Міністерство освіти і науки України

Харківський національний університет імені В.Н. Каразіна

Лабораторна робота #1

Інтерполяція функцій поліномами за формулами Лагранжа і Ньютона

Виконав:

Захаров Дмитро Олегович

Група МП-31

Харків – 2023

Зміст

1 Постановка задачі

Побудувати інтерполяційний поліном Лагранжа і Ньютона (вперед та назад) $L_n(x)$, $N_n^+(x)$, $N_n^-(x)$ для функції $f(x)$, заданої в вузлах відрізку $[\alpha, \beta]$ значеннями $f(x_i)$, $i \in \{0, \dots, n\}$:

1. $x_k = \alpha + k \cdot h,\ h = \frac{\beta - \alpha}{n},\ k \in \{0, \dots, n\}$
2. $\hat{x}_k = \frac{1}{2}\left(\beta + \alpha - (\beta - \alpha)\cos\frac{2k+1}{2(n+1)}\pi\right),\ k \in \{0, \dots, n\}$

На друк вивести результати у вигляді таблиць:

$$x_i^* \qquad f(x_i^*) \qquad P_n(x_i^*) \qquad |f(x_i^*) - P_n(x_i^*)|$$

де $x_i^* = x_i + \alpha h,\ i \in \{0, \dots, n-1\}, \alpha \in (0, 1)$, P_n кожен з трьох поліномів $L_n(x), N_n^+(x), N_n^-(x)$ для випадків 1 та 2.

Варіант 5.

$$f(x) = e^{\frac{x}{10}} \sin x + x^3 + \cos x,\ \alpha = -2,\ \beta = 2$$

2 Опис методів

2.1 Інтерполяційний поліном Лагранжа

Нехай маємо вузли $\{x_j\}_{j=0}^n \subset [\alpha, \beta]$ та значення функції $f : [\alpha, \beta] \to \mathbb{R}$ в них $\{f_j\}_{j=0}^n := \{f(x_j)\}_{j=0}^n$. Інтерполяційний поліном Лагранжа має вигляд:

$$L_n(x) = \sum_{i=0}^{n} f_i \ell_i(x),\ \ell_i(x) \triangleq \prod_{j \neq i}^{n} \frac{x - x_j}{x_i - x_j}$$

2.2 Інтерполяційний поліном Ньютона

Нехай маємо вузли $\{x_j\}_{j=0}^n \subset [\alpha, \beta]$ та значення функції $f : [\alpha, \beta] \to \mathbb{R}$ в них $\{f_j\}_{j=0}^n := \{f(x_j)\}_{j=0}^n$. Тоді, інтерполяційні поліноми Ньютона вперед та назад ($N_n^+(x)$ та $N_n^-(x)$, відповідно) мають вид:

$$N_n^+(x) = \varphi_0 + \sum_{k=1}^{n} \varphi_{[0:k]} \prod_{i=0}^{k-1} (x - x_i),\ N_n^-(x) = \varphi_n + \sum_{k=1}^{n} \varphi_{[n:n-k]} \prod_{i=0}^{k-1} (x - x_{n-i})$$

де

$$y_{[i:j]} \triangleq \frac{y_{[i+1:j]} - y_{[i:j-1]}}{x_j - x_i}$$

Значення розділених різниць можна рахувати і рекурсивно, але значно легше використати наступну формулу:

$$y_{[i:i+k]} = \sum_{j=0}^{k} \frac{y_{i+j}}{\prod_{l=0, l \neq j}^{k} (x_{i+j} - x_{i+l})}$$

3 Текст програми

Повний текст програми можна знайти за цим посиланням (← напис клікабельний) на *Github* сторінку.

