Практическое задание №3
Армбристер Никита Владиславович
Группа ПМИ-31
Вариант 1

Цель: сформировать практические навыки аппроксимации табличных функций с помощью сглаживающих сплайнов

Задание:

1. Разработать класс, реализующий интерфейс сглаживающего сплайна. На каждом сегменте разбиения использовать базисную систему финитных функций первого порядка. Сглаживающий сплайн g(x) строить как решение задачи о минимизации функционала в линейном подпространстве $\Phi = (1-p)\left\|f(x)-g(x)\right\|_2^2 + p\left\|g^{'(x)}\right\|_2^2$
2. В качестве входных данных сгенерируйте по нормальному закону последовательность случайных чисел (число наблюдений и параметры распределения взять из таблицы).
3. На одной диаграмме изобразить интерполяционный и сглаживающий сплайны: ось абсцисс – номер случайного числа, ось ординат – сгенерированные случайные числа. Параметр сглаживания p варьировать от 0 до 1. Выяснить, на что влияет варьирование весовых коэффициентов в дискретном скалярном произведении при построении сглаживающего сплайна.
4. Проанализируйте поведение графика сглаживающего сплайна при p = 0. Какими свойствами он обладает?
5. Решите предложенную задачу средствами языка Python. Какими недостатками обладает данный инструмент анализа данных? Как их обойти?

Данные варианта 1: n=1304; M=0.94; std=4.28

Ссылка на репозиторий с проектом: https://github.com/Gribnik24/NumericalMethods-PracticalTasks/tree/main/Task%20%232.3.%20Smoothing%20Splines

Решение:

Для решения данной лабораторной работы был разработан проект на языке C++.
Структура проекта представлена ниже:

```
├── Заголовки
│   └── smoothing_spline.h
├── Исходные
│   ├── smoothing_spline.cpp
│   └── main.cpp
```

Рассмотрим каждую составляющую проекта подробнее

- smoothing_spline.h
  Включает в себя два класса.
  Первый описывает сглаживающий реализацию сплайна. Присутствуют методы по переходу на мастер-элемент, определение базисных функций, инициализации сплайна и вычисление значения в необходимой точке, соответственно.

```
class SmoothingSpline {
private:
    std::vector<double> Points;
    std::vector<double> alpha;
    double SMOOTH;

    void Transition_To_Master_Element(int Seg_Num, double X, double& Ksi) const;


    double Basis_Function(int Number, double Ksi) const;


    double Der_Basis_Function(int Number, double Ksi) const;

public:
    SmoothingSpline(double smooth_param = 0.5) : SMOOTH(smooth_param) {}

    void initialize(const std::vector<double>& x_nodes,
        const std::vector<double>& y_nodes,
        double p = 0.5);

    double evaluate(double x) const;
};
```

Второй класс отвечает за создание нормального распределения и построения сетки

```
class RandomDataGenerator {
private:
    static std::mt19937& getGenerator() {
        static std::mt19937 gen(42);
        return gen;
    }

public:
    static std::vector<double> generateNormalData(int n, double mean = 0, double stddev = 1);
    static std::vector<double> generateUniformGrid(int n, double start = 0, double end = 1);
};
```

- smoothing_spline.cpp

Включает реализацию функции из представленных выше классов

  - Transition_To_Master_Element преобразуем координату x из реального сегмента в мастер-элемент на координатах [-1, 1] по формуле перехода

```
void SmoothingSpline::Transition_To_Master_Element(int Seg_Num, double X, double& Ksi) const {
    double h = Points[Seg_Num + 1] - Points[Seg_Num];
    Ksi = 2.0 * (X - Points[Seg_Num]) / h - 1.0;
}
```

  - Basis_Function

    Определяет линейные базисные функции на мастер-элементе по формулам

```
double SmoothingSpline::Basis_Function(int Number, double Ksi) const {
    switch (Number) {
    case 1: return 0.5 * (1 - Ksi);
    case 2: return 0.5 * (1 + Ksi);
    default: throw std::invalid_argument("Error in basis function number");
    }
}
```

  - Der_Basis_Function

Возвращает две константы в зависимости от базисной функции – производные базисных функций

```
double SmoothingSpline::Der_Basis_Function(int Number, double Ksi) const {
    switch (Number) {
    case 1: return -0.5;
    case 2: return 0.5;
    default: throw std::invalid_argument("Error in basis function derivative number");
    }
}
```

