Команда ГАДы:
Крылов Андрей m3234
Фунин Георгий m3234
Дударев Денис m3235

Отчет
Лабораторная работа №2
Методы нулевого и первого порядка

Введение

В этой лабораторной работе мы реализовали и исследовали эффективность продвинутых численных методов поиска минимума многомерных функций, а также сравнили их с методами из первой лабораторной работы. В лабораторной работе были использованы Scala 3 и библиотека breeze и Python 3 и библиотека scipy-optimyze.

1 Описание методов

Метод Ньютона

Реализация метода Ньютона

- Использует хвостовую рекурсию для оптимизации памяти
- Гессиан ищется с помощью библиотеки breeze
- Гиперпараметры в виде eps, h, maxiter

Листинг 1: Реализация метода Ньютона

```
def newtonMethod(
        x_0: Point,
        h: Scheduling = hConst
    ): (Point, Int) = {
      @tailrec
      def recursion(x_k: Point, k: Int): (Point, Int) = {
        val gradNorm = findGradient(x_k).N
        val next     = newtonStep(x_k, k, h, findHessian(x_k))
        if (gradNorm < eps * 1e-3 || k > max_iter || (next - x_k).N < eps * 1e-3)
          (x_k, k)
        else {
          recursion(next, k + 1)
        }
      }
      recursion(x_0, 0)
    }
```

BFGS

Реализация BFGS

- Использует хвостовую рекурсию для оптимизации памяти
- Гессиан ищется с помощью библиотеки breeze
- Гиперпараметры в виде eps, a-scale, h, (c1, c2) - Wolfe, maxiter

Листинг 2: Реализация метода Ньютона

```
def BFGS(x_0: Point, h: Scheduling = wolfeRule(f)): (Point, Int) = {
      val I: DenseMatrix[Double] = DenseMatrix.eye[Double](2)

      @tailrec
      def recursion(
          k: Int,
```

```
        x_k: Point,
        B_k_inverse: DenseMatrix[Double]
    ): (Point, Int) = {
      val grad_k   = findGradient(x_k)
      val gradNorm = grad_k.N

      if (gradNorm < eps * 1e-3 || k > max_iter) (x_k, k)
      else {
        val p_k       = B_k_inverse * DenseVector(grad_k.coords)
        val direction = Point(-p_k(0), -p_k(1))

        val alpha_k = h.func(k, x_k) * 0.7

        val x_k1 = x_k + direction * alpha_k

        val grad_k1 = findGradient(x_k1)
        val y_k     = grad_k1 - grad_k
        val s_k     = x_k1 - x_k

        val sTy   = s_k.coords.zip(y_k.coords).map { case (s, y) => s * y }.sum
        val rho_k = 1.0 / sTy

        val s_k_vec       = DenseVector(s_k.coords)
        val y_k_vec       = DenseVector(y_k.coords)
        val term1         = I - (s_k_vec * y_k_vec.t) * rho_k
        val term2         = I - (y_k_vec * s_k_vec.t) * rho_k
        val term3         = s_k_vec * s_k_vec.t * rho_k
        val B_k1_inverse = term1 * B_k_inverse * term2 + term3
        recursion(k + 1, x_k1, B_k1_inverse)
      }
    }

    recursion(0, x_0, I)
  }
```

2 Графики

Реализуем отображение графиков на Python, который:

- отображает визуализацию 3D
- отрисовывает траекторию градиентного спуска
- отображает линии уровня функции

Используемые библиотеки

- `numpy` — работа с массивами данных
- `matplotlib.pyplot` — создание 3D-графиков
- `matplotlib.colors.LightSource` — создание освещения для 3D-графиков

3 Описание результатов

3.1 метод Ньютона

Рассмотрим работу методов на примере обычной функции:

$$z = x^2 + y^2 \text{ в точке } (2,\ 2)$$

Метод	Параметры	Итерации	в.Функции	в.Градиента	x	y
Constant $\frac{1}{c}$	$c = 100$	1	21	1	0	0
Dec. seq. $\frac{1}{k+c}$	$c = 1$	1	22	2	0	0