3.1 Генерація вузлів

Для початку, створимо файл `generators.py`, завантажимо залежності та створимо свій тип для інтервалу:

```python
from math import cos, pi
from typing import Tuple, TypeAlias, List
from abc import ABC, abstractmethod

Interval: TypeAlias = Tuple[float, float]
```

Лістинг 1: Завантаження залежностей

Зробимо генерацію вузлів через абстрактний клас, котрий буде вміти створюватись та генерувати набір x координат вузлів через функцію `generate_nodes`. Також, тут же задамо функцію `generate_test_points`, котра буде видавати набір x координат точок, на котрих ми будемо оцінювати інтерполяцію (як сказано в умові, використовуючи формулу $x_i^* = x_i + \alpha h$ де x_i це координата вузла).

```python
class IDataPointsGenerator(ABC):
    """Interface for generating data points"""

    def __init__(self, interval: Interval, number: int) -> None:
        """
        Function initializing the generator

        ### Args:
        - interval (‘Interval‘): Interval on which the data points will be generated
        - number (‘int‘): Number of data points to generate
        """

        assert number > 0, "Number of data points must be greater than 0"
        assert interval[0] < interval[1], "Lower bound must be less than upper bound"

        self._lower, self._upper = interval
        self._number = number

    @abstractmethod
    def generate_nodes(self) -> List[float]:
        """
        Function generating data points
```

```
        ### Returns:
            `List[float]`: List of generated x coordinates
        """
        pass

    @abstractmethod
    def generate_test_points(self, alpha: float = 0.1) -> List[float]:
        """
        Function generating test points for evaluating the polynomial
        ### Args:
        - alpha (`float`): We will evaluate the polynomial at points
    x_i + alpha*h

        Returns:
        - List[float]: List of x coordinates
        """

        nodes = self.generate_nodes()
        h = (self._upper - self._lower) / self._number
        return [node + alpha*h for node in nodes]
```

Лістинг 2: Задання абстрактного класу

Далі, наводимо конкретні реалізації цього інтерфейсу.

3.1.1 Лінійно розбитий проміжок

```
class LinearDataPointsGenerator(IDataPointsGenerator):
    """Class for generating linearly spaced data points"""

    def __init__(self, interval: Interval, number: int) -> None:
        super().__init__(interval, number)

    def generate_nodes(self) -> List[float]:
        fn = lambda i: self._lower + i * (self._upper - self._lower) /
    (self._number)
        return [fn(i) for i in range(self._number + 1)]

    def generate_test_points(self, alpha: float = 0.1) -> List[float]:
        return super().generate_test_points(alpha)
```

Лістинг 3: Генерація лінійно розкинутих точок

3.1.2 Проміжок розбитий по гармонічному закону

```
class CosineDataPointsGenerator(IDataPointsGenerator):
    """Class for generating cosine spaced data points"""

    def __init__(self, interval: Interval, number: int) -> None:
```

```
        super().__init__(interval, number)

    def generate_nodes(self) -> List[float]:
        fn = lambda i: 0.5*(self._lower+self._upper-(self._upper-self._lower)*cos((2*i+1)*pi/(2*(self._number+1))))
        return [fn(i) for i in range(self._number + 1)]

    def generate_test_points(self, alpha: float = 0.1) -> List[float]:
        return super().generate_test_points(alpha)
```

Лістинг 4: Генерація точок по закону з косинусом

3.1.3 Перевірка генерації

Перевіримо роботу програми, поклавши $n := 4$ для нашого конкретного інтервалу $[-2, 2]$. В коді, це виглядає так:

```
linear_generator = LinearDataPointsGenerator((-2.0, 2.0), 4)
cosine_generator = CosineDataPointsGenerator((-2.0, 2.0), 4)

print(f"Linear generator nodes: {linear_generator.generate_nodes()}")
print(f"Cosine generator nodes: {cosine_generator.generate_nodes()}")
```

Лістинг 5: Використання генераторів

Якщо запустити, отримаємо наступний результат:

```
Linear generator nodes: [-2.0, -1.0, 0.0, 1.0, 2.0]
Cosine generator nodes: [-1.902113032590307, -1.1755705045849463, -1.2246467991473532e-16, 1.175570504584946, 1.902113032590307]
```

Лістинг 6: Результат запуску генераторів

Перевіримо аналітично. У випадку лінійного розбиття, маємо:

$$x_k = -2 + k \cdot \frac{2+2}{4} = -2 + k$$

Дійсно, якщо підставляти $k \in \{0, \dots, 4\}$, отримаємо точки $\{-2, -1, 0, 1, 2\}$.