- Initialize

  Функция по инициализации и построении сплайна. Она задает СЛАУ, далее решает ее через метод прогонки. Также данной работе для управления степенью сглаживания сплайна используется механизм регуляризации, зависящий от параметра p. Основная идея заключается в том, чтобы балансировать между точностью аппроксимации исходных данных и гладкостью результирующей кривой. Регуляризация реализована через добавление в систему линейных алгебраических уравнений (СЛАУ) дополнительных членов, которые штрафуют за резкие изменения производных.

```
void SmoothingSpline::initialize(const std::vector<double>& x_nodes,
                                 const std::vector<double>& y_nodes,
                                 double p)
{
    SMOOTH = p;
    Points.clear();

    // Создаем точки, определяем сегменты
    for (size_t i = 0; i < x_nodes.size(); ++i) {
        Points.push_back(x_nodes[i]);
    }

    int Num_Segments = Points.size() - 1;
    alpha.resize(Num_Segments + 1, 0.0);

    // Диагонали матрицы СЛАУ
    std::vector<double> a(Num_Segments + 1, 0.0);  // нижняя диагональ
    std::vector<double> b(Num_Segments + 1, 0.0);  // главная диагональ
    std::vector<double> c(Num_Segments + 1, 0.0);  // верхняя диагональ
```

```
    // Процедура ассемблирования СЛАУ
    auto Assembling = [&](int i, double X, double F_Val, double w) {
        double Ksi;
        Transition_To_Master_Element(i, X, Ksi);

        double f1 = Basis_Function(1, Ksi);
        double f2 = Basis_Function(2, Ksi);

        // Вклад в матрицу (интерполяционная часть)
        b[i] += (1.0 - SMOOTH) * w * f1 * f1;
        b[i + 1] += (1.0 - SMOOTH) * w * f2 * f2;
        a[i + 1] += (1.0 - SMOOTH) * w * f1 * f2;
        c[i] += (1.0 - SMOOTH) * w * f2 * f1;

        // Вклад в правую часть
        alpha[i] += (1.0 - SMOOTH) * w * f1 * F_Val;
        alpha[i + 1] += (1.0 - SMOOTH) * w * f2 * F_Val;
        };
```

```
// Сборка СЛАУ
for (int i = 0; i < Num_Segments; i++) {
    // Вклад от узлов сетки
    Assembling(i, Points[i], y_nodes[i], 1.0);
    Assembling(i, Points[i + 1], y_nodes[i + 1], 1.0);

    // Вклад от сглаживания (регуляризация)
    double h = Points[i + 1] - Points[i];

    // Плавное увеличение регуляризации при p от 0 до 1
    double reg_scale = 1.0 + 99.0 * p * p;
    double reg_weight = SMOOTH * reg_scale;

    b[i] += reg_weight / h;
    b[i + 1] += reg_weight / h;
    a[i + 1] -= reg_weight / h;
    c[i] -= reg_weight / h;
}

// Для очень больших p - дополнительная регуляризация, но плавная
if (p > 0.8) {
    double extra_scale = (p - 0.8) / 0.2; // От 0 до 1 при p от 0.8 до 1.0
    double extra_reg = extra_scale * extra_scale * 500.0; // Квадратичное увеличение

    for (int i = 0; i < Num_Segments; i++) {
        double h = Points[i + 1] - Points[i];
        b[i] += extra_reg / h;
        b[i + 1] += extra_reg / h;
    }
}
```

```
    // Метод прогонки: прямой ход
    for (int j = 1; j < Num_Segments + 1; j++) {
        if (std::abs(b[j - 1]) < 1e-12) {
            double sum = 0.0;
            for (double val : y_nodes) sum += val;
            double avg = sum / y_nodes.size();
            for (size_t i = 0; i < alpha.size(); ++i) alpha[i] = avg;
            return;
        }
        double m = a[j] / b[j - 1];
        b[j] -= m * c[j - 1];
        alpha[j] -= m * alpha[j - 1];
    }
```

```
    // Метод прогонки: обратный ход
    if (std::abs(b[Num_Segments]) < 1e-12) {
        double sum = 0.0;
        for (double val : y_nodes) sum += val;
        double avg = sum / y_nodes.size();
        for (size_t i = 0; i < alpha.size(); ++i) alpha[i] = avg;
        return;
    }

    alpha[Num_Segments] /= b[Num_Segments];
    for (int j = Num_Segments - 1; j >= 0; j--) {
        alpha[j] = (alpha[j] - c[j] * alpha[j + 1]) / b[j];
    }
}
```

- Evaluate

  Функция по вычислению значения сплайна. Она ищет сегмент, где находится исходная точка. При удачном сценарии нахождения вернет ее разложение через полином. Иначе вернет краевые значения, если точка вне диапазона

```cpp
// Вычисляет значение сплайна
double SmoothingSpline::evaluate(double x) const {
    if (Points.empty()) return 0.0;

    double eps = 1e-7;
    int Num_Segments = Points.size() - 1;

    // Поиск сегмента, содержащего x
    for (int i = 0; i < Num_Segments; i++) {
        if ((x >= Points[i] && x <= Points[i + 1]) ||
            std::abs(x - Points[i]) < eps ||
            std::abs(x - Points[i + 1]) < eps) {

            // Переход на мастер-элемент и вычисление полинома
            double Ksi;
            Transition_To_Master_Element(i, x, Ksi);
            return alpha[i] * Basis_Function(1, Ksi) + alpha[i + 1] * Basis_Function(2, Ksi);
        }
    }

    // Если точка вне диапазона
    if (x < Points[0]) return alpha[0];
    if (x > Points[Num_Segments]) return alpha[Num_Segments];

    return 0.0;
}
```

