Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

(Университет ИТМО)

Кафедра Систем Управления и Информатики

Лабораторная работа №2

Выполнил:

Таякин Д.Р.

Проверил:

Мусаев А.А.

Санкт-Петербург, 2022

1 Задание

В первом задании нужно было создать модуль, позволяющий производить возведение в квадрат, транспонирование и нахождение определителя матрицы. Подключить данный модуль к новой программе, а также проверить быстродейсвтвие реализованных функций с NumPy функциями.

На рисунке 1.1 представлен основной код программы с подключенным модулем “custom_module”, в котором содержится класс Matrix для выполнения ранее описанных действий с матрицами.

Lab-2 > task_1.py > ...

```python
import custom_module as cm
import time, numpy as np

M = cm.Matrix(2,2, [1,2,3,4])

print("Custom Matrix Module")
t_start1 = time.perf_counter()
print(M.squared())
print("Время работы squared: %s секунд " % (time.perf_counter() - t_start1))

t_start2 = time.perf_counter()
print(M.transpose())
print("Время работы transpose: %s секунд " % (time.perf_counter() - t_start2))

t_start3 = time.perf_counter()
print(M.det())
print("Время работы squared: %s секунд " % (time.perf_counter() - t_start3))

print("————————————————————————————————————————————————————")
print("NumPy")

A = np.matrix("1 2; 3 4")

t_start1 = time.perf_counter()
print(np.matmul(A, A))
print("Время работы squared: %s секунд " % (time.perf_counter() - t_start1))

t_start2 = time.perf_counter()
print(np.transpose(A))
print("Время работы transpose: %s секунд " % (time.perf_counter() - t_start2))

t_start3 = time.perf_counter()
print(np.linalg.det(A))
print("Время работы det: %s секунд " % (time.perf_counter() - t_start3))

```

Рисунок 1.1 – Основной код программы с подключенным модулем

На рисунках 1.2 и 1.3 изображен код из подключаемого модуля. В первой части кода находятся методы создания матрицы и транспонирования. Во второй части кода представлены метод нахождения определителя и вспомогательные функции.

Lab-2 > custom_module.py > Matrix > transpose

```python
class Matrix:
    def __init__(self, row_count: int, col_count: int, data: list):
        self.create(row_count, col_count, data)

    def create(self, row_count: int, col_count: int, data: list):
        self.rows = row_count
        self.cols = col_count
        self.mat = []
        for i in range(row_count):
            row_list = []
            for j in range(col_count):
                row_list.append(data[row_count * i + j])
            self.mat.append(row_list)

    def transpose(self):
        matrix_T = []
        for j in range(self.cols):
            row=[]
            for i in range(self.rows):
                row.append(self.mat[i][j])
            matrix_T.append(row)

        return matrix_T
```

Рисунок 1.2 – Класс Matrix в модуле “custom_module”

```python
    def det(self):
        if self.cols != self.rows:
            print('Number of columns must be equal to number of rows.')
            return

        if self.rows == 1:
            return self.mat[0]
        elif self.rows == 2:
            return self.mat[0][0] * self.mat[1][1] - self.mat[1][0] * self.mat[0][1]
        else:
            summ = 0
            for i in range(self.cols):
                minor = self.minor(0, i)
                flat_minor_list = [item for sublist in minor for item in sublist]
                summ += ((-1)**i) * self.mat[0][i] * Matrix(len(minor), len(minor[0]), flat_minor_list).det()
            return summ

    def minor(self, i, j):
        return [row[:j] + row[j+1:] for row in (self.mat[:i] + self.mat[i+1:])]

    def squared(self):
        if self.cols != self.rows:
            print('Number of columns must be equal to number of rows.')
            return

        C = self.zeros_matrix(self.rows, self.cols)
        for i in range(self.rows):
            for j in range(self.cols):
                total = 0
                for ii in range(self.cols):
                    total += self.mat[i][ii] * self.mat[ii][j]
                C[i][j] = total

        return C

    def zeros_matrix(self, rows, cols):
        return [[0 for _ in range(cols)] for _ in range(rows)]
```

Рисунок 1.3 – Основная логика фасетного поиска

На рисунке 1.4 и 1.5 представлены результаты работы программы с использованиям кастомного модуля и с использованием NumPy. На первом рисунке в качестве входных данных были матрицы размерностью 2x2. На втором рисунке - 5x5.

