Dokumentacja Zadanie 1.11

Ciesielski Mateusz
Kasztelan Damian
Matyjas Sebastian
Grupa 1

Treść:
Wejście: przedział **[a,b]**, liczba naturalna **n** , **n** liczb rzeczywistych $y_0, \dots, y_{n-1}$
Zadanie:
- Stworzyć siatkę równoodległą n elementów
- podać wielomian interpolacyjny w postaci Newtona interpolujący dane wejściowe:

$$P(x) = f[x_0] + f[x_0, x_1](x - x_0) + [x_0, x_1, x_2](x - x_0)(x - x_1) + \dots + [x_0, x_1, \dots, x_n](x - x_0) \cdot (x - x_{n-1})$$

- Wypisać ilorazy różnicowe
- Narysować wielomian interpolacyjny
- zezwolić użytkownikowi na dodanie kolejnego węzła x_n i wartości y_n
- Wyznaczyć metodą Newtona (bez ilorazów różnicowych) wielomian interpolujący dane wejściowe $y_0, \dots, y_n$
- Narysować obydwa wielomiany na jednym wykresie

Interpolacja wielomianowa – Niech $D \subset R$, oraz niech F będzie pewnym zbiorem funkcji $f: D \to R$. Niech $x_0, x_1, \dots, x_n$ będzie ustalonym zbiorem parami różnych punktów z D (węzłów interpolacji). Mówimy, że wielomian w interpoluje funkcję $f \in F$ w węzłach x_j , wtedy, gdy

$$w(x_j) = f(x_j). \qquad 0 \leq j \leq n$$

- **punkty:** $x_0, x_1, \dots, x_n$ **→ węzły interpolacji**
- **wartości:** $f(x_0), f(x_1), \dots, f(x_n)$ **→ wartości pewnej funkcji w tych węzłach**

Postać Newtona I – jedna z metod przedstawiania wielomianu interpolacyjnego. Dla węzłów $x_0, x_1, \dots, x_n$ parami różnych i dla wartości $y_0, y_1, \dots, y_n$ szukamy wielomianu $P \in \pi_n$ spełniającego warunek:

(1) $$P(x_i) = y_i, i = 0, 1, \dots, n$$

w postaci:

$$P(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1) + \ldots + b_n(x - x_0) \cdot \ldots \cdot (x - x_{n-1}).$$

z warunków (1) otrzymamy układ z niewiadomymi $b_0, b_1, \dots, b_n$.

PRZYKŁAD:

Szukamy wielomianu interpolującego dane:

x_i	1	2	4	5
y_i	0	2	12	20

wielomian interpolujący ma postać:

$$P(x)=b_0+b_1(x-1)+b_2(x-1)(x-2)+b_3(x-1)(x-2)(x-4)$$

$$P(x_0)=0\ ,\ P(x_1)=2\ ,\ P(x_2)=12\ ,\ P(x_3)=20$$

liczymy niewiadomą b_0 $\quad P(x_0)=b_0=0$

liczymy b_1 $\quad P(x_1)=b_0+b_1(x_1-1)$ *więc* $2=0+b_1(2-1)$ *stąd otrzymujemy, że* $b_1=2$

liczymy b_2 $\quad P(x_2)=b_0+b_1(x_2-1)+b_2(x_2-1)(x_2-2)$ *stąd otrzymujemy, że* $b_2=1$

liczymy b_3 $\quad P(x_3)=b_0+b_1(x_3-1)+b_2(x_3-1)(x_3-2)+b_3(x_3-1)(x_3-2)(x_3-4)(x_3-5)$

stąd otrzymujemy, że $b_3=0$

więc

$$P(x)=0+2(x-1)+1(x-1)(x-2)+0(x-1)(x-2)(x-4)=2x-2+x^2-3x+2=x^2-x$$

Odpowiedź: $P(x)=x^2-x$

Postać Newtona II – współczynniki wielomianu newtona można wyznaczyć przy użyciu ilorazów różnicowych. Wartości y_j traktujemy jako wartości pewnej funkcji f w punktach x_j .

