Национальный исследовательский университет ИТМО
Факультет информационных технологий и программирования
Прикладная математика и информатика

Методы оптимизации
Отчет по лабораторной работе №4

Работу выполнили:
Володько А.А., M32321
Ярунина К.А., M32371
Преподаватель:
Шохов М.Е.

Санкт-Петербург
2023

Постановка задачи:

1. Изучить использование вариантов SGD (torch.optim) из PyTorch. Исследовать эффективность и сравнить с собственными реализациями из 2 работы.

2. Изучить использование готовых методов оптимизации из SciPy (scipy.optimize.minimize, scipy.optimize.least_squares)

   (a) Исследовать эффективность и сравнить с собственными реализациями из 3 работы.

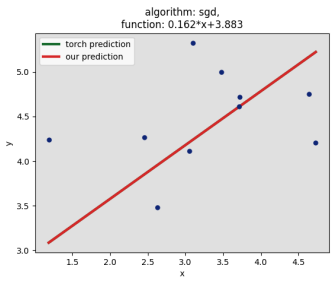
   (b) Реализовать использование PyTorch для вычисления градиента и сравнить с другими подходами.

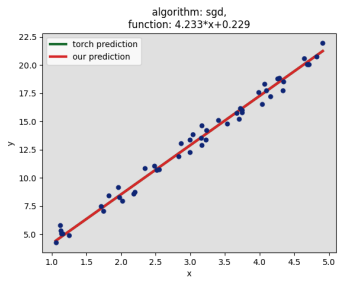
   (c) Исследовать как задание границ изменения параметров влияет на работу методов из SciPy.

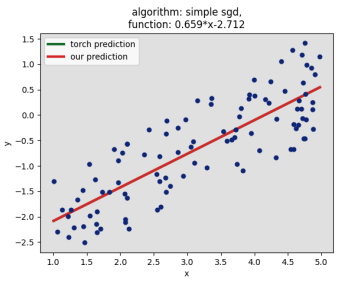
*. Исследовать использование линейных и нелинейных ограничений scipy.optimize.minimize из SciPy (scipy.optimize.LinearConstraint и scipy.optimize.NonlinearConstraint). Рассмотреть случаи когда минимум находится на границе заданной области и когда он расположен внутри.

1. Изучить использование вариантов SGD (torch.optim) из PyTorch. Исследовать эффективность и сравнить с собственными реализациями из 2 работы. рассмотрим на примере решения задачи нахождения линейной регрессии. здесь на графиках и таблицах приведены средние результаты из запусков 3 различных начальных точках (мы использовали (0, 0), (-4, 2) и (10, -10)) для линейной регрессии с 10, 50, и 100 данными точками и размерами батча 10.
   ниже представлена реализация алгоритма с использованием torch.optim

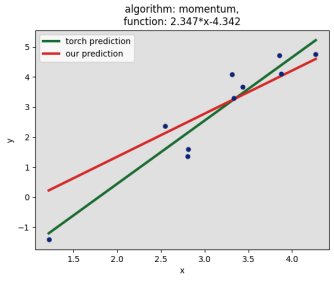
```python
def linear_torch(w, optimizer, X, y, batch_size, max_iter=1000):
    X_with_bias = np.concatenate((X, np.ones((X.size, 1))), axis=1)

    X_tensor = torch.from_numpy(X_with_bias).float()
    y_tensor = torch.from_numpy(y).float()

    num_samples = X.shape[0]
    num_batches = num_samples // batch_size

    for epoch in range(max_iter):
        for batch in range(num_batches):
            start = batch * batch_size
            end = (batch + 1) * batch_size

            X_batch = X_tensor[start:end]
            y_batch = y_tensor[start:end]

            predictions = torch.matmul(X_batch, w)
            loss = torch.mean((predictions - y_batch) ** 2)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
    return w
```

