МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ им. Р.Е.АЛЕКСЕЕВА

Институт радиоэлектроники и информационных технологий

Кафедра информатики и систем управления

Пояснительная записка

К лабораторной работе №3

по дисциплине

Технологии разработки программного обеспечения

РУКОВОДИТЕЛЬ:

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

(подпись) (фамилия, и.,о.)

СТУДЕНТ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Савкин А.Е.

(подпись) (фамилия, и.,о.)

М23-ИВТ-1

(шифр группы)

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

С оценкой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Задание к выполнению лабораторной работы №3**

**BFGS**

# Текст задания

* Реализовать алгоритм BFGS
* Выявить преимущества и недостатки по отношению к методу градиентного спуска

# Теоретическая часть:

Метод **BFGS**, итерационный метод численной оптимизации, назван в честь его исследователей: **B**royden, **F**letcher, **G**oldfarb, **S**hanno. Относится к классу так называемых квазиньютоновских методов. В отличие от ньютоновских методов в квазиньютоновских не вычисляется напрямую гессиан функции, т.е. нет необходимости находить частные производные второго порядка. Вместо этого гессиан вычисляется приближенно, исходя из сделанных до этого шагов.

Существует несколько модификаций метода:

* **L-BFGS** (ограниченное использование памяти) — используется в случае большого количества неизвестных.
* **L-BFGS-B** — модификация с ограниченным использованием памяти в многомерном кубе.

Метод эффективен и устойчив, поэтому зачастую применяется в функциях оптимизации. Например в SciPy, популярной библиотеки для языка python, в функции optimize по умолчанию применяется BFGS, L-BFGS-B.

### Алгоритм

Пусть задана некоторая функция  и мы решаем задачу оптимизации: .  
Где в общем случае  является не выпуклой функцией, которая имеет непрерывные вторые производные.  
  
**Шаг №1**  
Инициализируем начальную точку ;  
Задаем точность поиска > 0;  
Определяем начальное приближение , где  — обратный гессиан функции;  
  
Каким нужно выбрать начальное приближение ?  
К сожалению не существует общей формулы, которая хорошо бы работала во всех случаях. В качестве начального приближения можно взять гессиан функции, вычисленный в начальной точке . Иначе можно использовать хорошо обусловленную, невырожденную матрицу, на практике часто берут единичную матрицу.  
  
**Шаг №2**  
Находим точку, в направлении которой будем производить поиск, она определяется следующим образом:



**Шаг №3**  
Вычисляем  через рекуррентное соотношение:

  
Коэффициент  находим используя линейный поиск (line search), где  удовлетворяет условиям Вольфе ([Wolfe conditions](https://en.wikipedia.org/wiki/Wolfe_conditions)):



  
Константы  и  выбирают следующим образом: . В большинстве реализаций:  и .  
Фактически мы находим такое  при котором значение функции  минимально.  
  
**Шаг №4**  
Определяем вектора:





 — шаг алгоритма на итерации,  — изменение градиента на итерации.  
  
**Шаг №5**  
Обновляем гессиан функции, согласно следующей формуле:



где 



 — единичная матрица.

#### Замечание

Выражение вида  является внешним произведением ([outer product](https://en.wikipedia.org/wiki/Outer_product)) двух векторов.  
Пусть определены два вектора  и , тогда их внешнее произведение эквивалентно матричному произведению . Например, для векторов на плоскости:



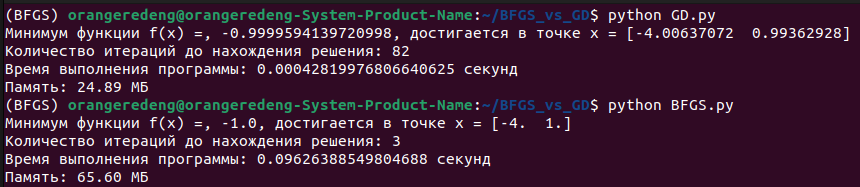
**Шаг №6**  
Алгоритм продолжает выполнятся до тех пор пока истинно неравенство: .

# Ход работы

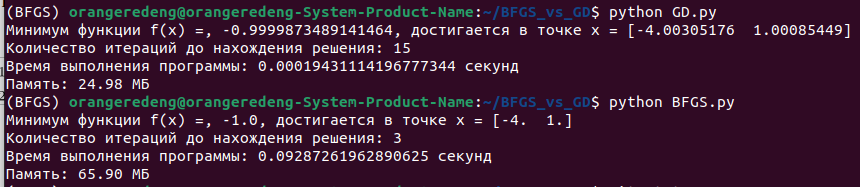
Пусть задана некоторая функция  и мы решаем задачу оптимизации: .  
Необходимая точность ;  
Градиент функции :



Скриншот результата работы скриптов GD и BFGS с указанием количества итераций на решение, минимума функции и точки в которой он достигается, также указано время исполнения программы (с помощью модуля time) и используемая память (с помощью модуля psutil).

