Научно-технологический университет «Сириус»

Научный центр информационных технологий и искусственного интеллекта

Направление «Математическая робототехника»

Отчёт

по заданию №1

Дисциплина: численные методы нелинейной и выпуклой оптимизации

Тема: аппроксимация

Выполнил

Студент группы М01МР-24 __________________А. С. Кондратьев

Преподаватель

Профессор, к.ф.-м.н. _______________________С. В. Гусев

Сириус

2025

Задача 1

Построим многочлен пятой степени $y = p(x)$ в точках (x_i, y_i), где $x_i = 0.1i$, $i = [0, 100]$, который удовлетворяет неравенствам

$$0 \le y \le 5$$

$$y_0 \le 1,\ y_{30} \ge 4$$

$$y_{70} \le 1$$

$$2 \le y_{100} \le 3$$

Для этого сведем данные неравенства к задаче минимизации константы с ограничениями. Для полинома пятой степени необходимо определить шесть коэффициентов (представлены в таблице 1).

$$p(x) = a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

Таблица 1 – Коэффициенты полинома

a_0	0.88465826
a_1	-0.1854891
a_2	1.97192655
a_3	-0.76100487
a_4	0.09711716
a_5	-0.00404296

График полинома представлен на рисунке 1. Красными точками отмечены значения полинома, на которые наложены ограничения-неравенства. Соответствие полинома ограничениям представлено в таблице 2.

Таблица 2 – Соответствие полинома ограничениям

Ограничение	Значение полинома
$y_0 \le 1$	$y_0 = 0.88$
$y_{30} \ge 4$	$y_{30} = 4.41$
$y_{70} \le 1$	$y_{70} = 0.41$
$2 \le y_{100} \le 3$	$y_{100} = 2.09$

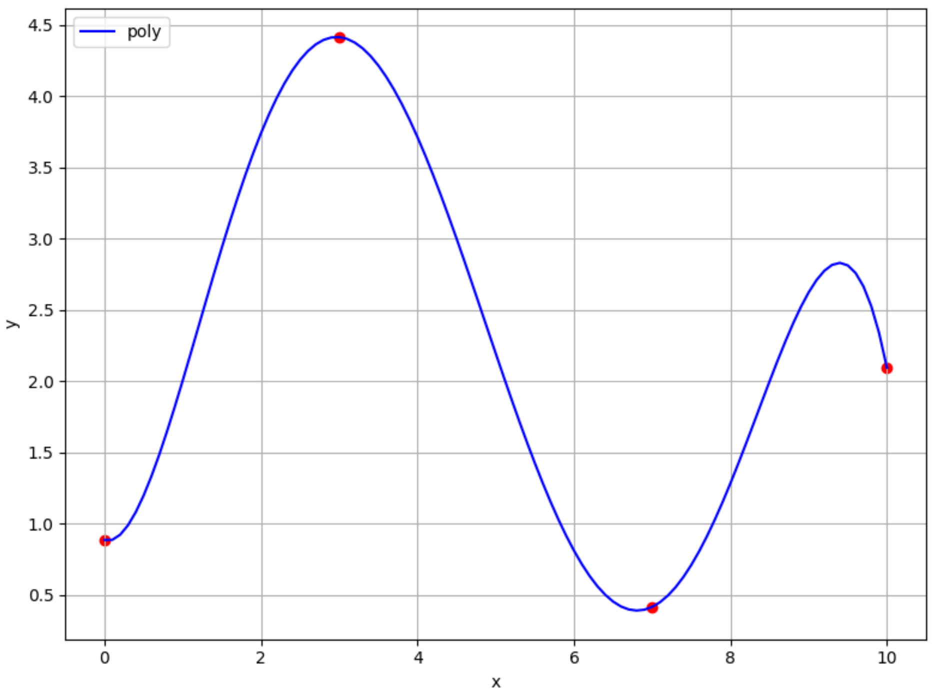

Рисунок 1 – Полином пятой степени

Задача 2

Для построенного полинома сгенерируем гауссовский шум w со средним 0 и дисперсией 0.3 и получим зашумленные измеренные значения (рисунок 2)

$$z_i = y_i + w_i$$

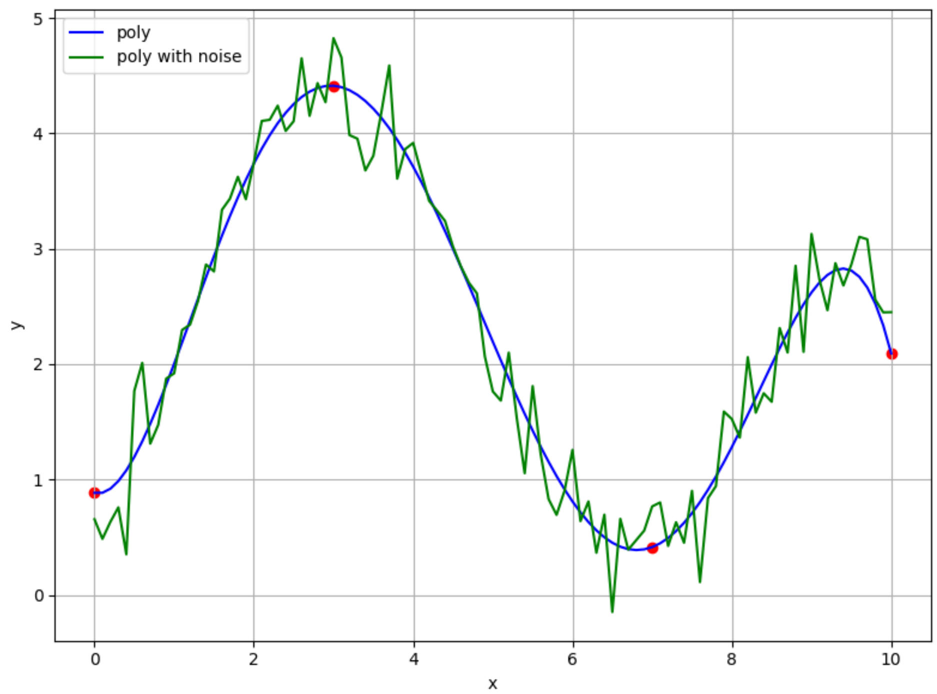

Рисунок 2 – Зашумленные измерения

Задача 3

Оценим значения коэффициентов полинома по зашумленным измерениям.

1. Метод наименьших квадратов (МНК). Для определения коэффициентов полинома с помощью МНК необходимо использовать формулу

$$x = (A^T A)^{-1} A^T z$$

где $A = 1 + x + x^2 + x^3 + x^4 + x^5$ – прямоугольная матрица, у которой число столбцов равно числу коэффициентов полинома.

2. Аппроксимация Чебышева. При использовании аппроксимации Чебышева для определения коэффициентов полинома необходимо минимизировать норму

$$\|Ax - z\|_\infty$$

Данную задачу минимизации можно свести к задаче линейного программирования

$$\min t \text{ при } -t \leq Ax - z \leq t$$

3. Минимизация суммы модулей ошибок. Сумму модулей ошибок можно представить в виде первой нормы

$$\|Ax - z\|_1$$

и свести к задаче линейного программирования

$$\min 1^T t \text{ при } -t \leq Ax - z \leq t$$

4. Минимизация суммы значений штрафной функции $\varphi = \sqrt{|t|}$, где $t = Ax - z$. Данную задачу нельзя свести к задаче линейного программирования.

