Национальный исследовательский университет ИТМО

Факультет информационных технологий и программирования

Прикладная математика и информатика

Отчет по лабораторной работе 4

по дисциплине «**Методы оптимизации в машинном обучении**»

Авторы: Багаутдинов Искандер Ильгизович M3237

Пан Артём Олегович M3237

Илляхунов Ансар Адылович M3237

Булавко Тимофей Евгеньевич M3237

Преподаватель: Андреев Юрий Александрович

Санкт-Петербург 2024

Цель работы:

Исследовать эффективность методов стохастической оптимизации на примере генетического алгоритма, а также фреймворков, применяющих стохастичекие оптимизации для нахождения гиперпараметров моделей и методов.

Задачи работы:

- Разобрать теоретическое описание генетического алгоритма
- Реализовать генетический алгоритм
- Сравнить эффективность генетического алгоритма на методах и примерах из лаб. 1 и 2
- Выбрать функции, на которых эффективность методов будет явно различаться
- Исследовать работу методов в зависимости от способа нахождения производной
- Проиллюстрировать примеры

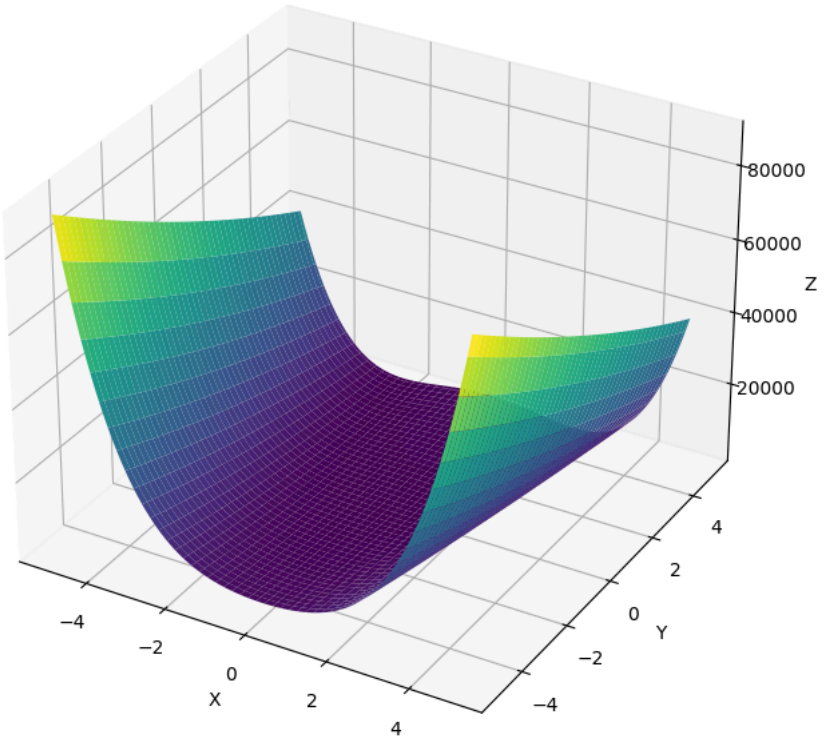
Подготовка. Выбор функций с ограниченной областью определения

Для проведения исследования эффективности нами были выбраны 3 стандартные тестовые функции в оптимизации:

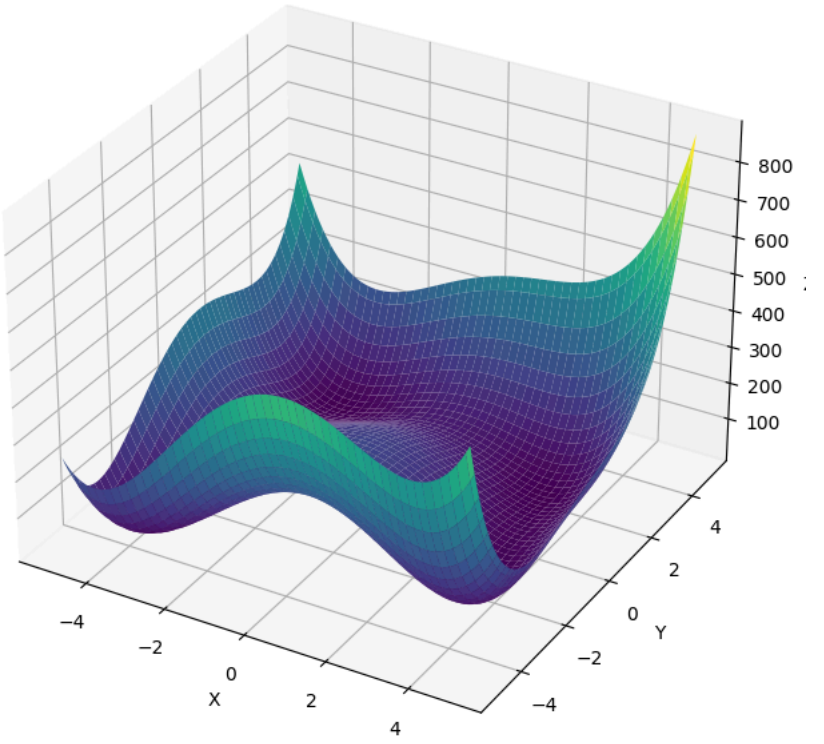
1. Функция Розенброка - функция вида (1-x)**2 + 100*(y-x**2)**2. Имеет единственный глобальный минимум в (1, 1). Рассматриваем куб [-5, 10]^2.
2. Функция Химмельблау - (x**2 + y – 11) **2 + (x + y**2 – 7)**2. Она имеет 4 локальных минимума с одинаковым значением - 0 и один глобальный максимум - (-0.270845…, -0.923039…). Рассматриваем куб [-5, 5]^2.
3. Функция Растригина для функции двух переменных: 20 + (x**2 - 10(cos(2pi*x)) + (y**2 - 10(cos(2pi*y)). Она имеет крайне нетривиальную поверхность и большое количество локальных минимумов и максимумов. Рассматриваем куб [-5, 5]^2.