- generateNormalData

  Задает нормальное распределение с заданным сидом

```cpp
std::vector<double> RandomDataGenerator::generateNormalData(int n, double mean, double stddev)
    std::vector<double> data;
    data.reserve(n);

    auto& gen = getGenerator();
    std::normal_distribution<double> dist(mean, stddev);

    for (int i = 0; i < n; ++i) {
        data.push_back(dist(gen));
    }

    return data;
}
```

- generateUniformGrid

  Формирует сетку по заданным параметрам

```cpp
// Генерирует сетку
std::vector<double> RandomDataGenerator::generateUniformGrid(int n, double start, double end)
    std::vector<double> grid;
    grid.reserve(n);

    double step = (end - start) / (n - 1);
    for (int i = 0; i < n; ++i) {
        grid.push_back(start + i * step);
    }

    return grid;
}
```

- main.cpp

  Включает функцию conductExperiement, которая формирует построение сплайнов для набора параметра p и дальнейший вывод в csv файл и дальнейший вызов данной функции

```cpp
void conductExperiment() {
    // Задаем параметры первого варианта
    const int n_observations = 1304;
    const double mean = 0.94;
    const double stdev = 4.28;

    // Создаем распределение и сетку
    auto random_data = RandomDataGenerator::generateNormalData(n_observations, mean, stdev);
    auto indices = RandomDataGenerator::generateUniformGrid(n_observations, 0, n_observations

    // Задаем значения параметра сглаживания для тестов и создаем набор сплайнов
    std::vector<double> p_values = { 0.0, 0.2, 0.5, 0.8, 0.99 };
    std::vector<SmoothingSpline> splines;
    for (double p : p_values) {
        SmoothingSpline spline(p);
        spline.initialize(indices, random_data, p);
        splines.push_back(spline);
    }
```

```cpp
    // Создаем файл для визуализации
    std::ofstream file("smoothing_splines.csv");
    file << "x,Original";
    for (double p : p_values) {
        file << ",p=" << p;
    }
    file << "\n";

    // Для каждого узла сохраняем исходную точку и значения сплайнов
    for (int i = 0; i < n_observations; ++i) {
        file << indices[i] << "," << random_data[i];
        for (const auto& spline : splines) {
            file << "," << spline.evaluate(indices[i]);
        }
        file << "\n";
    }

    file.close();
}

int main() {
    conductExperiment();
    return 0;
}
```

Далее через matplotlib мной была написан код graphs.ipynb. Там представлен вывод результатов работы сплайна. Для удобства и красоты будем брать каждый 20 элемент из 1304

[1]:
```
import pandas as pd
import matplotlib.pyplot as plt
```

[2]:
```
data = pd.read_csv('smoothing_splines.csv')
data
```

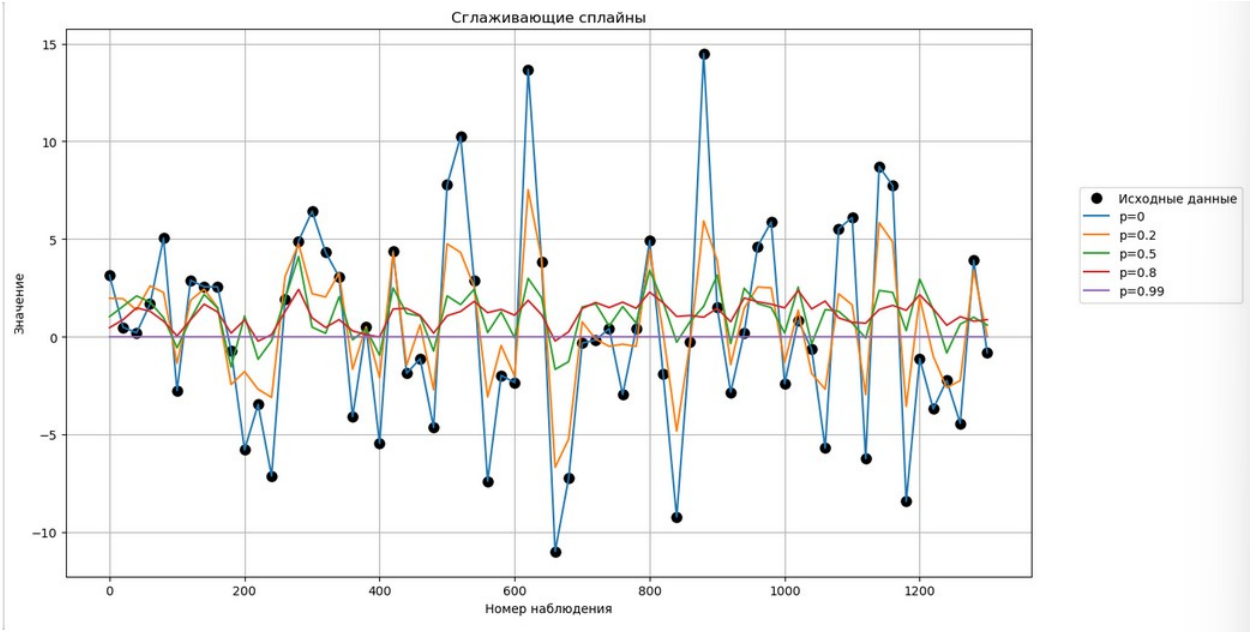
[2]:

	x	Original	p=0	p=0.2	p=0.5	p=0.8	p=0.99
0	0	3.146050	3.146050	1.973320	1.035430	0.469121	0.000056
1	1	-1.415000	-1.415000	1.027560	0.953464	0.458723	-0.000009
2	2	6.796970	6.796970	4.021420	1.055460	0.462882	0.000129
3	3	2.968120	2.968120	2.538600	0.711508	0.417832	0.000066
4	4	0.408650	0.408650	0.363005	0.192287	0.352969	0.000008
...	...	...	...	...	...	...	...
1299	1299	1.605230	1.605230	2.637360	1.269220	0.968532	0.000048
1300	1300	-0.805483	-0.805483	0.031843	0.601177	0.878946	-0.000012
1301	1301	-0.304650	-0.304650	-1.223150	0.042384	0.802445	-0.000017
1302	1302	-5.641990	-5.641990	-3.959590	-0.489455	0.734545	-0.000113
1303	1303	-4.010890	-4.010890	-3.982490	-0.621098	0.716183	-0.000093

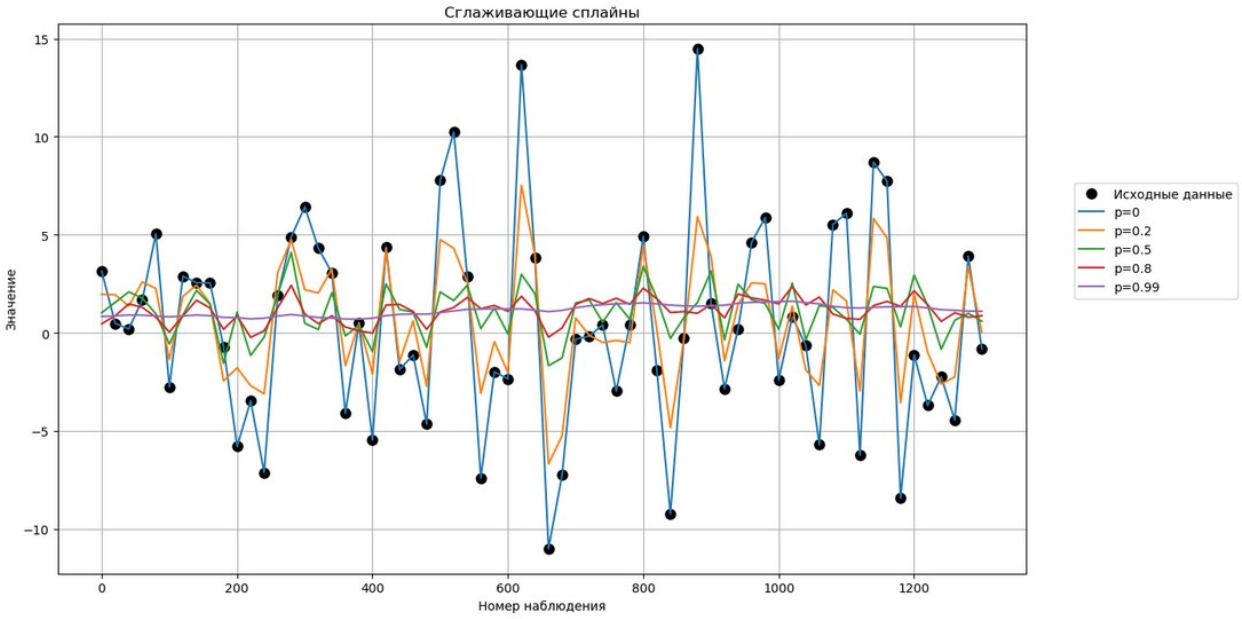
1304 rows × 7 columns

[3]:
```
data = data.iloc[::20]

plt.figure(figsize=(14, 8))
plt.plot(data['x'], data['Original'], 'ko', label='Исходные данные', markersize=8)
for col in data.columns[2:]:
    plt.plot(data['x'], data[col], label=col)

plt.legend(loc=(1.05, 0.5))
plt.grid(True)
plt.xlabel('Номер наблюдения')
plt.ylabel('Значение')
plt.title('Сглаживающие сплайны')
plt.show()
```

По итогу получился такой график



Если убрать более сильную регуляризацию при p>0.8, то разница будет заметна сразу же. Сплайны при более высоком параметре сглаживания работать стали хуже



Далее по заданию было необходимо реализовать сплайн на Python. Решение представлено в файле smoothing_spline.ipynb Я решил выполнить это через несколько способов:

1) Ручная реализация
2) scipy.make_smoothing_spline
3) scipy.UnivariateSpline

Прежде всего, я перенес построение графика в отдельную функцию. Так будет удобнее

```python
def splines_visualisation(df):
    data = df.iloc[::20]

    plt.figure(figsize=(14, 8))
    plt.plot(data['x'], data['Original'], 'ko', label='Исходные данные', markersize=8)
    for col in data.columns[2:]:
        plt.plot(data['x'], data[col], label=col)

    plt.legend(loc=(1.05, 0.5))
    plt.grid(True)
    plt.xlabel('Номер наблюдения')
    plt.ylabel('Значение')
    plt.title('Сглаживающие сплайны')
    plt.show()
```

Для начала я полностью перенес реализацию класса, написанную ранее на C++. Некоторые моменты я упрощал, используя numpy.

```python
class SmoothingSpline:
    def __init__(self, smooth_param = 0.5):
        self.Points = []
        self.alpha = None
        self.SMOOTH = smooth_param

    def _transition_to_master_element(self, seg_num, x):
        h = self.Points[seg_num + 1] - self.Points[seg_num]
        return 2.0 * (x - self.Points[seg_num]) / h - 1.0

    def _basis_function(self, number, ksi):
        if number == 1: return 0.5 * (1 - ksi)
        elif number == 2: return 0.5 * (1 + ksi)
        else: raise ValueError("Error in basis function number")

    def _der_basis_function(self, number, ksi):
        if number == 1: return -0.5
        elif number == 2: return 0.5
        else: raise ValueError("Error in basis function derivative number")

    def initialize(self, x_nodes, y_nodes, p = 0.5):
        self.SMOOTH = p
        self.Points = list(x_nodes) if hasattr(x_nodes, '__iter__') else [x_nodes]

        num_segments = len(self.Points) - 1
        self.alpha = np.zeros(num_segments + 1)

        a = np.zeros(num_segments + 1)
        b = np.zeros(num_segments + 1)
        c = np.zeros(num_segments + 1)
```

```
        def assembling(i, x_val, f_val, weight):
            ksi = self._transition_to_master_element(i, x_val)

            f1 = self._basis_function(1, ksi)
            f2 = self._basis_function(2, ksi)

            b[i] += (1.0 - self.SMOOTH) * weight * f1 * f1
            b[i + 1] += (1.0 - self.SMOOTH) * weight * f2 * f2
            a[i + 1] += (1.0 - self.SMOOTH) * weight * f1 * f2
            c[i] += (1.0 - self.SMOOTH) * weight * f2 * f1

            self.alpha[i] += (1.0 - self.SMOOTH) * weight * f1 * f_val
            self.alpha[i + 1] += (1.0 - self.SMOOTH) * weight * f2 * f_val

        for i in range(num_segments):
            assembling(i, self.Points[i], y_nodes[i], 1.0)
            assembling(i, self.Points[i + 1], y_nodes[i + 1], 1.0)

            h = self.Points[i + 1] - self.Points[i]

            reg_scale = 1.0 + 99.0 * p * p
            reg_weight = self.SMOOTH * reg_scale

            b[i] += reg_weight / h
            b[i + 1] += reg_weight / h
            a[i + 1] -= reg_weight / h
            c[i] -= reg_weight / h

        if p > 0.8:
            extra_scale = (p - 0.8) / 0.2
            extra_reg = extra_scale * extra_scale * 500.0

            for i in range(num_segments):
                h = self.Points[i + 1] - self.Points[i]
                b[i] += extra_reg / h
                b[i + 1] += extra_reg / h

        for j in range(1, num_segments + 1):
            if abs(b[j - 1]) < 1e-12:
                avg = np.mean(y_nodes)
                self.alpha.fill(avg)
                return

            m = a[j] / b[j - 1]
            b[j] -= m * c[j - 1]
            self.alpha[j] -= m * self.alpha[j - 1]

        if abs(b[num_segments]) < 1e-12:
            avg = np.mean(y_nodes)
            self.alpha.fill(avg)
            return
```

