

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

# Методы оптимизации

Отчет по лабораторной работе №4

⟨Собрано 28 июня 2023 г.⟩

**Работу выполнили:**

Бактурин Савелий Филиппович М32331

Вереня Андрей Тарасович М32331

Сотников Максим Владимирович М32331

**Преподаватель:**

Казанков Владислав Константинович

## Стохастический градиентный спуск vs. `torch.optim`

Настало время сравнить самописные реализации обычного и других модифицированных версий стохастического градиентного спуска с написанным серьезным дядьками из PyTorch. В качестве экспериментирования мы возьмем старую добрую функцию Розенброка, которая на плоскости  $\mathbb{R}^2$  определяется следующим образом:

$$f(x, w_0, w_1) = (1 - x \cdot x_0)^2 + 100 \cdot (x \cdot w_1 - x^2)^2,$$

где  $w_0$  и  $w_1$  – некие конечные константы. Теперь установим некие параметры для будущих тестов, а именно для генерации набора данных (*dataset*):

```
density = 8000 (плотность генерируемых точек),
dots_count = 1000 (количество генерируемых точек),
radius = 0.001 (радиус от функции генерации точек),
dist = 0.01 (расстояние относительно прямой генерации точек)
```

Также зададим постоянными параметрами двух конечных констант для функции Розенброка:

$$w_0 = 2 \quad w_1 = 3$$

Далее количество тестов на усреднение результатов – данный параметр говорит, сколько необходимо сделать независимых генераций и запусков алгоритмов с начальной точки нужно сделать, затем усреднить:

$$\text{test\_count} = 100$$

Наконец, зададим начальную точку. Поскольку наша задача состоит в исследовании работы алгоритмов в условиях генерируемых новых точек, умоляя тех или иных дополнительных эффектов от `numpy`-генерации, то мы будем брать в качестве начала одну и ту же точку:

$$\text{initial\_w} = \langle -1.0, 0.1 \rangle$$

Перед тем как мы обратимся непосредственно к самим результатам исследований скажем несколько слов по поводу реализуемых и используемых функций. По сути, `torch.optim` это такой конструктор, который необходимо собрать, чтобы получить подходящую под задачу модель. В частности реализация использует в качестве подсчета на `dataset` функцию `helper.train_torch_optim`:

```
function train_torch_optim( $f : \mathbb{R}^{k+1} \rightarrow \mathbb{R}$ ,
     $X_{\text{train}} \subset \mathbb{R}$ ,
     $Y_{\text{train}} \subset \mathbb{R}$ ,
     $\varepsilon_{\text{minimum}} \in \mathbb{R}$ ,
     $\langle x_0, y_0 \rangle \in \mathbb{R}^2$ ,
     $g_{\text{loss}} : \mathbb{R}^{k+1} \rightarrow \mathbb{R}$ ,
    optimizer_method : /*implementation defined*/,
    num_epochs (= 10000)  $\in \mathbb{N}$ ,
    learning_rate (= 0.01)  $\in \mathbb{N}$ ,
    momentum (=  $\emptyset$ )  $\in \mathbb{R}$ ,
    nesterov (= False)  $\in \mathbb{V}$ ,
    apply_min (= True)  $\in \mathbb{V}$ )
```

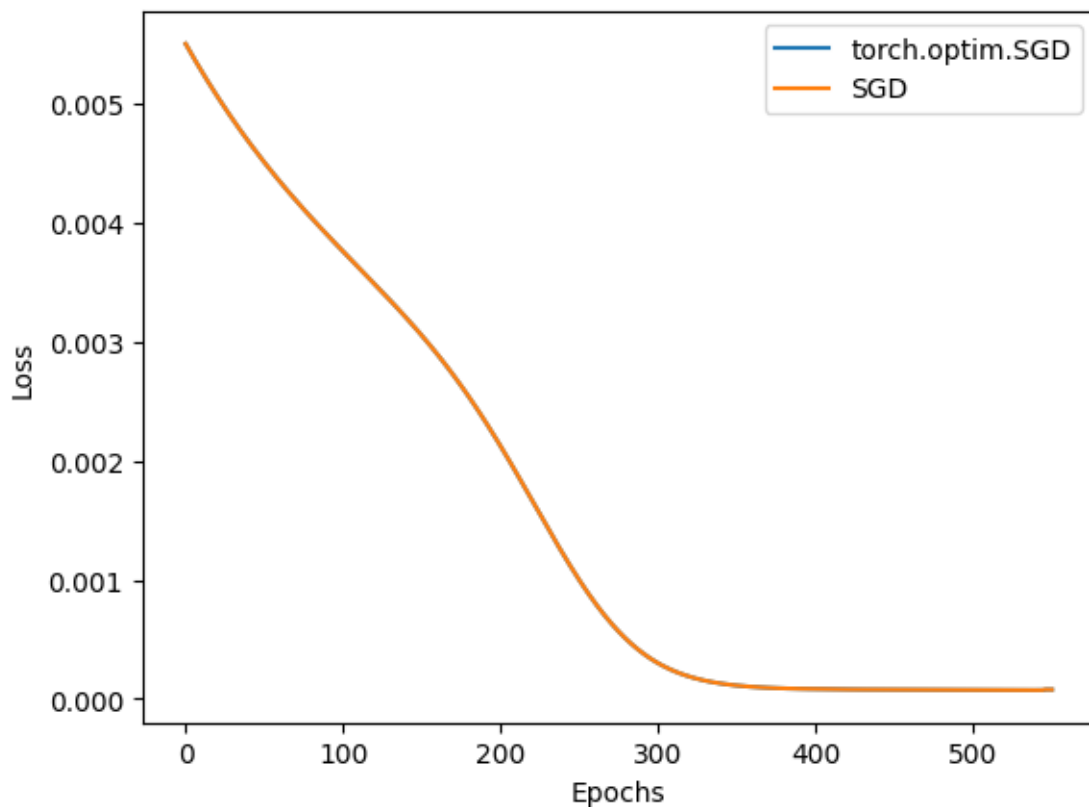
По сути, она выстраивает модель оптимизатора, определенный в библиотеке PyTorch, от `optimizer_method`, который подается как дождь - с небес, и далее по количеству, ограниченных по умолчанию в 10000 шагов, начинает обновлять параметры модели (от исходной точки) от композиции обнуления градиенты, вычисления функции потерь, проверки на сходимость и, соответственно, шага. Наконец, определим еще одну, усредняющую функцию, для вывода результатов – `get_average_loss_history` – она вычисляет среднее значение функции потерь `helper.mse_loss`, определяемая как

```
1 function mse_loss(X_dataset, Y_dataset, w, f):
2      $y' \leftarrow f(X_{\text{dataset}}, w)$ ;
3     return torch.mean((Y_dataset - y')2);
4
```

То есть, такая функция, которая вычисляет среднеквадратичную ошибку между предсказанными и реальными значениями на основе модели.

## SGD vs. torch.optim.SGD

Перейдем к исследованиям. Первым на подходе мы посмотрим на стандартный стохастический градиентный спуск (с размером батча, равный единице). Запустим наших зверей и посмотрим на вывод в виде графика, по оси Y которой мы выставим значения функции потерь, по оси X - количество шагов до остановки алгоритма.