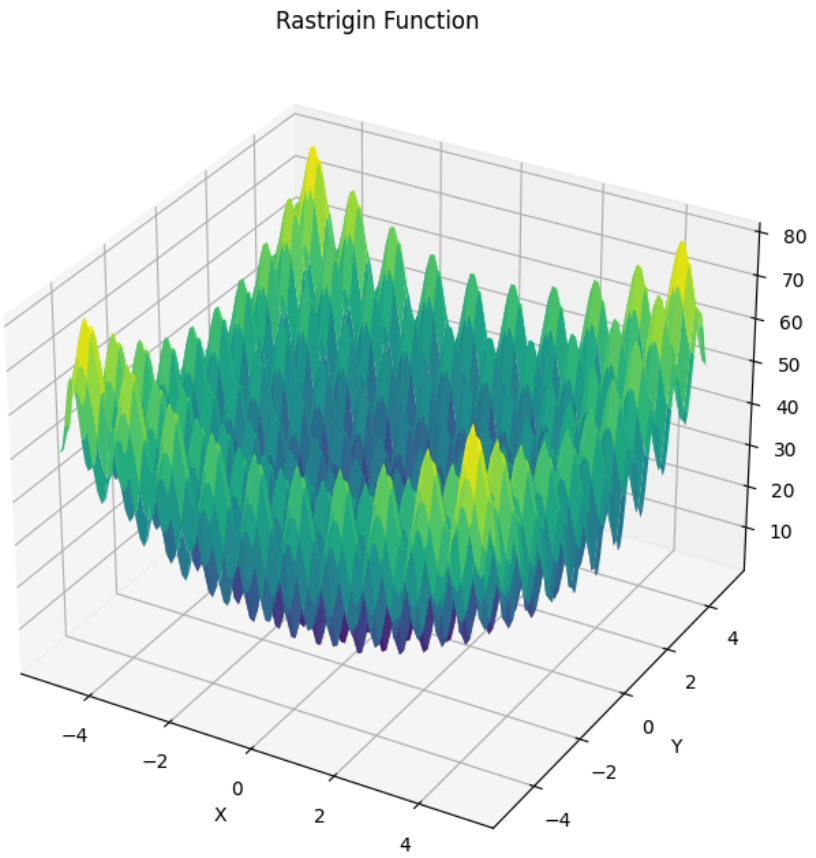
Визуализация функций:

Rosenbrock Function



Himmelblau Function





Основное задание. Пункт 1: Разбор генетического алгоритма

Генетический алгоритм - принадлежит классу эволюционных алгоритмов, симулирующих в своей работе процессы естественного отбора. С его помощью можно решать не только проблему оптимизации, но и некоторые другие, состояние в которых может быть представлено вектором (об этом ниже), в том числе проблемы на графах (salesman) и knapsack problem.

Стохастическая оптимизация - класс алгоритмов оптимизации, использующих случайность в процессе поиска оптимума.

Генетический алгоритм решает задачу поиска максимума, то есть оперирует с fitness function, что сводится к задаче поиска минимума умножением значения функции на -1.

Алгоритм оперирует с вектором, также называемым хромосомой, ограниченных значений, представляющим некоторое решение/состояние. Мы будем рассматривать наиболее классическую версию, где значениями выступают 0 и 1, то есть вектор является битовой строкой, однако есть и другие способы кодирования, показывающие себя более эффективно в определенных случаях в проблеме оптимизации, в частности кодирование floating point, но это выходит за рамки нашей работы.

Рассмотрим классический способ кодирования точки бинарной строкой, когда область определения целевой функции представляет собой гиперкуб ($a_i <= x_i <= b_i$). Разобьем каждый из отрезков значений на 2^n подотрезков вида

$[a_i + k\frac{b_i - a_i}{2^n}, a_i + (k+1)\frac{b_i - a_i}{2^n}]$ для $0 <= k < 2^n$. Будем кодировать k просто бинарным представлением для каждого параметра, а после сконкатенируем их. Для упрощения, будем считать, что k задает не интервал, а просто левую границу.

