**Федеральное государственное автономное образовательное учреждение**
**высшего образования**
**«Национальный исследовательский университет ИТМО»**
**Факультет Программной Инженерии и Компьютерной Техники**

**Вариант №9**
**Лабораторная работа №4**
по теме
**Аппроксимация функции методом наименьших квадратов**
по дисциплине
**Вычислительная математика**

Выполнил Студент группы P3212
**Кобелев Р.П.**
к. т. н. Преподаватель:
**Наумова Н.А.**

г. Санкт-Петербург
2024г.

# Содержание

**1 Цель работы** **2**

**2 Порядок выполнения работ** **2**

**3 Вычислительная реализация задачи** **3**
3.1 Табулирование функции . . . 3
3.2 Линейное и квадратичное приближения . . . 3
3.2.1 Линейное приближение . . . 3
3.2.2 Квадратичное приближение . . . 3
3.2.3 Нахождение СКО . . . 4
3.2.4 Расчет и Сравнение СКО . . . 4
3.3 Выбор наилучшего приближения . . . 4
3.4 Построение графиков . . . 4
3.5 Вывод . . . 5

**4 Программная реализация задачи** **6**
4.1 Листинг программы . . . 9
4.2 Пример работы программы . . . 14

**5 Github** **14**

**6 Вывод** **14**

# 1 Цель работы

Найти функцию, являющуюся наилучшим приближением заданной табличной функции по методу наименьших квадратов.

# 2 Порядок выполнения работ

**Программная реализация задачи:** Для исследования использовать:

- линейную функцию,
- полиномиальную функцию 2-й степени,
- полиномиальную функцию 3-й степени,
- экспоненциальную функцию,
- логарифмическую функцию,
- степенную функцию.

**Методика проведения исследования:**

1. Вычислить меру отклонения: $S = \sum_{i=1}^{n} [\varphi(x_i) - y_i]^2$ для всех исследуаемых функций;
2. Уточнить значения коэффициентов эмпирических функций, минимизируя функцию $S$;
3. Сформировать массивы предполагаемых эмпирических зависимостей $(\varphi(x_i), \varepsilon_i)$;
4. Определить среднеквадратичное отклонение для каждой аппроксимирующей функции. Выбрать наименьшее значение и, следовательно, наилучшее приближение;
5. Построить графики полученных эмпирических функций.

# 3 Вычислительная реализация задачи

## 3.1 Табулирование функции

Первым делом сформируем таблицу значений функции $y = \frac{4x}{x^4+9}$ на интервале $[0; 2]$ с шагом $h = 0.2$.

| $x$ | $y$ |
|---|---|
| 0.0 | 0.000 |
| 0.2 | 0.0888 |
| 0.4 | 0.1772 |
| 0.6 | 0.2628 |
| 0.8 | 0.34 |
| 1.0 | 0.4 |
| 1.2 | 0.4334 |
| 1.4 | 0.436 |
| 1.6 | 0.4114 |
| 1.8 | 0.3692 |
| 2.0 | 0.32 |

## 3.2 Линейное и квадратичное приближения

Теперь найдем линейное (первой степени) и квадратичное (второй степени) приближения для этой функции, используя метод наименьших квадратов. Приближения будут иметь вид $y = ax + b$ для линейного и $y = ax^2 + bx + c$ для квадратичного.

### 3.2.1 Линейное приближение

Для линейного приближения мы используем формулу метода наименьших квадратов:

$$na + \left(\sum x_i\right) b = \sum y_i$$
$$\left(\sum x_i\right) a + \left(\sum x_i^2\right) b = \sum x_i y_i$$

Подставляем реальные значения:

$$11a + (11)b = 3.2338$$
$$11a + 15.4b = 4.0096$$

Решая систему, находим $a$ и $b$. $a = 0.17631$, $b = 0.11766$

### 3.2.2 Квадратичное приближение

Аналогично, для квадратичного приближения найдем коэффициенты $a$, $b$, и $c$, решив соответствующую систему уравнений.