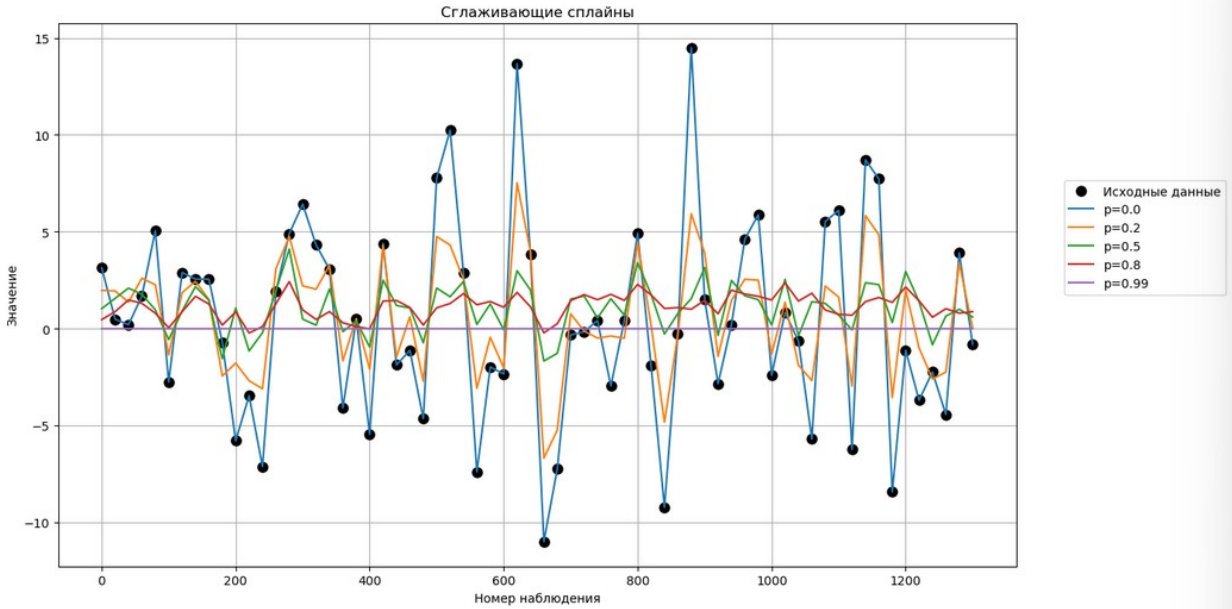
```
    self.alpha[num_segments] /= b[num_segments]
    for j in range(num_segments - 1, -1, -1):
        self.alpha[j] = (self.alpha[j] - c[j] * self.alpha[j + 1]) / b[j]

def evaluate(self, x):
    if len(self.Points) == 0:
        return 0.0

    eps = 1e-7
    num_segments = len(self.Points) - 1

    for i in range(num_segments):
        if (self.Points[i] <= x <= self.Points[i + 1] or
            abs(x - self.Points[i]) < eps or
            abs(x - self.Points[i + 1]) < eps):

            ksi = self._transition_to_master_element(i, x)
            return (self.alpha[i] * self._basis_function(1, ksi) +
                    self.alpha[i + 1] * self._basis_function(2, ksi))

    if x < self.Points[0]:
        return self.alpha[0]
    if x > self.Points[num_segments]:
        return self.alpha[num_segments]

    return 0.0
```

Результаты работы кода видны ниже. Я не стал задавать новое распределение с тем же сидом 42. Сомневаюсь, что они будут давать одинаковые данные для разных языков. Поэтому я просто выделил столбик с исходными значениями из smoothing_splines.csv и произвел вычисления по ним. График визуально вышел идентичным, но вот значения в таблице будут чуть отличаться.

	x	Original	p=0.0	p=0.2	p=0.5	p=0.8	p=0.99
0	0	3.146050	3.146050	1.973316	1.035431	0.469121	0.000056
1	1	-1.415000	-1.415000	1.027562	0.953465	0.458723	-0.000009
2	2	6.796970	6.796970	4.021425	1.055457	0.462881	0.000129
3	3	2.968120	2.968120	2.538601	0.711507	0.417831	0.000066
4	4	0.408650	0.408650	0.363004	0.192286	0.352969	0.000008
...	...	...	...	...	...	...	...
1299	1299	1.605230	1.605230	2.637354	1.269221	0.968530	0.000048
1300	1300	-0.805483	-0.805483	0.031841	0.601173	0.878944	-0.000012
1301	1301	-0.304650	-0.304650	-1.223150	0.042380	0.802443	-0.000017
1302	1302	-5.641990	-5.641990	-3.959593	-0.489459	0.734543	-0.000113
1303	1303	-4.010890	-4.010890	-3.982494	-0.621101	0.716181	-0.000093