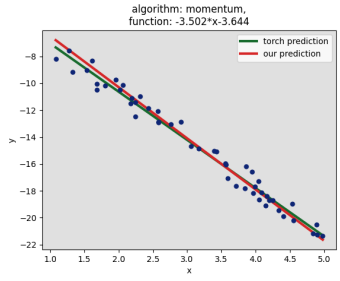


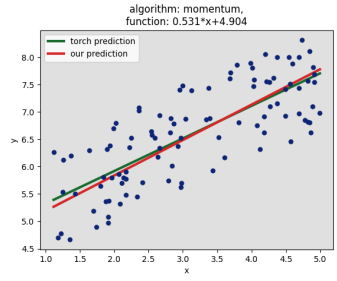




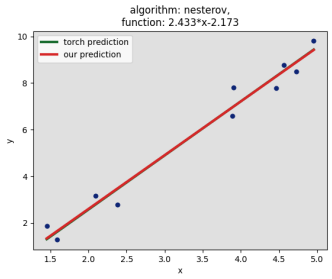
sgd				
f	avg torch time	avg our time	avg torch mem	avg our mem
f1	0.07869	0.20752	11829.33333	6485.33333
f2	0.0728	0.20428	7317.33333	2144.0
f3	0.07168	0.2036	7317.33333	2144.0

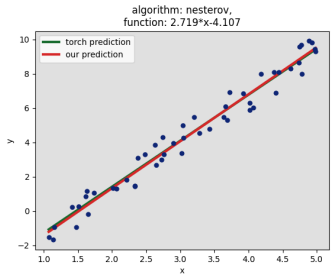


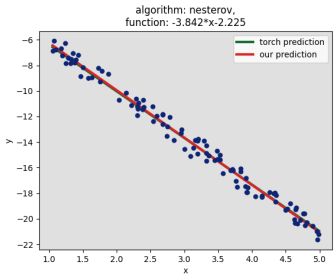




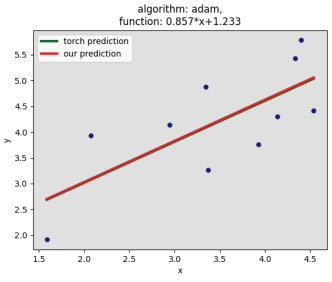
momentum				
f	avg torch time	avg our time	avg torch mem	avg our mem
f1	0.07631	0.43906	7434.66667	4117.33333
f2	0.07621	0.43612	7392.0	3072.0
f3	0.07631	0.43618	7392.0	3072.0

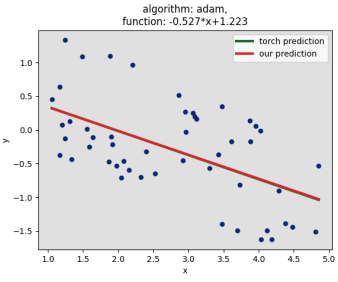


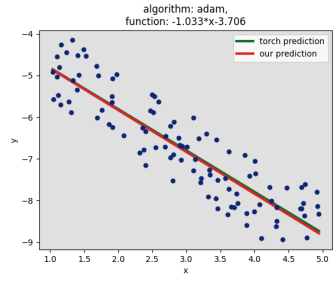




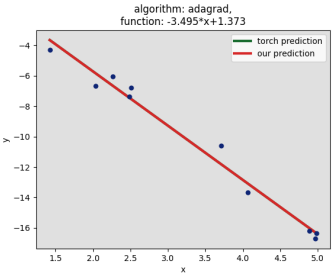
nesterov				
f	avg torch time	avg our time	avg torch mem	avg our mem
f1	0.07161	0.08328	7317.33333	3093.33333
f2	0.07136	0.11491	7317.33333	3093.33333
f3	0.07092	0.09377	7328.0	3072.0

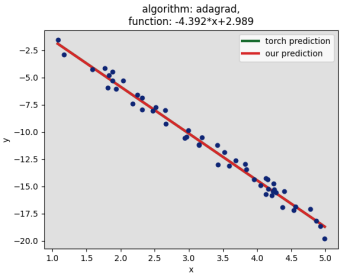


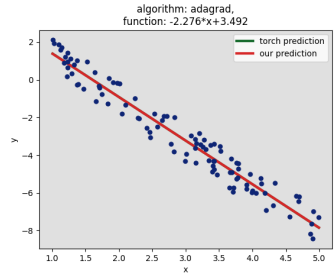




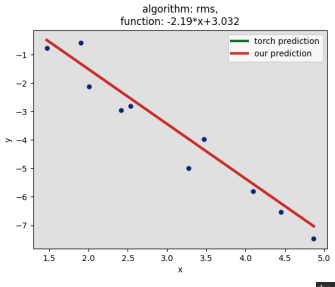
adam				
f	avg torch time	avg our time	avg torch mem	avg our mem
f1	0.09215	0.07879	8373.33333	3712.0
f2	0.09175	0.05201	7744.0	3690.66667
f3	0.09174	0.11256	7754.66667	3701.33333