При использовании МНК коэффициенты полинома вычислены аналитически. Для вычисления коэффициентов полинома с помощью Аппроксимации Чебышева, минимизации первой нормы использовался пакет CVX (Convex Optimization) [1], позволяющий решать задачу линейного программирования. Для минимизации штрафной функции использовались следующие пакеты:

1.SciPy Optimization [2], позволяющий решать невыпуклые задачи оптимизации (солвер L-BFGS-B).

2. PyTorch [3], позволяющий минимизировать целевую функцию с помощью метода градиентного спуска (выбран шаг 1e-5).

Для минимизации штрафной функции были выбраны разные начальные приближения. На рисунке 3 представлены результаты с начальным приближением в виде интерполяции по шести измерениям (индексы [0, 20, 40, 60, 80, 100]), рисунке 4 – вектора **0**, рисунке 5 – вектора **1**, рисунке 6 – результатов МНК.

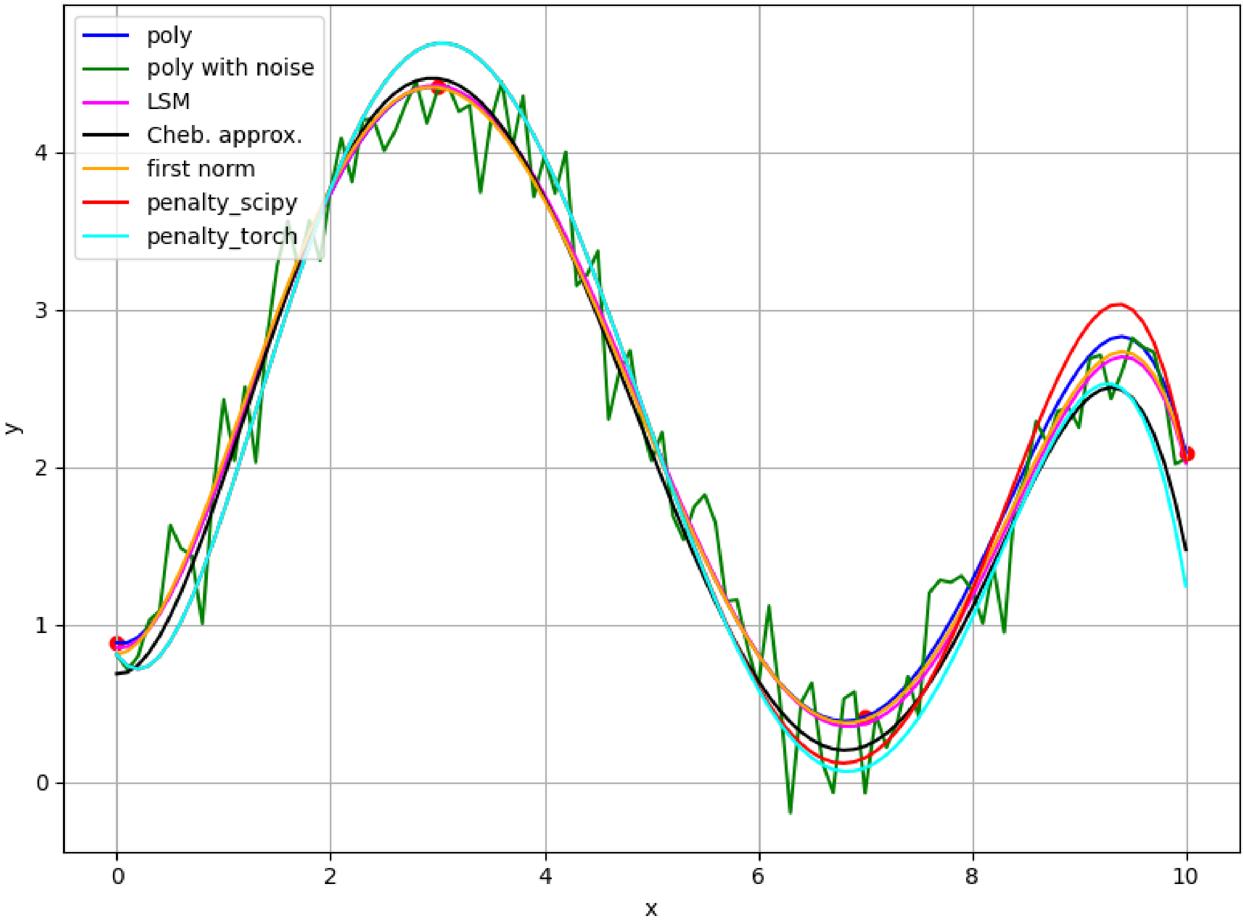

Рисунок 3 – Восстановленные полиномы (нач. приближение – интерполяция)

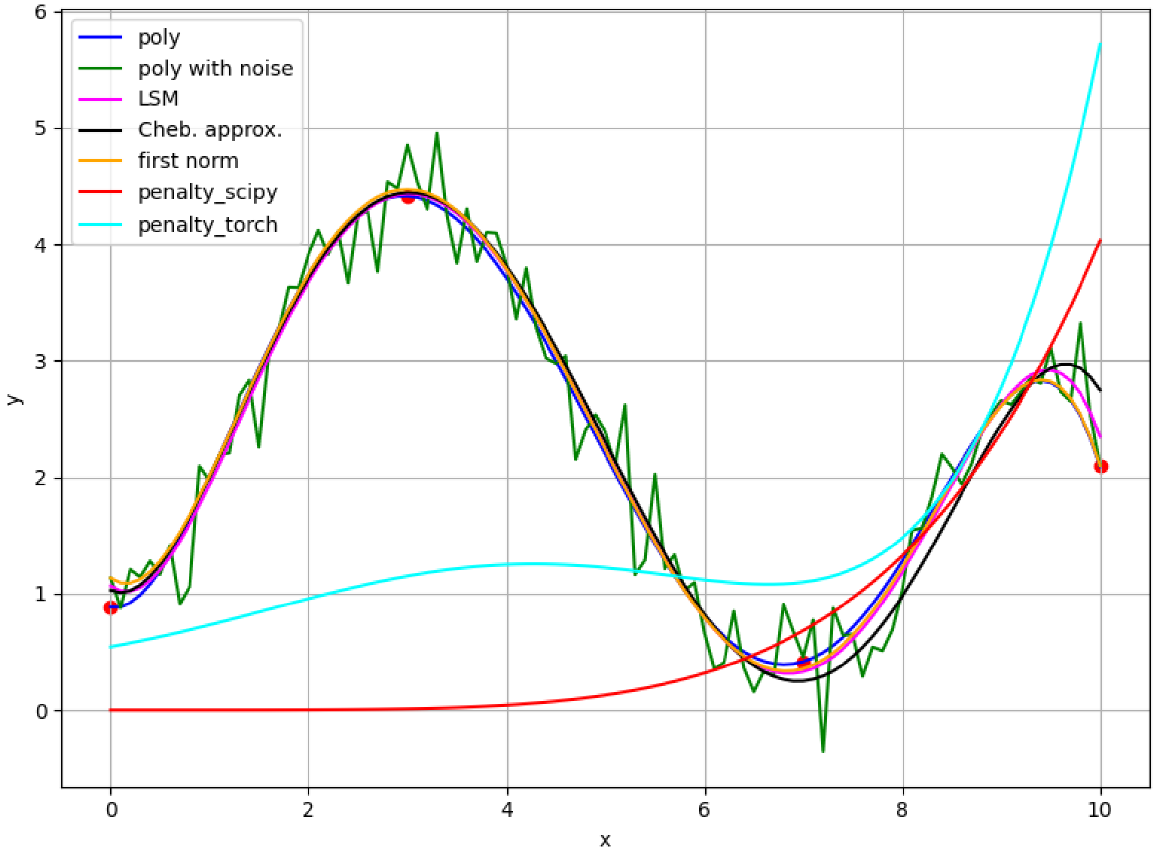

Рисунок 4 – Восстановленные полиномы (начальное приближение – вектор **0**)