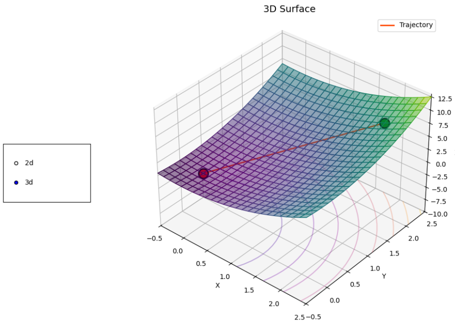

Рис. 1: 3D Constant

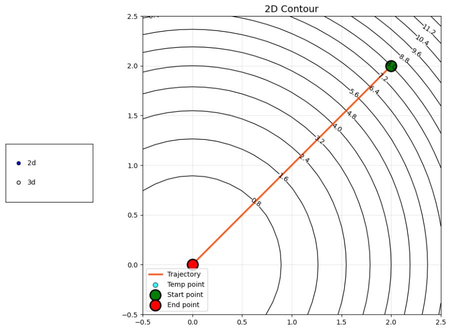

Рис. 2: 2D Constant

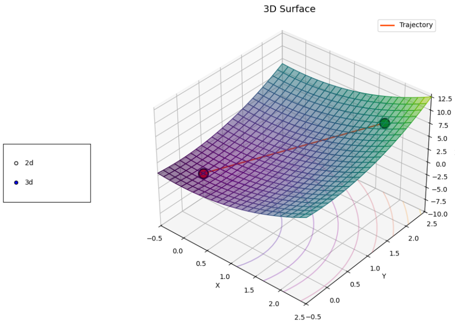

Рис. 3: 3D Sequence

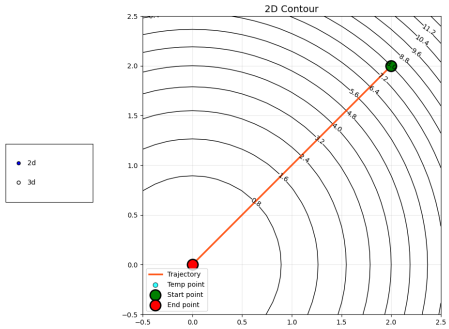

Рис. 4: 2D Sequence

Возьмемем фукнцию поинтереснее выберем мультимодальную функцию Химмельблау:

$$z = (x^2 + y - 11)^2 + (x + y^2 - 7)^2 \text{ в точке (-4, -5)}$$

Метод	**Параметры**	**Итерации**	**Вызовы функции**	**Вызовы градиента**	**x**	**y**
Constant $\frac{1}{c}$	$c = 100$	5	266	66	-3.77931	-3.28319
Dec. seq. $\frac{1}{k+c}$	$c = 1$	3058	33649	3058	-3.77938	-3.28348
Armijo rule	$c = 0.5$	1791	115476	21504	-3.77931	-3.28320
Wolfe rule	$c_1 = 0.001,\ c_2 = 0.9$	838	75510	15941	-3.77931	-3.28319