Приступим к вычислениям коэффициентов:

$$na + \left(\sum x_i\right) b + \left(\sum x_i^2\right) c = \sum y_i$$
$$\left(\sum x_i\right) a + \left(\sum x_i^2\right) b + \left(\sum x_i^3\right) c = \sum x_i y_i$$
$$\left(\sum x_i^2\right) a + \left(\sum x_i^3\right) b + \left(\sum x_i^4\right) c = \sum x_i^2 y_i$$

Сначала вычислим необходимые суммы:

- $S_x = 11$
- $S_{x^2} = 15.4$
- $S_{x^3} = 24.2$
- $S_{x^4} = 40.5328$
- $S_y = 3.2338$
- $S_{xy} = 4.0096$

- $S_{x^2y} = 5.75136$

Мы можем подставить их в систему уравнений для квадратичного приближения: 11x+11y+15.4z=3.2338 11x+15.4y+24.2z=4.0096 15.4x+24.2y+40.5328z=5.75136

$$11a + 11b + 15,4c = 3.2338$$
$$11a + 15.4b + 24.2c = 4.0096$$
$$15.4a + 24.2b + 40.5328c = 5.75136$$

Эту систему уравнений можно решить, чтобы найти коэффициенты $a$, $b$, и $c$. Решение системы дает:
- $a \approx -0.2386$
- $b \approx 0.6535$
- $c \approx -0.0255$

**Результаты**

Коэффициенты линейного приближения получились как $y = 0.17631x + 0.11766$, а коэффициенты квадратичного приближения как $y = -0.2386x^2 + 0.6535x - 0.0255$.

#### 3.2.3 Нахождение СКО

Для линейного приближения у нас есть формула $\hat{y}_i = 0.17631x + 0.11766$. СКО вычисляется как:

$$\text{СКО} = \sqrt{\frac{1}{n}\sum(y_i - \hat{y}_i)^2}$$

Для квадратичного приближения формула выглядит как $\hat{y}_i = -0.2386x^2 + 0.6535x - 0.0255$. Процесс аналогичен линейному приближению, но с использованием этой формулы. ‘

#### 3.2.4 Расчет и Сравнение СКО

После выполнения этих шагов для всех точек интервала, мы получаем СКО для линейного и квадратичного приближений:

СКО (линейное приближение) $\approx 0.086$

СКО (квадратичное приближение) $\approx 0.015$

Эти расчеты показывают, что квадратичное приближение обеспечивает меньшее среднеквадратическое отклонение, что говорит о его более высокой точности по сравнению с линейным приближением для данной функции и заданного интервала.

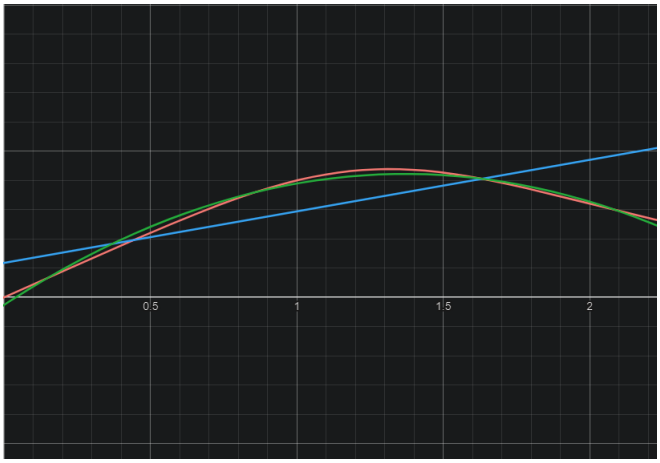
Это означает, что квадратичное приближение лучше аппроксимирует данную функцию на выбранном интервале, так как имеет меньшее среднеквадратическое отклонение.

### 3.3 Выбор наилучшего приближения

На основе среднеквадратических отклонений, наилучшим приближением является квадратичное, так как его СКО меньше.

### 3.4 Построение графиков

Теперь построим графики исходной функции и аппроксимаций.