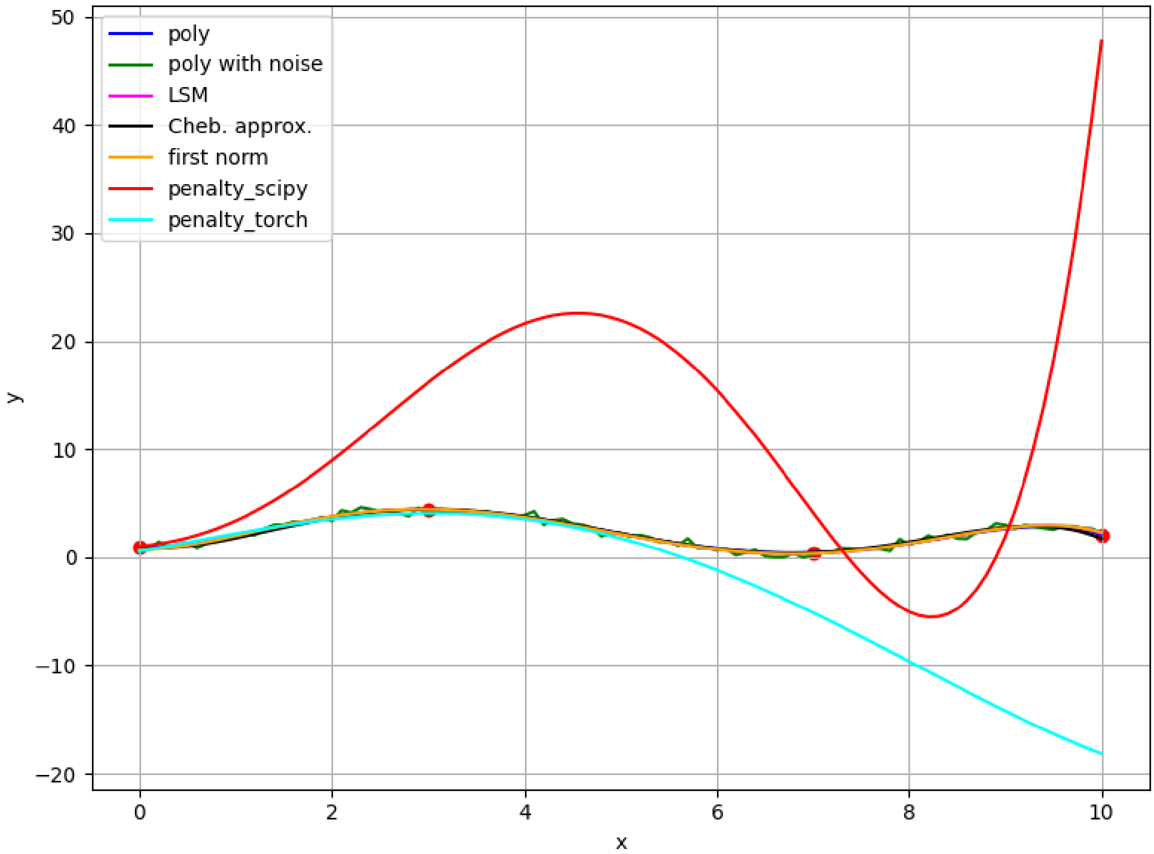

Рисунок 5 – Восстановленные полиномы (начальное приближение – вектор **1**)

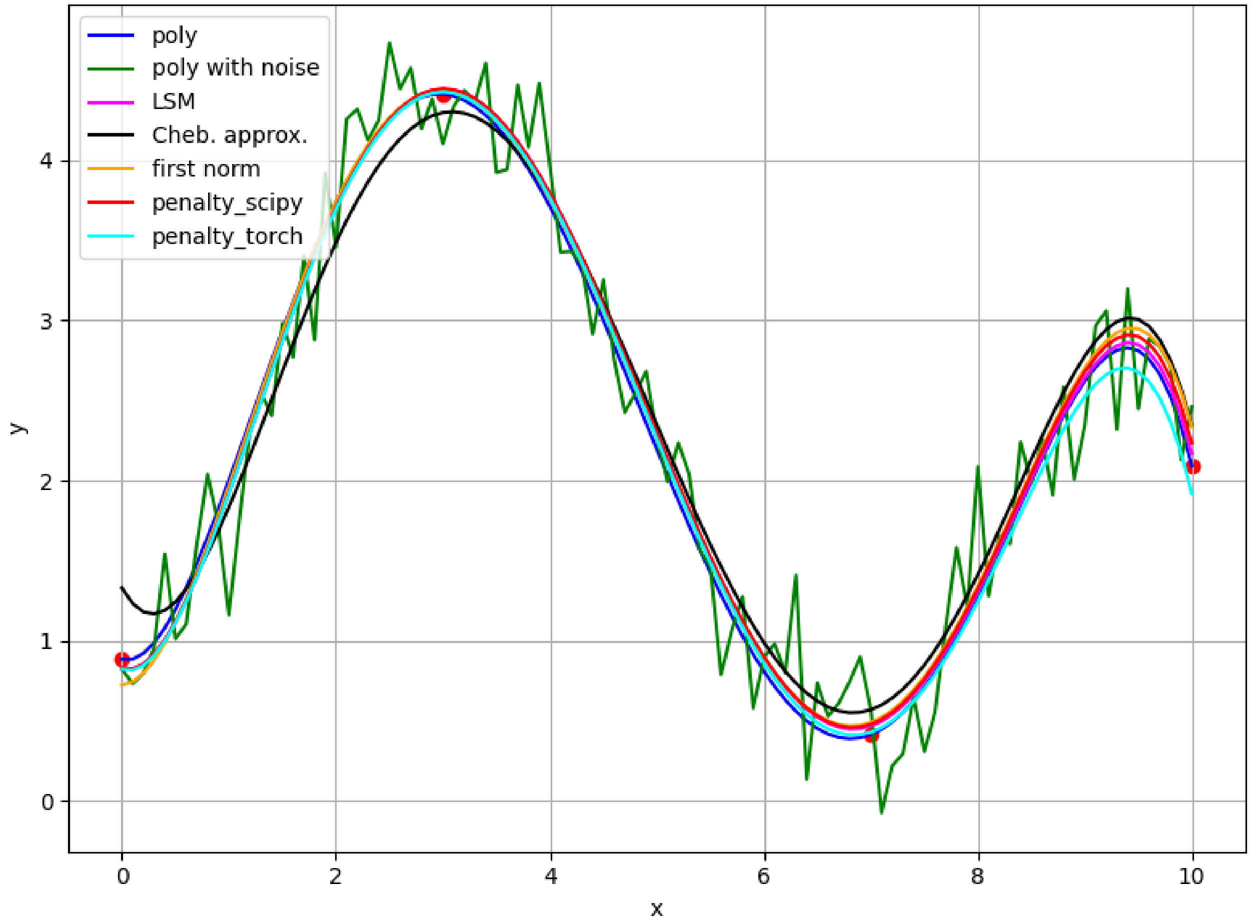

Рисунок 6 – Восстановленные полиномы (начальное приближение – МНК)

Проанализируем точность использованных методов с помощью вычисления максимальной для всех шести коэффициентов среднеквадратичной ошибки (MSE). Вычисленные значения представлены в таблице 3.