У випадку розбиття по косинусу:

$$\hat{x}_k = \frac{1}{2}\left(2 - 2 - (2+2)\cos\frac{2k+1}{10}\pi\right) = -2\cos\frac{(2k+1)\pi}{10}$$

Тому, наприклад, $\hat{x}_0 = -2\cos\frac{\pi}{10} \approx -1.902$. Аналогічно можна перевірити схожість для інших k. Отже, ми дійсно отримали схожі результати.

3.2 Поліноми

Створимо файл `polynomials.py` та завантажимо залежності:

```python
from math import prod
from abc import ABC, abstractmethod
from typing import List, Tuple, TypeAlias

Point: TypeAlias = Tuple[float, float]
```

Лістинг 7: Завантаження залежностей

Знову створимо абстракцію для поліномів. Кожен поліном має мати конструктор, а також функцію `evaluate`, котра рахує значення полінома в заданій точці:

```python
class IPolynomial(ABC):
    """Interface any polymial should implement"""

    def __init__(self, points: List[Point]) -> None:
        """
        Initializes the polynomial
        ### Args:
        - points (`List[Point]`): List of points to interpolate on
        """
        assert len(points) > 1, "At least two points are required"
        self._points = points

    @abstractmethod
    def evaluate(self, x: float) -> float:
        """
        Evaluates the polynomial at x
        ### Args:
        - x (`float`): Point to evaluate the polynomial at
        ### Returns:
            `float`: Value of the polynomial at x
        """
        pass
```

Лістинг 8: Інтерфейс для поліномів

Далі, наводимо конкретні реалізації для кожного з поліномів.

3.2.1 Поліном Лагранжа

```python
class LagrangePolynomial(IPolynomial):
    """Polynomial implementing Lagrange interpolation"""

    def __init__(self, points: List[Point]) -> None:
        super().__init__(points)

    def _lagrange_coefficient(self, i: int, x: float) -> float:
```

```
        """ Calculates the i-th Lagrange coefficient at x

        ### Args:
        - i (‘int‘): Number of the Lagrange coefficient to calculate
        - x (‘float‘): Point to calculate the Lagrange coefficient at

        ### Returns:
            float: Value of the i-th Lagrange coefficient at x
        """
        # Finding the i-th point's x coordinate
        x_i, _ = self._points[i]
        # Forming an array of product terms
        terms = [(x-x_j) / (x_i-x_j) for (j, (x_j,_)) in enumerate(self._points) if i!=j]
        # Finding their product
        return prod(terms)

    def evaluate(self, x: float) -> float:
        # Finding the y coordinate of each point and multiplying it by the corresponding Lagrange coefficient
        terms = [self._lagrange_coefficient(i, x) * y for (i, (_, y)) in enumerate(self._points)]
        # Summing the terms
        return sum(terms)
```

Лістинг 9: Реалізація полінома Лагранжа

3.2.2 Поліном Ньютона назад

Тут, при інціалізації (функція `__init__`), ми спочатку рахуємо і зберігаємо набір роздільних різниць за допомогою внутрішньої функції `_differential_difference(self, i: int, j: int) -> float`, а далі при обрахунку значення (функція `evaluate`) використовуємо збереженні значення

```
class NewtonBackwardsPolynomial(IPolynomial):
    """Polynomial implementing Newton's lower interpolation"""

    def __init__(self, points: List[Point]) -> None:
        # Calculating all differential differences beforehand
        super().__init__(points)
        n = len(points)
        self._differences = [self._differential_difference(n-1, i) for i in reversed(range(n))]

    def _differential_difference(self, i: int, j: int) -> float:
        """ Finds the differential difference y[i:j] at given x

        ### Args:
        - x (‘float‘): Point to find the differential difference at
        - i (‘int‘): From which index to start
```

```
        - j (`int`): At which index to stop

        ### Returns:
            `float`: Value of a differential difference at x
        """
        assert i >= 0, "i must be positive"
        assert j <= len(self._points), "j must be less than the number of points"

        # It is easier to deal with indices i,...,i-k
        k = i - j
        sum_terms: List[float] = [] # Defining a list of terms to sum
        for j in range(k+1):
            # Finding the denominator product terms
            denominator_terms = [self._points[i-j][0] - self._points[i-l][0] for l in range(k+1) if l!=j]
            # Dividing the numerator (self._points[i+j][1]) by the product of denominator terms
            sum_terms.append(self._points[i-j][1] / prod(denominator_terms))

        return sum(sum_terms)

    def evaluate(self, x: float) -> float:
        # Defining a list of terms to sum
        n = len(self._points)
        sum_terms: List[float] = []
        for i in reversed(range(n)):
            product_terms = [x - X for (X, _) in self._points[i+1:n]]
            # Finding the differential difference y[0:i+1] at x
            sum_terms.append(self._differences[n-1-i] * prod(product_terms))

        return sum(sum_terms)
```