Implemented SGD vs. PyTorch's

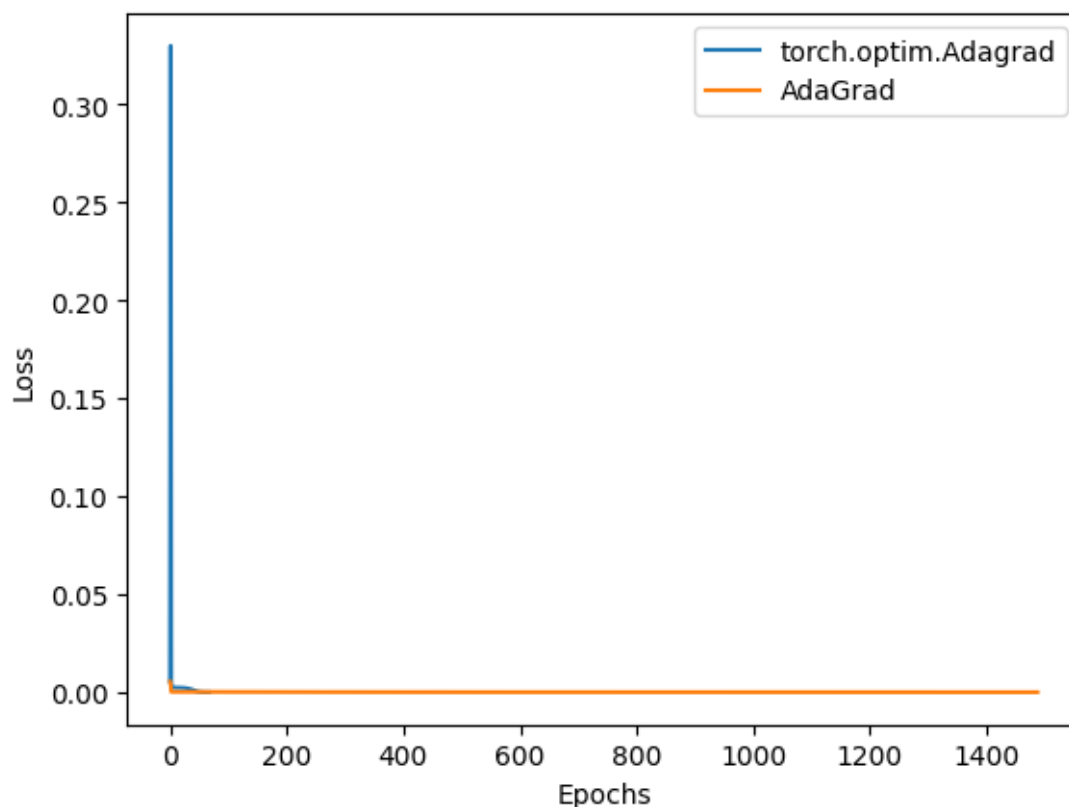
Графики почти полностью совпали. Полученные результаты соответствующие:

Испытуемый	Значение функции потерь	Среднее число шагов до схождения
Настоящие данные	0.00007481586253817499	-
<code>torch.optim.SGD</code>	0.00007753667274059714	551
Собственная реализация	0.00007756952443600056	551

Как мы видим, значения, особенно количество шагов до схождения алгоритма, почти полностью совпадают. Есть теория, что стохастический градиентный спуск был сделан правильно и, как сказал один из коллег по лабораторной работе, «по другому он не может быть сделан». По более формальной теории, поскольку dataset используется один и тот же, то есть вероятность, что точки были так удачно распределены даже спустя 100 тестов, что любая выбранная из батч размера нивелирует среднее значение.

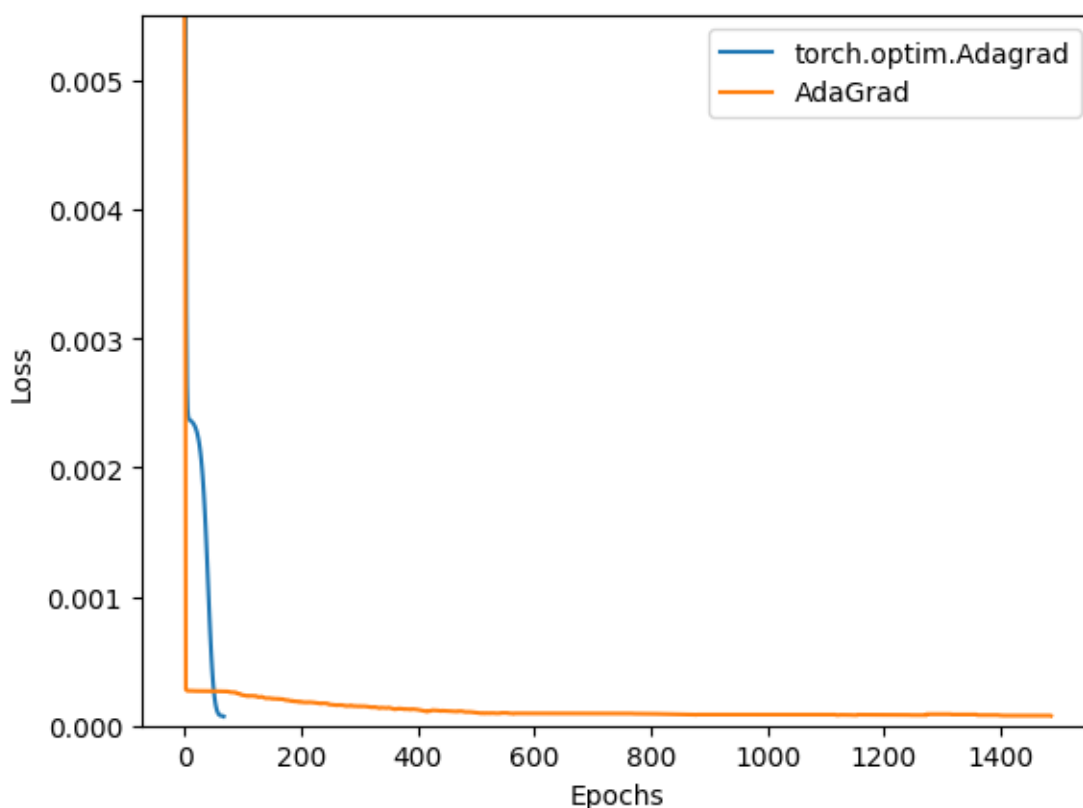
## AdaGrad vs. `torch.optim.AdaGrad`

С AdaGrad'ом ситуации поинтереснее. Запустим наших зверей и посмотрим на вывод в виде графика, по оси Y которой мы выставим значения функции потерь, по оси X - количество шагов до остановки алгоритма.



Implemented AdaGrad vs. PyTorch's

Также, для наглядности, немного приблизимся к почти незаметной горке, ограничивая по оси Y.