Таблица 3 – Среднеквадратичная ошибка значений полинома

Метод	MSE
МНК	0.233379
Аппроксимация Чебышева	0.180228
Первая норма	0.148730
Штрафная функция (SciPy) (нач. прибл. – **0**)	3.887829
Штрафная функция (PyTorch) (нач. прибл. – **0**)	3.585770
Штрафная функция (SciPy) (нач. прибл. – **1**)	2.021041
Штрафная функция (PyTorch) (нач. прибл. – **1**)	3.776992
Штрафная функция (SciPy) (нач. прибл. – МНК)	0.305351
Штрафная функция (PyTorch) (нач. прибл. – МНК)	0.279268
Штрафная функция (SciPy) (нач. прибл. – интерполяция)	1.698965
Штрафная функция (PyTorch) (нач. прибл. – интерполяция)	1.667321

Задача 4

Модифицируем шум. Для этого добавим к измерениям z случайную величину v, принимающую значение 0 с вероятностью 0.9, и величину в диапазоне $10 \leq |v| \leq 20$ с вероятностью 0.1. Зашумленные измерения представлены на рисунке 8.

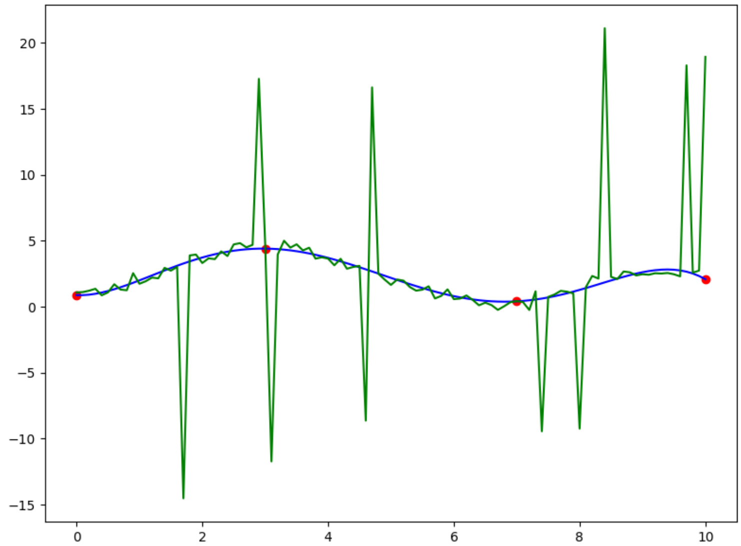

Рисунок 8 – Зашумленные измерения

Задача 5

Задача 5 аналогична задаче 3 за исключением характера шума. Оценка точности использованных методов в новых условиях с помощью вычисления максимальной среднеквадратичной ошибки (MSE) представлена в таблице 4. Графики полиномов с различными начальными приближениями для штрафной функции (с нач. прибл. на основе интерполяции, вектором 0, вектором 1, с нач. прибл. на основе МНК) представлены на рисунках 9, 10, 11, 12 соответственно.

Таблица 4 – Максимальная среднеквадратичная ошибка значений полинома

Метод	MSE
МНК	5.396803
Аппроксимация Чебышева	62.868552
Первая норма	0.078927
Штрафная функция (SciPy) (нач. прибл. – **0**)	3.887720
Штрафная функция (PyTorch) (нач. прибл. – **0**)	3.625318
Штрафная функция (SciPy) (нач. прибл. – **1**)	3.015516
Штрафная функция (PyTorch) (нач. прибл. – **1**)	3.666388
Штрафная функция (SciPy) (нач. прибл. – МНК)	5.396803
Штрафная функция (PyTorch) (нач. прибл. – МНК)	7.220629
Штрафная функция (SciPy) (нач. прибл. – интерполяция)	5.244933
Штрафная функция (PyTorch) (нач. прибл. – интерполяция)	5.161705

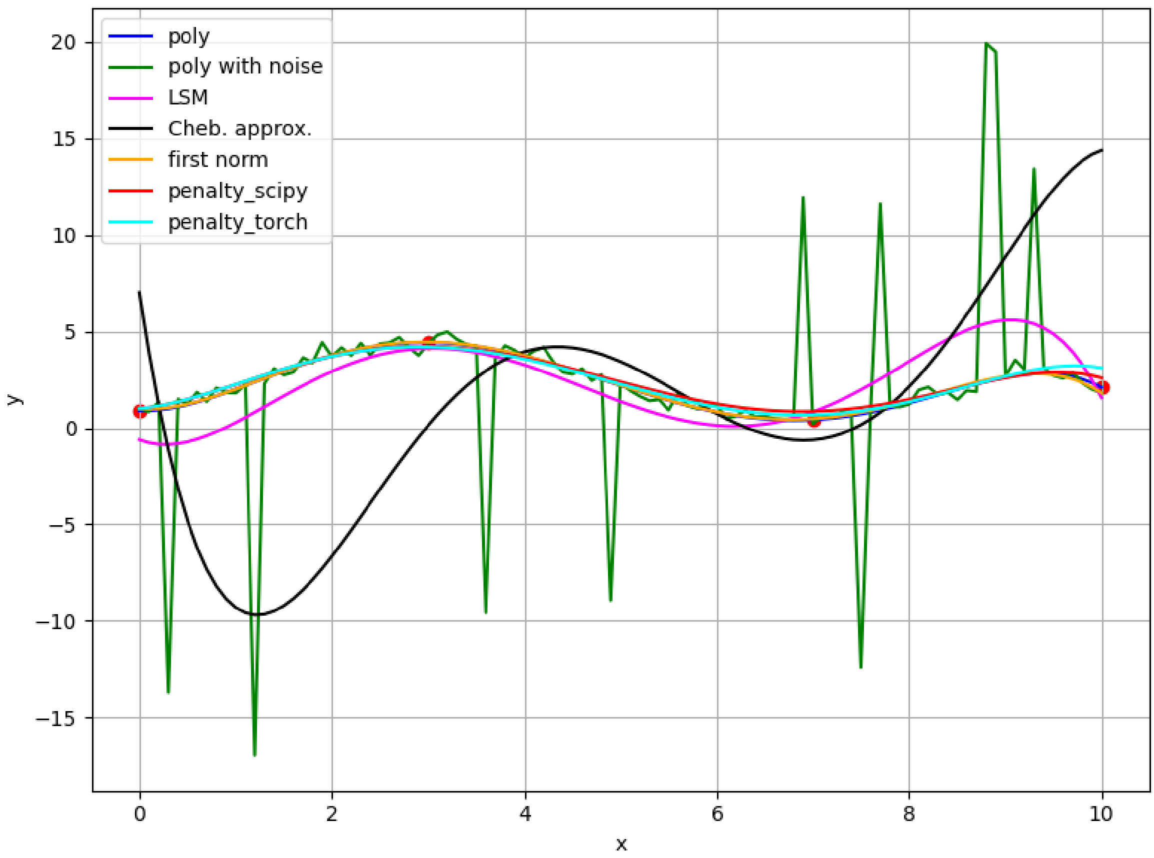

Рисунок 9 – Восстановленные полиномы (нач. приближение – интерполяция)