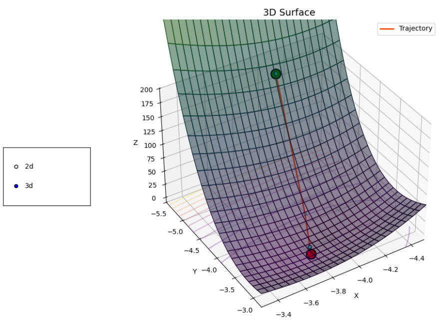

Рис. 5: 3D Constant

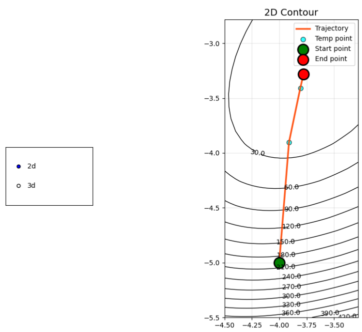

Рис. 6: 2D Constant

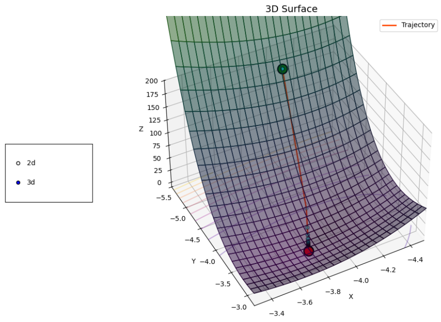

Рис. 7: 3D Sequence

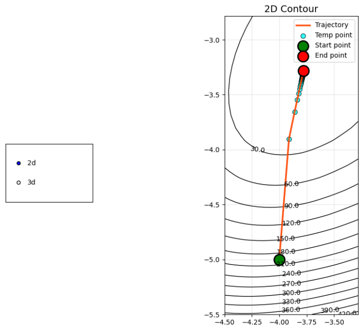

Рис. 8: 2D Sequence

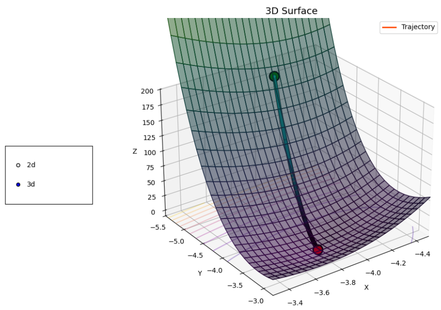

Рис. 9: 3D Armijo

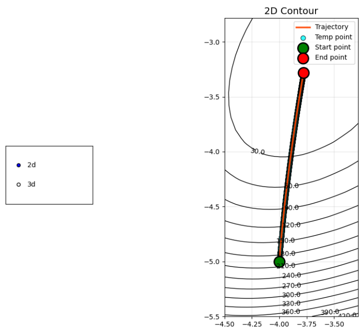

Рис. 10: 2D Armijo

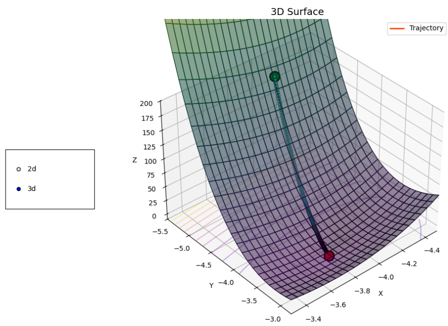

Рис. 11: 3D Wolfe

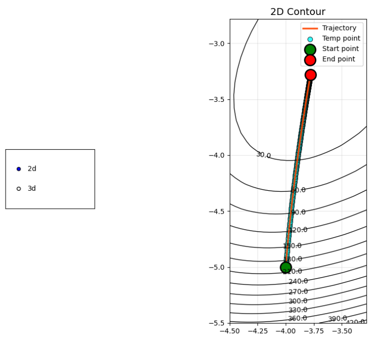

Рис. 12: 2D Wolfe

По полученной таблице видно, что лучше всего методе Ньютона в себя показал Constant, хуже всего это сделал Armijo. Все методы нашли нужные точки, но разницы в требуемых ресурсах очень велики.

3.2 BFGS

На примере функции Бута

$$z = (x + 2y - 7)^2 + (2x + y - 5)^2 \text{ в точке (-1, -3)}$$

Метод	**Параметры**	**Итерации**	**Вызовы функции**	**Вызовы градиента**	**x**	**y**
Constant $\frac{1}{c}$	$c = 100$	5	44	11	$1,00000$	$3,00000$
Dec. seq. $\frac{1}{k+c}$	$c = 1$	100001	800012	200003	$1,00065$	$3,00070$
Armijo rule	$c = 0.5$	629	15100	1888	$1,00000$	$3,00000$
Wolfe rule	$c_1 = 0.001,\ c_2 = 0.9$	291	12232	2330	$1,00000$	$3,00000$