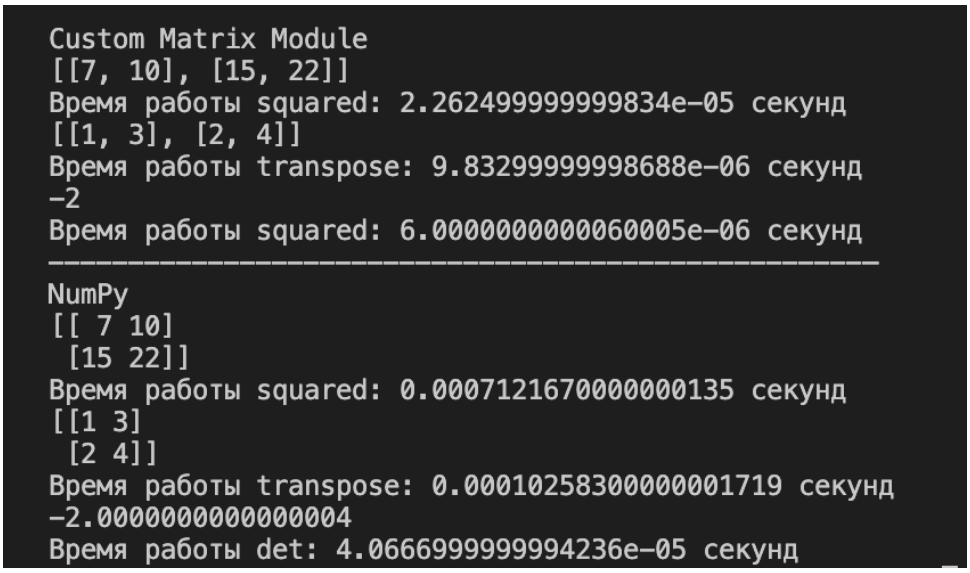

Рисунок 1.4 – Результаты работы программы с входными данными матриц 2x2

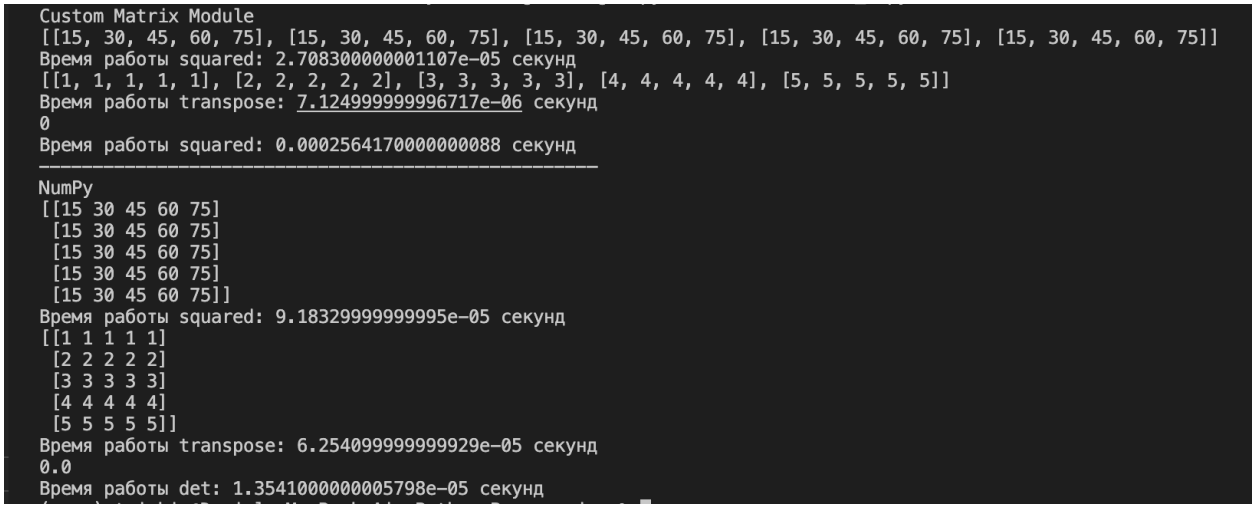

Рисунок 1.5 – Результаты работы программы с входными данными матриц 5x5

2 Задание

Во втором задании нужно было реализовать алгоритм, восстанавливающий данные путем линейной аппроксимации и реализовать алгоритм, восстанавливающий значения путем корреляционного восстановления. Входными данными является таблица nxn, заполненная случайнми величинами от 1 до 30 и пользовательская выборка.