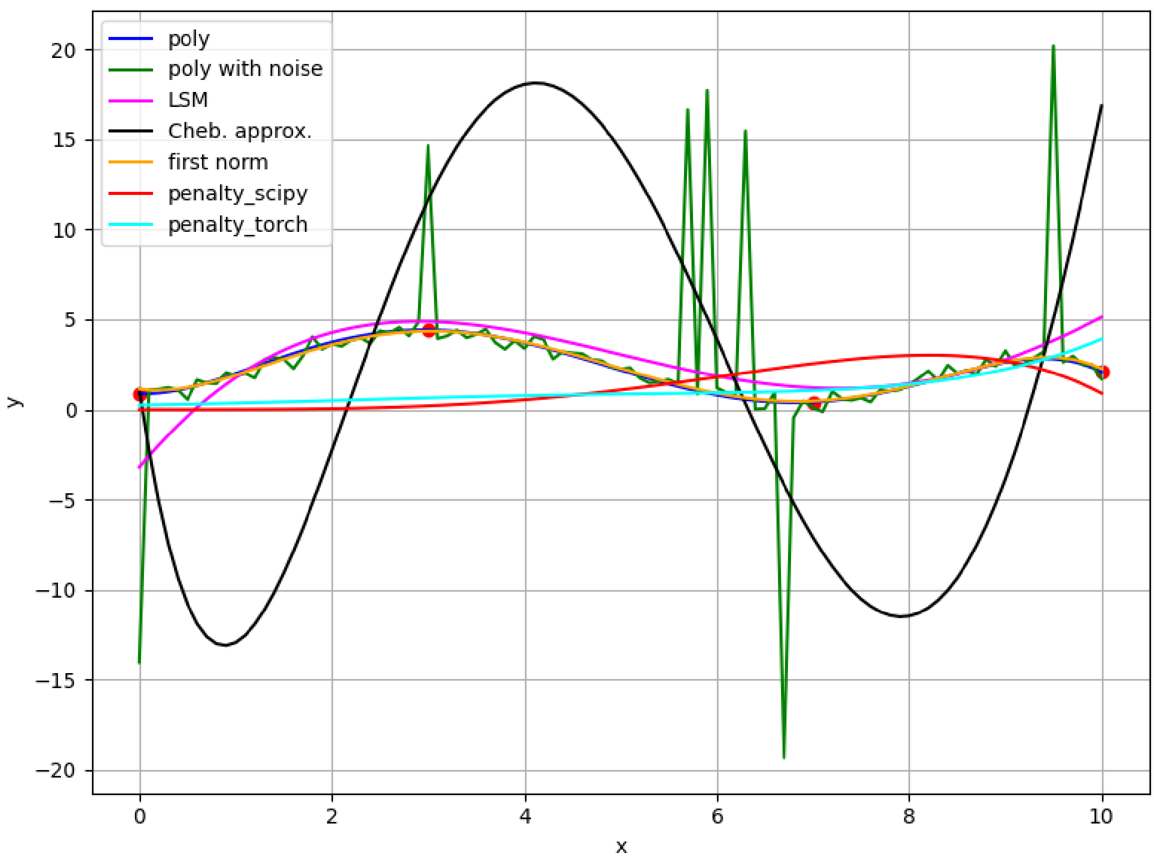

Рисунок 4 – Восстановленные полиномы (начальное приближение – вектор **0**)

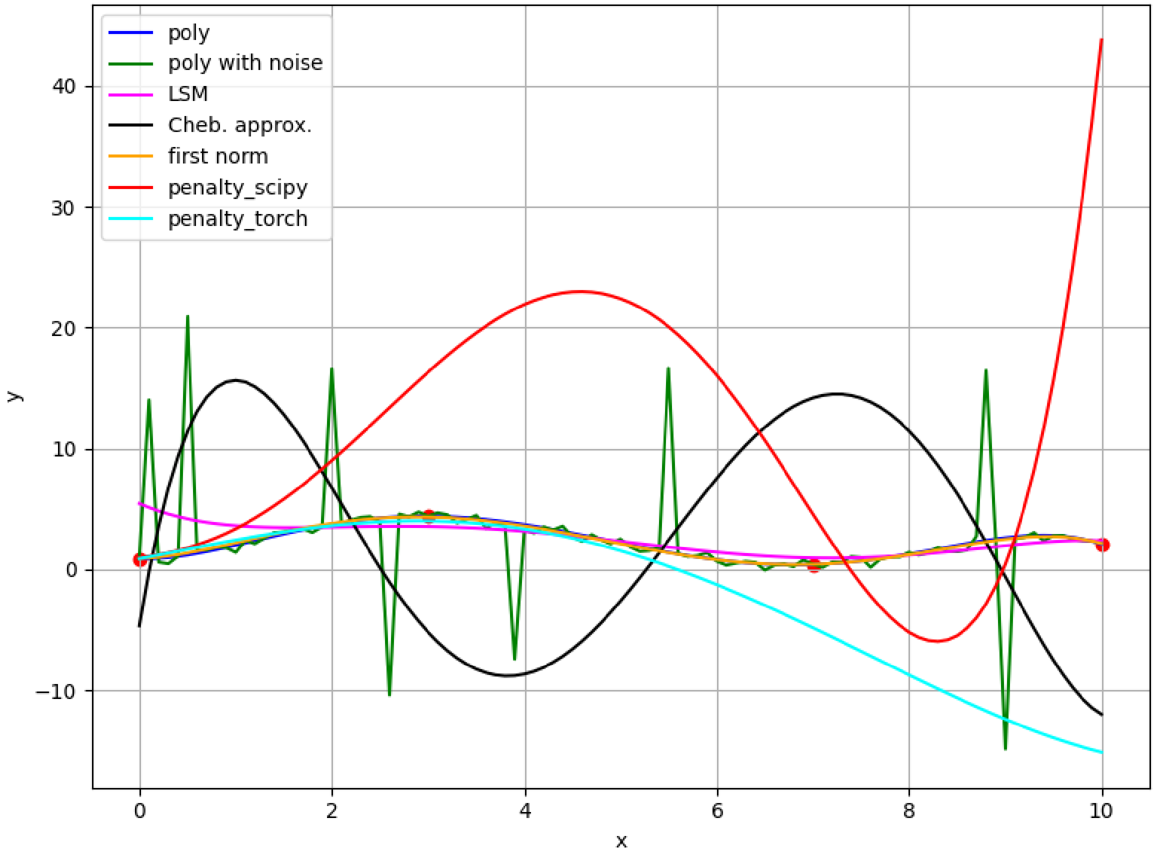

Рисунок 5 – Восстановленные полиномы (начальное приближение – вектор **1**)

