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:** d – O número (inteiro) de algarismos significativos a ser usado internamente pelo programa. $(0 < x \leq 100)$

  Exemplo: $(x = 100)$

```
1.1038770053475935828877005347593582887700534759358288770053475935828877005347593582887700534759358288770053475935829
```

  Exemplo: $(x = 5)$

```
1.1039
```

- **Number of Nodes:** n – O número de pares (x, y) a serem introduzidos. $(1 < x < \infty)$

  Exemplo: $n = 2$

```
<------>
x   y
1.0 3.0
2.0 4.0
<------>
```

  Exemplo: $n = 4$

```
<------>
x   y
1.0 5.0
2.0 6.0
3.0 7.0
4.0 8.0
<------>
```

- **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:

```
number of nodes: 0
Error: number of nodes should be a number greater than zero.
```

Exemplo:

```
precision: 5
number of nodes: 3
x values: 1 2 3
y values: 0.9 2.1 2.9
<------>
x   y
1.0 0.9
2.0 2.1
3.0 2.9
<------>
>-- interpolation (Neville) --<
       2
- 0.2*x  + 1.8*x - 0.7
>-- interpolation (Newton) --<
       2
- 0.2*x  + 1.8*x - 0.7
Starting iterative cycle of computing f(x) using the above interpolations. (Ctrl+C to stop)
x=1
Using Newton's...
f(x) = 0.90000
Using Neville's...
f(x) = 0.90000
error = ±0
x=0.999
Using Newton's...
f(x) = 0.89860
Using Neville's...
f(x) = 0.89860
error = ±0.00019932
x=-10
Using Newton's...
f(x) = -38.700
Using Neville's...
f(x) = -38.700
error = ±26.400
```

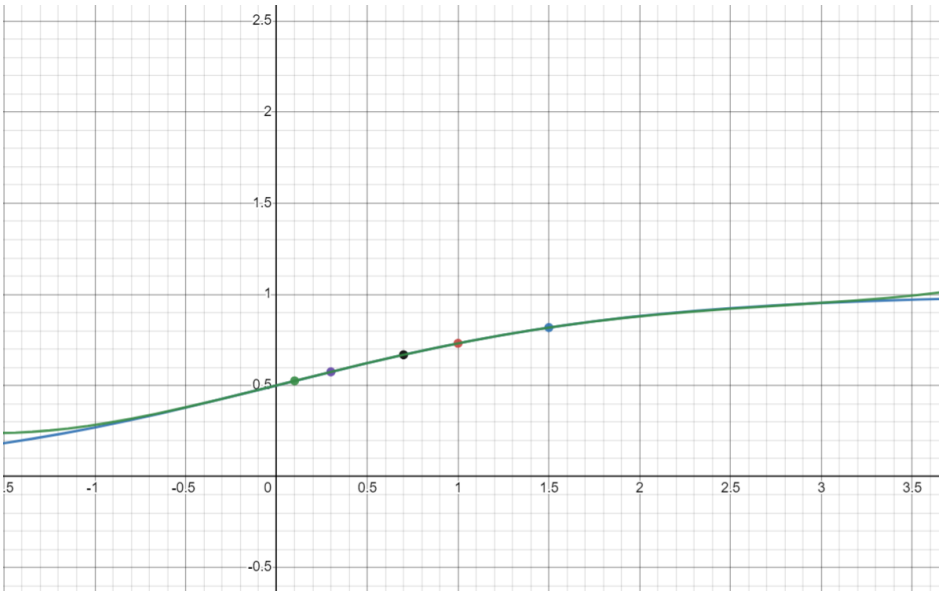
Explicação: Para um x que está nos valores originais, o erro será 0 porque se sabe o valor de y para esse ponto. Para um ponto proximo dos valores originais, o erro será muito baixo e por consequência a aproximação boa. O valor que está bastante longe dos valores originais vai ter uma incerteza enorme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