Implemented AdaGrad vs. PyTorch's (zoom)

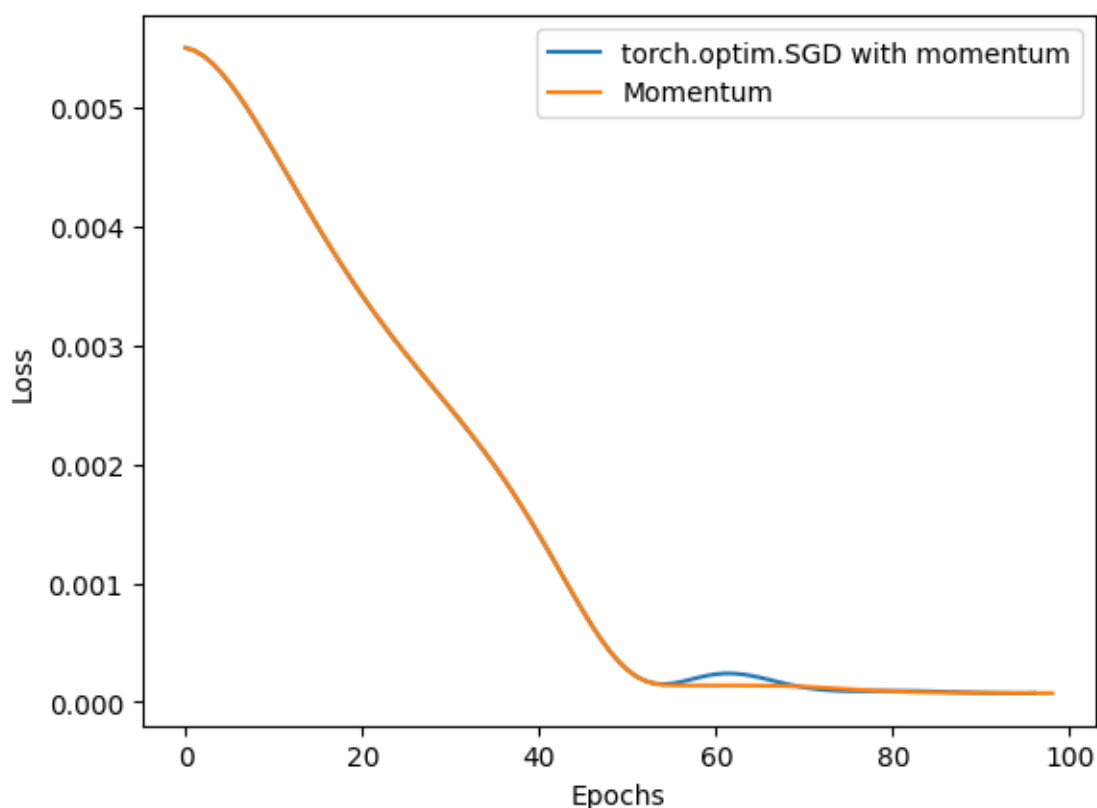
Полученные результаты соответствующие:

Испытуемый	Значение функции потерь	Среднее число шагов до схождения
Настоящие данные	0.00007481586253817499	-
<code>torch.optim.Adagrad</code>	0.00007116629643237779	68
Собственная реализация	0.00007581577268337732	1488

Ситуация обратная самому первому исследованию: собственная реализация сильно проигрывает библиотечной. Это происходит из-за неправильно установленного изначально `learning_rate`, отчего своя реализация уходит в небытие. Также, существует мнение, что библиотечная функция рассчитана на «неудобные» начальные данные (что-то вроде защиты от дураков). А поскольку в нашем случае такого не предполагалось (вернее, обработки столь щепетильной ситуации), то и в реализации такого и нету.

## Momentum vs. `torch.optim.SGD (with momentum = True)`

От более сложной модели мы перейдем, внезапно, к самой простой модификации к стохастическому градиентному спуску, основанный на физических моделях момента Momentum. Запустим наших зверей и посмотрим на вывод в виде графика, по оси Y которой мы выставим значения функции потерь, по оси X - количество шагов до остановки алгоритма.



Implemented Momentum vs. PyTorch's

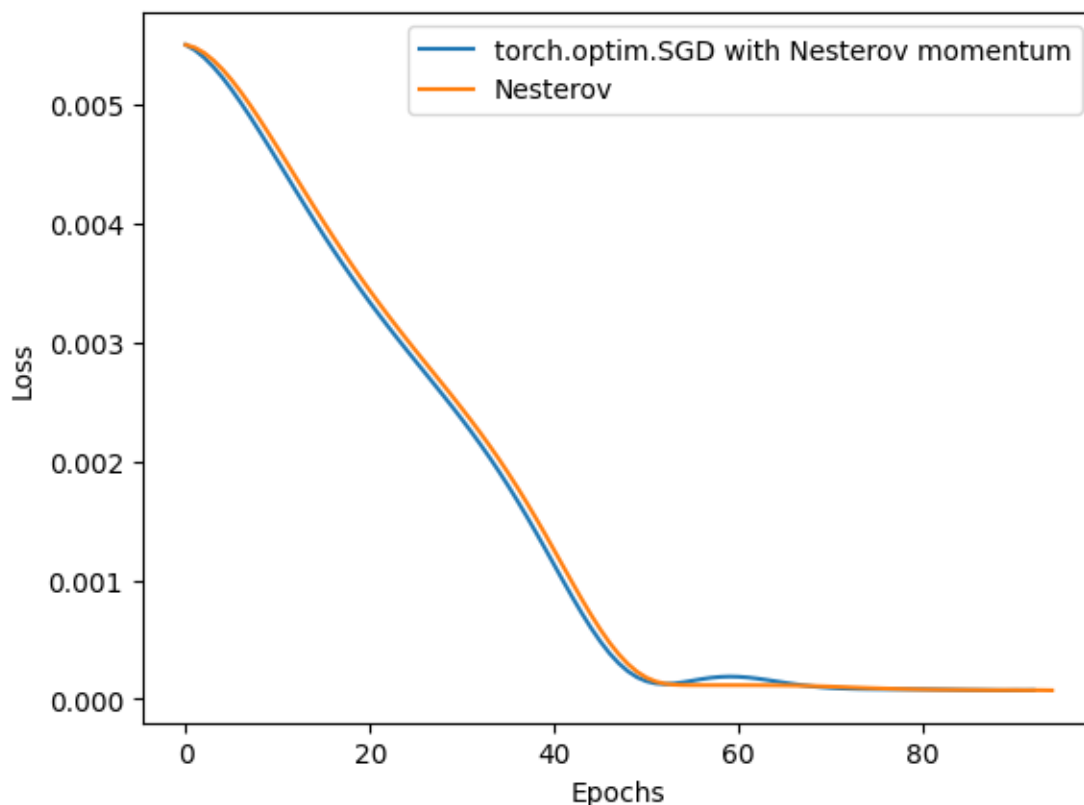
Полученные результаты соответствующие:

Испытуемый	Значение функции потерь	Среднее число шагов до схождения
Настоящие данные	0.00007481586253817499	-
<code>momentum = True</code>	0.00007458029055847868	97
Собственная реализация	0.00007460658043034181	99

Замечаем, что результаты, в особенности среднее значение функции потерь, почти не изменилось. В отличие от количества шагов, который совершенно мало увеличилось, что, опять же, как и в предыдущем случае, может быть связано с кривым поданным `learning_rate`. И всё же, почему полученные значения почти совпадают? Ситуация аналогична с ситуацией при исследовании обычного стохастического спуска: простота написания функции уменьшает количество ошибок и неточностей далее.

**Nesterov vs. `torch.optim.SGD` (with `momentum = True`, `nesterov = True`)**

В реализации библиотеки PyTorch функция SGD, для работы того самого Nesterov требует два флага: `momentum` и `nesterov` – это объясняется тем, модификация Nesterov берет за основу и использует идею Momentum. Запустим наших зверей и посмотрим на вывод в виде графика, по оси Y которой мы выставим значения функции потерь, по оси X - количество шагов до остановки алгоритма.



