МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ им. Р.Е.АЛЕКСЕЕВА

Институт радиоэлектроники и информационных технологий

Кафедра «Информатика и системы управления»

ОТЧЕТ

**Лабораторная работа №3**

По дисциплине: «Технологии разработки ПО»

Тема: «Исследование особенностей алгоритма Бройдена — Флетчера — Гольдфарба — Шаннои сравнение его с методом градиентного спуска»

РУКОВОДИТЕЛЬ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Чернобаев И.Д.

СТУДЕНТ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Дубков И.А.

Группа М23-ИВТ-1

Работа защищена «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_

Нижний Новгород 2024

Цель работы:

Реализовать алгоритм BFGS

Выявить преимущества и недостатки по отношению к методу градиентного спуска

Теоретическая часть:

### Основные шаги алгоритма BFGS

1. **Инициализация**: Начинаем с начальной точки x0​ и задаем начальную матрицу Гессиана H0 (обычно можно взять единичную матрицу).
2. **Итерации**:
   * Вычисляем градиент функции в текущей точке
   * Определяем направление поиска:
   * Находим шаг ​ с помощью метода поиска по линии.
   * Обновляем параметры:
   * Обновляем градиент:
   * Обновляем матрицу Гессиана:
     + Вычисляем ,
     + Обновляем матрицу ​ с использованием формул Бройдена.
3. **Условие остановки**: Процесс продолжается, пока не будет достигнута заданная точность или максимальное количество итераций.

Ход работы:

Реализуем наш алгоритм на языке программирования python:

Введем функцию и ее производную:

def analyzed\_func(x):  
 *#Функция чашки.* return (x[0]-1)\*\*2+x[1]\*\*2  
def analyzed\_func\_grad(x):  
 *#Градиент функции чашки* dfdx0 = 2\*(x[0]-1)  
 dfdx1 = 2\*x[1]  
 return np.array([dfdx0, dfdx1])

Функция линейного поиска:

def line\_search(f, grad, x, p):alpha = 1.0  
 c = 1e-4  
 while f(x + alpha \* p) > f(x) + c \* alpha \* grad(x) @ p:  
 alpha \*= 0.5 # Уменьшаем шаг  
 return alpha

Реализуем функцию bfgs, со всеми вышеуказанными шагами в ней:

def bfgs(initial\_x, max\_iter=1000, tol=1e-6):n = len(initial\_x)  
 x\_k = initial\_x  
 g\_k = analyzed\_func\_grad(x\_k)  
 H\_k = np.eye(n) # Начальный Гауссиан  
 for i in range(max\_iter):  
 if np.linalg.norm(g\_k) < tol:  
 print(f"Сошлось за {i} итераций.")  
 return x\_k  
  
 p\_k = -H\_k @ g\_k # Направление поиска  
  
 # Линейный поиск для нахождения alpha  
 alpha\_k = line\_search(analyzed\_func, analyzed\_func\_grad, x\_k, p\_k)  
 x\_k\_new = x\_k + alpha\_k \* p\_k  
  
 g\_k\_new = analyzed\_func\_grad(x\_k\_new)  
 s\_k = x\_k\_new - x\_k  
 y\_k = g\_k\_new - g\_k  
  
 # Обновление Гауссиана  
 rho\_k = 1.0 / (y\_k @ s\_k)  
 H\_k = (np.eye(n) - rho\_k \* np.outer(s\_k, y\_k)) @ H\_k @ (np.eye(n) - rho\_k \* np.outer(y\_k, s\_k)) + rho\_k \* np.outer(s\_k, s\_k)  
  
 x\_k = x\_k\_new  
 g\_k = g\_k\_new  
  
 print("Достигнуто максимальное количество итераций.")  
 return x\_k

Реализуем градиентный спуск:

def gradient\_descent(initial\_x, max\_iter=10000, tol=1e-6, alpha=0.001):x\_k = initial\_x  
 for i in range(max\_iter):  
 g\_k = analyzed\_func\_grad(x\_k)  
 if np.linalg.norm(g\_k) < tol:  
 print(f"Сошлось за {i} итераций.")  
 return x\_k  
 x\_k = x\_k - alpha \* g\_k # Обновление точки  
 print("Достигнуто максимальное количество итераций.")  
 return x\_k

Затем сравниваем результаты:

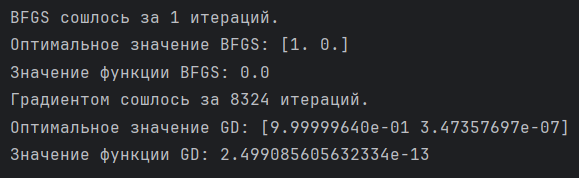
# Начальная точка  
initial\_x = np.array([-5.21, 6.0])  
  
# Запуск BFGS  
result\_bfgs = bfgs(initial\_x)  
print("Оптимальное значение BFGS:", result\_bfgs)  
print("Значение функции BFGS:", analyzed\_func(result\_bfgs))  
  
# Запуск градиентного спуска  
result\_gd = gradient\_descent(initial\_x)  
print("Оптимальное значение GD:", result\_gd)  
print("Значение функции GD:", analyzed\_func(result\_gd))

Анализ полученных результатов:

Программа была протестирована на нескольких функциях:

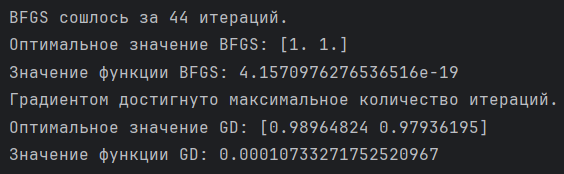
def analyzed\_func(x):  
 *#Функция чашки.* return (x[0]-1)\*\*2+x[1]\*\*2  
def analyzed\_func\_grad(x):  
 *#Градиент функции чашки* dfdx0 = 2\*(x[0]-1)  
 dfdx1 = 2\*x[1]  
 return np.array([dfdx0, dfdx1])

Результаты:



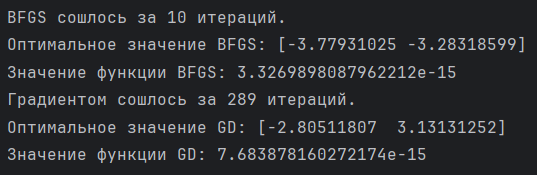
def analyzed\_func(x):  
 *#Функция Розенброк* return (1 - x[0])\*\*2 + 100 \* (x[1] - x[0]\*\*2)\*\*2  
def analyzed\_func\_grad(x):  
 *#Градиент функции Розенброк* dfdx0 = -2 \* (1 - x[0]) - 400 \* x[0] \* (x[1] - x[0]\*\*2)  
 dfdx1 = 200 \* (x[1] - x[0]\*\*2)  
 return np.array([dfdx0, dfdx1])

Результаты:

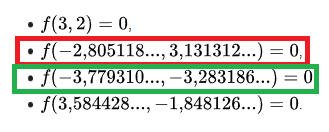


def analyzed\_func(x):  
 *#Функция Химмельблау.* return (x[0]\*\*2 + x[1] - 11)\*\*2 + (x[0] + x[1]\*\*2 - 7)\*\*2  
def analyzed\_func\_grad(x):  
 *#Градиент функции Химмельблау* dfdx0 = 4\*x[0]\*(x[0]\*\*2 + x[1] - 11) + 2\*(x[0] + x[1]\*\*2 - 7)  
 dfdx1 = 2\*(x[0]\*\*2 + x[1] - 11) + 4\*x[1]\*(x[0] + x[1]\*\*2 - 7)  
 return np.array([dfdx0, dfdx1])

Результаты:



\*В этом случае результаты расходятся у двух методов, потому что функция Химмельблау имеет четыре минимума, и наши методы нашли два разных:



Заметим, что во всех случаях BFGS показал себя намного более быстрым по количеству итераций алгоритмом.

Вывод:

#### Преимущества BFGS по сравнению с методом градиентного спуска:

* **Скорость сходимости**: BFGS, как правило, сходится быстрее, особенно вблизи минимума, благодаря использованию информации о вторых производных.
* **Адаптивность**: BFGS автоматически адаптирует направление и величину шага, что делает его более устойчивым к проблемам, связанным с выбором шага.
* **Меньшая вероятность застревания**: Использование информации о кривизне позволяет избегать локальных минимумов лучше, чем метод градиентного спуска.

#### Недостатки BFGS:

* **Сложность**: Реализация BFGS сложнее, чем метод градиентного спуска.
* **Вычислительные ресурсы**: BFGS требует больше вычислений для обновления матрицы Гессиана, что может быть проблемой для больших задач.
* **Память**: Нужно хранить матрицу Гессиана, что увеличивает требования к памяти.
* **Чувствительность к начальным условиям**: часто метод выйдет из-под контроля, если дать ему начальную точку с экстремальными значениями первой или второй производной, в то время как метод градиентного спуска намного более стабилен и может работать с почти любой точкой, где производная первого порядка существует.