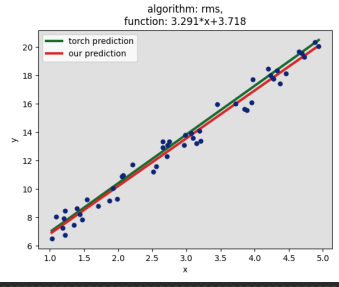


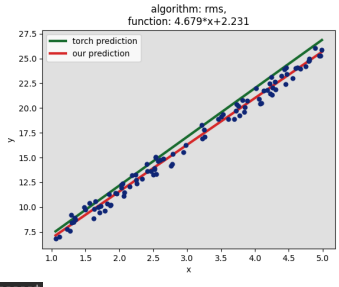




adagrad				
f	avg torch time	avg our time	avg torch mem	avg our mem
f1	0.08195	0.11256	8053.33333	3072.0
f2	0.08166	0.09653	7680.0	3050.66667
f3	0.082	0.07252	7680.0	3040.0







rms				
f	avg torch time	avg our time	avg torch mem	avg our mem
f1	0.08009	0.0568	7840.0	2858.66667
f2	0.07956	0.19984	7488.0	2762.66667
f3	0.07958	0.198	7488.0	2762.66667

по графикам и таблицам можно сделать следующие выводы: в целом, в большинстве своем приведенные алгоритмы в нашей реализации и в PyTorch выдают около одинаковые результаты на различных функциях. что заметно отличается – это показатели затраченных времени и памяти. по памяти наша реализация оказывается в 2-3 раза эффективнее, но по времени либо тратит столько же, либо больше. почему это так можно понять из официальной документации: алгоритмы pythorch имеют различные реализации, оптимизированные для производительности, читаемости и/или общности, и используется наиболее быстрая под конкретное устройство. в целом в алгоритмах pytorch есть 3 основных категории реализаций: for-loop, foreach (multi-tensor) и fused. for-looping обычно медленнее, чем foreach, которые объединяют параметры в мульти-тензор и выполняют большие блоки вычислений одновременно, тем самым экономя множество последовательных вызовов ядра. некоторые из оптимизаторов имеют еще более быстрые сливаемые реализации, которые объединяют большие блоки вычислений в одно ядро. мы пользовались обычным for-loop ом в наших алгоритмах, этим объясняется отставание по времени.

2. Изучить использование готовых методов оптимизации из SciPy (scipy.optimize.minimize, scipy.optimize.least_squares)