В начале популяция, множество хромосом, будет сформирована из случайных хромосом, а далее на каждой итерации над популяцией будут осуществляться селекция, скрещивание и мутация.

Селекция представляет собой операцию отбора хромосом из предыдущей итерации для последующего скрещивания. Существуют разные стратегии проведения итерации, но во всех верно, что хромосомы с лучшим значением функции имеют больший шанс быть выбранными. Заметим, что одна и та же хромосома может быть выбрана для скрещивания несколько раз, так как на каждой итерации размер популяции остается прежним (хотя и на это есть модификации).

На этапе скрещивание отобранные пары родителей производят пару детей. Вырожденный случай скрещивания - с некоторой вероятностью изменений не происходит и хромосомы-родители и представляют собой пару детей. Классический способ скрещивания, когда оно происходит, - случайным образом выбирается место точка между двумя битами, разграничивающая строки родителей:

$$\boldsymbol{v_2} = (00000|0111000000010000),$$
$$\boldsymbol{v_3} = (11100|0000111111000101).$$

и один ребенок получает левую часть от первого родителя и правую от второго, а другой - наоборот:

$$\boldsymbol{v_2}' = (00000|0000111111000101),$$
$$\boldsymbol{v_3}' = (11100|0111000000010000).$$

В случае с нашим кодированием скрещивание, если точка разграничения попадет ровно между коэффициентами разных параметров, у детей будут перемешаны первые координаты и последний, если же точка попадет на представление какого-то коэффициента, то то же самое, только коэффициенты одного из параметров также будут перемешан на старшие и младшие биты.

На этапе мутации некоторые значения векторов будут изменены. В случае с бинарным кодированием - это замена 0 на 1 и наоборот. Это позволяет случайным образом попробовать попасть в “неизученные” текущей популяции места.

Основное задание. Пункт 1: Реализация генетического алгоритма.

Метод кодирования точек уже был описан. В качестве бинарной строки используются int'ы до 2^64 (в надежде, что под капотом будет оптимизировано под int64), то есть 32 бита на каждый коэффициент.

Взятие отрицательного значения функции для сведения к минимизации осуществляется в конструкторе класса chromosome.

В качестве алгоритма селекции используется турнир: для выбора каждого будущего родителя (при том выбор пар не связен) выбирается случайное подмножество из k (гиперпараметр) особей текущей популяции и среди них берется с наибольшим значением функции.

В мутации каждый бит может быть заменен с вероятностью 2^(-degree), где degree - еще один гиперпараметр. В тестах будем использовать degree=4, то есть каждый бит может быть заменен с вероятность 1/16 и математическое ожидание числа мутаций в одной хромосоме - 4.

Подбор критерия остановки - нетривиальная задача для генетического алгоритма, так как из того, что лучший результат не изменился между итерациями не следует, что алгоритм "угасает": продвижения популяции может дать свои плоды далее. Поэтому помимо tol используется параметр tries (=20 на тестах): если лучший результат не изменился на tol tries итераций подряд, то только тогда поиск завершается.

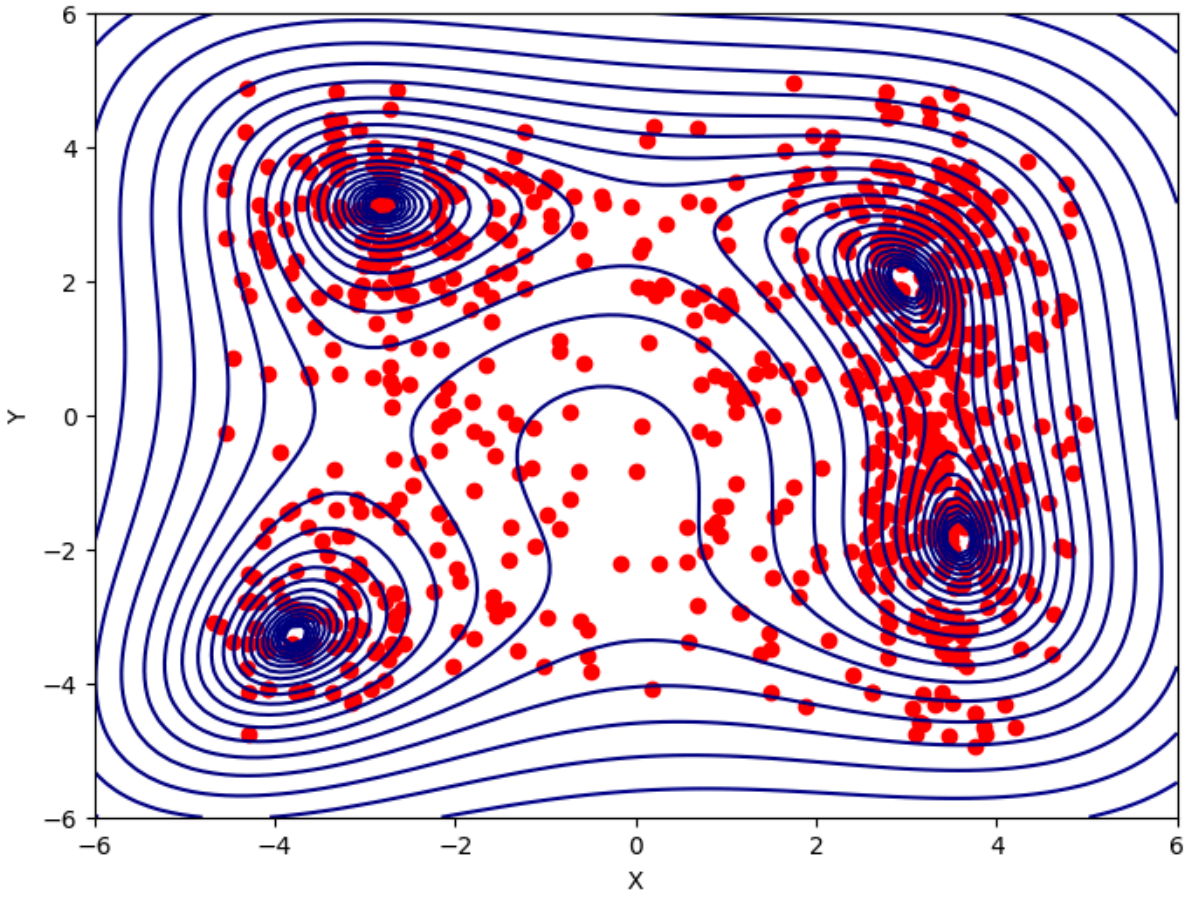
Основное задание. Пункт 2: сравнение с методом Ньютона и квазиньютоновскими методами