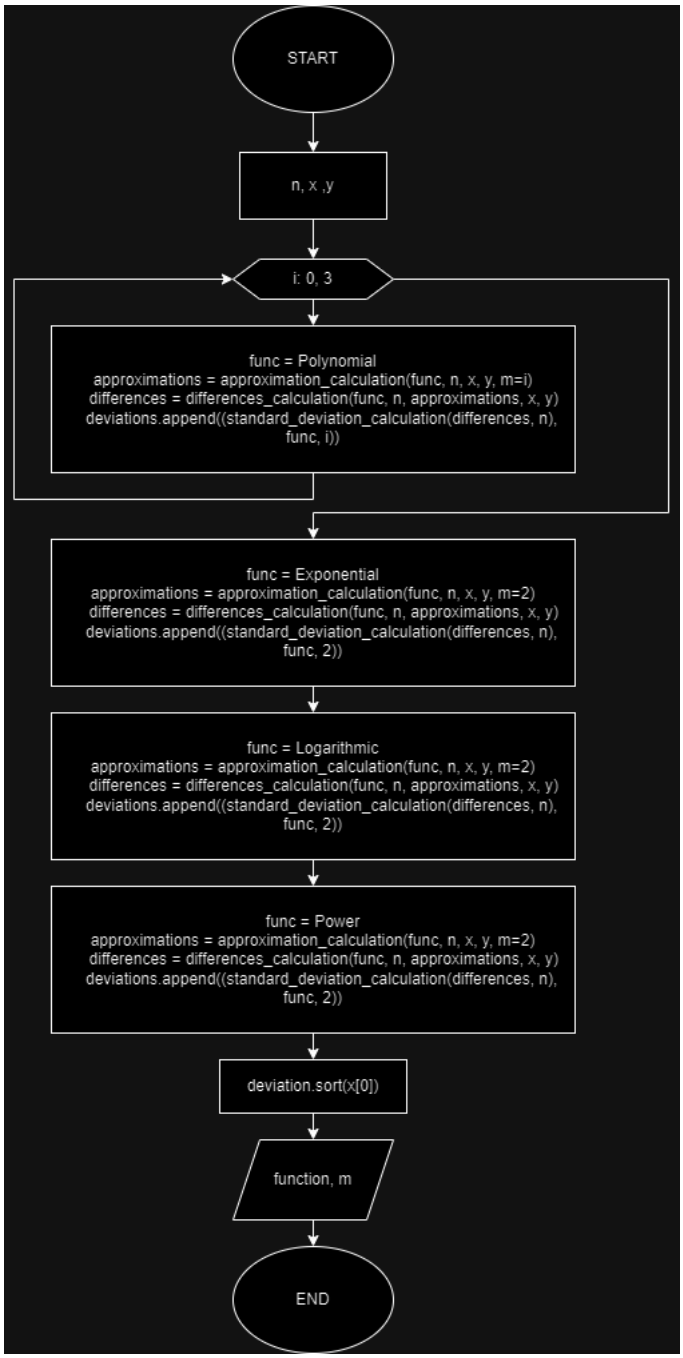


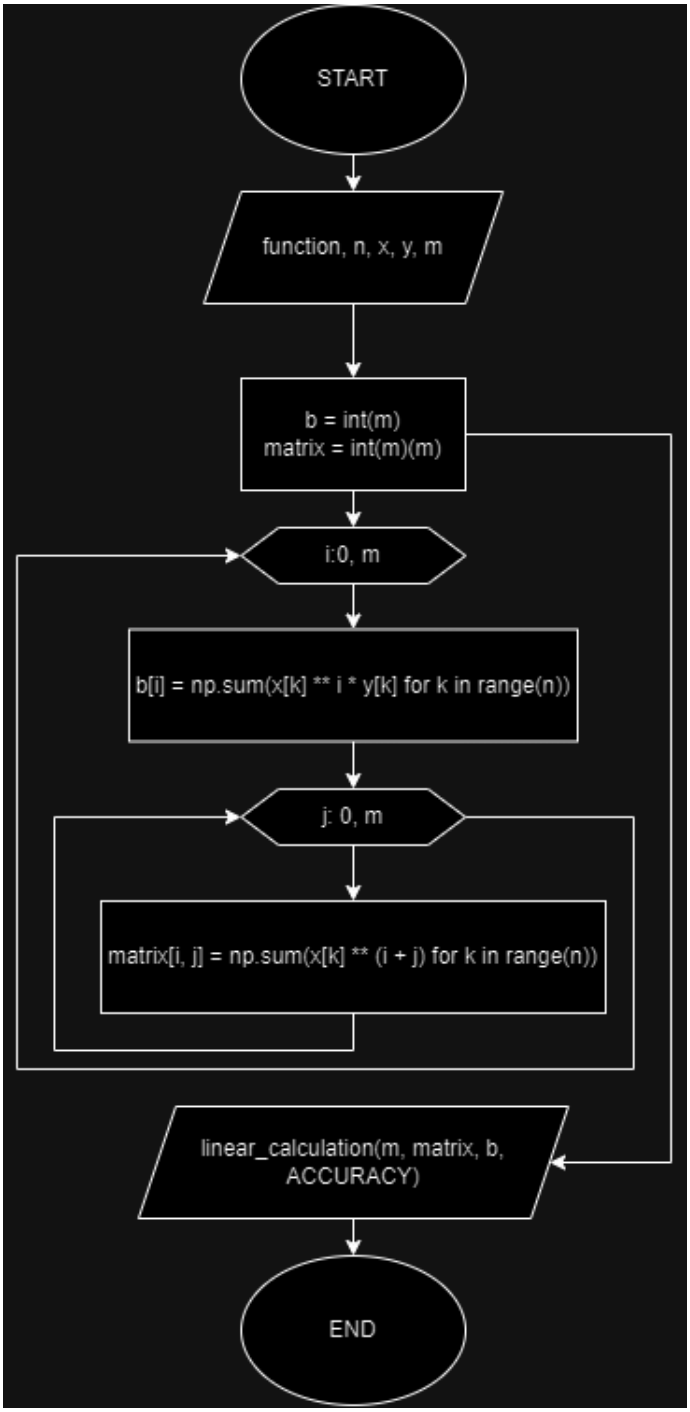

Где красный график - оригинальная функция, синия - линейная, а зелёная - квадратичная