(начальная точка [24, -99], lr = 0.1)

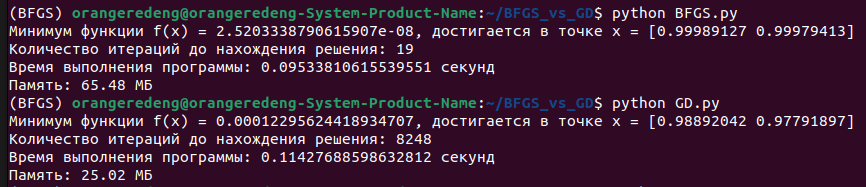
Как видим результаты за исключением количества итераций и точности не за BFGS. Ситуация усугубляется если увеличить LR для метода градиентного спуска:

(начальная точка [24, -99], lr = 0.5)

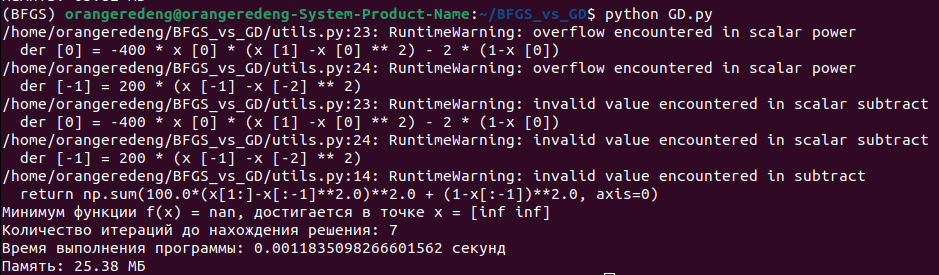
На более «сложных» функциях, таких как

*np.sum(100.0\*(x[1:]-x[:-1]\*\*2.0)\*\*2.0 + (1-x[:-1])\*\*2.0, axis=0)*

BFGS может показывать незначительное приемущество по скорости (в удачном запуске) при количестве итераций на два порядка меньше, чем в алгоритме градиентного спуска.

(начальная точка [0.1, 0.1], lr = 0.001 — иначе алгоритм неустойчив)

GD также показывает себя неустойчиво и требует подбора lr .

(начальная точка [0.1, 0.1], lr = 0.1 — выпадаем по переполнению int32 и даже int64)

# Вывод

**BFGS** - это квазиньютоновский метод, который сходится **за меньшее количество шагов,** чем **GD**, и **имеет немного меньшую склонность к "застреванию"** и **требует** **меньших настроек гиперпараметров** для достижения значительного снижения на каждой итерации. Каждая итерация может быть совершена со стоимостью  ( плюс стоимость вычисления функции и оценки градиента). Здесь нет  операций таких, как решение линейных систем или сложных математических операций. Алгоритм **устойчив** и **имеет сверхлинейную сходимость**. Даже если методы Ньютона сходятся гораздо быстрее (квадратично), стоимость каждой итерации выше, поскольку необходимо решать линейные системы. Неоспоримое преимущество алгоритма, конечно, состоит в том, что нет необходимости вычислять вторые производные.

Методы, подобные **GD**, дешевле, если матрично-векторные вычисления дешевы, а ваша задача настолько велика, что хранение гессиана затруднено или невозможно. **BFGS** использует несколько векторных вычислений для обновления своего приблизительного гессиана, поэтому каждая итерация **BFGS** будет стоить дороже, но вам потребуется меньшее их количество, чтобы достичь локального минимума.

Формула **BFGS** имеет самокорректирующиеся свойства. Если матрица  не верно оценивает кривизну функции и если эта плохая оценка замедляет алгоритм, тогда апроксимация гессиана стремится исправить ситуацию за несколько шагов. Самокорректирующие свойства алгоритма работают только в том случае, если реализован соответствующий линейный поиск (соблюдены условия Вольфе).

**Непонятно как в дальнейшем поведет себя алгоритм в сравнении с более продвинутыми версиями градиентного спуска, есть вероятность, что те преимущества, объявленные выше, перестанут ими быть, а в сухом остатке будет только скорость работы и потребление памяти. Интересно апробировать алгоритм в реальной задаче обучения нейронной сети, в которой чрезвычайно важна скорость работы и потребление памяти. Судя по тому, что градиентный спуск до сих пор является основным алгоритмом в обучении нейросетей, можно сделать вывод, что BFGS на сегодня "не взлетел" и требует дополнительных исследований.**

# Код

**BFGS.py**

'''

Реализация алгоритма BFGS на Python/Numpy/Scipy.