$$y_j=f(x_j)\ \ dla\ \ j=0,1,\dots,n,\ \ gdzie\ \ x_i\neq x_j,\ \ gdy\ i\neq j$$

Definiujemy ilorazy różnicowe rzędu 0 jako $f[x_j]=f(x_j)$, ilorazy różnicowe rzędu 1 jako:

$$f[x_j,x_{j+1}]=\frac{f[x_{j+1}]-f[x_j]}{x_{j+1}-x_j}$$

ilorazy różnicowe rzędu k jako:

$$f[x_j,x_{j+1},\dots,x_{j+k-1},x_{j+k}]=\frac{f[x_{j+1},\dots,x_{j+k}]-f[x_j,\dots,x_{j+k-1}]}{x_{j+k}-x_j}.$$

PRZYKŁAD:

x_i	1	2	4	5
$f(x_i)$	0	2	12	20

Obliczamy następujące ilorazy różnicowe

$x_0 = 1$, $f[x_0] = 0$ $\quad$ $x_1 = 2$, $f[x_1] = 2$ $\quad$ $x_2 = 4$, $f[x_2] = 12$ $\quad$ $x_3 = 5$, $f[x_3] = 20$

$$f[x_0, x_1] = \frac{2-0}{2-1} = 2 \quad f[x_1, x_2] = \frac{12-2}{4-2} = 5 \quad f[x_2, x_3] = \frac{20-12}{5-4} = 8$$

$$f[x_0, x_1, x_2] = \frac{5-2}{4-1} = 1 \quad f[x_1, x_2, x_3] = \frac{8-5}{5-2} = 1$$

$$f[x_0, x_1, x_2, x_3] = \frac{1-1}{5-1} = 0$$

Wielomian interpolujący obliczamy ze wzoru:

$$P(x) = f[x_0] + f[x_0, x_1](x-x_0) + f[x_0, x_1, x_2](x-x_0)(x-x_1) \\ + f[x_0, x_1, x_2, x_3](x-x_0)(x-x_1)(x-x_2)$$

$$P(x) = 0 + 2(x-1) + 1(x-1)(x-2) + 0(x-1)(x-2)(x-4) \\ = 2x - 2x^2 + 2 - 3x = x^2 - x$$

Odpowiedź: $P(x) = x^2 - x$

Postać Newtona IIa - Postać Newtona jest wygodna w sytuacji, gdy zmieniamy zadanie dokładając dodatkowy węzeł x_{n+1} **(różny od pozostałych) i wartość** y_{n+1} **. Niech :**

$$Q(x) = P(x) + b_{n+1}(x - x_0) \cdot \ldots \cdot (x - x_{n-1})(x - x_n),$$

gdzie P **interpoluje dane** (x_i, y_i) **dla** $i=0,1,\ldots,n$ **. Wtedy warunki (1) dla** Q **są spełnione, bo realizuje je wielomian** P **a funkcja:**

$$\omega_n(x) = \prod_{j=0}^{n} (x - x_j)$$

zeruje się dla każdego $x_i,\ i=0,1,\ldots,n$ **. Warunek dodatkowy:**

$$Q(x_{n+1}) = y_{n+1}$$

daje $y_{n+1} = P(x_{n+1}) + b_{n+1} w_n(x_{n+1})$ **i można wyznaczyć:**

$$b_{n+1} = \frac{y_{n+1} - P(x_{n+1})}{w_n(x_{n+1})}$$

PRZYKŁAD:

x_i	1	2	4
y_i	0	2	12

Powyższe dane interpolują wielomian:

$$x^2 - x$$

Podanie wielomianu interpolującego w $Q(x)$ *dane:*

x_i	1	2	4	5
y_i	0	2	12	20

$$Q(x) = P(x) + b_3(x-x_0)(x-x_1)(x-x_2)$$

Obliczamy podstawiając nowo dodany węzeł x do wzoru wielomianu P(x)