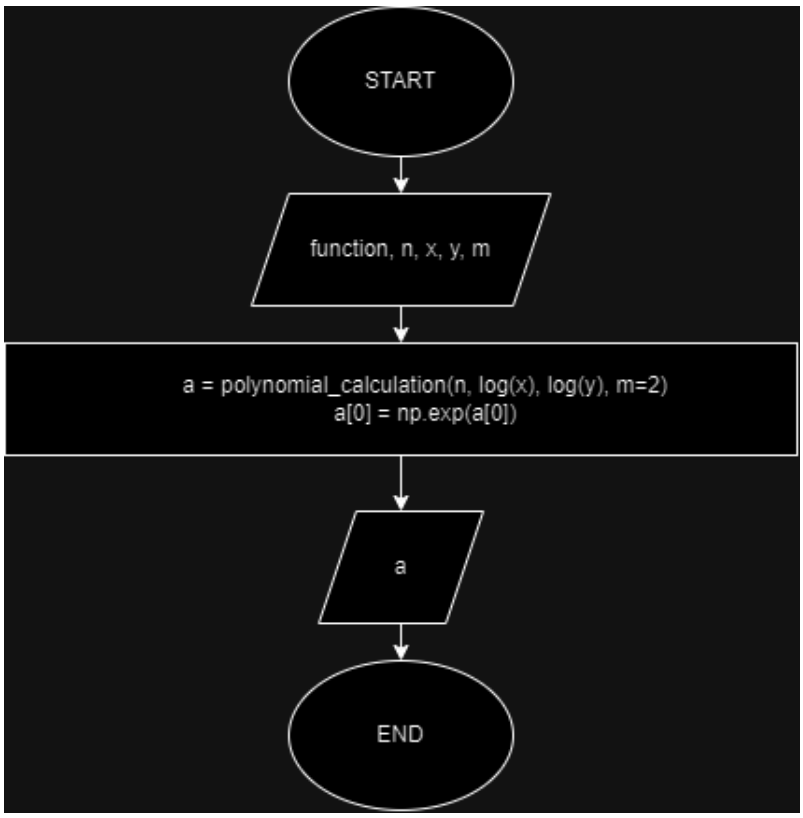
### 3.5 Вывод

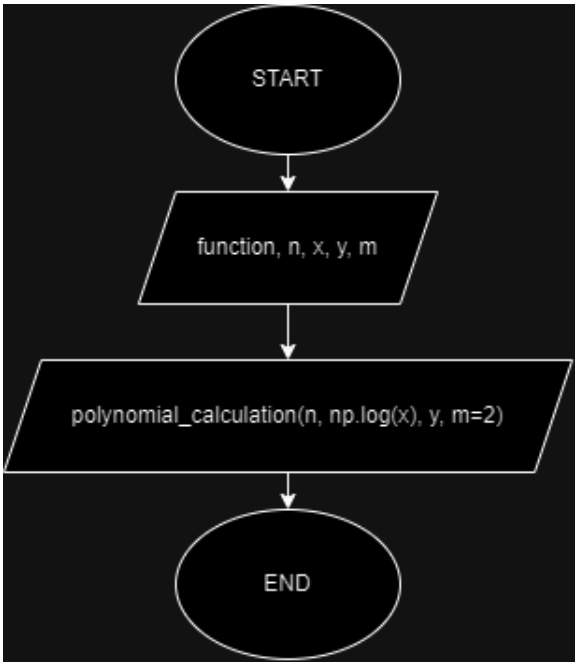
На основе среднеквадратического отклонения мы определили, что квадратичное приближение лучше подходит для аппроксимации данной функции, поскольку оно дает меньшее значение СКО.

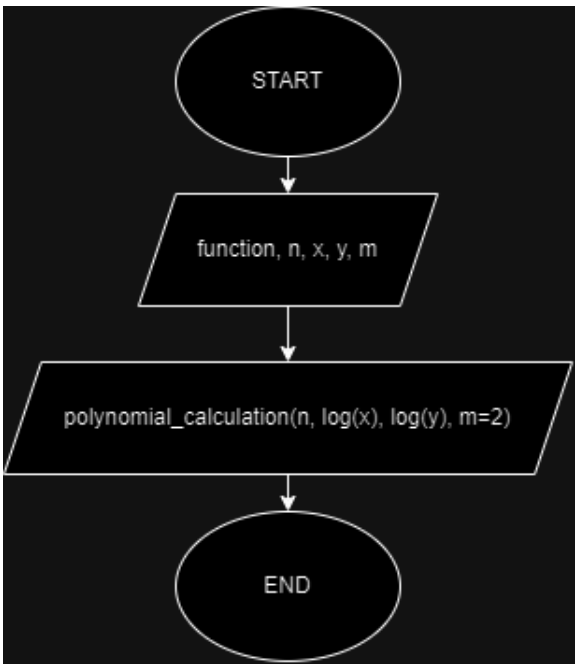
# 4 Программная реализация задачи




START
function, n, x, y, m
b = int(m)
matrix = int(m)(m)
i:0, m
b[i] = np.sum(x[k] ** i * y[k] for k in range(n))
j: 0, m
matrix[i, j] = np.sum(x[k] ** (i + j) for k in range(n))
linear_calculation(m, matrix, b, ACCURACY)
END


START
function, n, x, y, m
a = polynomial_calculation(n, log(x), log(y), m=2)
a[0] = np.exp(a[0])
a
END