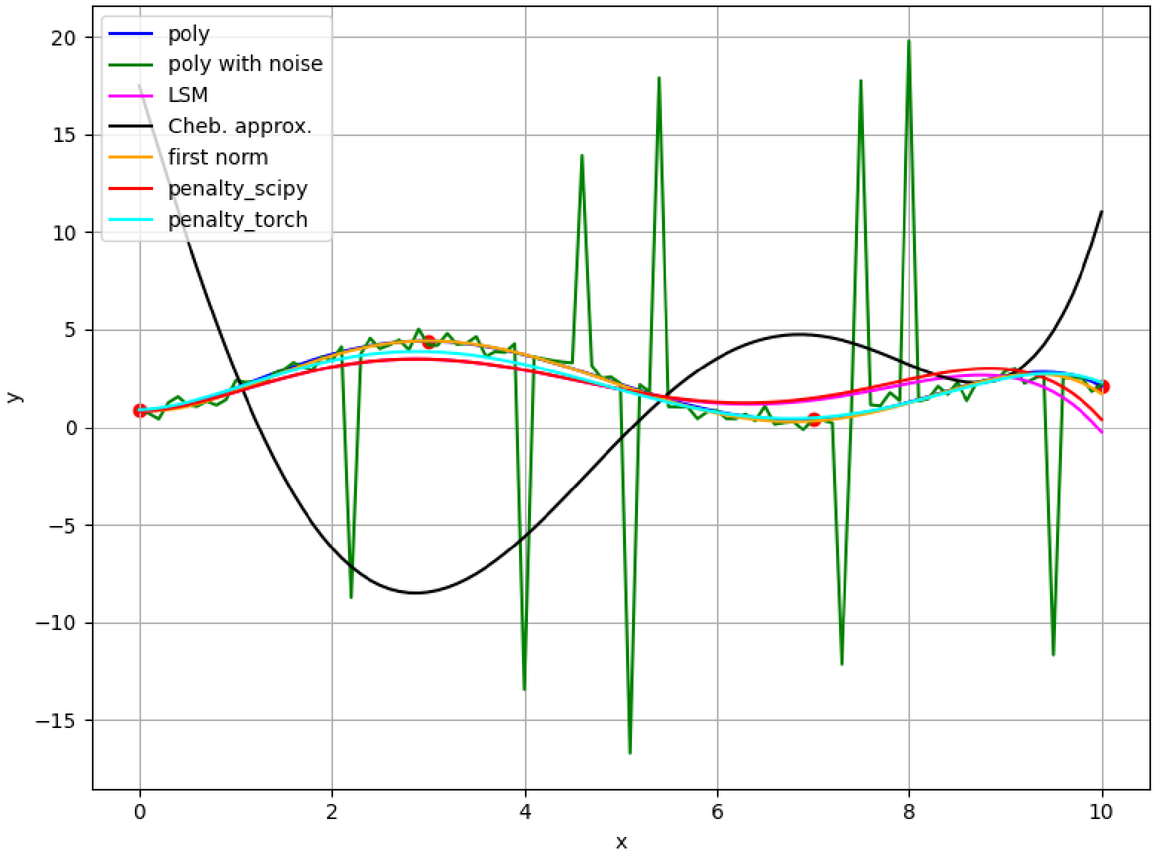

Рисунок 6 – Восстановленные полиномы (начальное приближение – МНК)

Заключение

Результаты оценки точности работы рассмотренных алгоритмов показывают, что в условиях белого шума методы выпуклой минимизации работают лучше: результаты минимизации штрафной функции сильно зависят от выбора начального приближения, в то время как МНК, аппроксимация Чебышева, а также минимизация первой нормы выдают стабильный результат.

В случае добавления к белому шуму выбросов метод наименьших квадратов стал показывать результаты хуже, т.к. он «притягивается» к большим выбросам. Аппроксимация Чебышева не может работать в условиях выбросов. Использование первой нормы показало наилучший результат в условиях присутствия выбросов в измерениях. Методы невыпуклой оптимизации показывают лучшие, по сравнению с МНК и аппроксимацией Чебышева, результаты, но все так же сильно зависят от начального приближения и шага.

Таким образом, можно сделать вывод о том, что использование таких методов, как градиентный спуск в условиях невыпуклой оптимизации не гарантирует достижение глобального минимума и сильно зависит как от начального приближения, так и от шага. Однако в случае успешного подбора параметров возможно достижение достаточных результатов.