Implemented Nesterov vs. PyTorch's

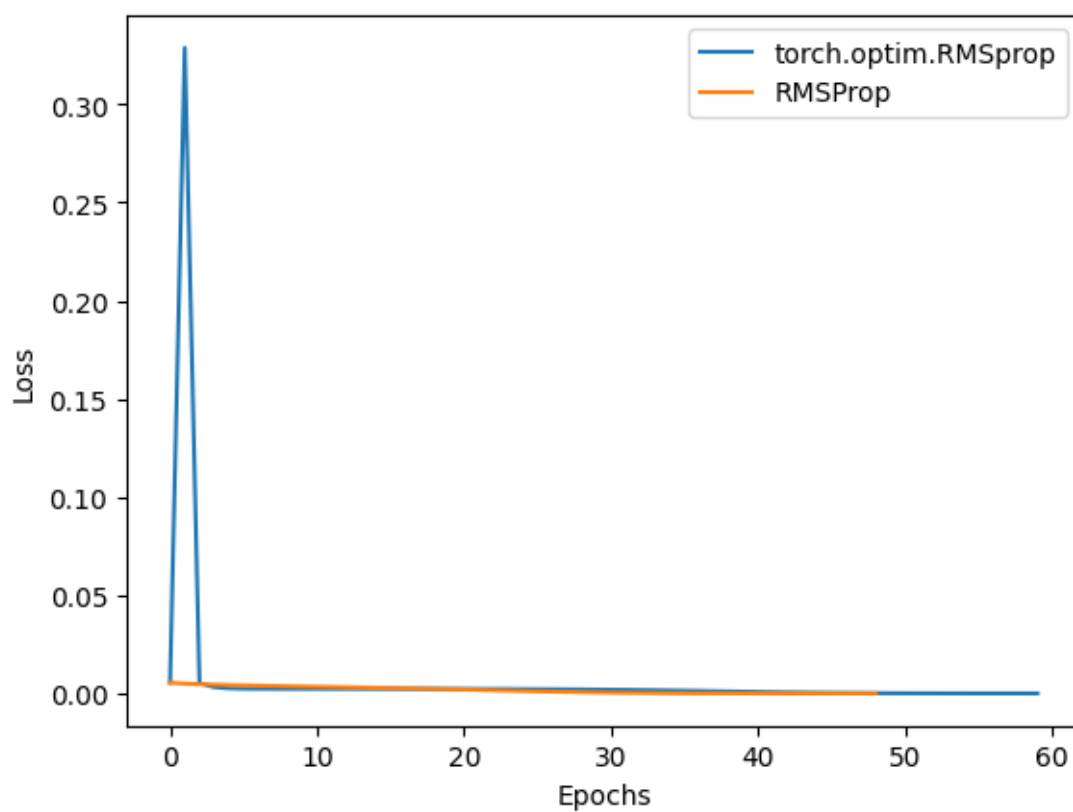
Полученные результаты соответствующие:

Испытуемый	Значение функции потерь	Среднее число шагов до схождения
Настоящие данные	0.00007458029055847868	-
<code>nesterov = True</code> и ...	0.00007412807086024441	93
Собственная реализация	0.00007271430540397701	95

Ситуация аналогична с предыдущим исследованием: наши средние значения функции потерь также почти одинаковы, отличающиеся лишь в шестой цифре, и по среднему числу шагов до схождения, отличающиеся лишь на некоторую константу. Полученные эквивалентные результаты обусловлены простотой реализации данной модификации к стохастическому градиентному спуску.

## RMSProp vs. torch.optim.RMSProp

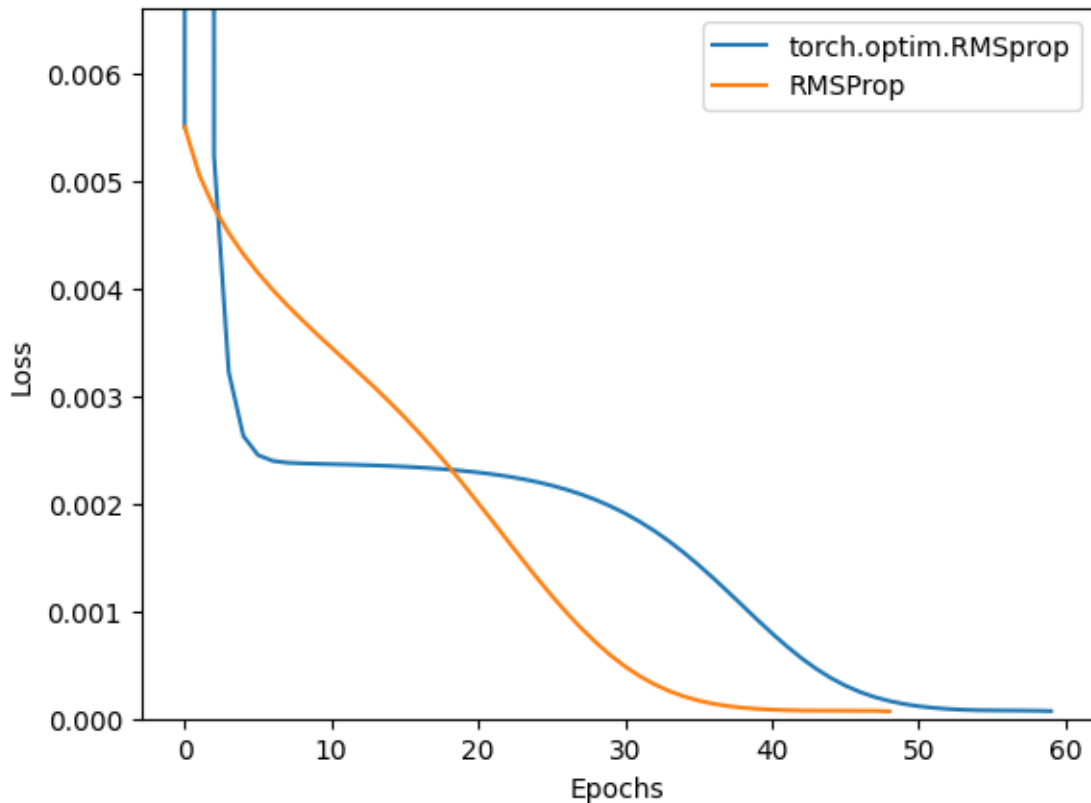
А вот мы и добрались до того самого момента, когда самописная реализация выходит победителем в межвидовой борьбе. Рассмотрим одного из двух гигантов модификаций – RMSProp. Запустим наших зверей и посмотрим на вывод в виде графика, по оси Y которой мы выставим значения функции потерь, по оси X - количество шагов до остановки алгоритма.



Implemented RMSProp vs. PyTorch's

Также, для наглядности, немного приблизимся к почти незаметной горке, ограничивая по оси  $Y$ .





Implemented RMSProp vs. PyTorch's (zoom)