$$P(x_3) = 5^2 - 5 = 20$$

Q(x) to wartość nowo dodanego węzła y

$$Q(x_3) = 20$$

$$b_3 = \frac{Q(x_3) - P(x_3)}{(x_3 - x_0)(x_3 - x_1)(x_3 - x_2)}$$

$$b_3 = \frac{20 - 20}{(5-1)(5-2)(5-4)}$$

$$b_3 = \frac{0}{4 \cdot 3 \cdot 1} = 0$$

$$Q(x) = x^2 - x + 0 = x^2 - x$$

Krótki (bardzo krótki) opis najważniejszych struktur danych oraz procedur i funkcji użytych w algorytmie realizującym metody:

DividedDifferenceNode – Reprezentuje węzeł w drzewie różnicowym. Obiekt posiada:

- Divided_difference – Wartość ilorazu różnicowego
- X – Wartość x'owa węzła, jeśli jest węzłem na najwyższym poziomie, albo wartość x'owa ojca w innym przypadku
- Left_parent – Zmienna wskazująca lewego ojca
- Right_parent – Zmienna wskazująca prawego ojca

Calculate_value – Funckja obliczająca wartość węzła.

Prepare_initial_nodes – Oblicza wartości X dla danej listy wartości Y w zakresie określonym przez parametry a i b. Wartości X są w prosty sposób obliczane przez podzielenie danego zakresu węzłów X, więc są one rozmieszczane na siatce równoodległej.

- Param x_start - Początek zakresu wartości X
- Param x_end - Koniec zakresu wartości X
- Param nodes_y - Lista wartości Y
- Rtype – Zwracana wartość

Calculate_divided_differences_row - Pobiera liste ilorazów różnicowych i oblicza nowy węzeł z każdej pary węzłów. Innymi słowy oblicza następny poziom drzewa Newtona II.

Calculate_divided_differences - Oblicza ilorazy różniocwe dla danych węzłów. Program zakłada, że co najmniej dwa węzły interpolacyjne są wprowadzone. Każda krotka zwracanej listy reprezentuje jeden poziom drzewa ilorazów różnicowych.

Calculate_newton_interpolation - Tworzy wielomian z podanych ilorazów różnicowych. Wielomian jest obliczany na podstawie wzoru przewidzianego w dokumentacji projektu. Zwarca obliczony wielomian Newtona.

Draw_interpolation_plot - Rysuje nowy wykres interpolacyjny dla danego wielomianu interpolacyjnego i danych węzłów.

Add_new_node_to_interpolation - Budowanie nowego wielomianu interpolacyjnego z nowo dodanym węzłem.

Parseargs – Parsowanie argumentów wprowadzonych przez użytkownika tzn. sprawdzanie poprawności wpisanych przez użytkownia wartości. Jest to zabezpieczenie przed wprowadzeniem niepoprawnych dla programu danych.

Opis "wejścia - wyjścia" czyli jakich danych potrzebuje program i z jakimi komunikatami na nie oczekuje, oraz co będzie wynikiem jego działania; w szczególnoúci umieszczamy tu opis zaprogramowanych zabezpieczeń przed wprowadzaniem nieodpowiednich dla algorytmu danych powodujących np. zapętlenie programu czy niewłaściwe obliczenia:

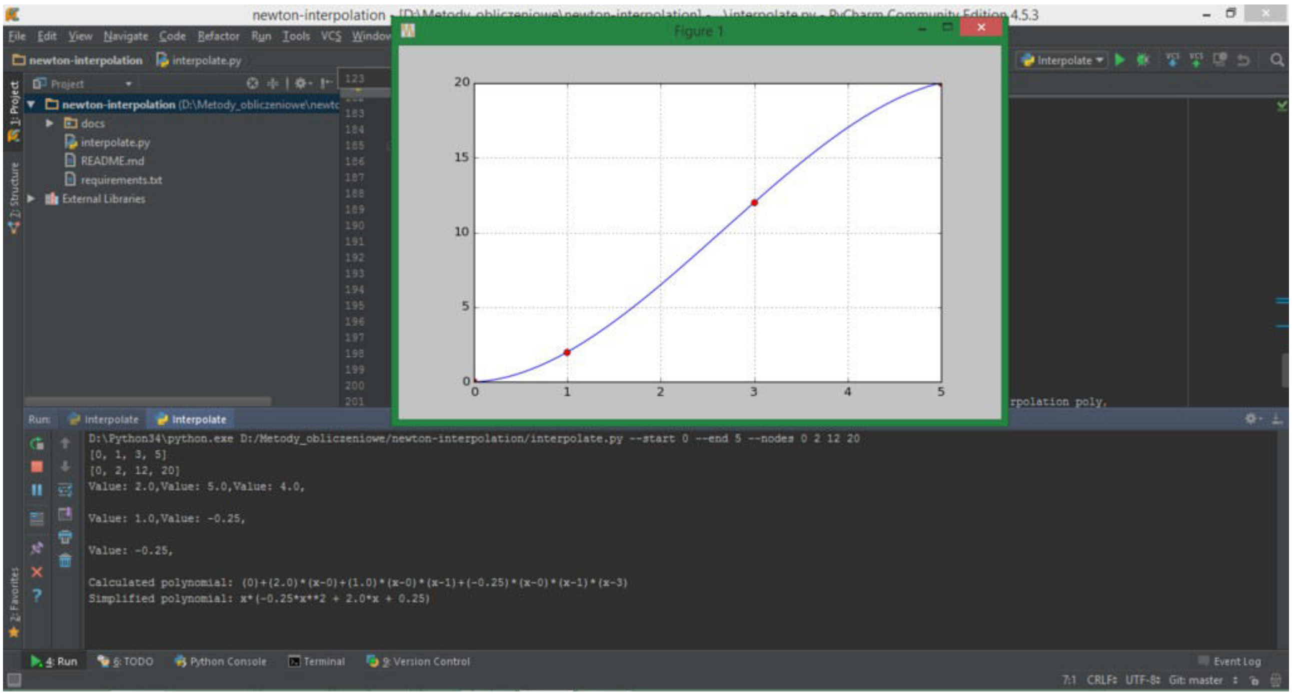
Dane wejściowe:

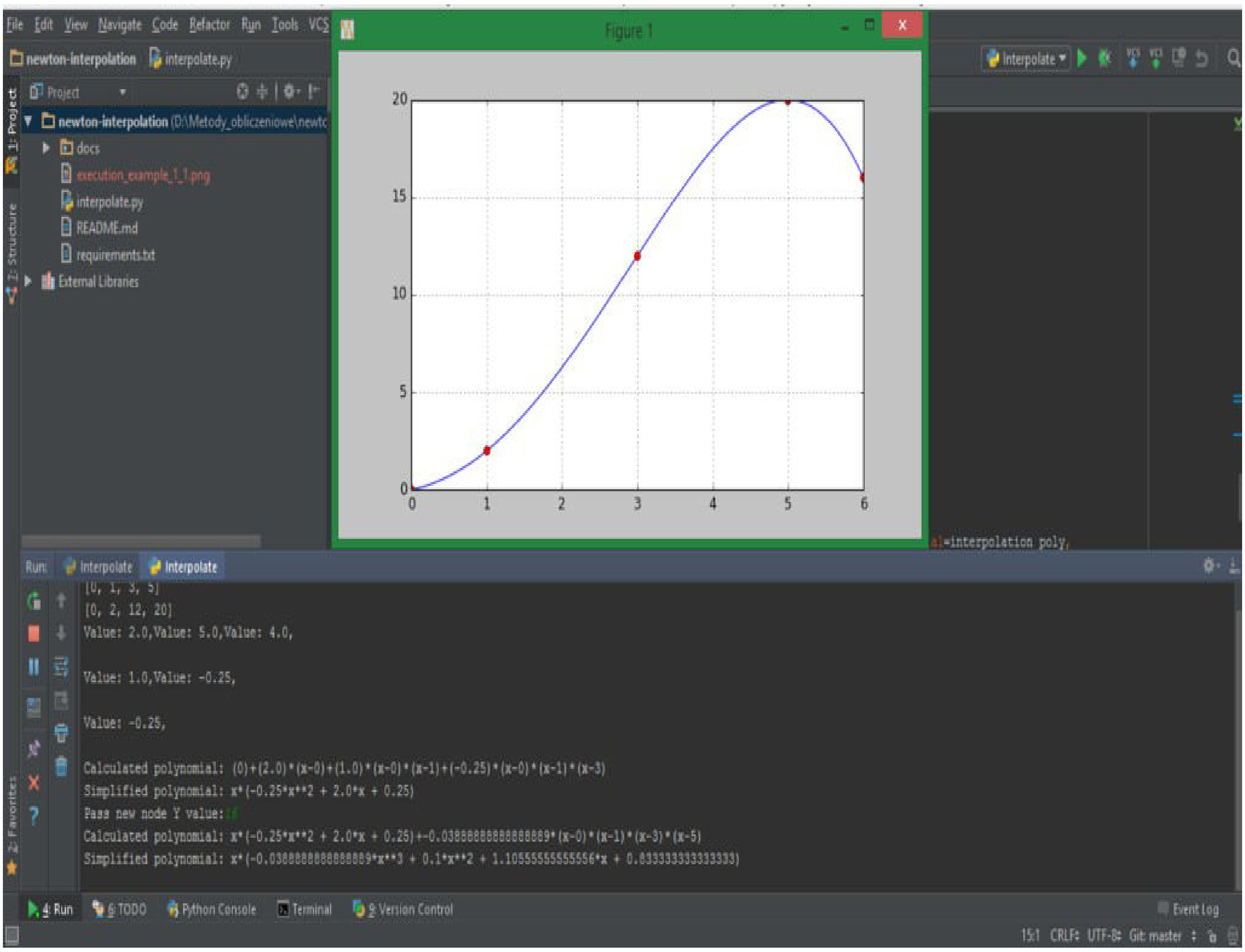
- Start – początek zakresu wartości na osi X (Wartość startowa musi być mniejsza od wartości końcowej, w przeciwnym razie otrzymamy komunikat : „Range of X values must be greater than 0"
- End – koniec zakresu wartości na osi X
- N – Liczbą węzłów
- Nodes – Wartości węzłów Y (Jeśli podamy mniej niż 2 węzły otrzymamy komunikat : „Provide at least two nodes"