Гиперпараметры генетического алгоритма на всех запусках выставлены так: размер популяции = 1000, k = 30, degree = 4, вероятность скрещивания = 0.5

Для сравнения с другими методами будем использовать одинаковое значение tol=1e-9.

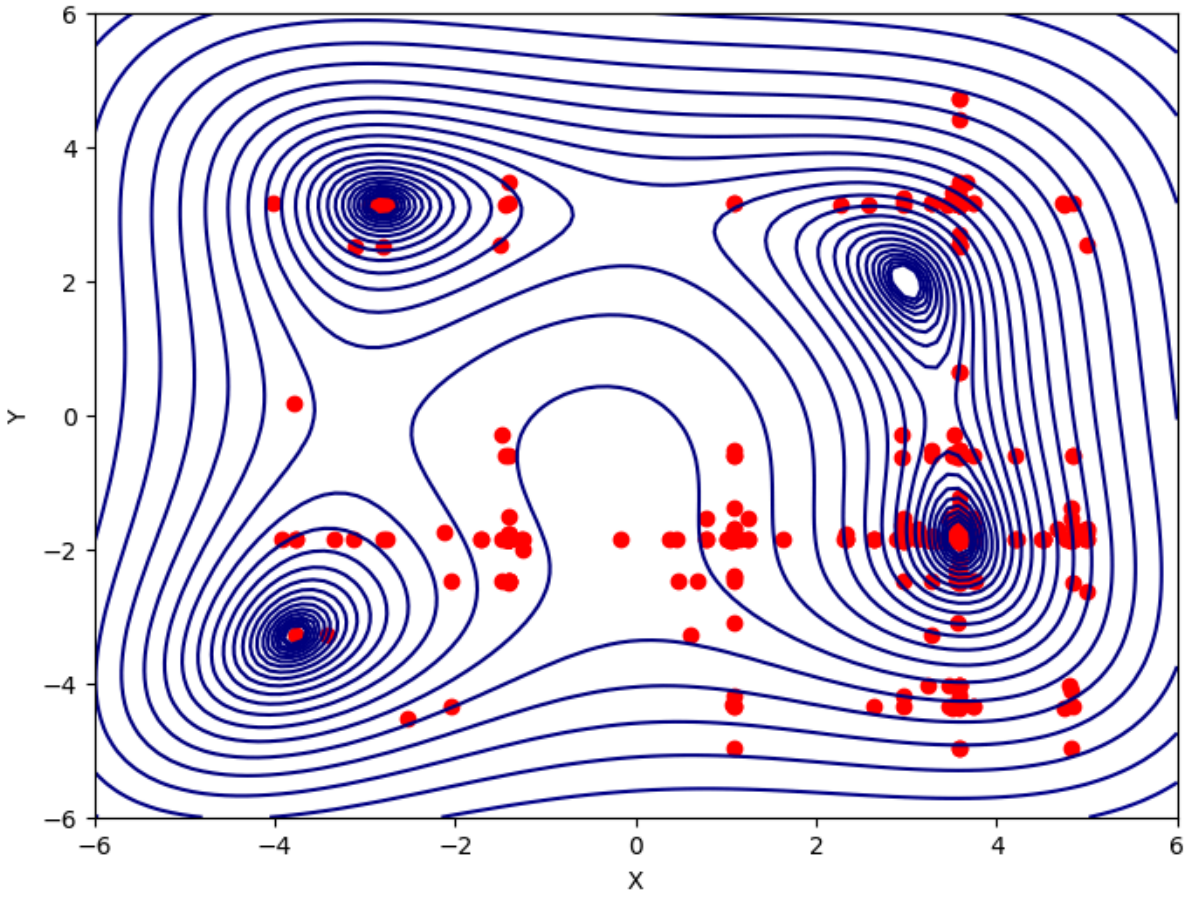
Функция Химмельблау

Спустя 10 итераций:



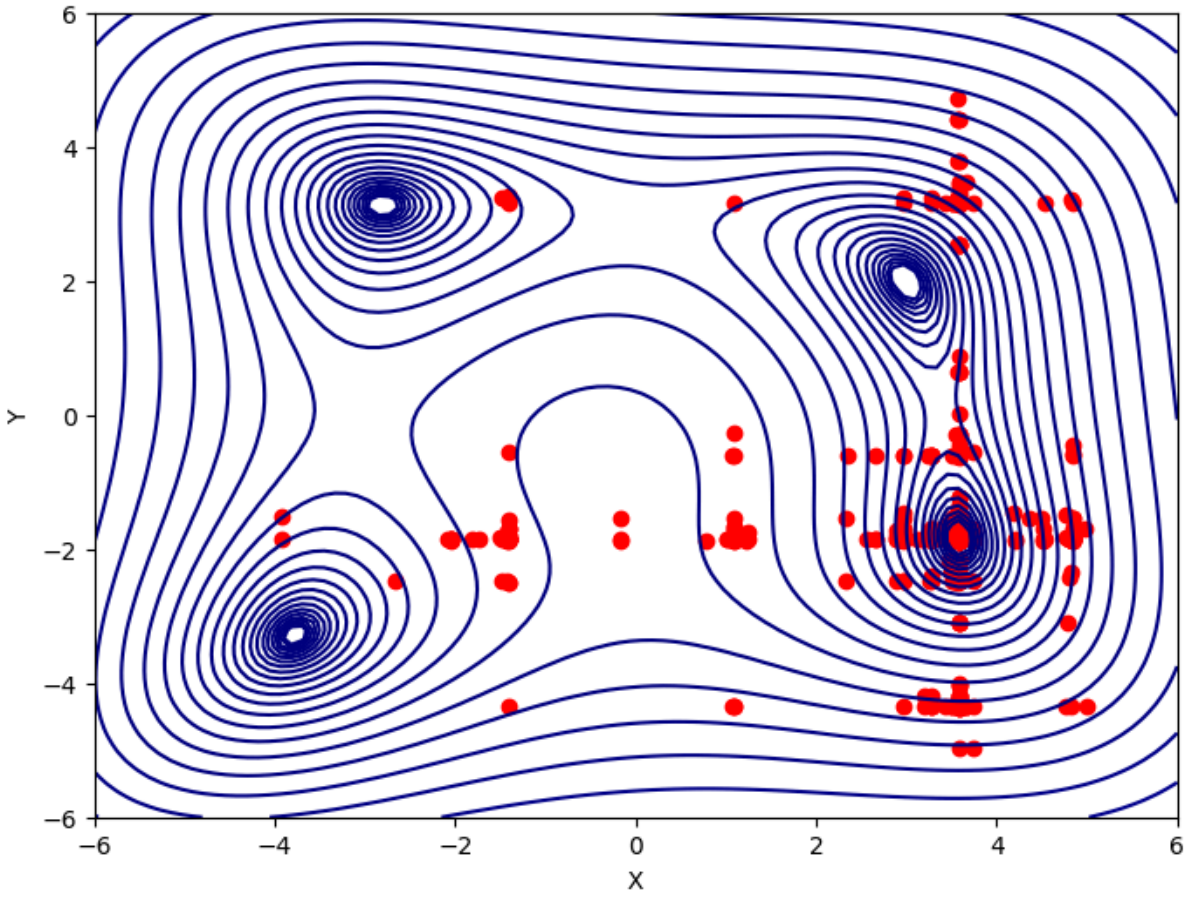
Уже можем наблюдать, как хромосомы концентрируются около оптимумов.

Спустя 20 итераций:



Можем наблюдать, что точки, по сути, разделились на те, что непосредственно у оптимума и те, что разбросаны достаточно произвольно, часто формируя “линии”, проходящие через оптимум. Это может быть объяснено работой мутации на нашей кодировке: вполне вероятно, что мутации придутся на один из коэффициентов, двигая точка только по одной оси.

На финальной, 72-ой итерации, популяция выглядит следующим образом:



Видим, что популяции в других оптимумах не сохранились.