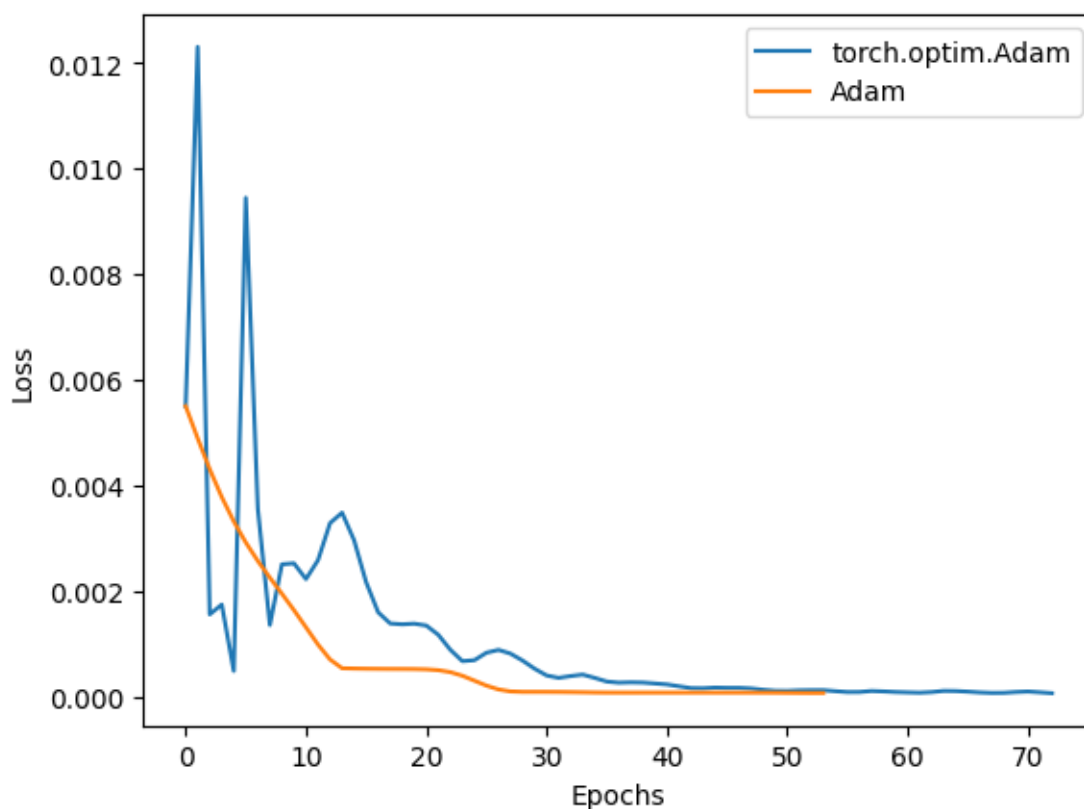
Полученные результаты соответствующие:

Испытуемый	Значение функции потерь	Среднее число шагов до схождения
Настоящие данные	0.00007481586253817499	-
<code>torch.optim.RMSProp</code>	0.00007001601600272134	60
Собственная реализация	0.00006762658137925631	49

Неожиданно, теперь в исследовании мы получаем обратную картину об общем строении мира. Дело в том, что мы все это время имели в том или ином смысле функции, которые направлены на решение более общих задач, даже для случаев, когда в функцию минимизации намеренно дают «плохой» вариант `learning_rate`, она за конечное число шагов выровняет в нормальный размер шага. Однако, посматривая на реализацию `torch.optim.RMSProp` <https://pytorch.org/docs/stable/generated/torch.optim.RMSprop.html>, мы понимаем, что есть некоторые параметры, которые оказались никоим образом не тронутые в исследовании и возможно они бы уменьшили число шагов до или даже меньше собственной реализации. С другой стороны, поскольку предоставленные значения все же являются усредненными, мы можем предположить, что та  $\delta$ -разница между средним числом шагов своей реализации и библиотечной, является константной и никуда не меняется (следовательно, если бы мы немного изменили начальный шаг, мы бы получили совершенно иные результаты).

## Adam vs. torch.optim.Adam

Еще один гигант и еще одна жертва падения (на самом деле, не совсем) перед своей реализацией библиотечной версии. Запустим наших зверей и посмотрим на вывод в виде графика, по оси Y которой мы выставим значения функции потерь, по оси X - количество шагов до остановки алгоритма.



Implemented Adam vs. PyTorch's

Полученные результаты соответствующие:

Испытуемый	Значение функции потерь	Среднее число шагов до схождения
Настоящие данные	0.00007481586253817499	-
torch.optim.Adam	0.00006828866230090804	73
Собственная реализация	0.00007292588434103817	54

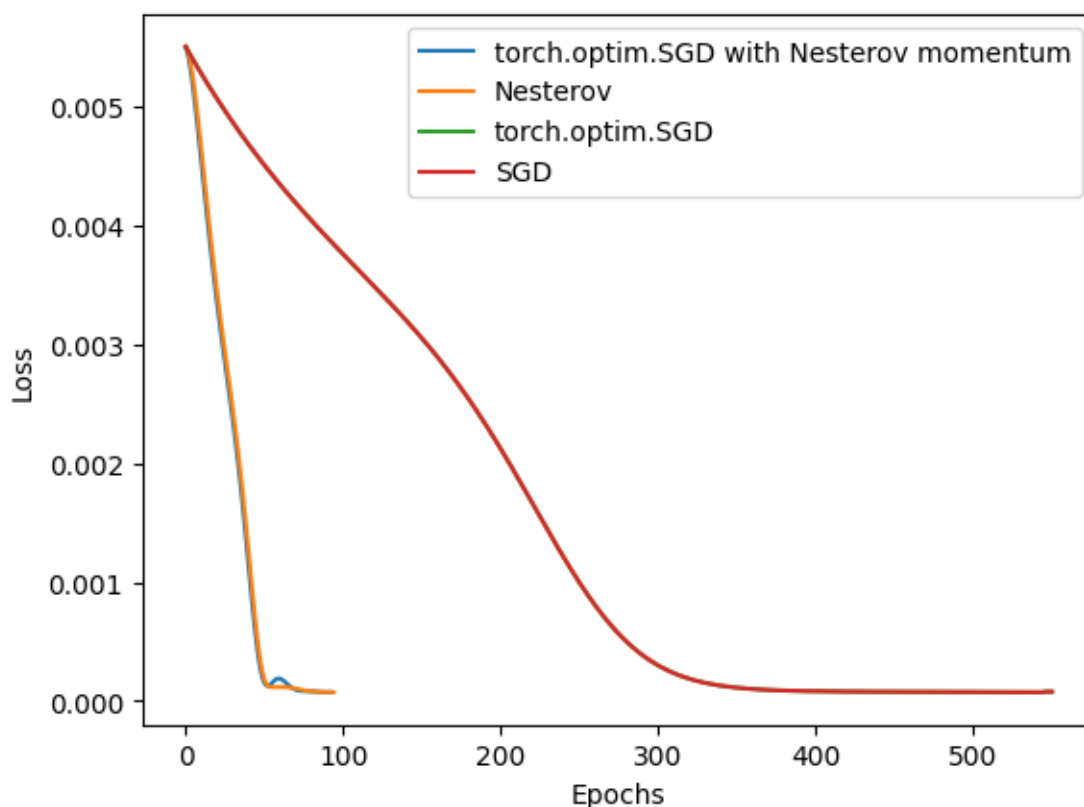
Казалось бы, свой Adam побеждает в числе шагов до вхождения, но тут мы внезапно осознаем, что не все так просто, как кажется. Дело в том, что библиотечный Adam, внезапно, оказывается куда точнее, нежели чем тем же настоящие данные и своя реализация. Это связано опять же с нашим вездесущим другом по имени `learning_rate`, если в собственной функции, вероятнее всего, по задумке изученного алгоритма, считает правильно точно, то вот новый, который по документации – <https://pytorch.org/docs/stable/generated/torch.optim.Adam.html> – включает в себя дополнительный, еще один, последний, `if` возвращает, очевидно, еще более точный минимум. Объяснение, почему количество шагов в реализации с нуля оказывается меньше, чем в библиотечной функции, аналогично предыдущим суждениям.

## Дополнительные исследования

Дальнейшие исследования не буду включать в себя многословные попытки объяснить, что к чему, а лишь будут иметь информативные представления о том, какие функции используются и какие результаты по итогам исследования вышли. Данный блок скорее нужен только для того, чтобы увидеть разницу между методами и их работой на общей поле битвы.