Лістинг 10: Реалізація полінома Ньютона назад

3.2.3 Поліном Ньютона вперед

Реалізація ідейно така сама, як і в попередній секції.

```
class NewtonForwardPolynomial(IPolynomial):
    """Polynomial implementing Newton's upper interpolation"""

    def __init__(self, points: List[Point]) -> None:
        # Calculating all differential differences beforehand
        super().__init__(points)
        self._differences = [self._differential_difference(0, i) for i in range(len(points))]

    def _differential_difference(self, i: int, j: int) -> float:
```

```
        """ Finds the differential difference y[i:j] at given x

        ### Args:
        - x (`float`): Point to find the differential difference at
        - i (`int`): From which index to start
        - j (`int`): At which index to stop

        ### Returns:
            `float`: Value of a differential difference at x
        """
        assert i >= 0, "i must be greater than 0"
        assert j <= len(self._points), "j must not exceed the number of points"

        # It is easier to deal with indices i,...,i+k
        k = j - i

        sum_terms: List[float] = [] # Defining a list of terms to sum
        for j in range(k+1):
            # Finding the denominator product terms
            denominator_terms = [self._points[i+j][0] - self._points[i+l][0] for l in range(k+1) if l!=j]
            # Dividing the numerator (self._points[i+j][1]) by the product of denominator terms
            sum_terms.append(self._points[i+j][1] / prod(denominator_terms))

        return sum(sum_terms)

    def evaluate(self, x: float) -> float:
        # Defining a list of terms to sum
        sum_terms: List[float] = []
        for i in range(len(self._points)):
            product_terms = [x - x_j for (x_j, _) in self._points[:i]]
            # Finding the differential difference y[0:i+1] at x
            sum_terms.append(self._differences[i] * prod(product_terms))

        return sum(sum_terms)
```

Лістинг 11: Реалізація полінома Ньютона вперед

3.3 Програма для оцінки

Тепер напишемо програму, котра оцінює поліноми і видає потрібні нам таблиці.

Використаємо пакет `rich` для красивого виведення таблиць у консоль :)

Далі алгоритм простий: створюємо масив (або, аналогічно, словник) з генераторів та поліномів, а потім попарно проходимось і генеруємо 6 таблиць. Скоріше за все,

логіку з оцінювання по конкретним `IPolynomial` та `IDataPointsGenerator` можна було виділити окремо (власне, для цього і існують інтефрейси та абстрактні класи). Проте, не будемо і далі ускладнювати структуру програми і просто запишемо все в одній функції:

```
# Math imports
from math import exp, sin, cos

# Internal imports
from generators import IDataPointsGenerator, LinearDataPointsGenerator, CosineDataPointsGenerator
from polynomials import LagrangePolynomial, NewtonForwardPolynomial, NewtonBackwardsPolynomial

# Rich logging
from rich.console import Console
from rich.table import Table

if __name__ == "__main__":
    # Defining the task parameters
    interval = (-2.0, 2.0)
    segments_number = 20
    alpha = 0.3
    fn = lambda x: exp(x / 10.0) * sin(x) + x**3 + cos(x)

    # Defining the generator and defining a set of points
    generators: dict[str, IDataPointsGenerator] = {
        "linear generation": LinearDataPointsGenerator(interval, segments_number),
        "cosine generation": CosineDataPointsGenerator(interval, segments_number)
    }

    # For rich logging
    console = Console()

    for generator_name, generator in generators.items():
        # Defining the points on which to interpolate the polynomial
        node_x = generator.generate_nodes()
        node_points = [(x, fn(x)) for x in node_x]

        # Defining the polynomials
        polynomials = {
            "Lagrange Polynomial": LagrangePolynomial(node_points),
            "Newton Forward Polynomial": NewtonForwardPolynomial(node_points),
            "Newton Backwards Polynomial": NewtonBackwardsPolynomial(node_points)
        }