   (a) Исследовать эффективность и сравнить с собственными реализациями из 3 работы.
   мы рассматривали методы dogleg, BFGS и L-BFGS-B из scipy.optimize.minimize в сравнении с нашими реализациями из предыдущей лабороторной. для нижеприведенных функцих здесь показааны средние значения из нескольких начальных точек

```python
# f(x, y) = w[0] * x ** 2 + w[1] * y ** 2
def func(w, X):
    return w[0] * X[:, 0] ** 2 + w[1] * X[:, 1] ** 2

def loss_function(x, data):
    y_predicted = func(x, data)
    residuals = data[:, -1] - y_predicted
    return np.sum(residuals ** 2)

def jacobian(w, data):
    return approx_fprime(w, loss_function, 1e-8, data)

def hessian(w, data):
    return Hessian(loss_function)(w, data)  # imported from numdifftools

w_start = np.ones(3)
num_features = 2
data = generate_data(func, num_features)

result_bfgs = minimize(loss_function, w_start, args=(data,), method='BFGS')
result_lbgs = minimize(loss_function, w_start, args=(data,), method='L-BFGS-B')
result_dogleg = least_squares(loss_function, w_start, args=(data,), method='dogbox', jac=jacobian)
```

```
f1: f(x, y, z, t) = w0 * x ** 2 + w1 * y ** 3 + w2 * 10 * sin(z) + w3 * t ** 2
f2: f(x, y, z, t, u) = w0 * x ** 2 + w1 * y ** 3 + w2 * 10 * sin(z) + w3 * t ** 2 + w4 * u
f3: f(x, y, z, t, u, v) = w0 * x ** 2 + w1 * sin(y) + w2 * sqrt(z) ** 3 + w3 * t + w4 * cos(u) + w5 * v + w6
+-----------------------------------------------------------------------------------------------------------------------+
|                                                BFGS algorithm result                                                  |
+-----------------+-------------------------+-----------------------------------+-----------------------------------------+
|        f        |           f1            |                f2                 |                   f3                    |
+-----------------+-------------------------+-----------------------------------+-----------------------------------------+
|     true res    |  (-2.663  0.764 -4.982) |  (-3.223  1.220  3.227 -3.977  1.335) |  (1.280  2.418  2.635  4.545 -3.686 -0.606)  |
|     our res     | (-2.625,  0.765, -5.114) | (-3.584,  1.221,  3.248, -3.628, 1.274) | (1.839,  3.556, 2.757, 3.935,  -3.669, -1.629) |
|    scipy res    |  (-2.679  0.763 -4.777) |  (-3.223  1.220  3.240 -3.976  1.257) |  (1.280  2.337  2.631  4.534 -3.778 -0.406)  |
|  our deviation  |          0.018          |               0.255               |                  3.04                   |
| scipy deviation |          0.042          |               0.006               |                  0.05                   |
|     our time    |          0.527          |               1.634               |                 2.226                   |
|    scipy time   |          0.015          |               0.031               |                 0.031                   |
+-----------------+-------------------------+-----------------------------------+-----------------------------------------+
+-----------------------------------------------------------------------------------------------------------------------+
|                                               L-BFGS algorithm result                                                 |
+-----------------+-------------------------+-----------------------------------+-----------------------------------------+
|        f        |           f1            |                f2                 |                   f3                    |
+-----------------+-------------------------+-----------------------------------+-----------------------------------------+
|     true res    |  (0.598  3.303 -1.407)  |  (-2.279  3.278 -2.558 -1.220  0.022)  |  (-0.831 -3.198 -1.287  4.917  3.358  3.287)  |
|     our res     | (0.650,  3.298, -1.434) | (-1.771,  3.280, -2.539, -1.771,  0.811) | (-0.753, -3.842, -1.763,  5.659,  3.392, 3.085) |
|    scipy res    |  (0.584  3.303 -1.163)  |  (-2.279  3.278 -2.572 -1.221  0.137)  |  (-0.830 -3.175 -1.287  4.900  3.379  3.388)  |
|  our deviation  |          0.003          |               1.18                |                  1.24                   |
| scipy deviation |           0.05          |               0.01                |                  0.01                   |
|     our time    |          0.485          |              1.667                |                 6.149                   |
|    scipy time   |          0.015          |              0.015                |                 0.029                   |
+-----------------+-------------------------+-----------------------------------+-----------------------------------------+
```

```
f1: f(x, y) = w0 * x ** 2 + w1 * y ** 2
f2: f(x, y, z, t) = w0 * x ** 2 + w1 * y ** 3 + w2 * 10 * sin(z) + w3 * t ** 2
f3: f(x, y, z, t, u) = w0 * x ** 2 + w1 * y + w2 * sqrt(z) ** 3 + w3 * t  ** 2 + w4 * u ** 2
```

DogLeg algorithm result

f	f1	f2	f3
true res	(1.894 2.129)	(3.903 4.602 1.055 -2.632)	(-4.000 -2.803 0.814 4.675)
our res	(1.679, 2.149)	(0.625, 4.603, 1.046, 0.625)	(-3.557, -2.600, 0.695, 2.557)
scipy res	(1.516 2.159)	(3.386 4.601 0.031 -2.102)	(-4.022 -1.582 0.677 4.655)
our deviation	0.143	21.3	4.73
scipy deviation	0.04	1.59	1.5
our time	0.01	0.181	0.015
scipy time	0.995	3.468	3.796

по таблицам выше видим, что наша реализация в основном имеет худшие показатели затраченных времени и памяти, опять же, библиотечные методы постарались оптимизировать. особенно это заметно в результатах алгоритма BFGS, он является дефолтным методом scipy.optimize.minimize и очевидно должен показывать отличные показатели. Однако стоит отметить, что метод Gauss-Newton в нашей реализации требует меньше итераций, чем в библиотеке. так как в библиотеке присутствует только реализация L-BFGS-B, мы использовали ее, но сам L-BFGS-B является более оптимальным, чем стандартный L-BFGS, поэтому наша реализация уступает в общей производительности.