Screenshots

$$f(x) = \frac{1}{1 + e^{-x}}$$

x	$f(x)$
0.1	0.525
0.3	0.5744
0.7	0.6682
1	0.7312
1.5	0.8176

```
>-----------------------------------<
precision: 5
number of nodes: 5
x values: 0.1 0.3 0.7 1.0 1.5
y values: 0.5250 0.5744 0.6682 0.7312 0.8176
<------------>
x      y
0.1    0.525
0.3    0.5744
0.7    0.6682
1.0    0.7312
1.5    0.8176
<------------>
>-- interpolation (Neville) --<
           4             3             2
0.0043888*x  - 0.024947*x  + 0.0026507*x  + 0.24901*x + 0.5001
>-- interpolation (Newton) --<
           4             3             2
0.0043878*x  - 0.024946*x  + 0.002655*x  + 0.24901*x + 0.5001
Starting iterative cycle of computing f(x) using the above interpolations. (Ctrl+C to stop)
x=0.2
Using Newton's...
f(x) = 0.54981
Using Neville's...
f(x) = 0.54981
error = ±1.6906e-5
```



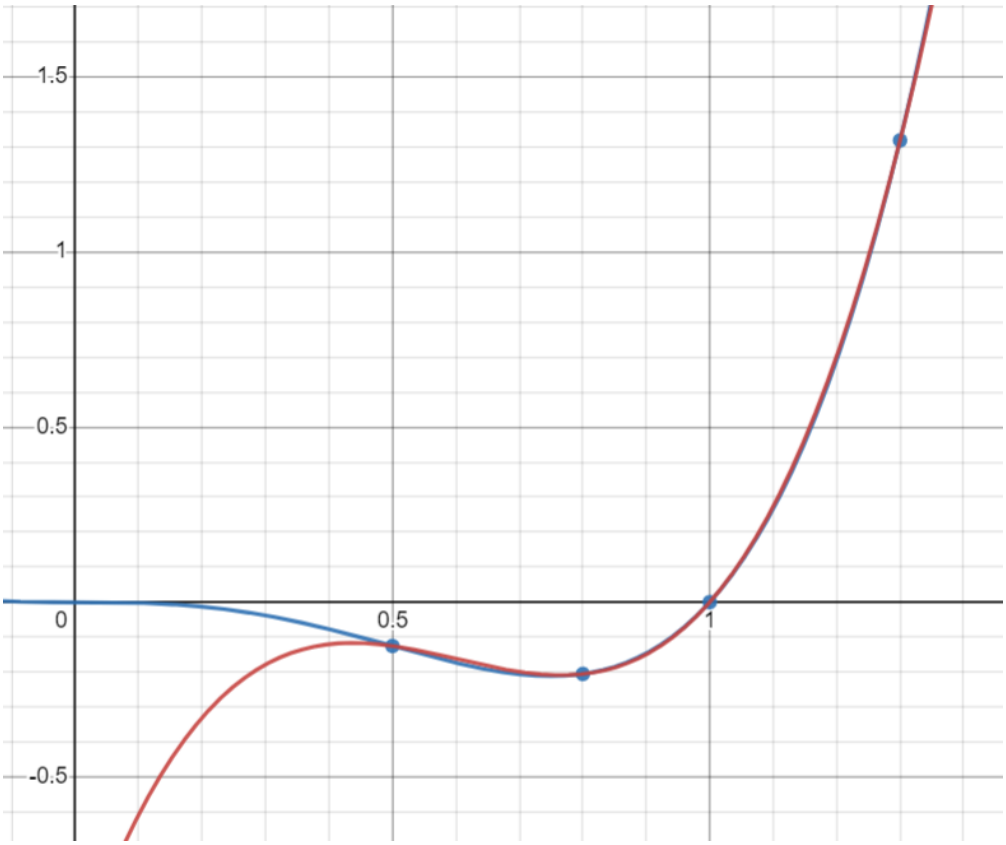
x interpolado: $0.54981 \pm 1.6906 \times 10^{-5}$

$f(x)$= 0.54983

$$f(x) = \frac{1}{1 + e^{-x}}$$

x	$f(x)$
0.5	-0.125
0.8	-0.2048
1	0
1.3	1.3182

```
precision: 10
number of nodes: 4
x values: 0.5 0.8 1.0 1.3
y values: -0.125 -0.2048 0 1.3182
<-------------->
x       y
0.5     -0.125
0.8     -0.2048
1.0     0
1.3     1.3182
<-------------->
>-- interpolation (Neville) --<
     3         2
5.2*x  - 9.38*x  + 5.22*x - 1.04
>-- interpolation (Newton) --<
     3                  2
5.2*x  - 9.379999999*x  + 5.22*x - 1.04
Starting iterative cycle of computing f(x) using the above interpolations. (Ctrl+C to stop)
x=0.75
Using Newton's...
f(x) = -0.2075000000
Using Neville's...
f(x) = -0.2075000000
error = ±0.01624999993
```