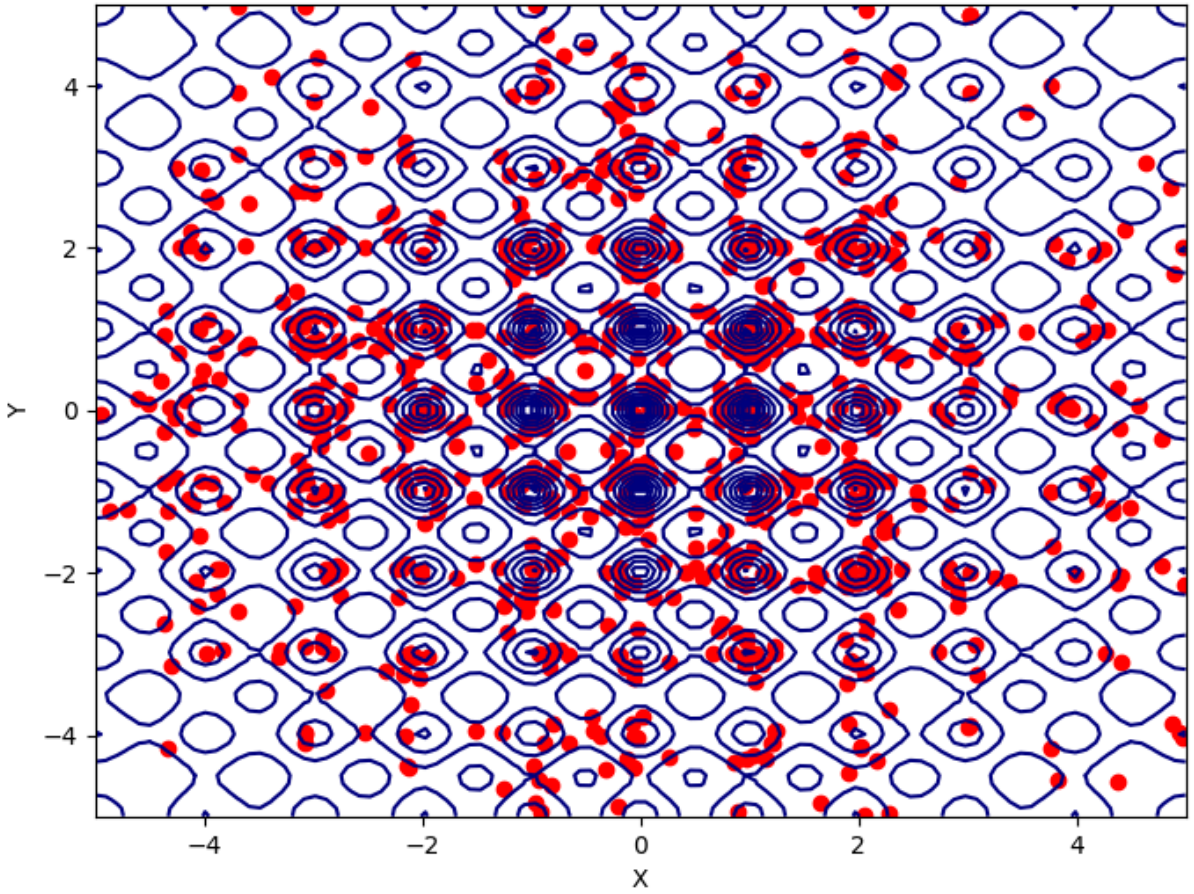
Метод	Предполагаемая x* (округлено)	f(x*) (округлено)	Количество итераций	Количество вычислений f	Количество вычислений градиента	Количество вычислений гессиана	Время, сек.
свой Ньютон с фиксированным шагом	(-0.27084, -0.92304)	181.61652	7	0	7	7	<1
свой Ньютон с дихотомией	(3, 2)	0	10	770	10	10	<1
Newton-CG	(3, 2)	0	9	13	13	9	<1

Генеративный алгоритм	(3.584, -1.848)	0	72	73000	0	0	5

По времени и количеству вычислений функции генеративный алгоритм заметно проигрывает.

