ПРОГРАММИРОВАНИЕ И АЛГОРИТМИЗАЦИЯ

Методические указания к лабораторным работам для студентов бакалавриата направления 15.03.04

САНКТ-ПЕТЕРБУРГ
2019

Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное
учреждение высшего образования
Санкт-Петербургский горный университет

Кафедра автоматизации технологических процессов и производств

ПРОГРАММИРОВАНИЕ И АЛГОРИТМИЗАЦИЯ

Методические указания к лабораторным работам
для студентов бакалавриата направления 15.03.04

САНКТ-ПЕТЕРБУРГ
2019

УДК 004.423 (073)

ПРОГРАММИРОВАНИЕ И АЛГОРИТМИЗАЦИЯ: Методические указания к лабораторным работам / Санкт-Петербургский горный университет. Сост.: *В.В. Губин, А.А. Дарьин.* СПб. 2019. 44 с.

Тематика представленных работ направлена на изучение основ программирования на языке Python, изучения алгоритмов и принципов их реализации.

Предназначены для студентов бакалавриата направления 15.03.04 «Автоматизация технологических процессов и производств»

Научный редактор проф. *В.Ю. Бажин*

Рецензент канд. техн. наук *В.В. Васильев* («ТОМС инжиниринг»)

ПРОГРАММИРОВАНИЕ И АЛГОРИТМИЗАЦИЯ

Методические указания к лабораторным работам
для студентов бакалавриата направления 15.03.04

Сост.: *В.В. Губин, А.А. Дарьин*

Печатается с оригинал-макета, подготовленного кафедрой
автоматизации технологических процессов и производств

Ответственный за выпуск *В.В. Губин*

Лицензия ИД № 06517 от 09.01.2002

Подписано к печати 06.09.2019. Формат 60×84/16.
Усл. печ. л. 2,6. Усл.кр.-отт. 2,6. Уч.-изд.л. 2,0. Тираж 50 экз. Заказ 761. С 264.

Санкт-Петербургский горный университет
РИЦ Санкт-Петербургского горного университета
Адрес университета и РИЦ: 199106 Санкт-Петербург, 21-я линия, 2

ЛАБОРАТОРНАЯ РАБОТА №1. ОРГАНИЗАЦИЯ ЦИКЛОВ

1. Цель работы – Получение навыка использования операторов цикла.

2. Постановка задачи. Используя оператор цикла, найти сумму элементов, указанных в конкретном варианте. Результат напечатать, снабдив соответствующим заголовком

3. Основные теоретические положения:

Циклические конструкции в языке программирования Python.

В языке программирования Python существуют два вида циклических конструкций.

Цикл **for**

Часто цикл **for** называют циклом со счетчиком. В языке Python оператор for служит для перебора всех элементов в объекте-последовательности. Последовательность это специальный тип объекта в Python, который поддерживает итерирование. Количество выполняемых циклов всегда равно количеству элементов в последовательности, поэтому если она конечна, то наш цикл также всегда конечен.

Синтаксис конструкции указан в листинге 1.

Листинг 1

```
for <имя переменной> in <последовательность>:
     <тело цикла>
```

Переменная по порядку принимает значение всех элементов последовательности, что соответствует простой операции присваивания.

Листинг 2

```
for i in [1,2,3,4,5]:
    print(i, end = ' ')

>> 1 2 3 4 5
```

Для того, чтобы использовать конструкцию **for in** именно в качестве цикла со счетчиком, можно использовать следующую форму:

Листинг 3

```
for i in range(5):
    print(i, end = ' ')

>> 0 1 2 3 4
```

Range – это команда, возвращающая объект класса range. Range – интервал – неизменяемая последовательность целых чисел. В скобках указываются следующие аргументы:

```
range(start_or_stop, stop[, step])  -> range
```

start_or_stop=0 : Целое число, которое должно явиться началом последовательности. Если тип инициализируется с одним аргументом, то значение трактуется как stop, а начало последовательности при этом — 0.

stop: Целое число, на котором должно завершиться формирование последовательности. Не входит в последовательность.

step=1: Целое число — шаг, с которым должна формироваться последовательность. При попытке задать нуль, возбуждается ValueError.

Цикл **while**

Цикл **while** называется также циклом с предусловием и имеет следующий вид (листинг 3).

Участок кода, находящийся в теле цикла, выполняется итерация за итерацией, пока выражение, описанное в заголовке цикла равно **True**. Когда логическое выражение вернет значение **False**, произойдет выход из цикла. Для цикла **while** очень важно в теле цикла предусмотреть изменение переменной, фигурирующей в заголовке цикла, таким образом, чтобы когда-нибудь обязательно наступала ситуация **false**. Иначе произойдет так называемое **зацикливание**, одна из самых неприятных ошибок в программировании.

Листинг 4

```
n = 0
while n < 10:
    n = n + 1                #без этой операции
цикл будет бесконечным
    print('*', end = ' ')

Out - > * * * * * * * * * *
```

После ключевого слова **while** должно располагаться выражение, возвращающее логическое выражение, но также могут быть указаны переменные других типов, которые могут быть неявно преобразованы в логический тип. При явном преобразовании числовых типов данных в bool возвращается False, если значение равно нулю (0 или 0.0) и True в любом другом значении. При преобразовании типов-контейнеров False возвращается, если переменная не содержит элементов, т.е. преобразуется пустая строка, пустой кортеж, пустой список, пустое множество и т.д. (Листинг 4).

Листинг 5

```
print(bool(10))#целое число не равно нулю -> True
print(bool(0)) #целое число равно нулю -> False
print(bool(10.3)) #вещественное число не равно
                  #нулю -> True
print(bool(0.0)) #вещественное число  равно нулю
                  # -> False
print(bool('10'))  #не пустая строка -> True
print(bool(''))    #пустая строка -> False
print(bool([1,2])) #не пустой список -> True
print(bool([]))    #пустой список -> False
```

Таким образом, можно организовать цикл так, чтобы он выполнялся пока переменная не примет значение 0, для числового типа:

Листинг 6

```
m = 10
while m:
    print(m, end = ' ')
    m = m - 1

Out - > 10 9 8 7 6 5 4 3 2 1
```

или пока в списке есть хоть один элемент:

Листинг 7

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
while nums:
    print(nums.pop(), end = ' ')
    #метод pop() объектов-списков возвращает
    #элемент списка по его
#индексу и удаляет его, если индекс не указан
#метод возвращает #последний элемент списка

Out - > 10 9 8 7 6 5 4 3 2 1
```

Оператор **break**

Появление оператора **break** внутри цикла означает, что цикл в этом месте будет прерван и преждевременно завершен. Например,

Листинг 8

```
for i in range(5):
    if i==3:
        break
    print(i, end = ' ')

Out - > 0 1 2
```

Оператор **continue**

Появление оператора **continue** внутри цикла означает, что данная итерация в этом месте будет прервана, но цикл продолжит работу со следующей итерации. Например,

Листинг 9