1304 rows × 7 columns

```
splines_visualisation(data)
```



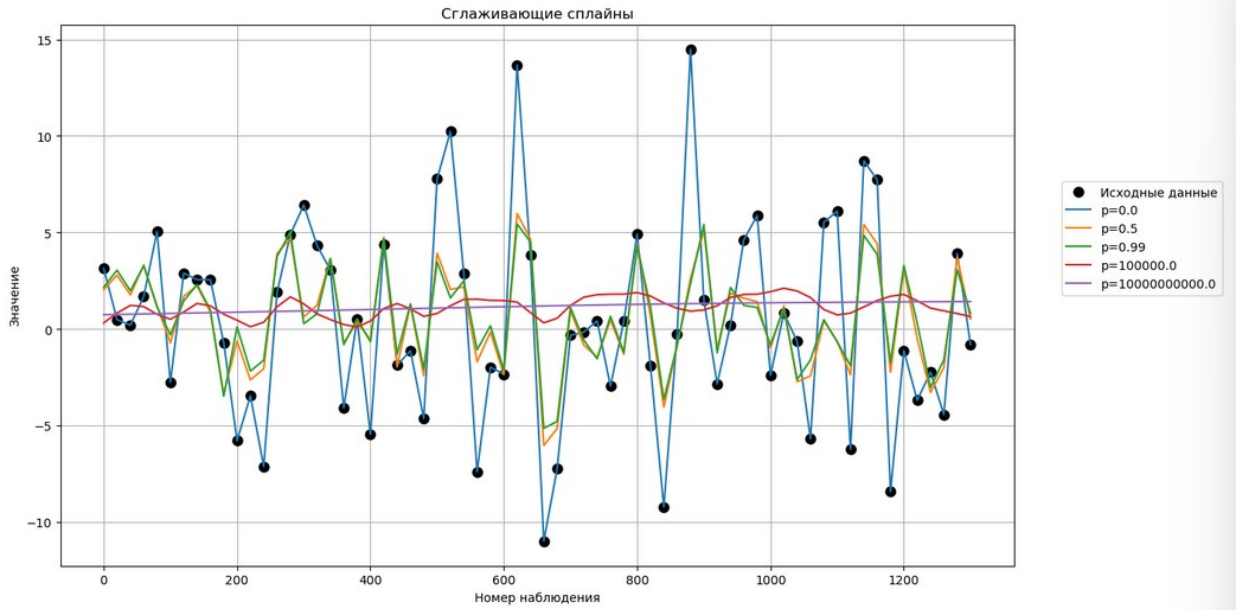
Далее я решил попробовать реализацию сплайна scipy.interpolate.make_smoothing_spline. Тут пришлось отойти от привычного набора значений параметра сглаживания p в силу определения параметра. Были убраны некоторые промежуточные значения и добавлены новые

```python
from scipy.interpolate import make_smoothing_spline

scipy_splines = []
p_values = [0.0, 0.5, 0.99, 1e5, 1e10]

for p in p_values:
    # Для make_smoothing_spline используем непосредственно p как параметр сглаживания
    # Чем больше p, тем более гладкий сплайн
    spline = make_smoothing_spline(indices, random_data, lam=p)
    scipy_splines.append(spline)

data_scipy = {'x': indices, 'Original': random_data}

for i, p in enumerate(p_values):
    data_scipy[f'p={p}'] = scipy_splines[i](indices)

data_scipy = pd.DataFrame(data_scipy)
```

```python
data_scipy
```

	x	Original	p=0.0	p=0.5	p=0.99	p=100000.0	p=10000000000.0
0	0	3.146050	3.146050	2.038792	2.168716	0.323296	0.745452
1	1	-1.415000	-1.415000	2.293365	2.521353	0.346792	0.746103
2	2	6.796970	6.796970	3.526332	3.198510	0.370314	0.746755
3	3	2.968120	2.968120	2.861815	2.479767	0.393885	0.747407
4	4	0.408650	0.408650	0.584100	0.487436	0.417574	0.748059
...	...	...	...	...	...	...	...
1299	1299	1.605230	1.605230	3.715405	3.567783	0.664006	1.425242
1300	1300	-0.805483	-0.805483	0.519179	0.800248	0.652176	1.425413
1301	1301	-0.304650	-0.304650	-1.889698	-1.641071	0.640070	1.425584
1302	1302	-5.641990	-5.641990	-3.909084	-3.708728	0.627807	1.425755
1303	1303	-4.010890	-4.010890	-4.842628	-5.051116	0.615488	1.425926

```
splines_visualisation(data_scipy)
```



Тут уже график заметно отличается и по логике работы. По нему также можно увидеть работу параметра p – если принять его за 1, то сглаживания особо не будет. Но при увеличении сглаживание будет увеличиваться. Но даже при достаточно больших числах мы не придем к прямой y=0, как это было в ручной реализации.
Связано это с тем, что данный сплайн работает чуть по-другому. Если заглянуть в документацию, то можно понять, что там происходит на самом деле. Формула, по которой рассчитывается данный сплайн выглядит по-другому. Ниже представлен скриншот

Given the data arrays `x` and `y` and the array of non-negative *weights*, `w`, we look for a cubic spline function `g(x)` which minimizes

$$\sum_{j=1}^{n} w_j |y_j - g(x_j)|^2 + \lambda \int_{x_1}^{x_n} \left(g^{(2)}(u) \right)^2 du$$

where $\lambda \geqslant 0$ is a non-negative penalty parameter, and $g^{(2)}(x)$ is the second derivative of $g(x)$. The summation in the first term runs over the data points, (x_j, y_j), and the integral in the second term is over the whole interval $x \in [x_1, x_n]$.

За параметр сглаживания отвечает параметр lam. Если его не задать, то он найдется сам через критерий GCV – способ нахождения оптимального параметра сглаживания. Если же он есть, то будет использоваться взвешенная линейная регрессия. Там нет никакого упоминания мастер-элементов. В этом и причина наклона и положении линии при p=1e10