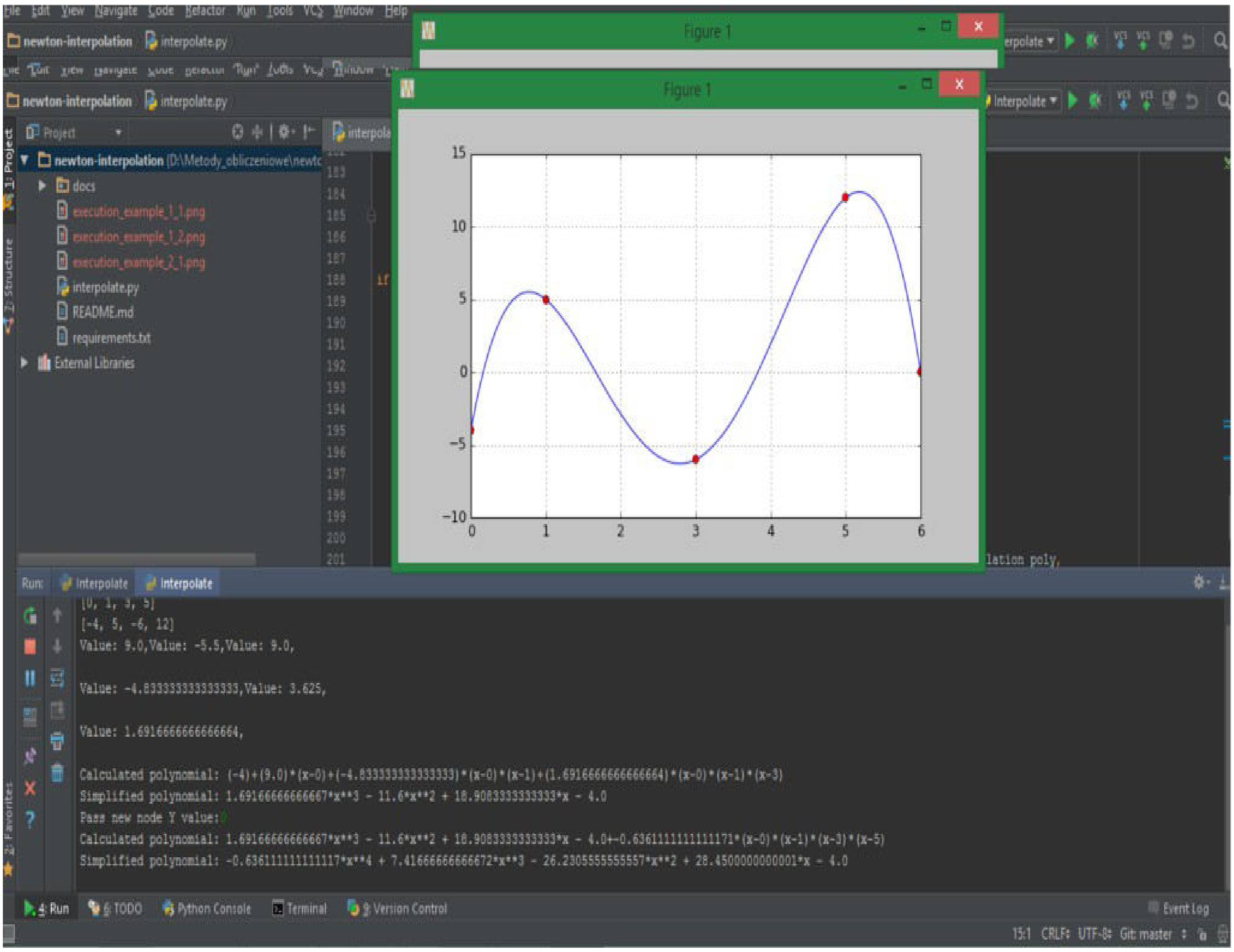
Dane wyjściowe:

- Narysowane na n-elementowej siatce równoodległej wielomiany interpolacyjne
- Wyliczony wielomian interpolacyjny

Wyniki przykładowych ”uruchomień” programu, ilustrujących różne sytuacje:







Tekst kodu źródłowego ostatecznej, poprawnej wersji programu:

```
import sympy as sy
import argparse
from matplotlib import pyplot as plt
import numpy

class DividedDifferenceNode(object):
"""
Represents node in divided differences tree.
"""
divided_difference = None
x = None
left_parent = None
right_parent = None

def __init__(self, left_parent=None, right_parent=None, x=None,
divided_difference=None):
self.left_parent = left_parent
self.right_parent = right_parent
self.x = x
self.divided_difference = divided_difference

def __str__(self):
return "Value: {0},".format(self.divided_difference)
```

```
def calculate_value(self):
self.divided_difference = ((self.right_parent.divided_difference -
self.left_parent.divided_difference) / (
self.get_right_x() - self.get_left_x()))

def get_left_x(self):
if self.x is not None:
return self.x
else:
return self.left_parent.get_left_x()

def get_right_x(self):
if self.x:
return self.x
else:
return self.right_parent.get_right_x()

@staticmethod
def create_child_node(left_parent, right_parent):
return DividedDifferenceNode(left_parent=left_parent,
right_parent=right_parent)


def prepare_initial_nodes(x_start, x_end, nodes_y):
"""
Calculates X values for given list of Y values in range defined by a and b
parameters. X values are
simply calculated by dividing given X range by number of nodes, so they are
distributed in even range.
:param x_start: Start of X values range
:param x_end: End of X values range
:param nodes_y: List of Y values
:return: List of nodes with X and Y values
"""
nodes_x = [float(x_start + ((x_end - x_start) / (len(nodes_y) - 1)) * i) for
i in range(0, len(nodes_y))]
nodes_y = [float(y) for y in nodes_y]
print(nodes_x)
print(nodes_y)
nodes = list(zip(nodes_x, nodes_y))
return nodes


def calculate_divided_differences_row(nodes_to_compute):
"""
Takes list of divided differences nodes and calculates new divided
differences node from each pair
of nodes_to_compute.
In other words, it computes next level of so called Newton's second
interpolation form tree.
:return : list of calculated divided differences
"""
divided_differences = []

if len(nodes_to_compute) == 1:
return None

for i in range(0, len(nodes_to_compute) - 1):
child = DividedDifferenceNode.create_child_node(nodes_to_compute[i],
nodes_to_compute[i + 1])
child.calculate_value()
```

```
divided_differences.append(child)

for node in divided_differences:
print(node, end='')

print('\n')
return divided_differences

def calculate_divided_differences(nodes):
"""
Calculates divided differences for given interpolation nodes.
It is assumed, that at least two interpolation nodes are provided.
Each tuple of returned list represents one level of divided differences
tree.
:return : list of tuples of divided differences
"""
nodes_to_compute = []
divided_differences = []
for node in nodes:
nodes_to_compute.append(DividedDifferenceNode(x=node[0],
divided_difference=node[1]))

divided_differences.append(tuple(nodes_to_compute))

while len(nodes_to_compute) > 1:
next_node_row = calculate_divided_differences_row(nodes_to_compute)
divided_differences.append(tuple(next_node_row))
nodes_to_compute = next_node_row

return divided_differences

def calculate_newton_interpolation(divided_differences):
"""
Creates polynomial from given list of divided differences. Polynomial string
is created according to equation
provided in project docs.
:return : String representing calculated polynomial
"""
polynomial = []
for i, divided_differences_row in enumerate(divided_differences):
polynomial_part =
'({0})'.format(divided_differences_row[0].divided_difference)

for j in range(0, i):
polynomial_part += '*(x-{0})'.format(divided_differences[0][j].x)

polynomial_part += '+'
polynomial.append(polynomial_part)
polynomial_str = ''.join(polynomial)[:-1]

print('Calculated polynomial: {0}'.format(polynomial_str))
# Heuristic simplification of calculated polynomial
simplified_polynomial = sy.simplify(polynomial_str)
print("Simplified polynomial: {0}".format(simplified_polynomial))
return simplified_polynomial

def draw_interpolation_plot(start_x, end_x, interpolation_polynomial, nodes,