(b) Реализовать использование PyTorch для вычисления градиента и сравнить с другими подходами.
рассмотрим три подхода: `torch`, `numdifftools.Gradient` и численное дифференцирование следующего вида

```python
def get_torch_grad(f, dot):
    x = torch.tensor(dot, requires_grad=True)
    y = f(x)
    y.backward()
    return x.grad.item()

def get_numdifftools_grad(f, dot):
    return nd.Gradient(f)(dot)

def get_numerical_grad(f, dot, eps=1e-5):
    return (f(dot+eps)-f(dot-eps)) / (2*eps)

# рассмотренные функции, ищем производную в точке x=1
def f1(xx):
    return 4*xx**4 - 7*xx

def f2(xx):
    return (torch.exp(-xx**2 + 4) if torch.is_tensor(xx) else math.exp(-xx**2 + 4)) - 7*xx**2 + 4

def f3(xx):
    return (0.7*xx - 1.4)**2 - (0.2*xx - 0.1) ** 4
```

method	f	result	deviation	time	mem
wolfram	f1	9	0		
	f2	-54.171073846375336	0		
	f3	-0.9808	0		
torch	f1	9.0	0	0.01265216	2160
	f2	-54.17107391357422	4.51569e-15	0.00530505	1520
	f3	-0.9807999730110168	7.284052e-16	0.00530601	1488
nd	f1	8.999999999999998	3.155444e-30	0.09933591	109248
	f2	-54.17107384637556	5.169879e-26	0.10430193	29600
	f3	-0.980799999996005	1.596024e-23	0.10428905	61472
numerical	f1	9.000000001568877	2.461374e-18	0.0001092	976
	f2	-54.17107384531036	1.134173e-18	0.00019407	880
	f3	-0.9808000000027777	7.715434e-24	0.00019217	880

на табличке ниже представлены средние значения из 100 запусков каждого метода, а также суммарное время работы и затраченной памяти. мы видим, что в случае относительно несложных функций все методы справляются с подсчетом градиента и выдают схожие результаты. численное дифференцирование слегка проигрывает по точности, однако это по необходимости исправляется изменением параметра `eps`. зато памяти он затрачивает меньше всего. torch в целом выгляжит наиболее оптимальным в плане отношения точности к времени и памяти, так как все еще затрачивает в разы меньше памяти чем numdifftools, но при этом выдает неплохо приближенные к реальным результаты.

(c) Исследовать как задание границ изменения параметров влияет на работу методов из SciPy.
посмотрим на методы trf и dogbox, наиболее напоминающие методы, которые мы рассматривали в предыдущих работах. здесь мы задавали ограничения на модуль каждой координаты в некотором диапазоне вида $[-a, a]$ для $a \in \{2, 6, 10\}$ и тестировали на нескольких стандартных задачах нахождения линейной регрессии.

bounds	method	param	f1	f2	f3
[-2,2]	trf	loss	32.224486	2474.949471	1137.12501
		time	0.025003	0.158615	0.125714
		mem	28480	19936	17120
	dogbox	loss	31.611832	2687.706782	8267.976095
		time	0.025676	0.157206	0.165088
		mem	26272	25184	24096
[-6,6]	trf	loss	29.806259	1012.860972	6746.143051
		time	0.034143	0.11623	0.123151
		mem	15040	17344	16224
	dogbox	loss	30.176816	32265.626554	1828.467428
		time	0.033181	0.155815	0.171388
		mem	16320	24224	16480
[-10,10]	trf	loss	32.639507	10207.243895	13071.086975
		time	0.025655	0.118503	0.130729
		mem	16064	16224	16192
	dogbox	loss	32.391989	11218.060826	2270.027171
		time	0.043078	0.152761	0.168541
		mem	16320	16480	16480

по таблице можно сделать следующие выводы: в целом, более конкретные ограничения позволяют приблизиться к минимуму намного лучше: это видно на f_1 и f_2: первый диапазон не захватывал минимум, из-за чего значение loss функции чуть больше, второй позволил минимум найти, а третий оказался несколько большим, из-за чего возникли небольшие проблемы с точностью. однако мы также видим, что в случаях, когда значение loss функции мало, мы видим увеличение параметров затраченных времени и памяти. (в целом, алгоритм дошел дальше, поэтому и итераций -> времени/памяти потратил больше). более подробно влияние ограничений рассмотрим в следующем пункте.

*. Исследовать использование линейных и нелинейных ограничений scipy.optimize.minimize из SciPy (scipy.optimize.LinearConstraint и scipy.optimize.NonlinearConstraint). Рассмотреть случаи когда минимум находится на границе заданной области и когда он расположен внутри.