```

```
        # Defining the test points
        test_x = generator.generate_test_points(alpha=alpha)
        test_points = [(x, fn(x)) for x in test_x]

        for polynomial_name, polynomial in polynomials.items():
            table = Table(title=f"{polynomial_name} evaluation using {generator_name}")
            table.add_column("x", justify="center", style="cyan", no_wrap=True)
            table.add_column("f(x)", justify="center", style="magenta")
            table.add_column("P(x)", justify="center", style="green")
            table.add_column("|f(x)-P(x)|", justify="center", style="blue")

            for test_point in test_points:
                x_label = "{:.18f}".format(test_point[0])
                f_x_label = "{:.18f}".format(test_point[1])
                p_x_label = "{:.18f}".format(polynomial.evaluate(test_point[0]))
                difference_label = "{:.18f}".format(abs(test_point[1] - polynomial.evaluate(test_point[0])))

                table.add_row(x_label, f_x_label, p_x_label, difference_label)

            console.print(table)
```

Лістинг 12: Реалізація побудови таблиць з результатами

4 Результати

Для експериментів ми взяли $n = 20$ та $\alpha = 0.3$. Якщо ви хочете спробувати вибрати інші параметри, то можете запустити програму, що прикріплена у додатку :)

4.1 Лінійно розбитий проміжок

Lagrange Polynomial evaluation using linear generation

x	f(x)	P(x)	|f(x)-P(x)|
-1.939999999999999947	-8.430412608078702519	-8.430412608079912218	0.000000000001209699
-1.739999999999999991	-6.264718215887008057	-6.264718215887046249	0.000000000000038192
-1.540000000000000036	-4.478338069773848851	-4.478338069773834640	0.000000000000014211
-1.339999999999999858	-3.028751100403195728	-3.028751100403194396	0.000000000000001332
-1.139999999999999902	-1.874684961932285399	-1.874684961932284510	0.000000000000000888
-0.939999999999999947	-0.975902193275575591	-0.975902193275576035	0.000000000000000444
-0.739999999999999991	-0.292947538298825694	-0.292947538298825971	0.000000000000000278
-0.540000000000000036	0.213135735763524070	0.213135735763524126	0.000000000000000056
-0.339999999999999913	0.581111545100612692	0.581111545100612581	0.000000000000000111
-0.139999999999999958	0.849868873543164649	0.849868873543164205	0.000000000000000444
0.059999999999999998	1.058741411967588197	1.058741411967588641	0.000000000000000444
0.260000000000000175	1.247818275597583959	1.247818275597583515	0.000000000000000444
0.459999999999999909	1.458235200121894115	1.458235200121894337	0.000000000000000222
0.660000000000000142	1.732436033520094787	1.732436033520094343	0.000000000000000444
0.859999999999999876	2.114395089066301914	2.114395089066303246	0.000000000000001332
1.060000000000000053	2.649791991222090193	2.649791991222091525	0.000000000000001332
1.260000000000000231	3.386132005438906578	3.386132005438909243	0.000000000000002665
1.459999999999999964	4.372806461384308285	4.372806461384302956	0.000000000000005329
1.660000000000000142	5.661089714230100434	5.661089714230129744	0.000000000000029310
1.859999999999999876	7.304071090027932200	7.304071090028141811	0.000000000000209610
2.060000000000000053	9.356522371777739266	9.356522371769131041	0.000000000008608225