**Приложение А** – полный код программы

import numpy as np  
  
# def analyzed\_func(x):  
# #Функция Розенброк.  
# return (1 - x[0])\*\*2 + 100 \* (x[1] - x[0]\*\*2)\*\*2  
# def analyzed\_func\_grad(x):  
# #Градиент функции Розенброк  
# dfdx0 = -2 \* (1 - x[0]) - 400 \* x[0] \* (x[1] - x[0]\*\*2)  
# dfdx1 = 200 \* (x[1] - x[0]\*\*2)  
# return np.array([dfdx0, dfdx1])  
  
# def analyzed\_func(x):  
# #Функция чашки.  
# return (x[0]-1)\*\*2+x[1]\*\*2  
# def analyzed\_func\_grad(x):  
# #Градиент функции чашки  
# dfdx0 = 2\*(x[0]-1)  
# dfdx1 = 2\*x[1]  
# return np.array([dfdx0, dfdx1])  
  
def analyzed\_func(x):  
 #Функция Химмельблау.  
 return (x[0]\*\*2 + x[1] - 11)\*\*2 + (x[0] + x[1]\*\*2 - 7)\*\*2  
def analyzed\_func\_grad(x):  
 #Градиент функции Химмельблау  
 dfdx0 = 4\*x[0]\*(x[0]\*\*2 + x[1] - 11) + 2\*(x[0] + x[1]\*\*2 - 7)  
 dfdx1 = 2\*(x[0]\*\*2 + x[1] - 11) + 4\*x[1]\*(x[0] + x[1]\*\*2 - 7)  
 return np.array([dfdx0, dfdx1])  
  
def line\_search(f, grad, x, p):  
 alpha = 1.0  
 c = 1e-4  
 while f(x + alpha \* p) > f(x) + c \* alpha \* grad(x) @ p:  
 alpha \*= 0.5 # Уменьшаем шаг  
 return alpha  
  
def bfgs(initial\_x, max\_iter=1000, tol=1e-6):  
 n = len(initial\_x)  
 x\_k = initial\_x  
 g\_k = analyzed\_func\_grad(x\_k)  
 H\_k = np.eye(n) # Начальный гауссиан  
  
 for i in range(max\_iter):  
 if np.linalg.norm(g\_k) < tol:  
 print(f"BFGS cошлось за {i} итераций.")  
 return x\_k  
  
 p\_k = -H\_k @ g\_k # Направление поиска  
  
 # Линейный поиск для нахождения alpha  
 alpha\_k = line\_search(analyzed\_func, analyzed\_func\_grad, x\_k, p\_k)  
 x\_k\_new = x\_k + alpha\_k \* p\_k  
  
 g\_k\_new = analyzed\_func\_grad(x\_k\_new)  
 s\_k = x\_k\_new - x\_k  
 y\_k = g\_k\_new - g\_k  
  
 # Обновление Гауссиана  
 rho\_k = 1.0 / (y\_k @ s\_k)  
 H\_k = (np.eye(n) - rho\_k \* np.outer(s\_k, y\_k)) @ H\_k @ (np.eye(n) - rho\_k \* np.outer(y\_k, s\_k)) + rho\_k \* np.outer(s\_k, s\_k)  
  
 x\_k = x\_k\_new  
 g\_k = g\_k\_new  
  
 print("BFGS достигнуто максимальное количество итераций.")  
 return x\_k  
  
def gradient\_descent(initial\_x, max\_iter=10000, tol=1e-6, alpha=0.001):  
 x\_k = initial\_x  
 for i in range(max\_iter):  
 g\_k = analyzed\_func\_grad(x\_k)  
 if np.linalg.norm(g\_k) < tol:  
 print(f"Градиентом сошлось за {i} итераций.")  
 return x\_k  
 x\_k = x\_k - alpha \* g\_k # Обновление точки  
 print("Градиентом достигнуто максимальное количество итераций.")  
 return x\_k  
  
# Начальная точка  
initial\_x = np.array([-3.21, 5.0])  
  
# Запуск BFGS  
result\_bfgs = bfgs(initial\_x)  
print("Оптимальное значение BFGS:", result\_bfgs)  
print("Значение функции BFGS:", analyzed\_func(result\_bfgs))  
  
# Запуск градиентного спуска  
result\_gd = gradient\_descent(initial\_x)  
print("Оптимальное значение GD:", result\_gd)  
print("Значение функции GD:", analyzed\_func(result\_gd))