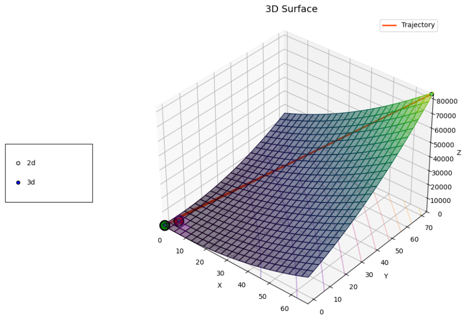

Рис. 13: 3D Constant

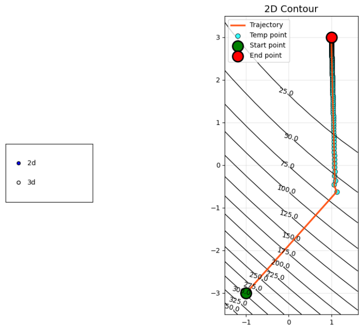

Рис. 14: 2D Constant

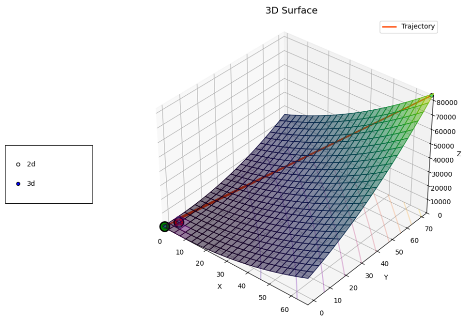

Рис. 15: 3D Sequence

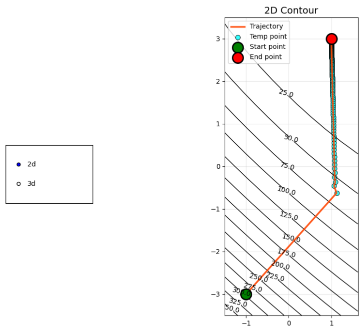

Рис. 16: 2D Sequence

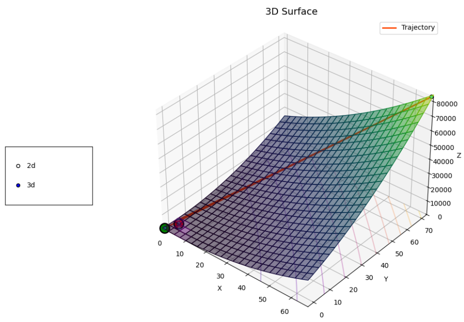

Рис. 17: 3D Armijo

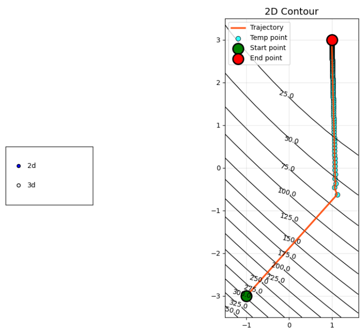

Рис. 18: 2D Armijo

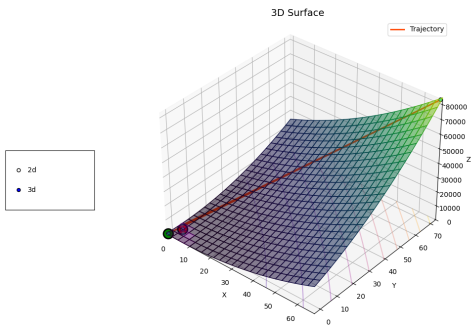

Рис. 19: 3D Wolfe

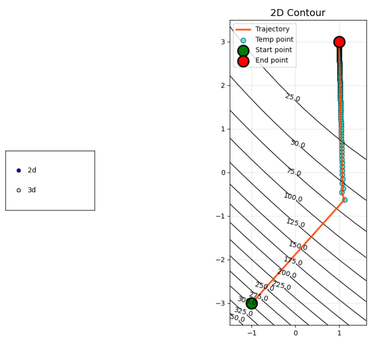

Рис. 20: 2D Wolfe

По полученной таблице видно, что лучше всего в методе BFGS себя показал Constant, хуже всего это сделал Dec. Sequence. Все методы нашли нужные точки, но разницы в требуемых ресурсах очень велики.

3.3 Scipy-optimize

Функция Химмельблау:

$$z = (x^2 + y - 11)^2 + (x + y^2 - 7)^2 \text{ в точке (10, -5)}$$

Метод	**Итерации**	**Вызовы функции**	**Вызовы градиента**	**x**	**y**
Constant $\frac{1}{c}$	77	620	155	3,00000	2,00000
Dec. seq. $\frac{1}{k+c}$	5	44	11	1,00013	3,00021
Armijo rule	100001	658994	200324	3,00000	2,00000
Wolfe rule	100001	4365798	304404	3,00000	2,00000
L-BFGS-B	14	15	15	3,00000	2,00000
BFGS	22	25	25	2.99999	2,00000
Newton-CG	12	12	12	2.99999	2,00000

Как видим по результатам вычислений, наши и библиотечные методы находят одинковые результаты, но совершают достаточно много избыточных вычислений.

3.4 Оптимизация Гиперпараметров

В свзяи с тем, что мы не имеем доступа к Optuna будем использовать `smile-scala 4.3.0` — модуль `smile.hpo`

3.4.1 Рассмотрим алгоритм на примере Армихо

1. Для каждой тестовой функции запустили градиентный спуск, где шаг выбирается по правилу Армихо.
2. В качестве базовой конфигурации использовали значение $c = 0.5$.
3. С помощью `Hyperparameters.random()` провели $N = 40$ случайных испытаний в диапазоне $c \in [0.1, 0.9]$; метрика — число итераций до достижения $\|\nabla f\| < 10^{-11}$.
4. Зафиксировали пару «лучший c / число итераций» результата.

Полученный результат c = 0.4125358104705811

4 Вывод

В процессе работы мы исследовали метод Ньютона и квазиньютоновские методы, а также сравнили собственные реализации с библиотечными. На плохо обусловленных и сложных функциях метод Ньютона показывает более медленную сходимость и требует больше вычислений, однако всё же способен найти минимум. Библиотечные реализации методов Ньютона, BFGS и стратегий линейного поиска в среднем демонстрируют лучшую эффективность по числу итераций и вычислений. Тем не менее, наши алгоритмы также достигают корректного результата.