x interpolado: -0.2075 ± 0.01625
$f(x)$= -0.2109375

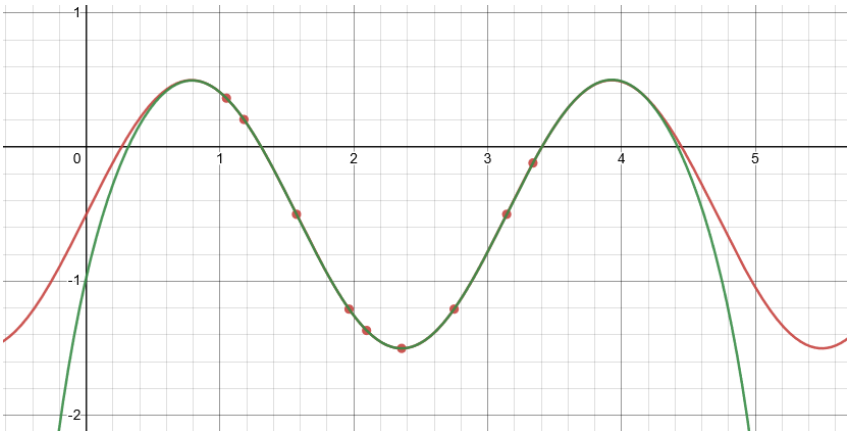
$$f(x) = \sin(2x) - \frac{1}{2}$$

x	$f(x)$
$\frac{\pi}{3}$	0.3660254
$\frac{3\pi}{8}$	0.20710678
$\frac{\pi}{2}$	-0.5
$\frac{5\pi}{8}$	-1.2071068
$\frac{2\pi}{3}$	-1.3660254
$\frac{3\pi}{4}$	-1.5
$\frac{7\pi}{8}$	-1.2071068
π	-0.5
$\frac{17\pi}{16}$	-0.11731657

```
>-----------------------------------<
precision: 10
number of nodes: 9
x values: pi/3 3*pi/8 pi/2 5*pi/8 2*pi/3 3*pi/4 7*pi/8 pi 17*pi/16
y values: 0.3660254 0.20710678 -0.5 -1.2071068 -1.3660254 -1.5 -1.2071068 -0.5 -0.11731657
<---------------------->
x           y
1.047197551 0.3660254
1.178097245 0.20710678
1.570796327 -0.5
1.963495408 -1.2071068
2.094395102 -1.3660254
2.35619449  -1.5
2.748893572 -1.2071068
3.141592654 -0.5
3.337942194 -0.11731657
<---------------------->
>-- interpolation (Neville) --<
                  8                  7                6                5
- 0.005206277785*x  + 0.09835094957*x  - 0.725373576*x  + 2.601888132*x  - 4.7

          4                3                2
08891116*x  + 4.779932382*x  - 4.976102026*x  + 4.313123446*x - 0.9684685439
>-- interpolation (Newton) --<
                  8                  7                6                5
- 0.005206277807*x  + 0.09835094989*x  - 0.7253735783*x  + 2.601888144*x  - 4.

          4                3                2
70889116*x  + 4.779932477*x  - 4.976102141*x  + 4.313123515*x - 0.9684685593
Starting iterative cycle of computing f(x) using the above interpolations. (Ctrl+C to stop)
x=pi
Using Newton's...
f(x) = -0.5000000000
Using Neville's...
f(x) = -0.5000000000
error = ±0
x=15*pi/16
Using Newton's...
f(x) = -0.8826733042
Using Neville's...
f(x) = -0.8826733041
error = ±0.0004552919418
x=11*pi/32
Using Newton's...
f(x) = 0.3314796892
Using Neville's...
f(x) = 0.3314796892
error = ±3.231986011e-5
```



x_0 interpolado: -0.5 ± 0

$f(x_0)$= -0.5

x_1 interpolado: $-0.8826733042 \pm 0.0004552919418$

$f(x_1)$= -0.8826834324

x_2 interpolado: $0.3314796892 \pm 0.00003231986011$

$f(x_2)$= 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
```