Приложение №1

```
import cvxpy as cp
import matplotlib.pyplot as plt
import numpy as np
import torch

from scipy.optimize import minimize


def gradient_descent_torch(loss_function, x0 : torch.tensor, h : float, max_iter
: int, tolerance : float) -> torch.tensor:
    """
    Минимизирует значение вектора x

    Аргументы:
    loss_function (np.array): функция потерь (целевая функция)
    x0 (np.array): начальное приближение
    h (float): шаг градиентного спуска (learning rate)
    max_iter (int): максимальное количество итераций
    tolerance (float): минимальный порог изменения функции, при котором алгоритм
продолжает работу

    Возвращаемое значение:
    x (np.array): минимизированное значение
    """

    x0.requires_grad_(True)

    # optimizer = torch.optim.SGD([x0], lr=h)
    optimizer = torch.optim.Adam([x0], lr=h)

    prev_loss = float('inf')

    for _ in range(max_iter):
        optimizer.zero_grad()
        loss = loss_function(x0)
        loss.backward()

        optimizer.step()

        if abs(prev_loss - loss.item()) < tolerance:
            break
```

```
        prev_loss = loss.item()

    return x0


# Задача 1. Генерация значений полинома

# x_i = 0.1 * i, i = [0, .., 100]
x = np.arange(0, 10.1, 0.1)
x = np.reshape(x, (x.shape[0], 1))

# y = p(x) = a5 * x^5 + a4 * x^4 + a3 * x^3 + a2 * x^2 + a1 * x + a0
A = np.hstack([np.ones(x.shape), x, x ** 2, x ** 3, x ** 4, x ** 5])

# Описание ограничений на значение полинома

# y >= 0
A_ub1 = -A
b_ub1 = np.zeros(x.shape)

# y <= 5
A_ub2 = A
b_ub2 = np.zeros(x.shape) + 5

# y(0) <= 1
A_ub3 = np.array([[1, 0, 0, 0, 0, 0]])
b_ub3 = 1

# y(3) >= 4
A_ub4 = -np.array([[3 ** i for i in range(6)]])
b_ub4 = -4

# y(7) <= 1
A_ub5 = np.array([[7 ** i for i in range(6)]])
b_ub5 = 1

# y(10) <= 3
A_ub6 = np.array([[10 ** i for i in range(6)]])
b_ub6 = 3

# y(10) >= 2
A_ub7 = -np.array([[10 ** i for i in range(6)]])
```

```
b_ub7 = -2

# Общие ограничения
A_ub = np.vstack([A_ub1, A_ub2, A_ub3, A_ub4, A_ub5, A_ub6, A_ub7])
b_ub = np.vstack([b_ub1, b_ub2, b_ub3, b_ub4, b_ub5, b_ub6, b_ub7])

# Поиск коэффициентов полинома сводится к решению задачи оптимизации константы,
# т.к. в таком случае коэффициенты будут любыми из множества, заданного ограничениями, что нас устраивает
coeffs = cp.Variable(6)
prob = cp.Problem(cp.Minimize(0), [A_ub @ coeffs <= b_ub])
result = prob.solve(verbose=False)

coeffs = np.reshape(coeffs.value, (6, 1))  # приведение к вектору-столбцу

def poly(a, x):
    """
    Функция возвращает вектор значений полинома с коэффициентами a в точках x

    Аргументы:
    a (np.array): вектор коэффициентов полинома размера (6, 1)
    x (np.array): вектор аргументов полинома

    Возвращаемое значение:
    y (np.array): вектор значений полинома
    """
    return a[5] * x ** 5 + a[4] * x ** 4 + a[3] * x ** 3 + a[2] * x ** 2 + a[1] * x + a[0]

y = poly(coeffs, x)  # генерация значений полинома

fig1, ax1 = plt.subplots(figsize=(8, 6))

ax1.plot(x, y, color='blue', label='poly')  # вывод значений полинома
# вывод точек, участвующих в ограничениях
ax1.scatter([0, 3, 7, 10], [poly(coeffs, 0), poly(coeffs, 3), poly(coeffs, 7), poly(coeffs, 10)], color='red')

print('Task 1: ok')

# Задача 2. Генерация шума
```

```
noise = np.random.normal(loc=0, scale=0.3, size=x.shape)  # генерация шума с mu =
0, std = 0.3
z = y + noise  # генерация зашумленных измерений

ax1.plot(x, z, color='green', label='poly with noise')

print('Task 2: ok')

# Задача 3. Поиск коэффициентов полинома

calc_mse = False  # если True, расчет коэффициентов каждым методом производится
10 раз, иначе - 1 раз
N = 1 if not calc_mse else 10

lsm = np.linalg.inv(A.T @ A) @ A.T @ z   # Поиск коэффициентов с помощью
аналитического решения МНК
lsm_list = np.array([lsm for _ in range(N)])

# Аппроксимация Чебышева: min ||Ax - b||_inf -> min t при -t <= Ax - b <= t, t -
скаляр
cheb_list = []
for _ in range(N):
    t = cp.Variable()
    cheb = cp.Variable(6)
    cheb_problem = cp.Problem(cp.Minimize(t), [cp.abs(A @ cheb - z) <= t])
    cheb_problem.solve(verbose=False)
    cheb = np.reshape(cheb.value, (6, 1))
    cheb_list.append(cheb)
cheb_list = np.array(cheb_list)

# Минимизация первой нормы: min ||Ax - b||_1 -> -1^T * t <= Ax - b <= 1^T * t, t
- вектор
first_norm_list = []
for _ in range(N):
    t = cp.Variable(A.shape[0])
    first_norm = cp.Variable(6)
    first_norm_problem = cp.Problem(cp.Minimize(cp.sum(t)), [A @ first_norm - z
<= t, z - A @ first_norm <= t])
    first_norm_problem.solve(verbose=False)
    first_norm = np.reshape(first_norm.value, (6, 1))
    first_norm_list.append(first_norm)
first_norm_list = np.array(first_norm_list)
```

```
# Минимизация суммы значений штрафной функции sqrt(abs(t)), t = Ax - b

# целевая функция для солвера из scipy
# min sum(sqrt(abs(t))), t = Ax - b
def penalty_objective(coeffs):
    t = A @ coeffs - z
    # return np.sqrt(np.sum(np.abs(t)))
    return np.sum(np.sqrt(np.abs(t)))

A_interp = np.vstack([A[0], A[20], A[40], A[60], A[80], A[100]])  # матрица A для поиска начального приближения в виде интерполяции
z_interp = np.vstack([z[0], z[20], z[40], z[60], z[80], z[100]])  # вектор z для поиска начального приближения в виде интерполяции

penalty_scipy_coeffs_init_zeros = np.zeros(6)  # начальное приближение в виде вектора нулей
penalty_scipy_coeffs_init_ones = np.ones(6)  # начальное приближение в виде вектора единиц
penalty_scipy_coeffs_init_lsm = lsm.reshape(lsm.shape[0])  # начальное приближение на основе МНК
penalty_scipy_coeffs_init_interp = np.linalg.solve(A_interp, z_interp).reshape(6)  # начальное приближение на основе интерполяции

penalty_scipy_coeffs_init = penalty_scipy_coeffs_init_interp  # выбор начального приближения для scipy

penalty_scipy_coeffs_list = []
for _ in range(N):
    penalty_scipy_result = minimize(penalty_objective, penalty_scipy_coeffs_init, method='L-BFGS-B')
    penalty_scipy_coeffs = np.reshape(penalty_scipy_result.x, (6, 1))
    penalty_scipy_coeffs_list.append(penalty_scipy_coeffs)
penalty_scipy_coeffs_list = np.array(penalty_scipy_coeffs_list)

A_torch = torch.tensor(A, dtype=torch.float64)
z_torch = torch.tensor(z, dtype=torch.float64)

penalty_torch_coeffs_init_zeros = torch.zeros(6, 1, dtype=torch.float64)  # начальное приближение в виде вектора нулей
penalty_torch_coeffs_init_ones = torch.tensor(np.ones((6, 1)), requires_grad=True, dtype=torch.float64)  # начальное приближение в виде вектора единиц
```

```
penalty_torch_coeffs_init_lsm = torch.tensor(lsm, requires_grad=True, dtype=torch.float64)  # начальное приближение на основе МНК
penalty_torch_coeffs_init_interp = torch.tensor(np.linalg.solve(A_interp, z_interp), requires_grad=True, dtype=torch.float64)  # начальное приближение на основе интерполяции

penalty_torch_coeffs_init = penalty_torch_coeffs_init_interp  # выбор начального приближения для torch
penalty_torch_coeffs_init.requires_grad_(True)

# функция потерь для градиентного спуска из torch
def penalty_loss_function(coeffs):
    t = A_torch @ coeffs - z_torch
    return torch.sum(torch.sqrt(torch.abs(t + 1e-6)))

h = 1e-5  # шаг градиентного спуска

# расчет коэффициентов полинома с помощью torch
penalty_torch_coeffs_list = []
for _ in range(N):
    penalty_torch_coeffs = gradient_descent_torch(penalty_loss_function, penalty_torch_coeffs_init, h, 40000, 1e-8)
    penalty_torch_coeffs = penalty_torch_coeffs.detach().numpy()
    penalty_torch_coeffs_list.append(penalty_torch_coeffs)
penalty_torch_coeffs_list = np.array(penalty_torch_coeffs_list)

ax1.plot(x, poly(lsm_list[0], x), color='magenta', label='LSM')
ax1.plot(x, poly(cheb_list[0], x), color='black', label='Cheb. approx.')
ax1.plot(x, poly(first_norm_list[0], x), color='orange', label='first norm')
ax1.plot(x, poly(penalty_scipy_coeffs_list[0], x), color='red', label='penalty_scipy')
ax1.plot(x, poly(penalty_torch_coeffs_list[0], x), color='cyan', label='penalty_torch')

ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.legend(loc='upper left')
ax1.grid(True)