START
function, n, x, y, m
polynomial_calculation(n, np.log(x), y, m=2)
END




### 4.1 Листинг программы

Approximation.py

```python
import numpy as np
from enum import Enum
from dataclasses import dataclass

ACCURACY = 0.001


class Function(str, Enum):
    Polynomial = '0'
    Exponential = '3'
    Logarithmic = '4'
    Power = '5'


@dataclass
class ApproximationCalculator:
    function: Function
    x: []
    y: []
    coefficients: []
    m: int = -1

    def calculate_coefficients(self):
        self.coefficients = approximation_calculation(
            self.function, len(self.x), self.x, self.y, m = self.m
        )
        return self.coefficients

    def calculate_differences(self):
        return differences_calculation(
            self.function, len(self.x), self.coefficients, self.x, self.y
```

```
        )

    def calculate_standard_deviation(self):
        diffs = self.calculate_differences()
        return standard_deviation_calculation(diffs, len(self.x))

    def calculate_pearson_correlation(self):
        n = len(self.x)
        sum_x = np.sum(self.x)
        sum_y = np.sum(self.y)
        sum_xy = np.sum(
            self.x[i] * self.y[j] if i == j else 0
            for i in range(len(self.x))
            for j in range(len(self.y))
        )
        sum_x_squared = np.sum(x**2 for x in self.x)
        sum_y_squared = np.sum(y**2 for y in self.y)

        numerator = n * sum_xy - sum_x * sum_y
        denominator = (
            (n * sum_x_squared - sum_x**2) * (n * sum_y_squared - sum_y**2)
        ) ** 0.5

        if denominator == 0.0:
            return [False,"Division by zero деление( наноль)"]

        r = numerator / denominator

        if abs(r) < 0.8:
            return [False, "No strong linear dependency линейная( зависимость)
                ↪ detected."]

        return [True,r]

    def get_phi_values(self):
        return np.array(
            [get_function_value(self.function, self.coefficients, x) for x in self.x]
        )

    def get_epsilon_values(self):
        return self.y - self.get_phi_values()

    def print_function(self):
        if self.function == Function.Polynomial:
            terms = []
            for i, coeff in enumerate(self.coefficients):
                if coeff == 0 and i != 0:
                    continue
                term = (
                    f"{coeff:.10f}"
                    if i == 0 or not terms
                    else f"{'+' if coeff >= 0 else ''}{coeff:.10f}"
                )
                if i == 1:
                    term += "x"
                elif i > 1:
```

```
                term += f"x^{i}"
            terms.append(term)
         if not terms:
            terms.append("0")
         return "".join(terms)
      elif self.function == Function.Exponential:
         return f"{self.coefficients[0]:.10f}e^{self.coefficients[1]:+.10f}x"
      elif self.function == Function.Logarithmic:
         return f"{self.coefficients[0]:.10f} + {self.coefficients[1]:+.10f}ln(x)"
      elif self.function == Function.Power:
         return f"{self.coefficients[0]:.10f}x^{self.coefficients[1]:+.10f}"


def approximation_calculation(f, n, x, y, m=1):
   if f == Function.Polynomial:
      b = np.zeros(m)
      matrix = np.zeros((m, m))

      for i in range(m):
         b[i] = np.sum(x[k] ** i * y[k] for k in range(n))
         for j in range(m):
            matrix[i, j] = np.sum(x[k] ** (i + j) for k in range(n))

      return linear_calculation(m, matrix, b, ACCURACY)

   elif f == Function.Exponential:
      a = approximation_calculation(Function.Polynomial, n, np.log(x), np.log(y), m
         ↪ =2)
      a[0] = np.exp(a[0])
      return a

   elif f == Function.Logarithmic:
      return approximation_calculation(Function.Polynomial, n, np.log(x), y, m=2)

   elif f == Function.Power:
      return approximation_calculation(
         Function.Polynomial, n, np.log(x), np.log(y), m=2
      )


def linear_calculation(n, a, b, e):
   v_x = np.zeros(n)
   while True:
      delta = 0.0
      for i in range(n):
         s = np.sum(a[i, j] * v_x[j] for j in range(0, i)) + np.sum(
            a[i, j] * v_x[j] for j in range(i + 1, n)
         )
         x = (b[i] - s) / a[i, i]
         d = abs(x - v_x[i])
         if d > delta:
            delta = d
         v_x[i] = x
      if delta < e:
         break
   return v_x
```