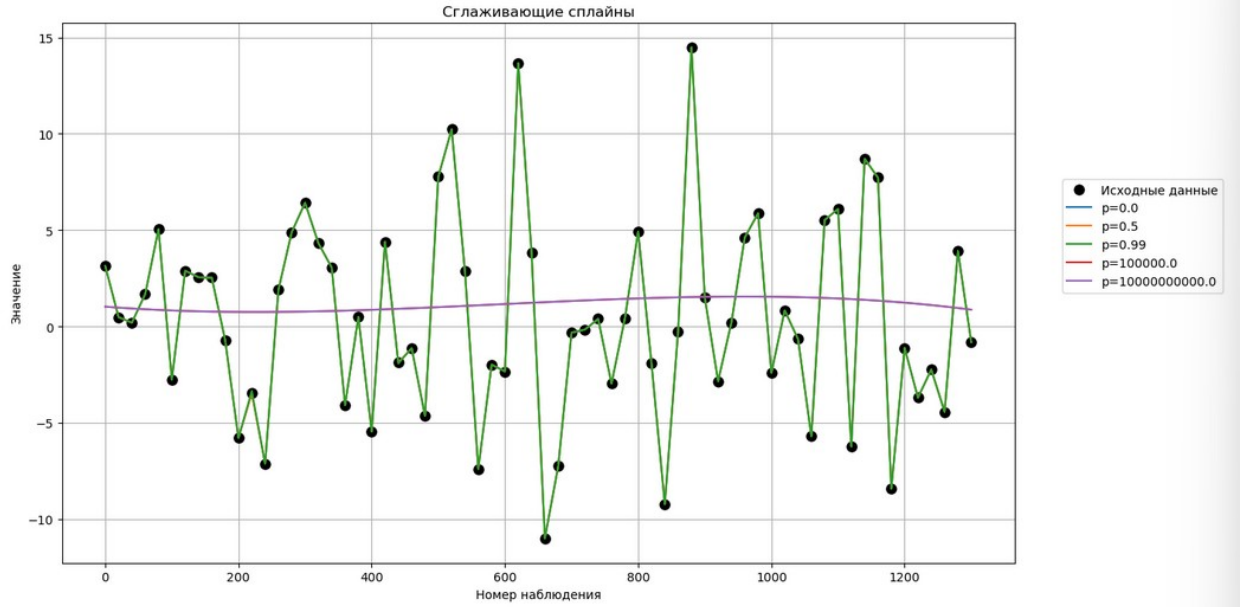
В ходе работы над данной лабораторной, я также опробовал реализацию scipy.UnivariateSpline

```
from scipy.interpolate import UnivariateSpline

scipy_splines = []
p_values = [0.0, 0.5, 0.99, 1e5, 1e10]

for p in p_values:
    spline = UnivariateSpline(x=indices, y=random_data, s=p)
    scipy_splines.append(spline)

data_scipy = {'x': indices, 'Original': random_data}

for i, p in enumerate(p_values):
    data_scipy[f'p={p}'] = scipy_splines[i](indices)

data_scipy = pd.DataFrame(data_scipy)
```

```
data_scipy
```

	x	Original	p=0.0	p=0.5	p=0.99	p=100000.0	p=10000000000.0
0	0	3.146050	3.146050	3.145825	3.145874	1.032924	1.032924
1	1	-1.415000	-1.415000	-1.413868	-1.414113	1.030287	1.030287
2	2	6.796970	6.796970	6.794469	6.795010	1.027665	1.027665
3	3	2.968120	2.968120	2.971583	2.970833	1.025057	1.025057
4	4	0.408650	0.408650	0.404718	0.405569	1.022463	1.022463
...	...	...	...	...	...	...	...
1299	1299	1.605230	1.605230	1.609777	1.608793	0.882637	0.882637
1300	1300	-0.805483	-0.805483	-0.807541	-0.807096	0.878183	0.878183
1301	1301	-0.304650	-0.304650	-0.304321	-0.304392	0.873712	0.873712
1302	1302	-5.641990	-5.641990	-5.641856	-5.641885	0.869223	0.869223
1303	1303	-4.010890	-4.010890	-4.010944	-4.010932	0.864718	0.864718

```
splines_visualisation(data_scipy)
```



Как можно заметить, результаты полного сглаживания вышел еще хуже. Теперь линия при p=1e10 даже не ровная. Можно проследить, что она следует за тенденцией изменения значений. Она чем-то похожа на вариант ручной реализации C++ без доп регуляризации при p>0.8. Подобное поведение явно не всегда будет удобно

Вывод:

В ходе работы над данной лабораторной работой была написана реализация сглаживающего сплайна на языке Python и C++. Было проведено сравнение ручной реализации и библиотечной на двух языках программирования. Все сплайны при значении параметра p=0 полностью интерполируют точки распределения

Если говорить про ручную реализую на обоих языках, то серьезного отличия я выделить не могу. Но, вероятнее всего, реализация на Python менее точна, чем на C++.

Говоря про реализацию сплайнов в библиотеке scipy, я могу выделить, что данные способы будут удобны далеко не во всех случаях. Они менее гибкие. Не так универсальны, как то, что было написано руками.

Но бы не сказал, что Make_Smoothing_Spline работает плохо. Он выполняет поставленную задачу, но делает это по-другому, при больших значениях параметра сглаживания больше как раз напоминая алгоритм линейной регрессии

В свою очередь scipy.UnivariateSpline явно уступает своим конкурентам. Перед использованием данного сплайна будет необходимо провести начальную обработку данных для достижения наилучшего результата. Например, избавить данные от выбросов или использовать разные веса для разных точек (для этого, кстати говоря, предусмотрен параметр в обоих методах scipy)