Рис. 1: Результат для полінома Лагранжа для лінійного розбиття

Newton Forward Polynomial evaluation using linear generation

x	f(x)	P(x)	|f(x)−P(x)|
−1.939999999999999947	−8.430412608078702519	−8.430412608078617254	0.000000000000085265
−1.739999999999999991	−6.264718215887008057	−6.264718215887017827	0.000000000000009770
−1.540000000000000036	−4.478338069773848851	−4.478338069773848851	0.000000000000000000
−1.339999999999999858	−3.028751100403195728	−3.028751100403198393	0.000000000000002665
−1.139999999999999902	−1.874684961932285399	−1.874684961932294502	0.000000000000009104
−0.939999999999999947	−0.975902193275575591	−0.975902193275610008	0.000000000000034417
−0.739999999999999991	−0.292947538298825694	−0.292947538298935328	0.000000000000109635
−0.540000000000000036	0.213135735763524070	0.213135735763201634	0.000000000000322437
−0.339999999999999913	0.581111545100612692	0.581111545099688098	0.000000000000924594
−0.139999999999999958	0.849868873543164649	0.849868873540588932	0.000000000002575717
0.059999999999999998	1.058741411967588197	1.058741411960700596	0.000000000006887602
0.260000000000000175	1.247818275597583959	1.247818275579912539	0.000000000017671420
0.459999999999999909	1.458235200121894115	1.458235200078044302	0.000000000043849813
0.660000000000000142	1.732436033520094787	1.732436033413022658	0.000000000107072129
0.859999999999999876	2.114395089066301914	2.114395088804005507	0.000000000262296407
1.060000000000000053	2.649791991222090193	2.649791990569838607	0.000000000652251586
1.260000000000000231	3.386132005438906578	3.386132003800092605	0.000000001638813973
1.459999999999999964	4.372806461384308285	4.372806457304263539	0.000000004080044747
1.660000000000000142	5.661089714230100434	5.661089704441307546	0.000000009788792887
1.859999999999999876	7.304071090027932200	7.304071068188909166	0.000000021839023034
2.060000000000000053	9.356522371777739266	9.356522329161711227	0.000000042616028040

Рис. 2: Результат для полінома Ньютона вперед для лінійного розбиття

Newton Backwards Polynomial evaluation using linear generation

x	f(x)	P(x)	|f(x)−P(x)|
−1.939999999999999947	−8.430412608078702519	−8.430412618982876083	0.000000010904173564
−1.739999999999999991	−6.264718215887008057	−6.264718217527679656	0.000000001640671599
−1.540000000000000036	−4.478338069773848851	−4.478338069532321164	0.000000000241527687
−1.339999999999999858	−3.028751100403195728	−3.028751100104076333	0.000000000299119396
−1.139999999999999902	−1.874684961932285399	−1.874684961808193773	0.000000000124091626
−0.939999999999999947	−0.975902193275575591	−0.975902193252467520	0.000000000023108071
−0.739999999999999991	−0.292947538298825694	−0.292947538308474975	0.000000000009649281
−0.540000000000000036	0.213135735763524070	0.213135735750189764	0.000000000013334306
−0.339999999999999913	0.581111545100612692	0.581111545091287263	0.000000000009325429
−0.139999999999999958	0.849868873543164649	0.849868873538095593	0.000000000005069056
0.059999999999999998	1.058741411967588197	1.058741411965345325	0.000000000002242873
0.260000000000000175	1.247818275597583959	1.247818275596787485	0.000000000000796474
0.459999999999999909	1.458235200121894115	1.458235200121668740	0.000000000000225375
0.660000000000000142	1.732436033520094787	1.732436033520038610	0.000000000000056177
0.859999999999999876	2.114395089066301914	2.114395089066278377	0.000000000000023537
1.060000000000000053	2.649791991222090193	2.649791991222074206	0.000000000000015987
1.260000000000000231	3.386132005438906578	3.386132005438895032	0.000000000000011546
1.459999999999999964	4.372806461384308285	4.372806461384309173	0.000000000000000888
1.660000000000000142	5.661089714230100434	5.661089714230070236	0.000000000000030198
1.859999999999999876	7.304071090027932200	7.304071090028236846	0.000000000000304645
2.060000000000000053	9.356522371777739266	9.356522371769781188	0.000000000007958079