```
for i in range(5):
    if i==3:
        continue
    print(i, end = ' ')

Out - > 0 1 2 4
```

Графическое описание алгоритма.

Алгоритм решения задачи можно представить в виде блок-схемы. Для этого используются специальные символы (рис.1):

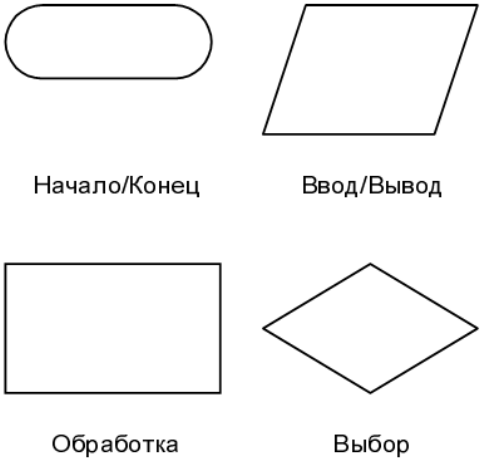

Рис.1. Символы блок-схемы

Например, так выглядит цикл While (цикл с предусловием) (рис. 2):

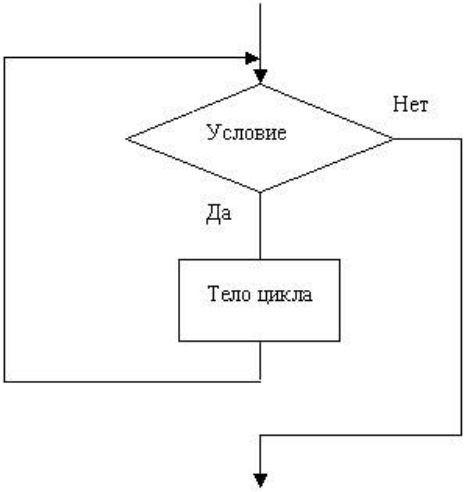

Рис.2. Блок-схема цикла while

4. Порядок выполнения работы:

При определении суммы членов ряда следует использовать рекуррентную формулу для получения следующего члена ряда.

Например, требуется найти сумму ряда с точностью $\varepsilon = 10^{-4}$, общий член которого равен $a_n = 2(n!)^2 /(3(2n)!)$.

На рисунке 3 представлена блок-схема алгоритма.

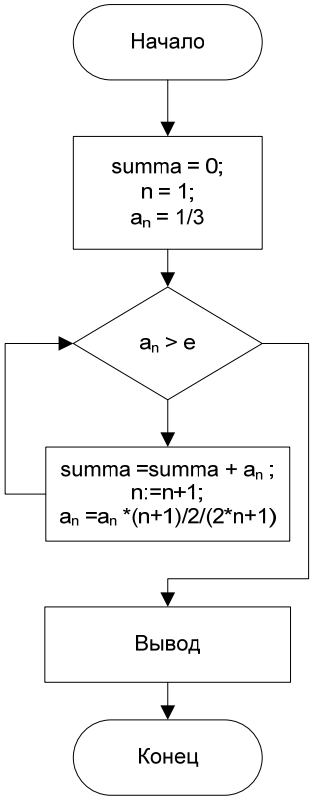

Рис. 3. Блок-схема алгоритма, представленного в листинге 5

Для получения рекуррентной формулы вычислим отношение следующего члена ряда к текущему:

$$\frac{a_{n+1}}{a_n} = \frac{2((n+1)!)^2 \cdot 3(2n)!}{3(2n+2)!\cdot 2(n!)^2} = \frac{n+1}{2(2n+1)},$$

откуда

$$a_{n+1} = a_n(n+1)/(4n+2).$$

Далее необходимо составить алгоритм вычисления суммы членов ряда по полученной рекуррентной формуле. Чтобы уяснить порядок действий, которые должны быть выполнены для решения задачи, удобно представить алгоритм в виде блок-схемы.

Ниже представлен пример программы на языке Python, реализующий приведенный выше алгоритм. При составлении программы будем считать, что точность достигнута, если $a_n < \varepsilon$.

Листинг 10

```
eps = 0.001      #требуемая точность
summa = 0        #текущее значение суммы ряда
a = 1/3          #первый член арифметического ряда
n = 1            #номер члена арифметического ряда
while a > eps:
    summa = summa + a
    n = n + 1
    a = a * (n+1) / 2 / (2*n+1)

# ниже для вывода на экран результатов работы
#программы в функции print просто
# перечисляются переменные, которые надо вывести
print('Сумма = ', summa,', последний член ряда = ', a)

#можно использовать довольно грубый способ
#форматирования строк - конкатенацию
```

```
print('Сумма = ' + str(summa) + ', последний\
член ряда = ' + str(a))

#более удобный способ - использование f-строк
print(f'Сумма = {summa}, последний член ряда =
{a}')

#можно указать точность, с какой надо выводить
#число. Сумму округлим до 3 знака после запятой,
#a - до 4 знака
print(f'Сумма = {summa:.3f}, последний член ряда\
= {a:.4f}')
```

Протокол работы программы:

```
Сумма = 0.472, последний член ряда = 0.0006
```

Теперь напишем для сравнения рекурсивную функцию для указанного алгоритма.

5. Содержание отчета:

- Название и цель работы
- Постановка задачи.
- Полученная рекуррентная формула
- Блок-схема алгоритма
- Текст программы.
- Результат решения конкретного варианта.

ЛАБОРАТОРНАЯ РАБОТА №2. ИСПОЛЬЗОВАНИЕ ОПЕРАТОРА МНОЖЕСТВЕННОГО ВЫБОРА

1. Цель работы – Получение навыков структурного программирования..

2. Основные теоретические положения:

Во многих языках программирования используется Switch-Case, который позволяет лучше структурировать программу, избегая сложных построений алгоритмов со многократным использованием вложенных операторов if.. else.

В языке Python непосредственно оператор case отсутствует, хотя речь о его введении в будущих релизах уже ведется.

В общем виде оператор выглядит следующим образом:

```
switch VAR:
    case VALUE1:
        ACT1( )
    case VALUE2:
        ACT2( )
    ...
    else:
        DEFAULT_ACT( )
```

Если VAR == VALUE1, выполняется соответствующее действие ACT1.

Такую конструкцию можно заменить имеющимися средствами языка

а) оператор if..elif..else.

Пример с этим оператором:

```
if VAR == VALUE1:
```

```
    ACT1( )
elif VAR == VALUE2:
    ACT2( )
else:
    DEFAULT_ACT( )
```

Здесь сравнение переменной VAR со значением происходит в каждой секции оператора условия.

б) с помощью словаря.

Словарь – ассоциативный массив, который хранит пары ключ-значение. В качестве ключа может использоваться VALUE, в качестве значения указанная функция.