### Nesterov vs. SGD vs. `torch.optim.SGD` vs. `torch.optim.SGD (momentum)`

Рассмотрим вот такую именитую четверку двух самых простых и неинтересных, двух сражающихся за право быть наименьшим по среднему числу шагов до сходимости: SGD и `torch.optim.SGD`, Nesterov и `torch.optim.SGD` (с `momentum = True` и `nesterov = True`). Запустим наших зверей и посмотрим на вывод в виде графика, по оси Y которой мы выставим значения функции потерь, по оси X - количество шагов до остановки алгоритма.



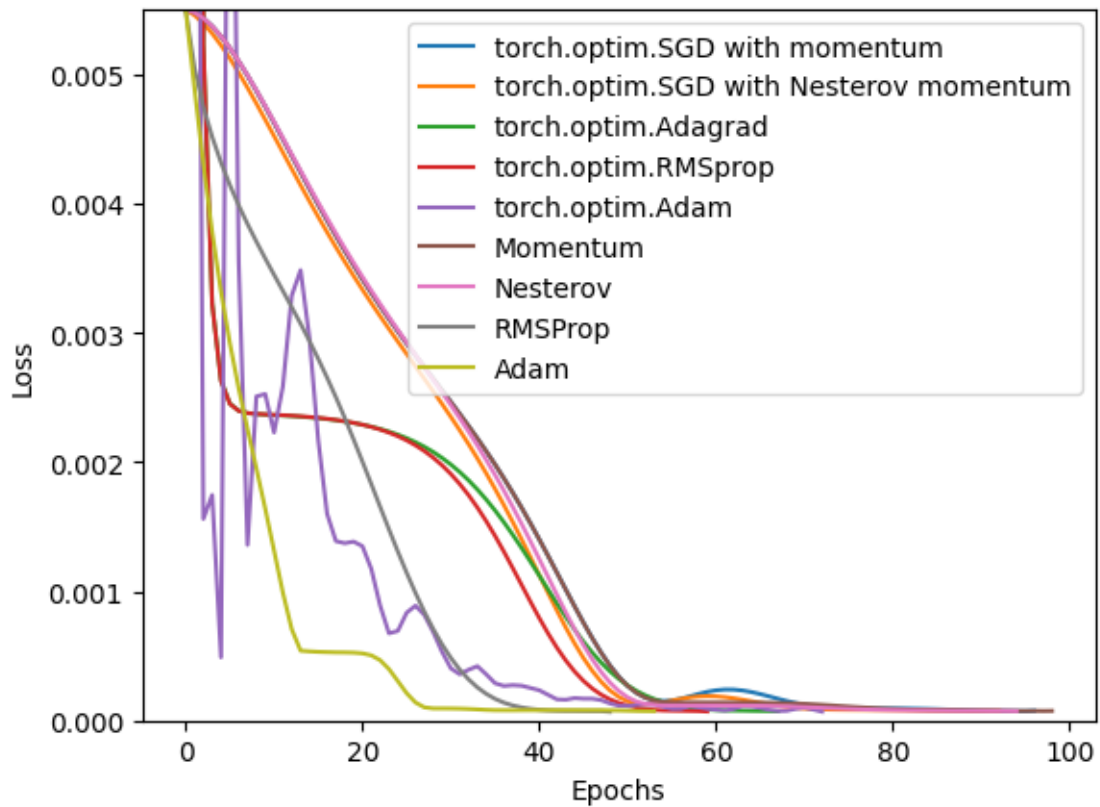
Implemented SGD and Nesterov vs. PyTorch's

Полученные результаты соответствующие:

Испытуемый	Значение функции потерь	Среднее число шагов до схождения
Настоящие данные	0.00007481586253817499	-
<code>nesterov = True</code> и ...	0.00007292588434103817	93
Свой Nesterov	0.00007271430540397701	95
<code>torch.optim.SGD</code>	0.00007753667274059714	551
Свой SGD	0.00007756952443600056	551

## Хаос

А теперь выпустим всех зверей наружу кроме трех неудачников, чтобы график не выглядел слишком плохо и ужасно: SGD (свой), `torch.optim.SGD` и AdaGrad (свой).



Implemented almost everyone vs. PyTorch's

Полученные результаты соответствующие:

Испытуемый	Значение функции потерь	Среднее число шагов до схождения
Настоящие данные	0.00007481586253817499	-
<code>momentum = True</code>	0.00007458029055847868	97
<code>nesterov = True</code> и ...	0.00007412807086024441	93
<code>torch.optim.Adagrad</code>	0.00007116629643237779	68
<code>torch.optim.RMSProp</code>	0.00007001601600272134	60
<code>torch.optim.Adam</code>	0.00006828866230090804	73
Свой Momentum	0.00007460658043034181	99
Свой Nesterov	0.00007271430540397701	95
Свой RMSProp	0.00006762658137925631	49
Свой Adam	0.00007292588434103817	54

## `scipy.optimize.minimize` и `scipy.optimize.least_squares`

### vs. ЛР №3

Настала пора проверить эквивалентность работы алгоритмов из великой и ужасной библиотеки SciPy, в котором, в отличие от PyTorch, уже за нас, ленивых разработчиков, сделано в виде удобных функций с бесконечным числом интуитивно понятных параметров. Здесь мы хотим рассмотреть и рассмотрим только два метода: BFGS и Powell Dog Leg. Почему их? С первым все понятно: в идеи работы (и, в каком-то смысле, еще и реализации) он почти эквивалентен своему другу/сопернику, оптимизированному в использовании памяти, L-BFGS. А вот со вторым ситуация примерно следующая: дело в том, как мы выяснили с предыдущей рассмотренной лабораторной работы, метод Powell Dog Leg является модифицированной версии Gauss-Newton и исследования показали, что свой туз в рукаве данный метод не часто использует при работе алгоритма (как правило, при первых  $k < n$  шагов, далее – Gauss-Newton). Именно поэтому мы рассматриваем именно их.

### vs. `scipy.optimize.minimize`

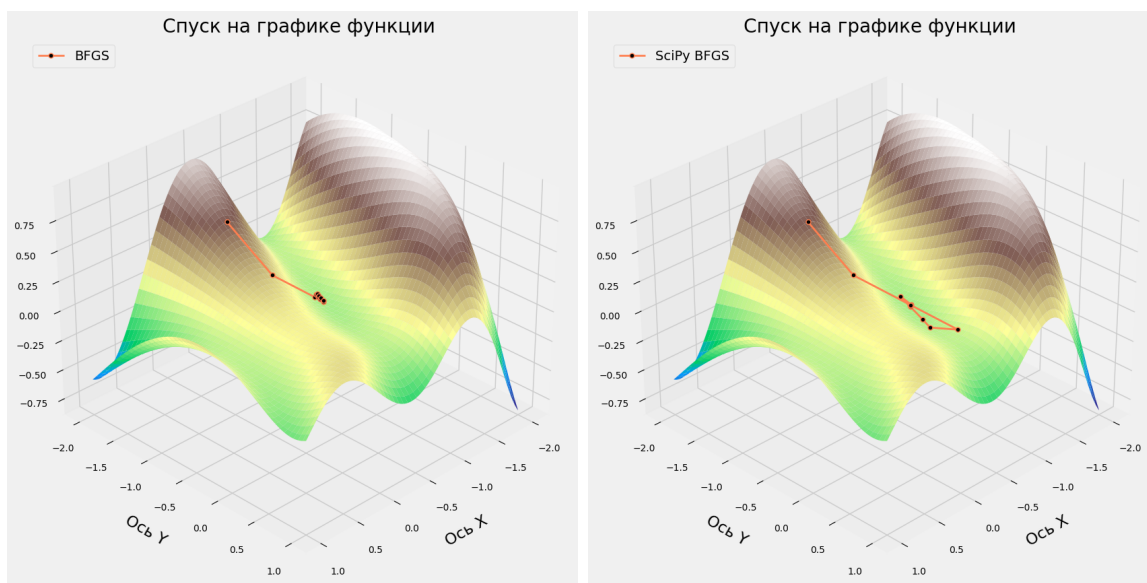
**BFGS** Начнем с BFGS или, как его еще называют в SciPy, `scipy.optimize.minimize(method='BFGS')`. Исследуем безымянную функцию  $f : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ , определенную следующим образом:

$$f(x, y) = \sin\left(\frac{1}{2} \cdot x^2 - \frac{1}{4} \cdot y^2 + 3\right) \cdot \cos(2 \cdot x + 1 - e^y)$$