Рис. 3: Результат для полінома Ньютона назад для лінійного розбиття

4.2 Проміжок розбитий по гармонічному закону

Lagrange Polynomial evaluation using cosine generation

x	f(x)	P(x)	|f(x)-P(x)|
-1.934407594362360205	-8.364311412317857020	-8.364311412317860572	0.000000000000003553
-1.889855824363647185	-7.849417908658865350	-7.849417908658865350	0.000000000000000000
-1.801747497288408439	-6.890856680001411050	-6.890856680001410162	0.000000000000000888
-1.672050807568877362	-5.617416988247773801	-5.617416988247774690	0.000000000000000888
-1.503662964936059554	-4.191157135139539136	-4.191157135139537360	0.000000000000001776
-1.300345475541838880	-2.777734004913427146	-2.777734004913426258	0.000000000000000888
-1.066640116127243854	-1.517467838569368510	-1.517467838569369842	0.000000000000001332
-0.807767478235116299	-0.502605792607142998	-0.502605792607143220	0.000000000000000222
-0.529510348821808252	0.235529300727764457	0.235529300727764707	0.000000000000000250
-0.238084532352349276	0.728002946559862196	0.728002946559862307	0.000000000000000111
0.059999999999999873	1.058741411967588197	1.058741411967587753	0.000000000000000444
0.358084532352348606	1.345743592684043755	1.345743592684043533	0.000000000000000222
0.649510348821808359	1.715767340607336600	1.715767340607338376	0.000000000000001776
0.927767478235116183	2.276284502078854910	2.276284502078854910	0.000000000000000000
1.186640116127244182	3.089626085588164983	3.089626085588162763	0.000000000000002220
1.420345475541838542	4.154858319891424401	4.154858319891426177	0.000000000000001776
1.623662964936059439	5.402238116765907705	5.402238116765910370	0.000000000000002665
1.792050807568877024	6.702725448732641311	6.702725448732640423	0.000000000000000888
1.921747497288408768	7.891453725521418328	7.891453725521416551	0.000000000000001776
2.009855824363647070	8.800410448902765026	8.800410448902743710	0.000000000000021316
2.054407594362360090	9.293067450377893834	9.293067450377982652	0.000000000000088818

Рис. 4: Результат для полінома Лагранжа для розбиття по косинусу

Newton Forward Polynomial evaluation using cosine generation

x	f(x)	P(x)	|f(x)−P(x)|
−1.934407594362360205	−8.364311412317857020	−8.364311412317857020	0.000000000000000000
−1.889855824363647185	−7.849417908658865350	−7.849417908658867127	0.000000000000001776
−1.801747497288408439	−6.890856680001411050	−6.890856680001411938	0.000000000000000888
−1.672050807568877362	−5.617416988247773801	−5.617416988247784460	0.000000000000010658
−1.503662964936059554	−4.191157135139539136	−4.191157135139547130	0.000000000000007994
−1.300345475541838880	−2.777734004913427146	−2.777734004913424926	0.000000000000002220
−1.066640116127243854	−1.517467838569368510	−1.517467838569417138	0.000000000000048628
−0.807767478235116299	−0.502605792607142998	−0.502605792607725865	0.000000000000582867
−0.529510348821808252	0.235529300727764457	0.235529300722015195	0.000000000005749262
−0.238084532352349276	0.728002946559862196	0.728002946509442750	0.000000000050419446
0.059999999999999873	1.058741411967588197	1.058741411640918839	0.000000000326669358
0.358084532352348606	1.345743592684043755	1.345743591104725745	0.000000001579318010
0.649510348821808359	1.715767340607336600	1.715767334640047048	0.000000005967289551
0.927767478235116183	2.276284502078854910	2.276284483743454601	0.000000018335400309
1.186640116127244182	3.089626085588164983	3.089626038414448495	0.000000047173716489
1.420345475541838542	4.154858319891424401	4.154858216194520004	0.000000103696904397
1.623662964936059439	5.402238116765907705	5.402237919425938451	0.000000197339969255
1.792050807568877024	6.702725448732641311	6.702725120904147182	0.000000327828494129
1.921747497288408768	7.891453725521418328	7.891453247646214031	0.000000477875204297
2.009855824363647070	8.800410448902765026	8.800409835611555209	0.000000613291209817
2.054407594362360090	9.293067450377893834	9.293066755968744985	0.000000694409148849