На рисунке 2.1 изображена первая часть кода. Здесь можно увидеть импортированные модули, и некоторые вспомогательные функции для генерации входных данных и просмотра графика для полученных данных.

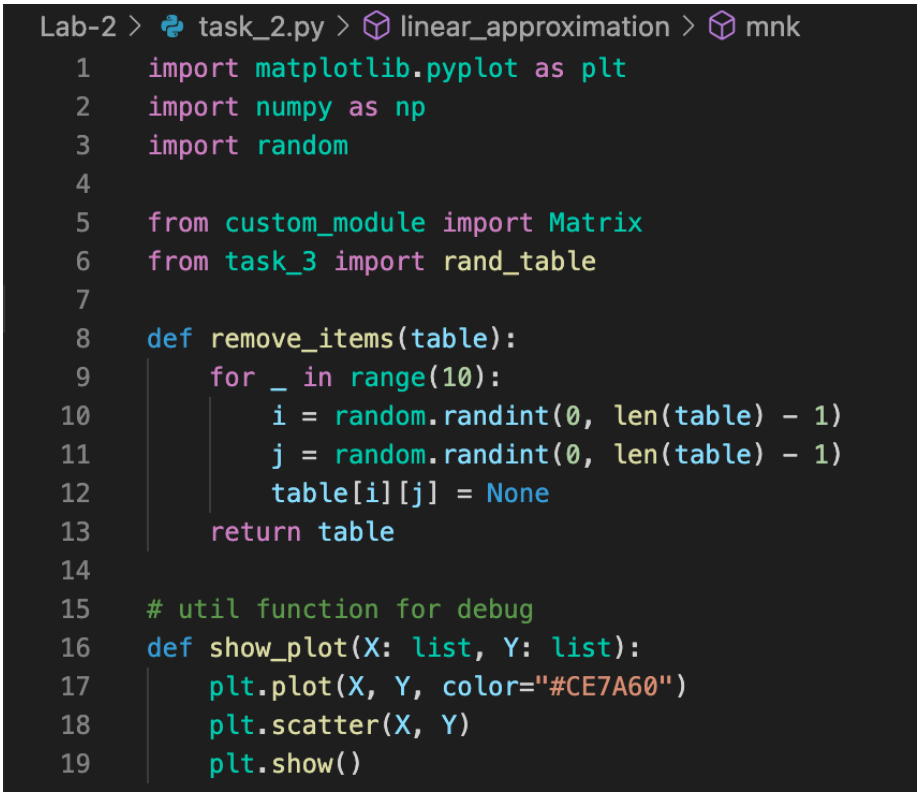

Рисунок 2.1 – Импортируемые модули и вспомогательные функции

На рисунке 2.2 представлена реализация линейной аппроксимации, которая использует метод МНК для нахождения аппроксимирующей прямой. На самом деле, можно было и обойтись без этой функции, но я решил включить ее для демонстрации работы линейной аппроксимации. Функция

find_nearest_elements находит ближайшие точки по которым можно аппроксимировать ряд.

```python
def linear_approximation(table):
    # Using MNK method for approximation
    def mnk(x: list, y: list):
        n = len(x)

        sum_x = sum(x); sum_y=sum(y)
        sum_xy = sum(map(lambda x, y: x * y, x, y))
        sum_x2 = sum(map(lambda x: x ** 2, x))

        det_m = Matrix(2, 2, [sum_x2, sum_x, sum_x, n]).det()
        det_a = Matrix(2, 2, [sum_xy, sum_x, sum_y, n]).det()
        det_b = Matrix(2, 2, [sum_x2, sum_xy, sum_x, sum_y]).det()

        a = det_a / det_m
        b = det_b / det_m

        return a, b

    def find_nearest_elements(Y: list, index):
        prv = index; nxt = index
        while True:
            if nxt == len(Y) - 1:
                nxt = 0

            prv -= 1 if Y[prv] == None else 0
            nxt += 1 if Y[nxt] == None else 0

            if Y[prv] != None and Y[nxt] != None:
                break

        return prv, nxt

    for i_Y, Y in enumerate(table):
        X = [x for x in range(len(Y))]

        indices = [i for i, y in enumerate(Y) if y == None]
        for i in indices:
            prv, nxt = find_nearest_elements(Y, i)

            a, b = mnk([X[prv], X[nxt]], [Y[prv], Y[nxt]])
            table[i_Y][i] = a*X[i] + b

    return table
```

