation (Newton) --<")

            result\_newton = newton.newton.solve(x\_values, y\_values)

            newton\_success = True

            pprint(simplify(result\_newton), *use\_unicode*=False)

        except Exception as e:

            print("Newton's method failed.")

            print(str(e))

        print("Starting iterative cycle of computing f(x) using the above interpolations. (Ctrl+C to stop)")

        if neville\_success or newton\_success:

            while exit\_x\_input == false:

                try:

                    x = math\_input.math\_input.get\_x\_value(precision)

                    print("Using Newton's...")

                    print("f(x) = ", *end*="")

                    pprint(result\_newton.subs(Symbol("x"), x))

                    print("Using Neville's...")

                    print("f(x) = ", *end*="")

                    pprint(result\_neville.subs(Symbol("x"), x))

                    if error\_neville < 0:

                        print("error could not be calculated: must give over two values for an error estimate.")

                    elif x in x\_values:

                        print("error = ±0")

                    else:

                        print("error = ±", *end*="")

                        pprint(error\_neville.subs(Symbol("x"), x))

                except KeyboardInterrupt:

                    exit\_x\_input = true

        else:

            print("both methods failed! repeating...")

    except KeyboardInterrupt:

        exit = true

    except Exception as e:

        print(str(e))

print("")

print("Exiting...")

math\_input.py

from sympy import \*

class math\_input:

    @staticmethod

    def get\_precision():

        this\_input = input("precision: ")

        try:

            try:

                this\_input = int(this\_input)

            except:

                raise Exception("Error: precision must be an integer.")

            if not ask(Q.integer(this\_input)):

                raise Exception("Error: precision must be an integer.")

            if this\_input < 0 or this\_input > 100:

                raise Exception("Error: Precision must be between 0 and 100.")

            return int(this\_input)

        except Exception as e:

            raise e

    @staticmethod

    def get\_num\_nodes():

        this\_input = input("number of nodes: ")

        try:

            try:

                this\_input = int(this\_input)

            except:

                raise Exception("Error: number of nodes must be an integer.")

            if this\_input <= 0:

                raise Exception("Error: number of nodes should be a number greater than zero.")

            return this\_input

        except Exception as e:

            raise e

    @staticmethod

    def get\_values(*num\_nodes*, *precision*, *value\_name*):

        values = [];

        this\_input = input(f"{value\_name} values: ")

        try:

            tokens = this\_input.split(" ")

            if len(tokens) != num\_nodes:

                raise Exception(f"Error: expected {num\_nodes} variables but got {len(tokens)}.")

            for token in tokens:

                value = N(token, precision)

                if not ask(Q.real(value)):

                    raise Exception(f"Error: Invalid node '{token}'.")

                values.append(value)

            return values

        except Exception as e:

            raise e

    @staticmethod

    def get\_x\_value(*precision*):

        this\_input = input(f"x=")

        try:

            value = N(this\_input, precision)

            if not ask(Q.real(value)):

                raise Exception(f"Error: Invalid x value '{value}'.")

            return value

        except Exception as e:

            raise e

neville.py

from sympy import \*

class neville:

    @staticmethod

    def solve(*x\_values*, *y\_values*):

        n = len(*x\_values*)

        Q = [[0 for i in range(n)] for j in range(n)]

        for column in range(n):

            if column == 0:

                for i in range(n):

                    Q[i][0] = *y\_values*[i]

            else:

                for row in range(n - column):

                    neville.Q\_calc(Q, *x\_values*, symbols('x'), column + row, column)

        error = -1

        if n > 2:

            error = simplify(abs(Q[n-1][n-1] - Q[n-2][n-2]))

        return [Q[n - 1][n - 1], error]

    @staticmethod

    def Q\_calc(*Q*, *x\_values*, *value*, *row*, *column*):

        x0 = *x\_values*[*row* - *column*]

        x1 = *x\_values*[*row*]

        previous\_Q\_upper = *Q*[*row* - 1][*column* - 1]

        previous\_Q\_lower = *Q*[*row*][*column* - 1]

*Q*[*row*][*column*] = ((*value* - x0)\*previous\_Q\_lower - (*value* - x1)\*previous\_Q\_upper) / (x1 - x0)

newton.py

from sympy import \*

class newton:

    @staticmethod

    def solve(*x\_values*, *y\_values*):

        n = len(*x\_values*)

        Q = [[0 for i in range(n)] for j in range(n)]

        for column in range(n):

            if column == 0:

                for i in range(n):

                    Q[i][0] = *y\_values*[i]

            else:

                for row in range(n - column):

                    newton.Q\_calc(Q, *x\_values*, column + row, column)

        result = 0

        for i in range(n):

            result += Q[i][i] \* newton.prod\_calc(i, *x\_values*)

        return result

    @staticmethod

    def Q\_calc(*Q*, *x\_values*, *row*, *column*):

        x0 = *x\_values*[*row* - *column*]

        x1 = *x\_values*[*row*]

        previous\_Q\_upper = *Q*[*row* - 1][*column* - 1]

        previous\_Q\_lower = *Q*[*row*][*column* - 1]

*Q*[*row*][*column*] = (previous\_Q\_lower - previous\_Q\_upper) / (x1 - x0)

    @staticmethod

    def prod\_calc(*iteration*, *x\_values*):

        result = 1

        for i in range(*iteration*):

            result \*= (symbols('x') - *x\_values*[i])

        return result