Рис. 5: Результат для полінома Ньютона вперед для розбиття по косинусу

Newton Backwards Polynomial evaluation using cosine generation

x	f(x)	P(x)	|f(x)−P(x)|
−1.934407594362360205	−8.364311412317857020	−8.364311392289165425	0.000000020028691594
−1.889855824363647185	−7.849417908658865350	−7.849417890183542745	0.000000018475322605
−1.801747497288408439	−6.890856680001411050	−6.890856665587505425	0.000000014413905625
−1.672050807568877362	−5.617416988247773801	−5.617416980014572125	0.000000008233201676
−1.503662964936059554	−4.191157135139539136	−4.191157132851595080	0.000000002287944056
−1.300345475541838880	−2.777734004913427146	−2.777734006154855440	0.000000001241428293
−1.066640116127243854	−1.517467838569368510	−1.517467840701276494	0.000000002131907983
−0.807767478235116299	−0.502605792607142998	−0.502605794202285128	0.000000001595142129
−0.529510348821808252	0.235529300727764457	0.235529299905034434	0.000000000822730023
−0.238084532352349276	0.728002946559862196	0.728002946239876492	0.000000000319985705
0.059999999999999873	1.058741411967588197	1.058741411871386262	0.000000000096201935
0.358084532352348606	1.345743592684043755	1.345743592662151045	0.000000000021892710
0.649510348821808359	1.715767340607336600	1.715767340604544833	0.000000000002791767
0.927767478235116183	2.276284502078854910	2.276284502079564120	0.000000000000709210
1.186640116127244182	3.089626085588164983	3.089626085588826232	0.000000000000661249
1.420345475541838542	4.154858319891424401	4.154858319891669538	0.000000000000245137
1.623662964936059439	5.402238116765907705	5.402238116765960108	0.000000000000052403
1.792050807568877024	6.702725448732641311	6.702725448732646640	0.000000000000005329
1.921747497288408768	7.891453725521418328	7.891453725521419216	0.000000000000000888
2.009855824363647070	8.800410448902765026	8.800410448902765026	0.000000000000000000
2.054407594362360090	9.293067450377893834	9.293067450377904493	0.000000000000010658

Рис. 6: Результат для полінома Ньютона назад для розбиття по косинусу

5 Висновки

В цій лабораторній роботі ми:

- навчилися будувати інтерполяційні поліноми Лагранжа, Ньютона вперед та Ньютона назад;
- писати комп'ютерну програму (на прикладі мови `Python`), що будує вищезгадані поліноми;
- оцінювати написану програму та діставати дані з експериментів.

Тепер більш детально про аналіз результатів. Як бачимо, в усіх випадках модуль різниці $|f(x^*)-P(x^*)|$ – дуже мала величина, що говорить про точність інтерполяції на заданому проміжку (оскільки ми брали набір точок x_i^*, що відносно близький до вузлів x_i).

Порівняти 3 поліноми по точності доволі складно, оскільки в усіх випадках різниця дуже маленька (часто порядком нижче за 10^{-8}): навіть якщо зменшити n до 5 з $\alpha = 0.3$, то різниця $|P_1(x_i^*) - P_2(x_i^*)|$ буде дуже маленькою для будь-яких двох поліномів P_1, P_2.

Проте, можна сказати про складність обчислень. Поліном Лагранжа не вимагає зберігання данних і обчислення потребує $\mathcal{O}(n^2)$ операцій (потрібно знайти суму $n+1$ доданків, кожен з яких містить ще порядка n доданків у добутку).

Що стосується поліномів Ньютона, то для них потрібно зберігати розділені різниці, кількість яких $\mathcal{O}(n)$. Також, обрахунок кожної різниці займає $\mathcal{O}(k^2)$ операцій (де $k \in \{1, \dots, n\}$), а враховуючи, що їх $\mathcal{O}(n)$, то складність ініціалізації $\mathcal{O}(n^3)$.

Проте, як і для полінома Лагранжа, складність обрахунку, маючи ці коефіцієнти, становить $\mathcal{O}(n^2)$. Проте, для полінома Ньютона легше додавання нової точки, в той час як для полінома Лагранжа це проблематично.