```
def ACT1():
    pass

def ACT2():
    pass

def DEFAULT_ACT():
    pass

SWITCH = {VALUE1: ACT1(),
                      VALUE2: ACT2()}

SWITCH.get(VAR,DEFAULT_ACT())
```

В начале примера описываются функции, в которых указаны выполняемые действия. Затем инициализируется словарь, содержащий пары {значение VAR: действие}. Действие по умолчанию, которое ранее указывалось после оператора else, теперь указывается в качестве параметра метода словаря get и выполняется, если текущее значение VAR не соответствует ни одному ключу словаря SWITCH.

3. Порядок выполнения работы

Дан алгоритм, который описывается следующей структурной схемой:

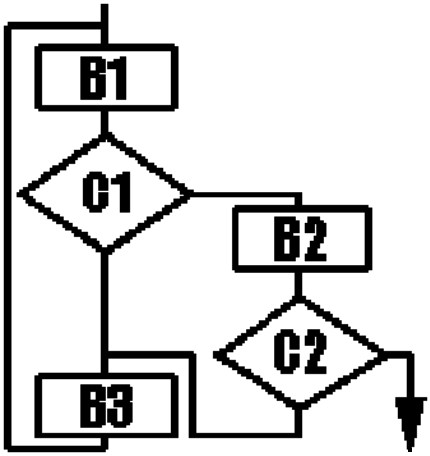

Рис.4 Пример алгоритма

Запишем его с помощью стандартных условной конструкции и циклов:

```
while True:
    B1
    if C1:
        B2
        if C2:
            break
        else:
            B3
    else:
        B3
```

Выделяем в нашем алгоритме фрагменты, которые хорошо укладываются в структурную модель (если такие есть).

В нашем случае такой фрагмент только один: B2 + C2, т.е. последовательность из блока и условия. Вне этих фрагментов ставим жирные точки в следующих местах:

- на входе в модуль (обозначим ее 1)
- на выходе модуля (обозначим 0)
- на входах и выходах всех фрагментов, что мы нашли
- во всех местах, где есть пересечение линий на блок-схеме

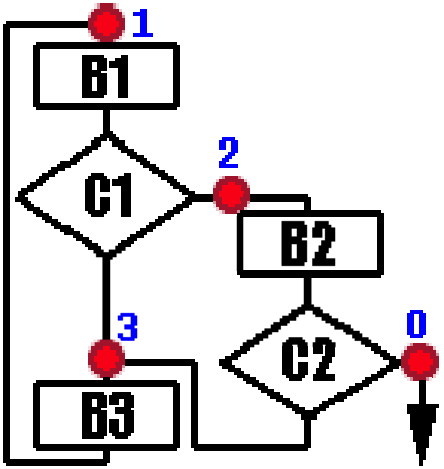

Рис.5 Разбиение алгоритма на фрагменты

Пронумеруем оставшиеся точки произвольно. В нашем примере получается 4 точки от 0 до 3. Теперь мы готовы перейти к модели конечного автомата и написать-таки нашу программу.

Алгоритм характеризует некую систему (объект), которая может находиться в одном из 4 состояний. И есть набор условий, в результате которых блок переходит из одного состояния в другое.

Для отображения этого самого состояния, заведем в программе некоторую переменную, скажем, state. И внутри нашей конструкции будем изменять ее состояние.

а) Напишем первый вариант программы, используя структуру if..elif..else.

Для начала опишем действия B1, B2, B3 в виде функций, симулирующий реальное действие. Пускай они просто выводят на экран номер соответствующего действия.

```
def B1():
    print('B1')

def B2():
    print('B2')

def B3():
    print('B3')
```

Условия C1, C2 – функции, возвращающие булевое значение, которое оно получает от пользователя, например:

```
def C1():
    answer = input('Условие C1[Yes or No]')
    if answer == 'Yes':
        return True
    elif answer == 'No':
        return False
    return C1()

def C2():
    answer = input('Условие C2[Yes or No]')
    if answer == 'Yes':
        return True
    elif answer == 'No':
        return False
    return C2()
```

Сама структура перехода из одного состояния в другое выглядит так:

```
state = 1               #начальное состояние \
начало алгоритма
while True:
    if state == 0:      #конец алгоритма
        break
    elif state == 1:
        pass
    elif state == 2:
        pass
    elif state == 3:
        pass
    print(f'текущая точка: {state}')
```

Вместо операторов pass теперь необходимо поместить соответствующие инструкции и условные операторы. Например, так:

```
elif state == 1:
        B1()
        if C1():
            state = 2
        else:
            state = 3
```

Допишем код и запустим приложение в консоли. Полный код программы расположен ниже:

```
# Описание функций

def B1():
    print('B1\n')

def B2():
    print('B2\n')
```

```
def B3():
    print('B3\n')

def C1():
    answer = input('Условие C1[Yes or No]:\t')
    if answer == 'Yes':
        return True
    elif answer == 'No':
        return False
    return C1()

def C2():
    answer = input('Условие C2[Yes or No]:\t')
    if answer == 'Yes':
        return True
    elif answer == 'No':
        return False
    return C2()

#Основная программа

state = 1             #начальное состояние \
начало алгоритма
while True:
    if state == 0:    #конец алгоритма
        break
    elif state == 1:
        B1()
        if C1():
            state = 2
        else:
            state = 3
    elif state == 2:
        B2()
        if C2():
            state = 0
        else:
```

```
            state = 3
    elif state == 3:
        B3()
        state = 1
    print(f'Переход -> {state}')
```

б) Разработаем другой вариант программы с использованием встроенного типа dict (словарь).

Функции B1, B2, B3, C1 и C2 выглядят так же, как и в предыдущем примере. Текущий вариант состоит из описания функций, соответствующим каждому из состояний, инициализации словаря и цикла:

```
# Описание блоков алгоритма

def func_state1():
    print(1)
    B1()
    if C1():
        return 2
    return 3

def func_state2():
    print(2)
    B2()
    if C2():
        return 0
    return 3

def func_state3():
    print(3)
    B3()
    return 1

#словарь
```

```
STATE = {1: func_state1, #каждому состоянию
соответствует своя функция
         2: func_state2,
         3: func_state3}