# расчет среднеквадратичного отклонения для каждого метода
def calculate_mse(coeffs, ref):
    return np.mean((coeffs - ref) ** 2, axis = 0)
```

```
if calc_mse:
    mse_lsm = calculate_mse(lsm_list, coeffs)
    mse_cheb = calculate_mse(cheb_list, coeffs)
    mse_first_norm = calculate_mse(first_norm_list, coeffs)
    mse_penalty_scipy = calculate_mse(penalty_scipy_coeffs_list, coeffs)
    mse_penalty_torch = calculate_mse(penalty_torch_coeffs_list, coeffs)

    # вывод максимального среди шести коэффициентов среднеквадратичного отклонения
    print('Task 3')
    print('LSM MSE: ', np.max(mse_lsm))
    print('Cheb. MSE: ', np.max(mse_cheb))
    print('first norm MSE: ', np.max(mse_first_norm))
    print('penalty scipy MSE: ', np.max(mse_penalty_scipy))
    print('penalty torch MSE: ', np.max(mse_penalty_torch))

print('Task 3: ok')

# Задача 4. Модификация шума
'''
z = y + w + v

вер. того, что v == 0: 0.9
вер. того, что 10 <= |v| <= 20: 0.1

необх. создать вектор случ. величин [0, 1] размером, равным x,
чтобы заполнить вектор шума нулями там, где случ. величина больше 0.9.
Для остальных ячеек необх. сгенерировать два вектора: первый - вектор знаков,
второй - вектор амплитуд из равномерного распр.
'''

p_zero = 0.9  # вероятность того, что шум равен 0
p_nonzero = 0.1  # вероятность того, что шум - случ. вел. из равн. распр.

random_values = np.random.rand(z.shape[0])  # массив вероятностей
zero_indices = random_values > p_nonzero  # массив, в котором лог. единица там,
где шум будет 0
nonzero_indices = np.invert(zero_indices)  # массив, в котором лог. единица там,
где шум будет из равном. распр.
num_nonzero = z.shape[0] - np.sum(zero_indices[zero_indices==True])  # количество
ненулевых элементов шума

v = np.zeros(z.shape)  # шаблон вектора шума
```

```
signs = np.random.choice([-1, 1], size=num_nonzero)  # вектор знаков для шума из равном. распр.
amplitudes = np.random.uniform(10, 20, size=num_nonzero)  # вектор амплитуд для шума из равном. распр.
v[nonzero_indices] = np.reshape(signs * amplitudes, (v[nonzero_indices].shape))  # заполнение вектора шума значениями из равном. распр.

z += v  # добавление к старому шуму нового

fig2, ax2 = plt.subplots(figsize=(8, 6))

ax2.plot(x, y, color='blue', label='poly')  # вывод значений полинома
# вывод точек, участвующих в ограничениях
ax2.scatter([0, 3, 7, 10], [poly(coeffs, 0), poly(coeffs, 3), poly(coeffs, 7), poly(coeffs, 10)], color='red')
# вывод зашумленных измерений
ax2.plot(x, z, color='green', label='poly with noise')

print('Task 4: ok')

# Задача 5. Поиск коэффициентов полинома с новым шумом
# Полное повторение задачи 3, но с модифицированным вектором z

calc_mse = False  # если True, расчет коэффициентов каждым методом производится 10 раз, иначе - 1 раз
N = 1 if not calc_mse else 10

lsm = np.linalg.inv(A.T @ A) @ A.T @ z  # Поиск коэффициентов с помощью аналитического решения МНК
lsm_list = np.array([lsm for _ in range(N)])

# Аппроксимация Чебышева: min ||Ax - b||_inf -> min t при -t <= Ax - b <= t, t - скаляр
cheb_list = []
for _ in range(N):
    t = cp.Variable()
    cheb = cp.Variable(6)
    cheb_problem = cp.Problem(cp.Minimize(t), [cp.abs(A @ cheb - z) <= t])
    cheb_problem.solve(verbose=False)
    cheb = np.reshape(cheb.value, (6, 1))
    cheb_list.append(cheb)
cheb_list = np.array(cheb_list)
```

```
# Минимизация первой нормы: min ||Ax - b||_1 -> -1^T * t <= Ax - b <= 1^T * t, t
- вектор
first_norm_list = []
for _ in range(N):
    t = cp.Variable(A.shape[0])
    first_norm = cp.Variable(6)
    first_norm_problem = cp.Problem(cp.Minimize(cp.sum(t)), [A @ first_norm - z
<= t, z - A @ first_norm <= t])
    first_norm_problem.solve(verbose=False)
    first_norm = np.reshape(first_norm.value, (6, 1))
    first_norm_list.append(first_norm)
first_norm_list = np.array(first_norm_list)

# Минимизация суммы значений штрафной функции sqrt(abs(t)), t = Ax - b

# целевая функция для солвера из scipy
# min sum(sqrt(abs(t))), t = Ax - b
def penalty_objective(coeffs):
    t = A @ coeffs - z
    # return np.sqrt(np.sum(np.abs(t)))
    return np.sum(np.sqrt(np.abs(t)))

A_interp = np.vstack([A[0], A[20], A[40], A[60], A[80], A[100]])  # матрица A для
поиска начального приближения в виде интерполяции
z_interp = np.vstack([z[0], z[20], z[40], z[60], z[80], z[100]])  # вектор z для
поиска начального приближения в виде интерполяции

penalty_scipy_coeffs_init_zeros = np.zeros(6)   # начальное приближение в виде
вектора нулей
penalty_scipy_coeffs_init_ones = np.ones(6)   # начальное приближение в виде
вектора единиц
penalty_scipy_coeffs_init_lsm = lsm.reshape(lsm.shape[0])   # начальное
приближение на основе МНК
penalty_scipy_coeffs_init_interp = np.linalg.solve(A_interp, z_interp).reshape(6)
# начальное приближение на основе интерполяции

penalty_scipy_coeffs_init = penalty_scipy_coeffs_init_interp  # выбор начального
приближения для scipy

penalty_scipy_coeffs_list = []
for _ in range(N):
    penalty_scipy_result = minimize(penalty_objective, penalty_scipy_coeffs_init,
method='L-BFGS-B')
```

```
    penalty_scipy_coeffs = np.reshape(penalty_scipy_result.x, (6, 1))
    penalty_scipy_coeffs_list.append(penalty_scipy_coeffs)
penalty_scipy_coeffs_list = np.array(penalty_scipy_coeffs_list)

A_torch = torch.tensor(A, dtype=torch.float64)
z_torch = torch.tensor(z, dtype=torch.float64)

penalty_torch_coeffs_init_zeros = torch.zeros(6, 1, dtype=torch.float64) # начальное приближение в виде вектора нулей
penalty_torch_coeffs_init_ones = torch.tensor(np.ones((6, 1)), requires_grad=True, dtype=torch.float64)  # начальное приближение в виде вектора единиц
penalty_torch_coeffs_init_lsm = torch.tensor(lsm, requires_grad=True, dtype=torch.float64)  # начальное приближение на основе МНК
penalty_torch_coeffs_init_interp = torch.tensor(np.linalg.solve(A_interp, z_interp), requires_grad=True, dtype=torch.float64)  # начальное приближение на основе интерполяции

penalty_torch_coeffs_init = penalty_torch_coeffs_init_interp  # выбор начального приближения для torch
penalty_torch_coeffs_init.requires_grad_(True)

# функция потерь для градиентного спуска из torch
def penalty_loss_function(coeffs):
    t = A_torch @ coeffs - z_torch
    return torch.sum(torch.sqrt(torch.abs(t + 1e-6)))

h = 1e-5  # шаг градиентного спуска

# расчет коэффициентов полинома с помощью torch
penalty_torch_coeffs_list = []
for _ in range(N):
    penalty_torch_coeffs = gradient_descent_torch(penalty_loss_function, penalty_torch_coeffs_init, h, 40000, 1e-8)
    penalty_torch_coeffs = penalty_torch_coeffs.detach().numpy()
    penalty_torch_coeffs_list.append(penalty_torch_coeffs)
penalty_torch_coeffs_list = np.array(penalty_torch_coeffs_list)

ax2.plot(x, poly(lsm_list[0], x), color='magenta', label='LSM')
ax2.plot(x, poly(cheb_list[0], x), color='black', label='Cheb. approx.')
ax2.plot(x, poly(first_norm_list[0], x), color='orange', label='first norm')
ax2.plot(x, poly(penalty_scipy_coeffs_list[0], x), color='red', label='penalty_scipy')
```

```
ax2.plot(x,      poly(penalty_torch_coeffs_list[0],      x),      color='cyan',
label='penalty_torch')

ax2.set_xlabel('x')
ax2.set_ylabel('y')
ax2.legend(loc='upper left')
ax2.grid(True)

# расчет среднеквадратичного отклонения для каждого метода
def calculate_mse(coeffs, ref):
    return np.mean((coeffs - ref) ** 2, axis = 0)

if calc_mse:
    mse_lsm = calculate_mse(lsm_list, coeffs)
    mse_cheb = calculate_mse(cheb_list, coeffs)
    mse_first_norm = calculate_mse(first_norm_list, coeffs)
    mse_penalty_scipy = calculate_mse(penalty_scipy_coeffs_list, coeffs)
    mse_penalty_torch = calculate_mse(penalty_torch_coeffs_list, coeffs)

    # вывод максимального среди шести коэффициентов среднеквадратичного отклонения
    print('Task 5')
    print('LSM MSE: ', np.max(mse_lsm))
    print('Cheb. MSE: ', np.max(mse_cheb))
    print('first norm MSE: ', np.max(mse_first_norm))
    print('penalty scipy MSE: ', np.max(mse_penalty_scipy))
    print('penalty torch MSE: ', np.max(mse_penalty_torch))

print('Task 5: ok')

plt.tight_layout()
plt.show()
```