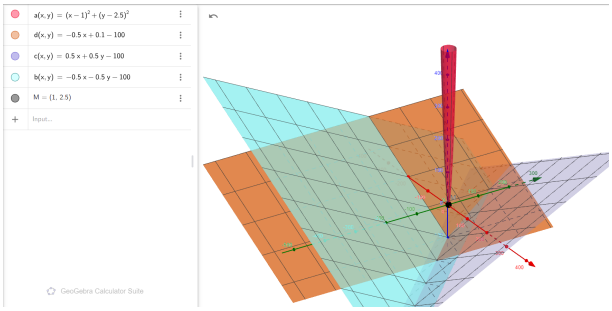
исследование сходимости градиентного спуска с линейными и нелинейными ограничениями проводилось на 2 функциях:

1. $f_1(x) = (x-1)^2 + (y-2.5)^2$, argmin $= (1, 2.5)$
2. $f_2(x) = 45x^2 + 42xy + 5y^2$, argmin $= (0, 0)$

рассмотрим результаты исследований на 1 функции.

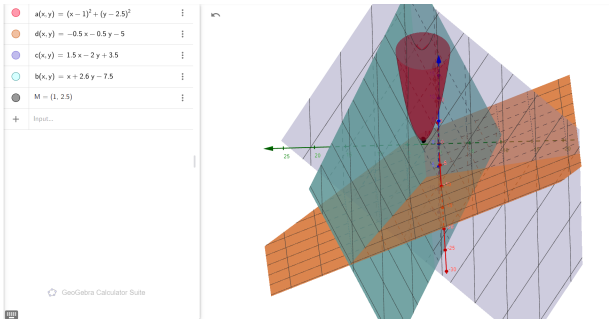
1. В качестве линейных ограничений, в случае, когда минимум не находится на границе заданной области были взяты следующие:

$$\begin{cases} -0.5x + 0.1 - 100 \leq 0 \\ 0.5x + 0.5y - 100 \leq 0 \\ -0.5x - 0.5y - 100 \leq 0 \end{cases}$$



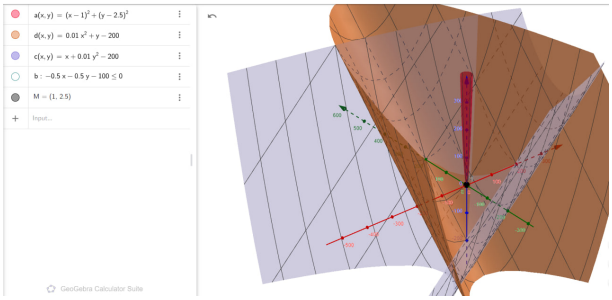
2. В качестве линейных ограничений, в случае, когда минимум находится на границе заданной области были взяты следующие:

$$\begin{cases} -0.5x - 0.5y - 5 \leq 0 \\ 1.5x - 2y + 3.5 \leq 0 \\ x + 2.6y - 7.5 \leq 0 \end{cases}$$



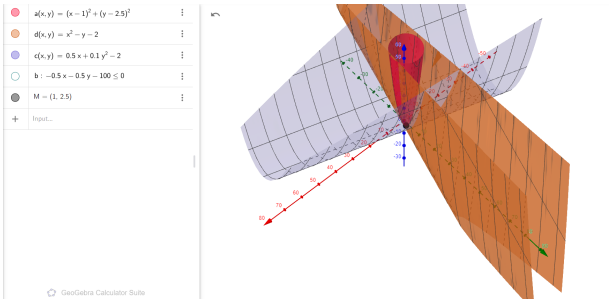
3. В качестве нелинейных ограничений, в случае, когда минимум не находится на границе заданной области были взяты следующие:

$$\begin{cases} 0.01x^2 + y - 200 \geq 0 \\ x + 0.01y^2 - 200 \geq 0 \end{cases}$$



4. В качестве нелинейных ограничений, в случае, когда минимум находится на границе заданной области были взяты следующие:

$$\begin{cases} x^2 - y - 2 \geq 0 \\ 0.5x + 0.1y^2 - 2 \geq 0 \end{cases}$$

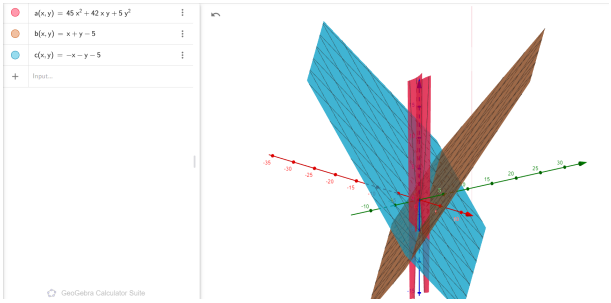


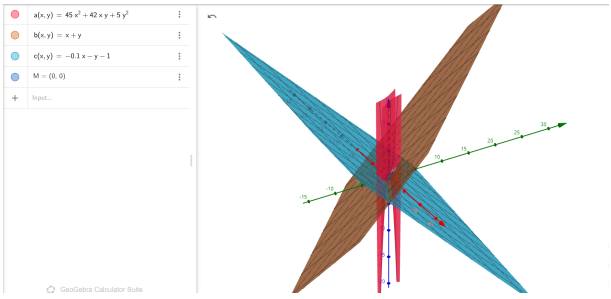
Рассмотрим результаты работы алгоритма без ограничений, а также с описанными выше ограничениями:

constraint	result	time, s	mem, kB
without constraints	[1. 2.5]	0.01563	34752
weak linear constraints	[0.99996 2.5]	0.0	28032
strong linear constraints	[1. 2.5]	0.0	24320
weak not linear constraints	[1. 2.49999]	0.03126	19712
strong not linear constraints	[0.99986 2.50016]	0.0368	18976

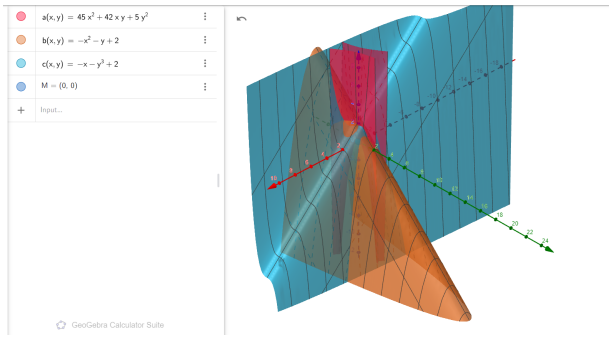
Как нетрудно заметить по результатам в таблице, во всех случаях минимум был найден корректно, что и следовало ожидать, поскольку функция простая и имеет маленькое число обусловленности. Более интересно обратить внимание на результаты измерений времени работы алгоритмов. Наиболее быстро справились алгоритмы с линейными ограничениями, более чем за 10^{-5} секунд, чуть дольше работал алгоритм без ограничений и наибольшее время работы у алгоритмов с нелинейными ограничениями, что объясняется сложностью проверки нелийных ограничений и их вычисления. Стоит также отметить, что в силу простоты функции для получения различных результатов раюоты алгоритмов за начальную точку была взята точка $(200000, 100)$. В случае выбора начальной точки с более близкими к искомому минимуму координатами алгоритмы работали одинаково хорошо и одинаково быстро, это не представляет такого интереса для анализа. Что касается результатов использованной памяти, здесь больше всего затрачено у алгоритмов без ограничений. Этого следовало ожидать, поскольку область вычислений не ограничена, а значения функции на первых шагах алгоритма значительно больше.

Перейдем к результатам исследования работы алгоритмов на 2 функции.
Стоит отметить, что данная функция была выбрана для исследований, поскольку имеет очень большое число обусловленности - около 100, что может значительно затруднить поиск минимума для алгоритмов.
Снова приведем ограничения, которые использовались для исследования.

1. В качестве линейных ограничений, в случае, когда минимум не находится на границе заданной области были взяты следующие:
$$\begin{cases} x + y - 5 \leq 0 \\ -x - y - 5 \leq 0 \end{cases}$$


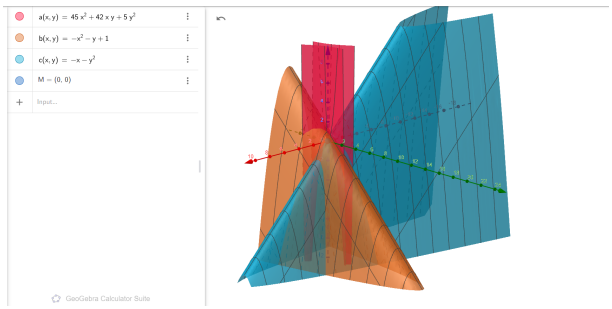
2. В качестве линейных ограничений, в случае, когда минимум находится на границе заданной области были взяты следующие:
$$\begin{cases} x + y \leq 0 \\ -0.1x - y - 1 \leq 0 \end{cases}$$