#Основная программа

state = 1              #начальное состояние \
начало алгоритма
while state:
    state = STATE[state]()   #словарь возвращает
объект-функцию
    #чтобы ее вызвать, добавляем круглые скобки в
конце
    print(f'Переход -> {state}')
```

4. Содержание отчета:

4.1 Название и цель работы
4.2 Блок-схема алгоритма с пронумерованными точками
4.3 Текст программы.
4.4 Результат решения конкретного варианта.

ЛАБОРАТОРНАЯ РАБОТА №3. АНАЛИЗ ВРЕМЕННОЙ СЛОЖНОСТИ АЛГОРИТМОВ

1. Цель работы – Научиться анализировать алгоритмы с точки зрения их временной сложности.

2. Постановка задачи. Реализовать заданный алгоритм, проанализировать его временную сложность, получить зависимость времени выполнения функции от размера входных данных.

3. Основные теоретические положения:

Рассмотрим алгоритм, который определяет, содержатся ли в массиве повторяющиеся элементы. (Стоит отметить, что это не самый эффективный способ обнаружения дубликатов.)

1. Ввод данных: массив A
2. Перебираем все элементы A, индекс храним в переменной i:
a. Внутри каждого цикла снова перебираем элементы A, индекс храним в переменной j:
- Если i не равен j то:
#Если индексы разные сравниваем сами элементы
Если A[i] равно A[j]:
#если будет найден дубликат, мы выйдем из алгоритма
Вернуть True
#если мы дошли до этой строки, то дубликатов нет.
3. Вернуть False.

Реализация на Python:

```
def ContainsDuplicates(A):
    for i in range(len(A)):
        for j in range(len(A)):
```

```
            if i!=j:
                if A[i]==A[j]:
                    return True
    return False
```

Алгоритм содержит два цикла, один из которых является вложенным. Внешний цикл перебирает все элементы массива N, выполняя $O(N)$ шагов. Внутри каждого такого шага внутренний цикл повторно пересматривает все N элементов массива, совершая те же $O(N)$ шагов. Следовательно, общая производительность алгоритма составит $O(N \cdot N) = O(N^2)$.

Для эмпирического анализа времени работы алгоритма используем следующий код:

```
def ContainsDuplicates(A):
    for i in range(len(A)):
        for j in range(len(A)):
            if i!=j:
                if A[i]==A[j]:
                    return True
    return False

def main(A):                    #функция расчета
времени выполнения
    a = timeit.default_timer()
    f = ContainsDuplicates(A)
    b = timeit.default_timer()
    return b-a

import numpy as np
import numpy.random as rand
import timeit
import matplotlib.pyplot as plot

fig = plot.figure()   # Создание объекта Figure
print (fig.axes)    # Список текущих областей
                    #рисования пуст
```

```
print (type(fig))   # тип объекта Figure
for i in range(100,10000,100):
    Arr = rand.choice(i,i,replace=False,p=None
    #генерируем массив случайных целых чисел
    #размера i
    time = main(Arr)
    plot.scatter(i,time, s = 0.5, color = 'black')
print (fig.axes)
plot.show()
```

Про функцию choice стоит рассказать подробнее.

```
numpy.random.choice(a, size=None, replace=True,
p=None)
```

a : одномерный массив или число. Если массив, будет производиться выборка из него. Если число, то выборка будет производиться из np.arange(a).

size : размерности массива. Если None, возвращается одно значение.

replace : если True, то одно значение может выбираться более одного раза, если False, то значения в массиве не повторяются.

p : вероятности. Это означает, что элементы можно выбирать с неравными вероятностями. Если не задано, используется равномерное распределение.

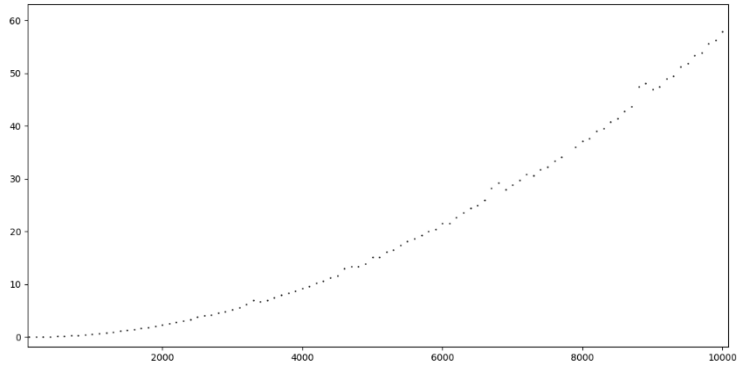

Рис. 6. График времени выполнения алгоритма в зависимости от размера входного массива

Так выглядит график зависимости времени выполнения алгоритма от размера массива при подаче на вход массива без элементов-дубликатов. Таким образом, алгоритм всегда выполняется максимально возможное время. То есть верхняя граница времени выполнения алгоритма равна $C*N^2$, где вычисляется несложно, например,:

$C = T/N^2 = 5.9375e-07$

Если брать разные пары чисел, то константа будет немного отличаться, это нормально. Каждый раз время выполнения даже одной и той же операции немного другое. Добавим к предыдущему коду строки:

```
t = np.arange(100,10100,100)
C = 5.9375e-07
plot.plot(t,C*t**2,'r--')
```

И нанесем, таким образом, на предыдущий график линию функции времени алгоритма.

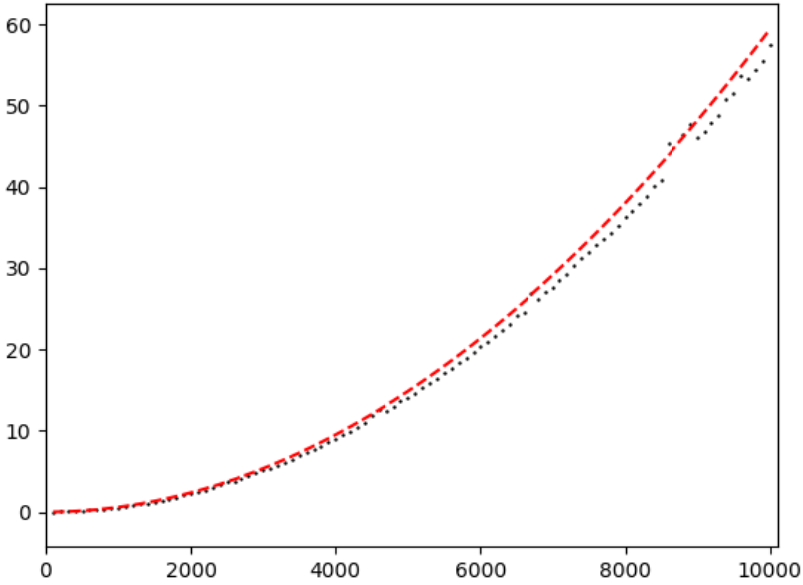

Рис. 7. График времени выполнения алгоритма в зависимости от размера входного массива

На графике пунктирной линией обозначена кривая функции $f(n)= 5.9375e\text{-}07 \cdot n^2$.

Поменяем в строке Arr = rand.choice(i,i,replace=False,p=None) значение параметра replace на True. Теперь в массиве числа могут повторяться. Это сразу скажется на времени выполнения алгоритма, так как повторяющийся элемент может находиться в равных местах массива, а обнаружение такого элемента является условием окончания поиска.

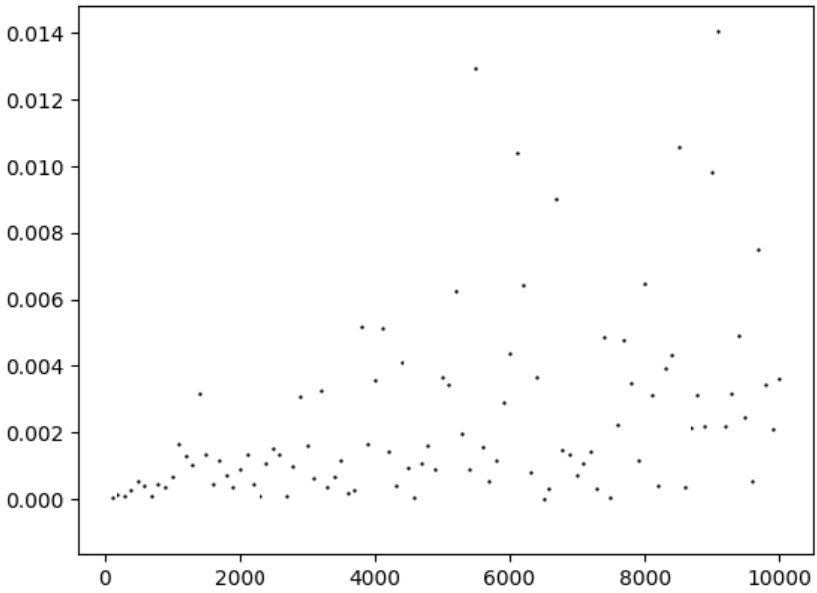

Рис. 8. График времени выполнения алгоритма в зависимости от размера входного массива

Поэтому график приобретает такой вид, как показано сверху. О чем он говорит? Верхняя граница времени выполнения по-прежнему имеет сложность $\Theta(n^2)$. А нижняя – $\Theta(1)$, так как элемент и его дубликат могут оказаться в начале массива.

3. Порядок выполнения работы:

1. Написать реализацию алгоритма. Структурировать программу следующим образом:

```
def alg(arg1,arg2,..):
        #какой-то алгоритм
        return #возвращаемое значение

def test(alg,arg1,arg2,…):
        f = alg(arg1,arg2,…)
        return f
```

2. Далее необходимо придумать тесты, проверяющие верную работу алгоритма и его устойчивость к различным вариантам входных данных (к примеру, что будет, если на вход алгоритма подать пустое множество, ноль, отрицательное число, неподходящий тип данных). На этом этапе вы дорабатываете алгоритм таким образом, чтобы он проходил все тесты.

```
def test(alg,arg1,arg2,…):
       assert alg(0) == 1
       assert alg(-10) == None
       assert alg('15') == 15
       return True
```

3. Далее выполняется загрузка необходимых модулей, генерирование входных данных, вывод результатов вычисления и времени выполнения алгоритма.

```
def test_time(alg,arg1,arg2,…):
       #возвращает время выполнения функции
       a = timeit.default_timer()
       alg(arg1, arg2, …)
       b = timeit.default_timer()
       return b-a

Arr = rand.choice(i,i,replace=False,p=None)
#генерируем массив случайных целых чисел
#размера i
time = test_time(Arr)
```

4. Используя библиотеку matplotlib и цикл со счетчиком для генерации входных данных разного размера, постройте график зависимости времени выполнения алгоритма от размера входных данных.

5. Знаю форму зависимости, итоговое время и входные данные, рассчитайте постоянный коэффициент для формулы времени выполнения алгоритма. Добавьте найденную функцию на итоговый график.

6. Сделайте вывод.

4. Пример варианта задания:

Реализовать и проанализировать Рекурсивный алгоритм вычисления факториала числа.

5. Содержание отчета:

5.1 Название и цель работы
5.2 Постановка задачи.
5.3 Блок-схема алгоритма
5.4 Текст программы.
5.5 Графики зависимости времени выполнения от размерности аргумента
5.6 Вывод о сложности алгоритма.

ЛАБОРАТОРНАЯ РАБОТА №4. РЕШЕНИЕ ЗАДАЧИ РЕГРЕССИИ МЕТОДОМ ГРАДИЕНТНОГО СПУСКА.

1. Цель работы – Научиться реализовывать алгоритмы оптимизации линейных функций на примере градиентного метода и задачи линейной регрессии.

2. Постановка задачи.

В этой работе мы рассмотрим простой градиентный метод оптимизации применительно к задаче линейной регрессии.

Решить задачу линейной регрессии – значит, установить линейную связь между входными переменными (X) и одной выходной переменной (Y).

Рассмотрим наш пример. У нас есть 2 входные независимые переменные x_1 и x_2, и одна выходная переменная y. Мы считаем, что связь между ними описывается зависимостью следующего вида:

$$y = f(x_1, x_2) = p_0 + p_1 x_1 + p_2 x_2$$

Мы найдем коэффициенты p_0, p_1, p_2, используя алгоритм градиентного спуска. Этот алгоритм позволяет найти экстремум функции, поэтому нам для начала надо определить функцию, которую мы хотим минимизировать.

В качестве меры отклонения данных, описываемых нашей моделью, от самой модели можно выразить с помощью функции суммы квадратов ошибки:

$$Q = \sum_{i=1}^{N} \left(y_i - f\left(x_1^i, x_2^i\right) \right)^2 = \sum_{i=1}^{N} (e_i)^2$$

В нашей реализации данного алгоритма мы будем использовать матричную форму записи данной формулы. Если E – вектор (матрица-столбец) значений e_i, т.е. вектор ошибки, то Q можно вычислить по следующей формуле:

$$Q = E^T E$$

Теперь рассмотрим из каких шагов состоит наш алгоритм градиентного спуска.

1) выбираем начальные значения параметров p_0, p_1, p_2. Существуют разные подходы к выбору этих значений. Целевая функция может иметь разную форму, и при поиске глобального оптимума функции мы может попадать в локальные. Если начинать поиск из одного и того же места, то результат поиска будет сильно зависеть от выбора начала. Будет рациональным выбирать эти значения случайно из какого-то диапазона чисел. Таким образом, при нескольких запусках алгоритма мы получим разное значение ошибки и сможем выбрать из полученных результатов по-настоящему оптимальный.

2) находим значения градиента функции на текущем шаге. Вычисляем $\frac{\partial Q}{\partial p_i}$ для каждого параметра. Частная производная в данном случае находится следующим образом:

$$\frac{\partial Q}{\partial p_j} = \frac{\partial\left(\sum_{i=1}^{N}\left(y_i - f\left(x_1^i, x_2^i\right)\right)^2\right)}{\partial p_j} = 2\sum_{i=1}^{N}\left(y_i - f\left(x_1^i, x_2^i\right)\right)\frac{\partial f\left(x_1^i, x_2^i\right)}{\partial p_j} =$$

$$= 2\sum_{i=1}^{N}\left(y_i - f\left(x_1^i, x_2^i\right)\right)\frac{\partial \sum_{j=0}^{2} p_j x_j^i}{\partial p_j} = 2\sum_{i=1}^{N}\left(y_i - f\left(x_1^i, x_2^i\right)\right)\cdot x_j^i$$

Где x_j^i – независимая переменная со множителем pj, x_0^i = 1 при любом i.

Представим эту формулу в матричном виде:

$$\frac{\partial Q}{\partial p_j} = 2X_j^T E$$

Если представить $\frac{\partial Q}{\partial p_0}$, $\frac{\partial Q}{\partial p_1}$ и $\frac{\partial Q}{\partial p_2}$ в виде вектора $\nabla Q = \left[\frac{\partial Q}{\partial p_0} \quad \frac{\partial Q}{\partial p_1} \quad \frac{\partial Q}{\partial p_2} \right]$, то итоговая формула будет выглядеть следующим образом:

$$\nabla Q = 2X^T E,$$

Где $X = \begin{bmatrix} 1 & x_0^1 & x_0^2 \\ 1 & x_1^1 & x_1^2 \\ 1 & x_2^1 & x_2^2 \\ \dots & \dots & \dots \\ 1 & x_n^1 & x_n^2 \end{bmatrix}$

3) посчитаем новое значение параметров p:

$$p_j = p_j - \lambda \cdot \frac{\partial Q}{\partial p_j}$$

Если отобразить значения параметров в виде вектора $P = \begin{bmatrix} p_0 & p_1 & p_2 \end{bmatrix}$, то предыдущую формулу можно переписать в виде:

$$P = P - \lambda \cdot \nabla Q$$

4) Рассчитаем новое значение f(x1,x2) по найденным значениям параметров.

5) Рассчитаем новое значение ошибки и целевой функции Q

6) Проверим выполнение условия выхода из цикла:

- введем условие достижения максимального количества циклов для предотвращения попадания в бесконечный цикл и переполнения переменных

- и условие достижения экстремума функции, т.е. $\Delta Q < \varepsilon$, где ε – допустимая погрешность.

Если условие не выполняется, возвращаемся к п.2. Если выполняется – переходим к п.7.

7) возвращаем найденные значения: вектор P, текущее значение Q, количество пройденных итераций i.

3. Постановка задачи

0) Этот шаг не обязательный, но рассмотрим способ генерации выборки, которая будет использована в дальнейшем в качестве условия задачи.

Искомая зависимость описывается уравнением вида:

$$y = f(x_1,x_2) = p_0 + p_1*x_1 + p_2*x_2$$

Для начала выберем случайным образом коэффициенты полинома p_0, p_1, p_2 в диапазоне от -10 до +10.

Импортируем необходимые модули. Будем использовать библиотеку numpy, модуль random для генерации выборки, scipy для расчета коэффициента корреляции, pandas для представления данных в виде таблицы.