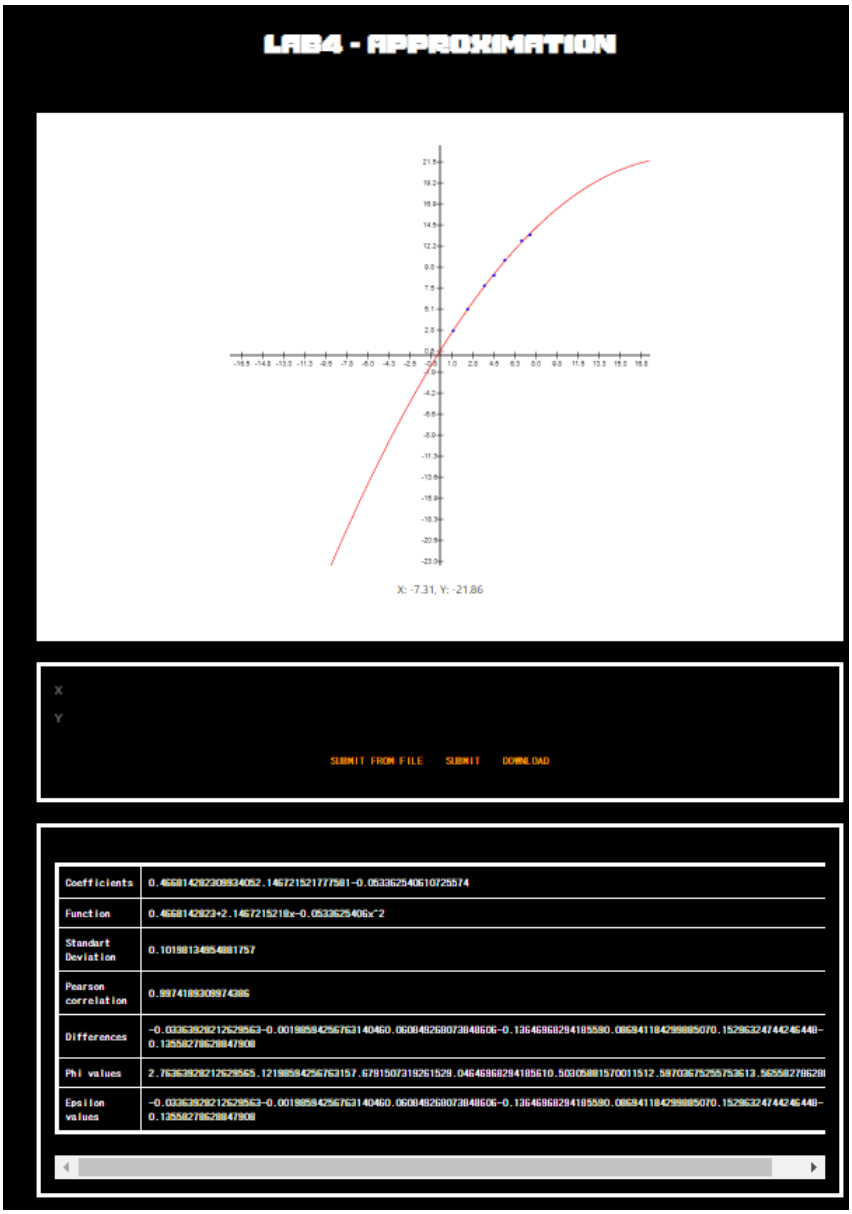
```
142
143
144 def differences_calculation(f, n, coefficients, x, y):
145     differences = np.zeros(n)
146     for i in range(n):
147         differences[i] = y[i] - get_function_value(f, coefficients, x[i])
148     return differences
149
150
151 def get_function_value(f, coefficients, x):
152     if f == Function.Polynomial:
153         return np.dot(coefficients, [x**i for i in range(len(coefficients))])
154     elif f == Function.Exponential:
155         return coefficients[0] * np.exp(coefficients[1] * x)
156     elif f == Function.Logarithmic:
157         return coefficients[0] + coefficients[1] * np.log(x)
158     elif f == Function.Power:
159         return coefficients[0] * x ** coefficients[1]
160
161
162 def standard_deviation_calculation(differences, n):
163     var = np.sum(differences**2) / n
164     return var**0.5
165
166
167 @staticmethod
168 def find_best_function(n, x, y):
169     deviations = []
170
171     # Polynomial of degree 1 to 3
172     for i in range(1, 4):
173         func = Function.Polynomial
174         approximations = approximation_calculation(func, n, x, y, m=i)
175         differences = differences_calculation(func, n, approximations, x, y)
176         deviations.append((standard_deviation_calculation(differences, n), func, i))
177
178     # Exponential
179     exponential_approximations = approximation_calculation(
180         Function.Exponential, n, x, y
181     )
182     exponential_differences = differences_calculation(
183         Function.Exponential, n, exponential_approximations, x, y
184     )
185     deviations.append(
186         (
187             standard_deviation_calculation(exponential_differences, n),
188             Function.Exponential,
189             2,
190         )
191     )
192
193     # Logarithmic
194     logarithmic_approximations = approximation_calculation(
195         Function.Logarithmic, n, x, y
196     )
197     logarithmic_differences = differences_calculation(
```

```
        Function.Logarithmic, n, logarithmic_approximations, x, y
    )
    deviations.append(
        (
            standard_deviation_calculation(logarithmic_differences, n),
            Function.Logarithmic,
            2,
        )
    )

    # Power
    power_approximations = approximation_calculation(Function.Power, n, x, y)
    power_differences = differences_calculation(
        Function.Power, n, power_approximations, x, y
    )
    deviations.append(
        (standard_deviation_calculation(power_differences, n), Function.Power, 2)
    )

    deviations.sort(key=lambda x: x[0])
    return [deviations[0][1], deviations[0][2]]
```

### 4.2 Пример работы программы




# 5 Github

Ссылка на GitHub

# 6 Вывод

В этой работе я ознакомился с различными методами квадратичной аппроксимации