3. В качестве нелинейных ограничений, в случае, когда минимум не находится на границе заданной области были взяты следующие:
$$\begin{cases} -x^2 - y + 2 \geq 0 \\ -x - y^3 + 2 \geq 0 \end{cases}$$



4. В качестве нелинейных ограничений, в случае, когда минимум находится на границе заданной области были взяты следующие:

$$\begin{cases} -x^2 - y + 1 \geq 0 \\ x - y^2 \geq 0 \end{cases}$$



Аналогично, рассмотрим результаты работы алгоритма без ограничений, а также с описанными выше ограничениями:

constraint	result	time, s	mem, kB
without constraints	[-183.18361 424.85063]	0.03778	40544
weak linear constraints	[-10.00001 15.00002]	0.01563	27808
strong linear constraints	[-0. 0.]	0.01563	24448
weak not linear constraints	[-0.67573 1.38829]	0.01563	19584
strong not linear constraints	[0.2129 -0.53582]	0.01563	18976

Результаты исследования результатов на данной функции значительно интереснее. В силу большого числа обусловленности в качестве точки старта была взята близкая к минимуму точка $(1, 1)$. При взятии точек, сильно отдаленных от точки минимума результаты были слишком плачевны. Для начала рассмотрим значение найденной точки минимума. Как видно из результатов, алгоритм без ограничений разошелся в силу большого числа обусловленности. Чуть лучше показал себя алгоритм с линейными слабыми ограничениями, достигнув результата $(-10, 15)$. Отдаленность найденной точки от реального минимума аналогично объясняется большим числом обусловленности выбранной функции, а так же достаточно большая область ограничений. Результаты работы нелинейных ограничений уже достаточно близки к правде. Как и следовало ожидать, чуть лучше результаты при ограничениях, пересекающих точку минимума. Лучший результат показал алгоритм с линейными ограничениями, пересекающими точку минимума, который смог точно найти его. Что касается, времени работы, алгоритм без ограничей работал дольше всех, затратив при это боьлше всего памяти.
Подводя итоги исследований сходимости алгоритмов с различными ограничениями, можно отметить, что ограничения могут вносить значительный вклад в случаях со сложными функциями. Также в среднем в случаях, когда минимум функции находился на границе ограниченной области результаты были лучше.
Ниже приведен код для исследований для первой функции. Реализация алгоритма для второй функции аналогична по модулю описанных ограничений.

```
def get_first_function_data():
    fun = lambda x: (x[0] - 1) ** 2 + (x[1] - 2.5) ** 2
    x_start_1 = np.array([200000, 100])
```

```
    A1_weak = np.array([[-0.5, 0.1],
                        [0.5, 0.5],
                        [-0.5, -0.5]])
    A1_strong = np.array([[-0.5, -0.5],
                          [1.5, -2],
                          [1, 2.6]])
    b1_weak = np.array([100, 100, 100])
    b1_strong = np.array([5, -3.5, 7.5])

    def strong_not_linear_first(X):
        return +X[0] ** 2 - X[1] + 2

    def strong_not_linear_second(X):
        return 0.5 * X[0] + 0.1 * X[1] ** 2 + 2

    def weak_not_linear_first(X):
        return 0.01 * X[0] ** 2 + X[1] + 200

    def weak_not_linear_second(X):
        return X[0] + 0.01 * X[1] ** 2 + 200

    weak_linear_constraints = LinearConstraint(A1_weak, -np.inf, b1_weak)
    strong_linear_constraints = LinearConstraint(A1_strong, -np.inf, b1_strong)

    weak_not_linear = [
        {'type': 'ineq', 'fun': weak_not_linear_first},
        {'type': 'ineq', 'fun': weak_not_linear_second}
    ]

    strong_not_linear = [
        {'type': 'ineq', 'fun': strong_not_linear_first},
        {'type': 'ineq', 'fun': strong_not_linear_second}
    ]

    return fun, x_start_1, weak_linear_constraints, strong_linear_constraints, weak_not_linear, strong_not_linear

def run_method(fun, x0, constraints=None):
    tracemalloc.start()
    a_time = time.time()
    if constraints:
        result = minimize(fun, x0, constraints=constraints)
    else:
        result = minimize(fun, x0)
    delta_time = time.time() - a_time
    mem = tracemalloc.get_tracemalloc_memory()
    tracemalloc.stop()
    return np.round(result.x, 5), mem, round(delta_time, 5)
```