```
import numpy as np
import numpy.random as rand
from scipy.stats.stats import pearsonr
import pandas as pd
```

Затем непосредственно получим значения коэффициентов.

```
p0 = rand.random() * 20 - 10
p1 = rand.random() * 20 - 10
p2 = rand.random() * 20 - 10
```

Получим случайную выборку значений входных переменных x_1 и x_2. Проверим, что входные переменные являются независимыми. Для этого рассчитаем коэффициент корреляции Пирсона. Конкретное значение нас не интересует, он должен быть достаточно близок к нулю.

```
X1 = rand.randint(-1000,1000,500)
X2 = rand.randint(-1000,1000,500)
print(pearsonr(x1,x2)[0])
```

Получим:

```
0.029452567671917593
```

Переменные x1 и x2 являются объектами класса <class 'numpy.ndarray'>, ими поддерживаются векторные и матричные операции, так что вектор Y мы можем получить с помощью такой операции:

```
Y = p0 + p1*X1 + p2*X2
```

Длина векторов X1 и Y должны совпадать:

```
>>> Y.size == X1.size
True
```

Полученные векторы X1, X2 и Y представим в виде словаря:

```
data = {'X1': X1,
        'X2': X2,
        'Y': Y}
```

Словарь преобразуем в DataFrame и запишем в csv-файл:

```
data = pd.DataFrame(data)
data.to_csv('data.csv')
```

В результате будет создан файл вида:

```
,X1,X2,Y
0,-466,-595,2815.927478464896
1,629,-483,812.8390326260446
2,487,691,-3176.3336633773456
```

3,-981,522,-423.5322318910014
4,920,431,-2880.6010912274687
5,-692,270,54.36561469240576
6,48,586,-2158.421669315061

Полученную выборку будем рассматривать в качестве данных для последующего анализа.

1) Имеется файл data.csv. Требуется его считать, извлечь из него данные и методом градиентного спуска найти такие коэффициенты полинома p_0, p_1, p_2, которые при подстановке в уравнение $f(x_1,x_2) = p_0 + p_1*x_1 + p_2*x_2$ давали наименьшее значение суммы квадратов ошибки.

Итак, в новом файл подключим необходимые библиотеки:

```
import numpy as np
import numpy.random as rand
import matplotlib.pyplot as plt
import pandas as pd
```

Загрузим файл data.csv:

```
data = pd.read_csv('data.csv')
```

Для того, чтобы извлечь из них векторы X1, X2 и Y в виде массивов (array) достаточно выполнить:

```
X1 = np.arrray(data['X1'])
X2 = np.array(data['X2'])
Y = np.array(data['Y'])
```

2) Поиск минимума функции методом градиентного спуска состоит из нескольких простых операций:

а) задать начальные значения аргументов функции

б) посчитать значение функции при текущих значениях аргументов

в) вычислить значение градиента функции на текущем шаге

г) изменить значение аргументов на величину –λ(δQ/δarg), где λ – постоянный множитель, коэффициент скорости сходимости функции.

Фактически, при малых λ скорость поиска экстремума слишком мала, а при больших λ алгоритм начинает расходиться, с каждым шагом ошибка только растет.

Например, если отложить по оси X значение коэффициента λ, а по оси Y количество итераций, ограниченное сверху 10000 итерациями, то получим следующий график:

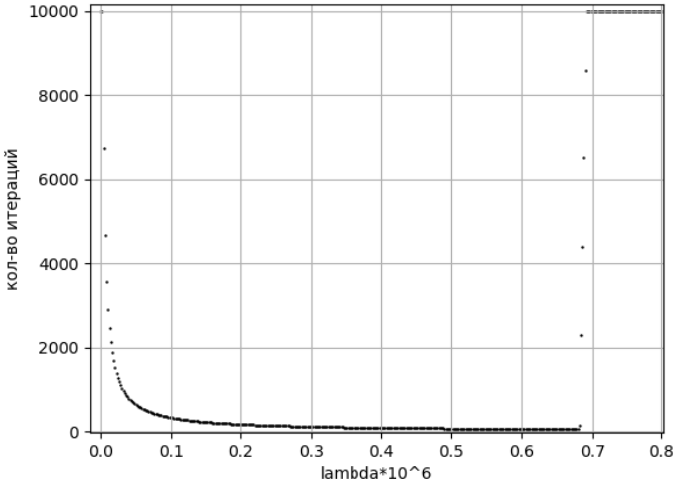

Рис. 9. График зависимости количества итераций алгоритма от значения постоянной скорости обучения λ

Зададим алгоритм в следующем виде:

```
def gradient_descent(X1, X2, Y, eps = 0.01, la = 0.0001,
max_iter = 1000):
    """алгоритм градиентного спуска"""
    #выбираем случайным образом нач. значения коэфф.
    p0, p1, p2 = tuple(rand.randint(-100,100,3))
    #X0 - единичный вектор того же размера, что и X1
    X0 = np.array([1]*X1.size)
```

```
#посчитаем значения функции при данных p0,p1,p2
F = X0*p0 + X1*p1 + X2*p2
#найдем вектор значений ошибки
e = Y - F
#посчитаем сумму квадратов ошибки
Q = e.dot(e.T)
#проведем n итераций для поиска p0,p1,p2
for i in range(max_iter):
    #здесь мы найдем градиент функции и
    #рассчитаем delta_p для всех p
    pass
return (p0, p1, p2), Q, i
```

3) Внутри цикла реализуем шаги в и г алгоритма, представленного выше:

а) считаем значение частной производной по каждой переменной $\delta Q/\delta arg$

б) изменяем текущее значение аргумента в соответствии с формулой $arg = arg - \lambda(\delta Q/\delta arg)$

в) рассчитываем новое значение функции F с новыми аргументами

г) рассчитываем новое значение вектор ошибки e

д) рассчитываем новое значение суммы квадратов ошибки Q_new.

е) выполняем проверку. Если $|Q\_new - Q| < eps$, то выходим из цикла, если нет, то обновляем значение Q_new = Q и начинаем новую итерацию.

4) Запустим алгоритм и проанализируем результаты. Пускай мы получили следующие результаты:

```
p0=73.99, p1=-1.46, p2=-3.57, Q=2256171.59, i = 336
```

где i – это количество итераций алгоритма.

Для начала посмотрим на график зависимости Q от номера итерации:

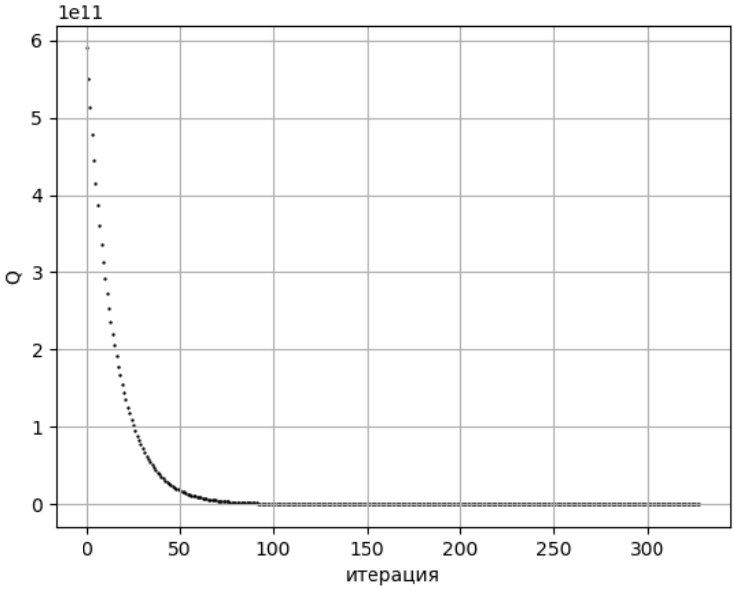

Рис. 10. График зависимости значения целевой функции от номера итерации.

До 100й итерации величина ошибки уменьшается экспоненциально, затем убывание происходит с линейной скоростью. Длина этого хвоста сильно зависит от заданной допустимой погрешности.

Получим график регрессии векторов Y и рассчитанного по найденным значениям коэффициентов вектора F.