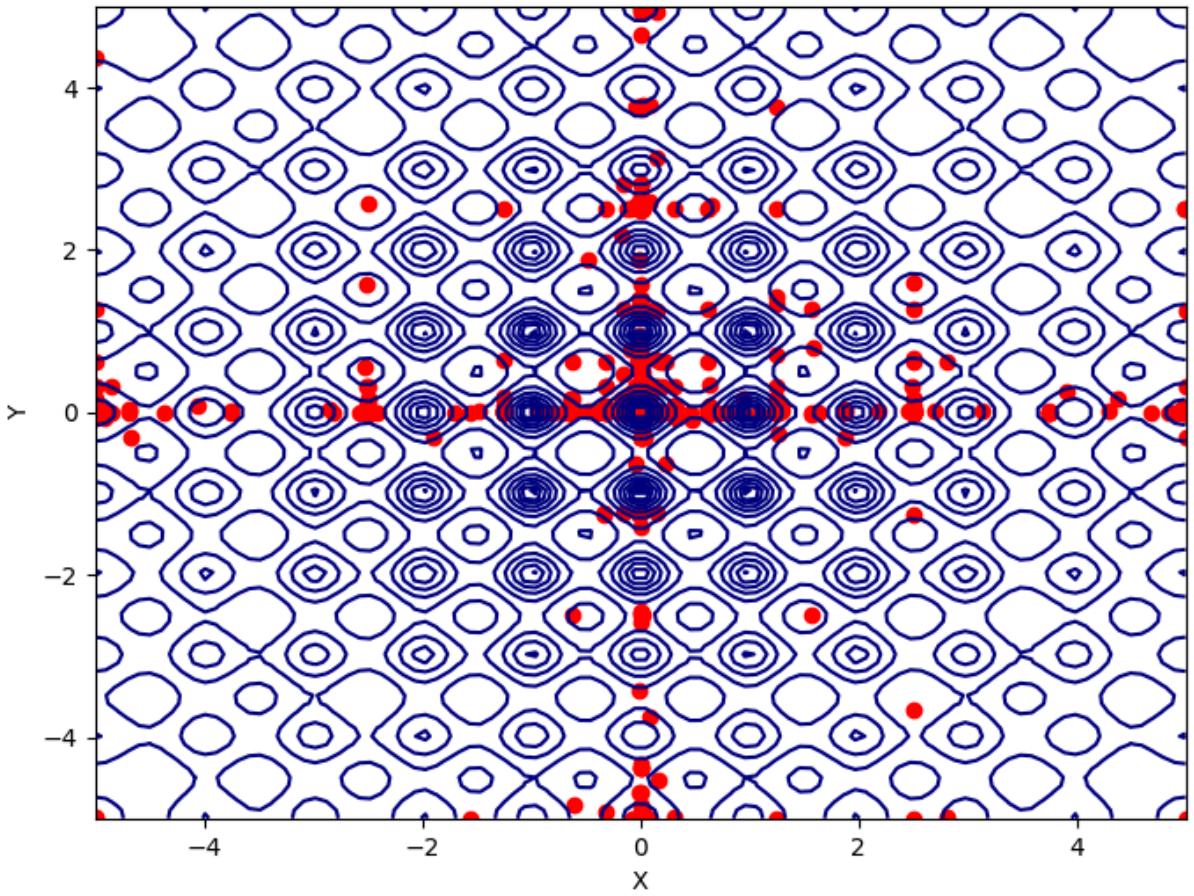
Функция Растригина

Спустя 10 итераций:



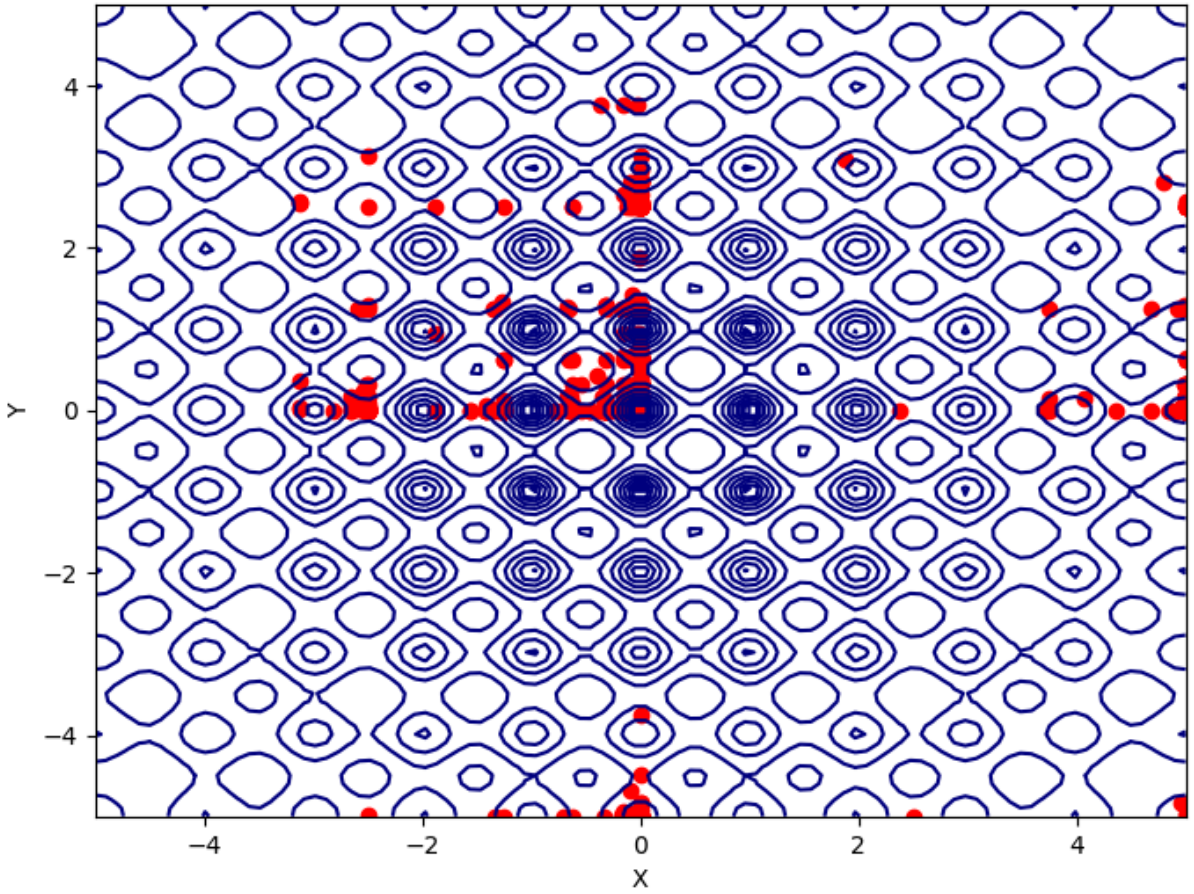
Видно, как популяция концентрируется у локальных минимумов.