В качестве исходного приближения положим точку  $\langle x_0, y_0 \rangle = \langle -0.3, -1.4 \rangle$ . Также, в качестве дополнительных параметров для работы алгоритмов мы положим следующие:

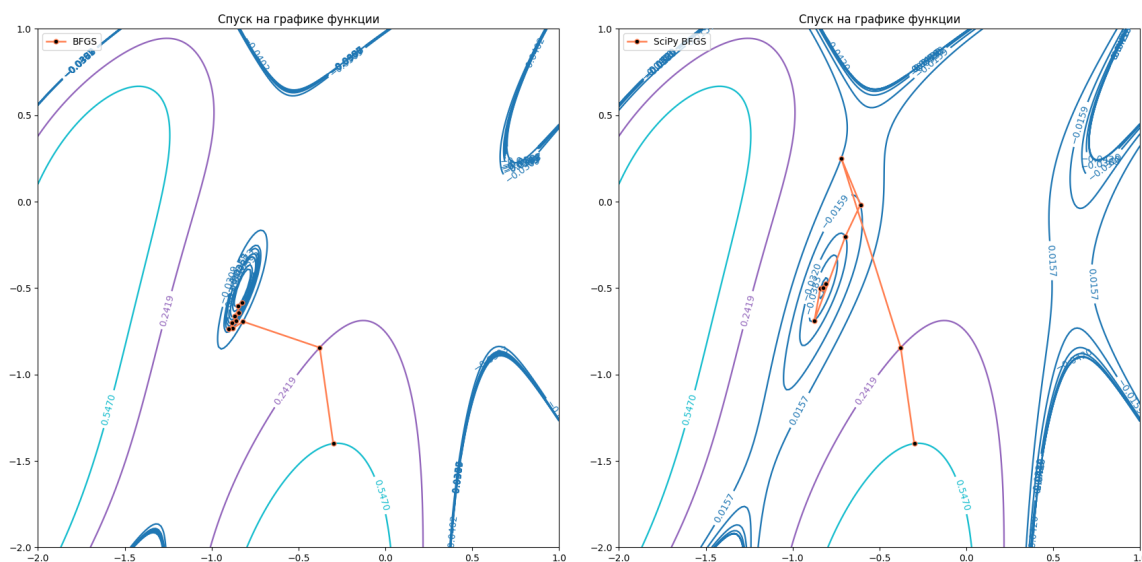
$$\begin{aligned} \text{max\_iter} &= 10 \text{ (максимальное количество шагов)} \\ \varepsilon &= 0.000000001 \text{ (условие сходимости)} \end{aligned}$$

Запустим наших зверей на бедную функцию и посмотрим на вывод путей в трехмерном виде:



BFGS vs. SciPy's on first function

Как мы видим, собственная реализация, которая слева, первыми несколькими уверенными шагами почти доходит минимума, а затем начинает «топтаться» на месте в надежде на то, что он найдет самый-самый из минимумов. А с библиотечной реализацией чуть поинтереснее: он делает первые несколько шагов немного в сторону, но затем, сбавляя скорость (уменьшая размер шага), доходит и заканчивает свой алгоритм, из-за условия сходимости. Более наглядно можно разглядеть путь соперников в виде линий уровней:



BFGS vs. SciPy's on first function (levels)

Наконец, посмотрим на полученные результаты работы в виде таблицы:

Испытуемый	Количество шагов	$\langle x, y \rangle$	$z$
Собственная реализация	11	$\langle -0.848456, -0.605206 \rangle$	$-0.040723$
SciPy	11	$\langle -0.822922, -0.499418 \rangle$	$-0.041983$

Теперь рассмотрим немного другую функцию, а именно – Химмельблау, у которого, как мы знаем, есть четыре эквивалентных минимума и определена она так:

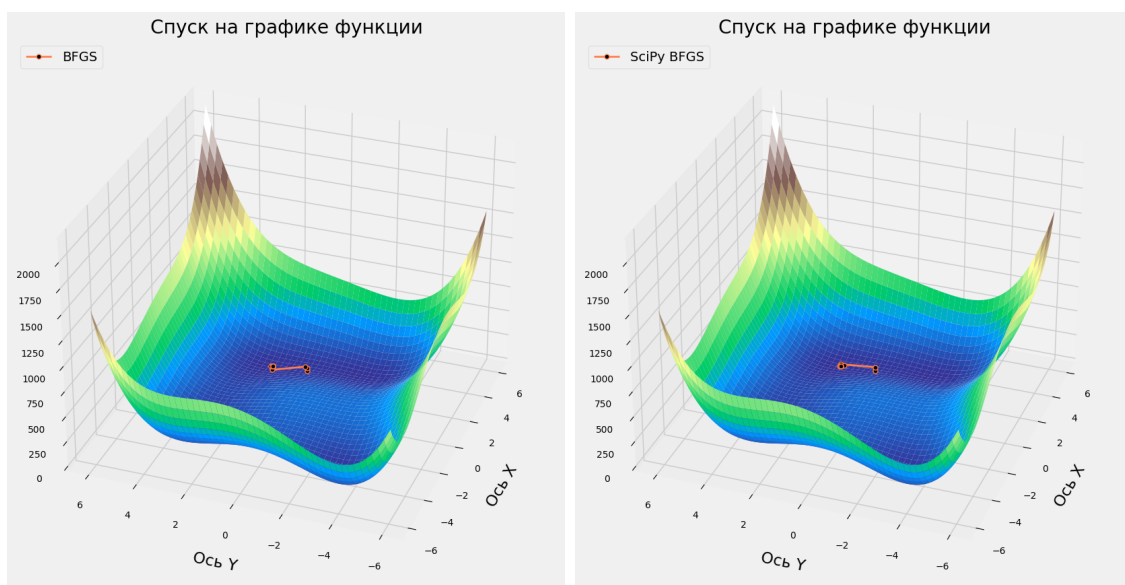
$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

Поставим здесь немного другие параметры на запуск алгоритмов:

$$\langle x_0, y_0 \rangle = \langle 3.0, 0.5 \rangle \text{ (начальное приближение)}$$