```
F = p0 + X1*p1 + X2*p2
#p0, p1, p2 – это найденные нами коэффициенты
fig, ax = plt.subplots()
ax.plot(F,Y,'.',Y,Y,'-')
ax.grid()
ax.set_xlabel('F')
ax.set_ylabel('Y')
plt.show()
```

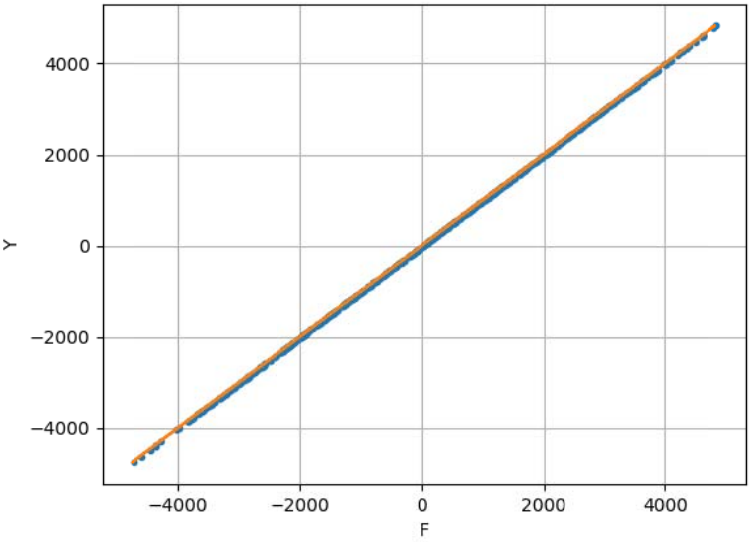

Рис. 11. График регрессии значений F и Y

По виду графика похоже, что функция достаточно точно описывает наши данные, хотя если его увеличить, то обнаружится существенный сдвиг относительно центра. Этот сдвиг конечно говорит о том, что мы не точно оценили параметр p_0.

Прежде чем корректировать наш алгоритм давайте посмотрим графики изменения других величин. Значение коэффициента p1 изменяется следующим образом:

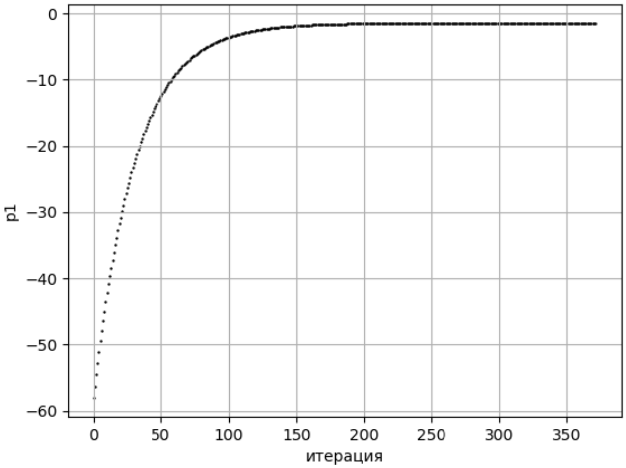

Рис. 12. График изменения коэффициента p_1

p_1 сходится к $p_{1(опт)}$, производная $\frac{\partial Q}{\partial p_1}$ сходится к нулю:

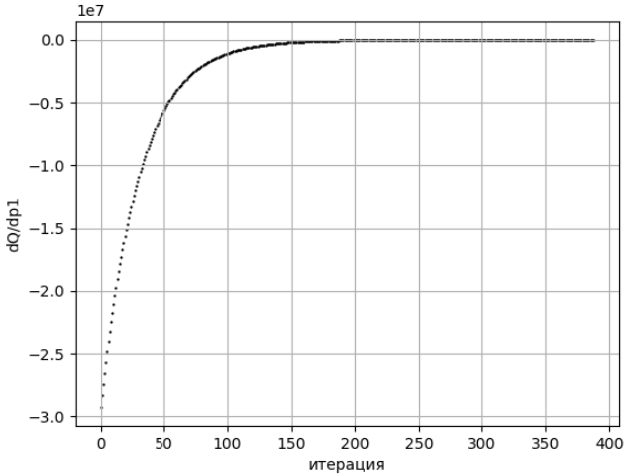

Рис. 13. График изменения $\frac{\partial Q}{\partial p_1}$

Ниже показан график для p_2

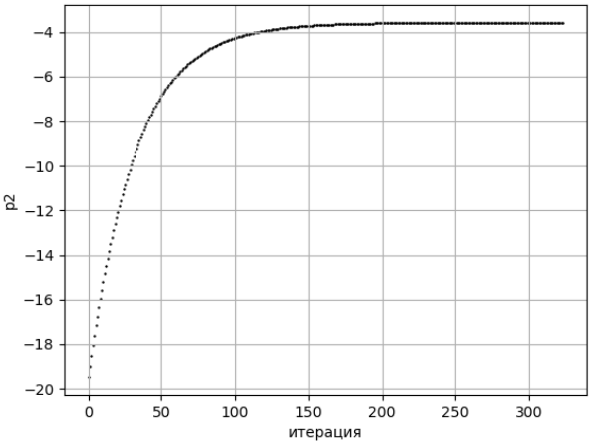

Рис. 14. График изменения коэффициента p_2

И график производной $\frac{\partial Q}{\partial p_2}$:

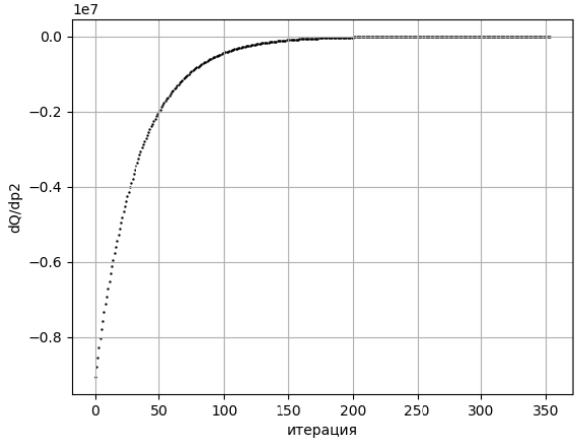

Рис. 15. График изменения $\frac{\partial Q}{\partial p_2}$

Ниже показан график для p_0

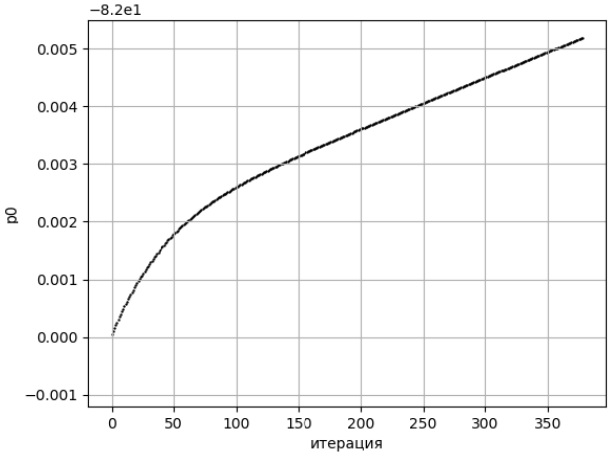

Рис. 16. График изменения коэффициента p_0

И график производной $\frac{\partial Q}{\partial p_0}$:

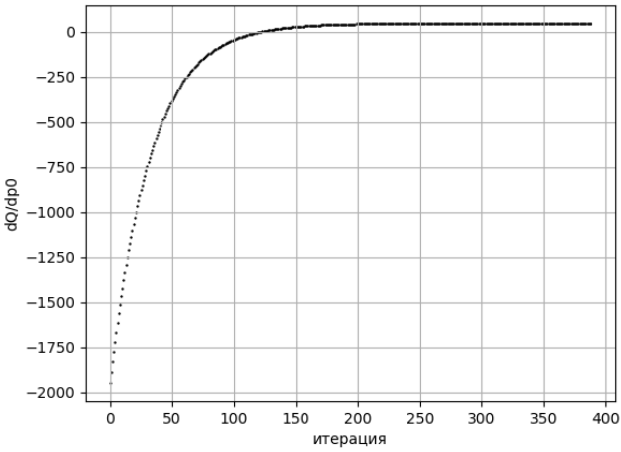

Рис. 17. График изменения $\frac{\partial Q}{\partial p_0}$

График на рис. 17 показывает, что значение сдвига p0 не устанавливается, а значение градиента $\frac{\partial Q}{\partial p_0}$ сходится к некоторому ненулевому значению.

5) Значение сдвига для нашей модели парной регрессии мы можем найти посчитав среднее значение разности $p_0 = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - p_1 x_1^i - p_2 x_2^i\right)$:

```
p0_ = np.mean(Y-(p1*X1 + p2*X2))
```

4. Содержание отчета:

- Название и цель работы
- Исходная таблица с данными
- Текст программы
- График регрессии данных модели и исходного вектора Y.
- График значения целевой функции Q от номера итерации.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Лутц Марк. Python. Карманный справочник. М.: Вильямс, 2015. — 320 с.

2. Лутц М. Изучаем Python. СПб.: Символ-Плюс, 2011. - 1280 с.

3. Python Notes for Professionals. GoalKicker.com, 2018. — 816 p. Python 3.x Version Release —3.7. Date — 2018-06-27

4. Томас Х. Кормен. Алгоритмы: вводный курс. М.: Вильямс, 2014. – 264 с.

СОДЕРЖАНИЕ