Спустя 20 итераций:



Заметна большая группа на (0, 0) - глобальном максимуме. Локальных минимумов, в которых концентрировалась бы популяция становится значительно меньше, и все они находятся на осях, проходящих через 0, что опять же объясняется алгоритмом мутации и скрещивания.

Спустя 100 итераций:



Можно заметить, что помимо (0, 0) многие хромосомы разбросаны в одной четверти. Это может объясняться работой оператора скрещивания, когда одна происходит обмен коэффициентами параметров.

При работе с этой функцией столкнулись с проблемой, что она не завершается или, по крайней мере, работа крайне долго при ограничении только по tol, поэтому было принято ограничить число итераций 1000-ью, что понизило точность.

Метод	Предполагаемая x* (округлено)	f(x*) (округлено)	Количество итераций	Количество вычислений f	Количество вычислений градиента	Количество вычислений гессиана	Время, сек.
свой Ньютон с фиксированным шагом	(0.99496, 0.99496)	1.98992	4	0	4	4	<1
свой Ньютон с	(0.99496, 0.99496	1.98992	7	294	7	7	<1

дихотомией	)						
Newton-CG	(0.99496, 0.99496)	1.98992	3	3	3	3	<1
Генетический алгоритм	(9e-6)	(2e-6)	1000	1e6	0	0	30

Вывод:

Таким образом, мы видим, что для получения той же точности, что детерминированным методами, генетическому алгоритму требуется много больше времени и вычислений функции.

Однако благодаря тому, что на начальных этапах алгоритм опробывает множество точек по всей области определения, он значительно реже попадает в ловушку локального минимума, когда таковых много.

Дополнительное задание. Пункт 1. Применение Optuna к примерам из лаб. 1. Параметры вызываемых библиотечных функций.

1. Метод **suggest_int(name, low, high, step=1, log=False)**:
   - name: Имя параметра.
   - low: Нижняя граница диапазона целых значений параметра.
   - high: Верхняя граница диапазона целых значений параметра.
   - step (необязательный): Шаг для дискретизации диапазона значений (по умолчанию 1).
   - log (необязательный): Если установлен в True, значения будут выбираться с логарифмическим распределением.
2. Метод **suggest_float(name, low, high, step=None, log=False)**:
   - name: Имя параметра.
   - low: Нижняя граница диапазона значений параметра.
   - high: Верхняя граница диапазона значений параметра.
   - step (необязательный): Шаг для дискретизации диапазона значений (по умолчанию None).
   - log (необязательный): Если установлен в True, значения будут выбираться с логарифмическим распределением.

Дополнительное задание. Пункт 2. Применение Optuna к методу Ньютона с фиксированным шагом спуска. Параметры вызываемых библиотечных функций.

1. Метод **create_study(direction='minimize', sampler=None, pruner=None, study_name=None, storage=None, load_if_exists=False)**:

- direction (необязательный): Направление оптимизации. Может быть 'minimize' для минимизации целевой функции или 'maximize' для максимизации. По умолчанию 'minimize'.
- sampler (необязательный): Сэмплер для выбора новых точек испытания. По умолчанию используется RandomSampler, который выбирает точки случайным образом.
- pruner (необязательный): Объект, который принимает решения о прекращении испытаний, основываясь на прогрессе оптимизации. По умолчанию используется SuccessiveHalvingPruner.
- study_name (необязательный): Имя исследования. По умолчанию генерируется автоматически.
- storage (необязательный): Объект, который определяет хранение исследования. По умолчанию используется локальное хранилище SQLite.
- load_if_exists (необязательный): Если True, исследование будет загружено из хранилища, если оно уже существует. По умолчанию False.

2. Метод **study.optimize(func, n_trials=100, timeout=None, n_jobs=1, catch=(<class 'Exception'>,), callbacks=(), gc_after_trial=True)**:
   - func: Целевая функция для оптимизации. Должна принимать объект Trial в качестве аргумента и возвращать значение, которое нужно минимизировать.
   - n_trials (необязательный): Количество пробных запусков для выполнения оптимизации. По умолчанию 100.
   - timeout (необязательный): Максимальное время (в секундах), в течение которого выполняется оптимизация. По умолчанию None.
   - n_jobs (необязательный): Количество потоков для распараллеливания вычислений. По умолчанию 1.
   - catch (необязательный): Исключения, которые следует перехватывать во время исполнения целевой функции. По умолчанию перехватываются все исключения (Exception).
   - callbacks (необязательный): Список функций обратного вызова, которые вызываются после каждого пробного запуска.
   - gc_after_trial (необязательный): Если True, вызывается сборка мусора после каждого пробного запуска. По умолчанию True.