'''

**import** numpy **as** np

**import** numpy.linalg **as** ln

**import** scipy **as** sp

**import** time

**import** psutil

**from** utils **import** f, f\_der

# Реализация BFGS

**def** **bfgs\_method**(f, fprime, x0, maxiter=None, epsi=10e-3):

"""

Минимизация функции f, используя алгоритм BFGS.

Параметры

----------

f : f(x)

Минимизируемая функция.

x0 : ndarray

Начальная точка.

fprime : fprime(x)

Градиент функции f.

"""

**if** maxiter **is** None:

maxiter = len(x0) \* 200

# Начальные значения

k = 0

gfk = fprime(x0)

N = len(x0)

# Создаем единичную матрицу I

I = np.eye(N, dtype=int)

Hk = I

xk = x0

**while** ln.norm(gfk) > epsi **and** k < maxiter:

# pk - направление поиска

pk = -np.dot(Hk, gfk)

# line\_search не только alpha

# но только это значение интересно для нас

line\_search = sp.optimize.line\_search(f, f\_der, xk, pk)

alpha\_k = line\_search[0]

xkp1 = xk + alpha\_k \* pk

sk = xkp1 - xk

xk = xkp1

# print(f'Итерация: {k}; найденная точка x: {xk}')

gfkp1 = fprime(xkp1)

yk = gfkp1 - gfk

gfk = gfkp1

k += 1

ro = 1.0 / (np.dot(yk, sk))

A1 = I - ro \* sk[:, np.newaxis] \* yk[np.newaxis, :]

A2 = I - ro \* yk[:, np.newaxis] \* sk[np.newaxis, :]

Hk = np.dot(A1, np.dot(Hk, A2)) + (ro \* sk[:, np.newaxis] \*

sk[np.newaxis, :])

**return** (xk, k)

**def** **main**():

ordinary\_F()

**def** **ordinary\_F**():

start\_time = time.time() # время начала выполнения

minimum\_x, iter\_count = bfgs\_method(f, f\_der, np.array([24, -99]))

minimum\_value = f(minimum\_x)

print(f'Минимум функции f(x) =, {minimum\_value}, достигается в точке x = {minimum\_x}')

print(f'Количество итераций до нахождения решения: {iter\_count}')

end\_time = time.time() # время окончания выполнения

execution\_time = end\_time - start\_time # вычисляем время выполнения

print(f"Время выполнения программы: {execution\_time} секунд")

print(f"Память: {psutil.Process().memory\_info().rss / 1024 \*\* 2:.2f} МБ")

**if** \_\_name\_\_ == "\_\_main\_\_":

main()

**GD.py**

'''

Реализация алгоритма Gradient Descend на Python/Numpy.

'''

**import** numpy **as** np

**import** numpy.linalg **as** ln

**import** time

**from** utils **import** f, f\_der

**import** psutil

**def** **gradient\_descent**(fprime, x0, lr, maxiter, epsi=10e-3):

"""

Минимизация функции f, используя алгоритм Gradien descend.

Параметры

----------

x0 : ndarray

Начальная точка.

fprime : fprime(x)

Градиент функции f.

lr: learning\_rate.

Cкорость обучения

"""

**if** maxiter **is** None:

maxiter = len(x0) \* 200

# Начальные значения

x = x0

gfk = fprime(x0)

k = 0

# for i in range(max\_iterations):

**while** ln.norm(gfk) > epsi **and** k < maxiter:

gradient = fprime(x)

x = x - lr \* gradient

gfkp1 = fprime(x)

gfk = gfkp1

k += 1

# print(f'Итерация: {k}; найденная точка x: {x}')

**return** x, k

**def** **main**():

ordinary\_F()

**def** **ordinary\_F**():

start\_time = time.time() # время начала выполнения

# Гиперпараметры

x0 = np.array([24, -99])

lr = 0.5

maxiter = 100

minimum\_x, iter\_count = gradient\_descent(f\_der, x0, lr, maxiter)

minimum\_value = f(minimum\_x)

print(f'Минимум функции f(x) =, {minimum\_value}, достигается в точке x = {minimum\_x}')

print(f'Количество итераций до нахождения решения: {iter\_count}')

end\_time = time.time() # время окончания выполнения

execution\_time = end\_time - start\_time # вычисляем время выполнения

print(f"Время выполнения программы: {execution\_time} секунд")

print(f"Память: {psutil.Process().memory\_info().rss / 1024 \*\* 2:.2f} МБ")

**if** \_\_name\_\_ == "\_\_main\_\_":

main()

**Utils.py**

**import** numpy **as** np

# Целевая функция

**def** **f**(x):

**return** x[0]\*\*2 - x[0]\*x[1] + x[1]\*\*2 + 9\*x[0] - 6\*x[1] + 20

# Производная

**def** **f\_der**(x):

**return** np.array([2 \* x[0] - x[1] + 9, -x[0] + 2\*x[1] - 6])