freq=200, additional_polynomial=None,
additional_nodes=None):
"""
Draws interpolation plot for given interpolation polynomial and nodes.
```

```
"""
# TODO: calculate figure size dynamically
plt.figure(figsize=(8, 6), dpi=80)
x = numpy.linspace(start_x, end_x, freq)
# TODO: eval should be changed to something more secure (like numexpr
evaluate())...
y = eval(str(interpolation_polynomial))
plt.subplot(211)
plt.plot(x, y, [node[0] for node in nodes], [node[1] for node in nodes],
'ro')
plt.grid(True)

if additional_polynomial:
poly_values = eval(str(additional_polynomial))
plt.subplot(212)
plt.plot(x, poly_values, [node[0] for node in additional_nodes], [node[1]
for node in additional_nodes], 'ro')
plt.grid(True)

plt.show()


def add_new_node_to_interpolation(polynomial, nodes):
new_node = nodes[-1]
# Calculate multiplier
# TODO: change eval to numexpr evaluate()
nominator = (float(new_node[1]) - eval(str(polynomial).replace("x",
str(new_node[0]))))
denominator = 1
for node in nodes[:-1]:
denominator = denominator * (new_node[0]-node[0])

multiplier = (nominator/denominator)

# build new polynomial
new_interpolation_polynomial = list()
new_interpolation_polynomial.append("{0}+({1})".format(str(polynomial),
multiplier))
for node in nodes[:-1]:
new_interpolation_polynomial.append("*(x-{0})".format(node[0]))

new_interpolation_polynomial_str = "".join(new_interpolation_polynomial)
print("Calculated polynomial: {0}".format(new_interpolation_polynomial_str))
new_interpolation_polynomial_str =
sy.simplify(new_interpolation_polynomial_str)
print("Simplified polynomial: {0}".format(new_interpolation_polynomial_str))

return new_interpolation_polynomial_str


def parse_user_provided_float(label):
val = None
while True:
try:
val = float(input("Type {0}:".format(label)))
except ValueError:
print("Type correct {0} value.".format(label))
continue
else:
break

return val
```

```
def parse_user_provided_float_list(label):
val_list = list()
while True:
try:
val_list.append(float(input("Type {0}:".format(label))))
except ValueError:
response = input("Stop (Y/N)?.".format(label))
if response == 'Y':
break
else:
continue

return val_list

def parseargs():
parser = argparse.ArgumentParser(description='Newton\'s Interpolation .')
parser.add_argument('--start', type=float, help='Start of X values range.')
parser.add_argument('--end', type=float, help='End of X values range.')
parser.add_argument('--nodes-y-values', type=float, nargs='+', help='Y
values of interpolation nodes.')
parsed_args = parser.parse_args()

if not parsed_args.start:
parsed_args.start = parse_user_provided_float("start of X values range")

if not parsed_args.end:
parsed_args.end = parse_user_provided_float("end of X values range")

if parsed_args.start >= parsed_args.end:
print("Range of X values must be greater than 0.")
exit(2)

if not parsed_args.nodes_y_values:
print("Enter Y values of interpolation nodes. Type Y to stop.")
parsed_args.nodes_y_values = parse_user_provided_float_list("Y value of
interpolation node")

if len(parsed_args.nodes_y_values) < 2:
print("Provide at least two nodes.")
exit(3)

return parsed_args

if __name__ == '__main__':
args = parseargs()

args.start = int(args.start)
args.end = int(args.end)

init_nodes = prepare_initial_nodes(args.start, args.end,
args.nodes_y_values)
divided_diffs = calculate_divided_differences(init_nodes)
interpolation_poly = calculate_newton_interpolation(divided_diffs)

if input("Add new node (Y/N)?") == "Y":
new_node_x = parse_user_provided_float("X value of new node")
new_node_y = parse_user_provided_float("Y value of new node")

new_initial_nodes = list(init_nodes)
new_initial_nodes.append((float(new_node_x), new_node_y))
new_polynomial = add_new_node_to_interpolation(interpolation_poly,
new_initial_nodes)
```

```
draw_interpolation_plot(start_x=args.start, end_x=args.end,
interpolation_polynomial=interpolation_poly,
nodes=init_nodes, additional_polynomial=new_polynomial,
additional_nodes=new_initial_nodes)
else:
draw_interpolation_plot(start_x=args.start, end_x=args.end,
interpolation_polynomial=interpolation_poly,
nodes=init_nodes)
```