Рисунок 2.2 – Реализация линейной аппроксимации

```python
def correlation(r_table, a, b):
    items1 = [0 if x == None else x for x in r_table[a]]
    items2 = [0 if x == None else x for x in r_table[b]]

    indices1 = [i for i, y in enumerate(r_table[a]) if y == None]
    indices2 = [i for i, y in enumerate(r_table[b]) if y == None]

    coef = np.corrcoef(items1, items2)[0][1]

    for i in indices1:
        if r_table[a][i] is None and r_table[b][i] is not None:
            r_table[a][i] = r_table[b][i] * coef
        elif r_table[b][i] is None and r_table[a][i] is not None:
            r_table[b][i] = r_table[a][i] * coef

    for i in indices2:
        if r_table[a][i] is None and r_table[b][i] is not None:
            r_table[a][i] = r_table[b][i] * coef
        elif r_table[b][i] is None and r_table[a][i] is not None:
            r_table[b][i] = r_table[a][i] * coef

    return r_table
```

Рисунок 2.3 – Реализация корреляции

На рисунке 2.3 представлена реализация корреляции. Аргументы функции представляют собой таблицу, с котрой идет взаимодействие и номера рядов, которые пользователь хочет коррелировать. В качестве вычисления коэфициентов я использовал встроенную функцию Numpy, которая использует вычисления Пирсона.

```
88  if __name__ == "__main__":
89      table = remove_items(rand_table())
90
91      print(
92  """
93  1. Linear approximation
94  2. Correlation
95  """)
96
97      opt = int(input("Select option: "))
98
99      if opt == 1:
100         print("Before:", table, "\n")
101         linear_table = linear_approximation(table)
102         print("After:", linear_table)
103     elif opt == 2:
104         r_table = table.copy()
105         while True:
106             print(
107 """
108 Select two columns that you want correlate:
109 (Enter two numbers with a space between)
110 """)
111
112             for i, x in enumerate(r_table):
113                 print(i, x)
114
115             a, b = map(int, input("Columns: ").split())
116
117             r_table = correlation(r_table)
118             print(r_table)
119     else:
120         exit(1)
121
```

Рисунок 2.4 – Обработка данных введенных пользователем и использование функций

На рисунке 2.4 представлена обработка входных данных и вывод результатов. У пользователя есть выбор алгоритма: линейная аппроксимация или корреляция. Если пользователь выбирает первый метод, то результатом будет таблица с линейно восстановленными значениями. Если пользователь выбрал второй метод, то он далее должен выбрать ряды, которые он хочет коррелировать. После каждой корреляцией двух рядов будет выводиться результат и продолжение работы программы, которая будет ожидать новые ряды для корреляции.

3 Задание

В третьем задании нужно было реализовать алгоритм поиска математического ожидания и дисперсии для каждого ряда таблицы размера nxn, заполненной случайными величинами от 1 до 30.

Lab-2 > task_3.py > D

```python
import random

def rand_table(n=10):
    return [[random.randint(1, 30) for _ in range(n)] for _ in range(n)]

def M(X, n):
    p = 1/n
    return p*sum(X)

def D(X, n):
    p = 1/n
    result = 0
    for a in X:
        result += (a - M(X, n))**2
    return p*result

table = rand_table()
for i, col in enumerate(table):
    print("————————————————————————————————————")
    print(f"Column {i}\n")
    print("Expected value: ", M(col, len(col)))
    print("Variance: ", D(col, len(col)))
    print("————————————————————————————————————")
```

Рисунок 3.1 – Код программы 3-го здания

На рисунке 3.1 предтавлен код программы. Функция rand_table создает таблицу со случайными величинами (стандартная размерность 10x10). Функция M представляет собой поиск математического ожидания, на вход которой подается ряд чисел и размер этого ряда. Для вычисления данного значения используется формула из определения дискретной величины: $M[X] = \Sigma_{i=1}^{\infty} x_i p_i$. Распределения вероятностей $P(X) = p_i = \frac{1}{n}$, $\Sigma_{i=1}^{\infty} p_i = 1$. Функция D также принимает ряд и размер и вычисляет дисперсию по формуле $D[X] = \Sigma_{i=1}^{n} p_i (x_i - M[X])^2$.