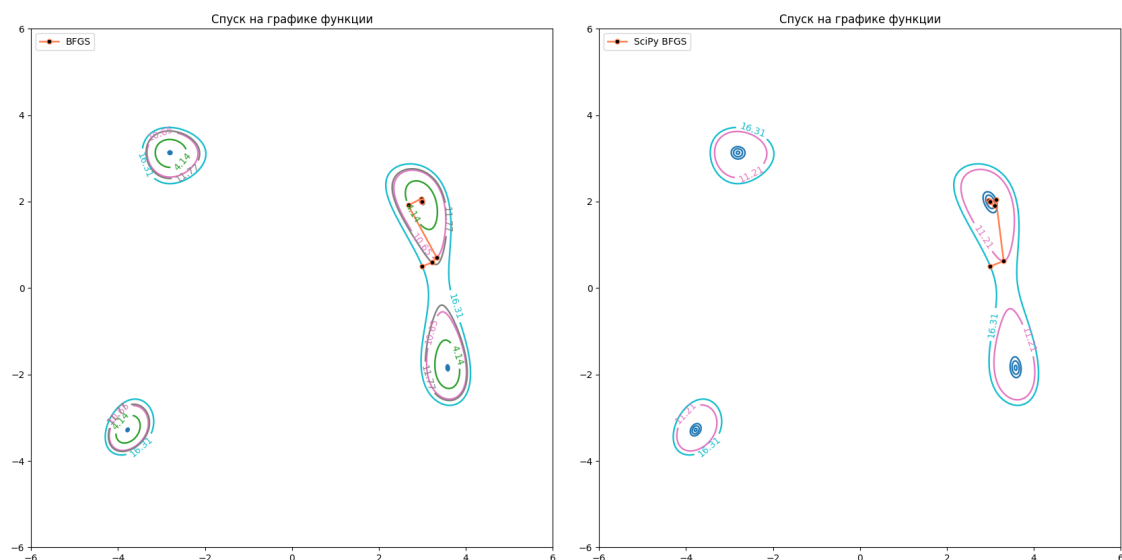
$$\text{max\_iter} = 13 \text{ (максимальное количество шагов)}$$

Запустим наших зверей на бедную функцию и посмотрим на вывод путей в трехмерном виде:



BFGS vs. SciPy's on second function

А вот здесь своя реализация, которая опять же слева, начинает проигрывать библиотеке. Заметим, что с начальной позиции алгоритм начал неестественно себя вести и делать, внезапно, слишком малые шаги до тех пор, пока не понял, что он не прав. Это связано с тем, что исходная точка лежит примерно в окрестности локального минимума относительно одной из четырех настоящих. При этом заметим, что написанный вариант вручную упирается в количество данных ему шагов. Библиотечная же реализация не ошибается и сразу делает уверенный шаг к предполагаемой точке, а в следующие еще несколько итераций приближает минимум к самому-самому. Более наглядно можно разглядеть путь соперников в виде линий уровней:

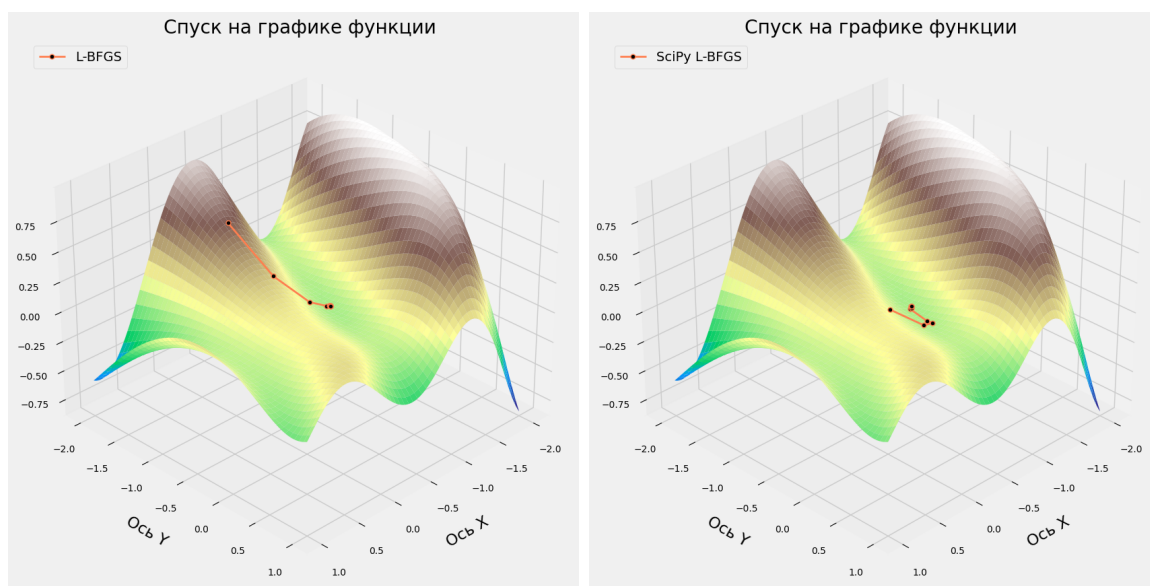


BFGS vs. SciPy's on second function (levels)

Наконец, посмотрим на полученные результаты работы в виде таблицы:

Испытуемый	Количество шагов	$\langle x, y \rangle$	$z$
Собственная реализация	14	$\langle 3.000000, 2.000000 \rangle$	0.000000
SciPy	10	$\langle 3.000000, 2.000000 \rangle$	0.000000

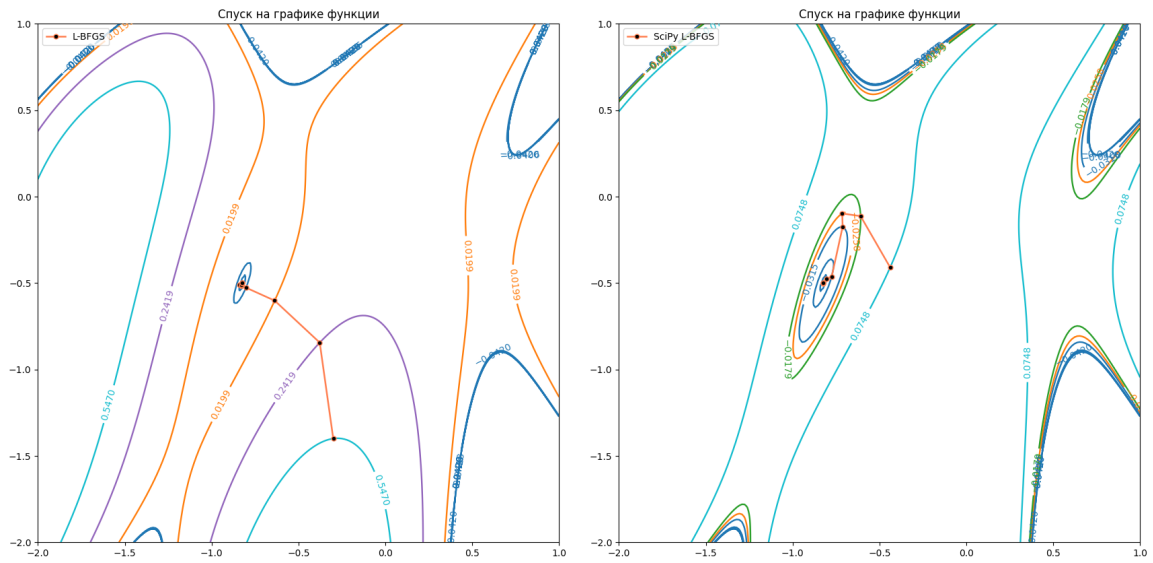
**L-BFGS** Да-да, мы, конечно, обманули друг друга, сказав, что будет рассмотрен только BFGS, но тут мы просто обозначим и вспомним, что столь не очевидная модификация существует и пользуется популярностью, чем его предшественник. Функции и параметры мы возьмем теми же, поэтому, не будем на этом останавливаться и сразу выдадим результаты. Пути сходимости по первой функции:



L-BFGS vs. SciPy's on first function

Линии уровней путей сходимости по первой функции:



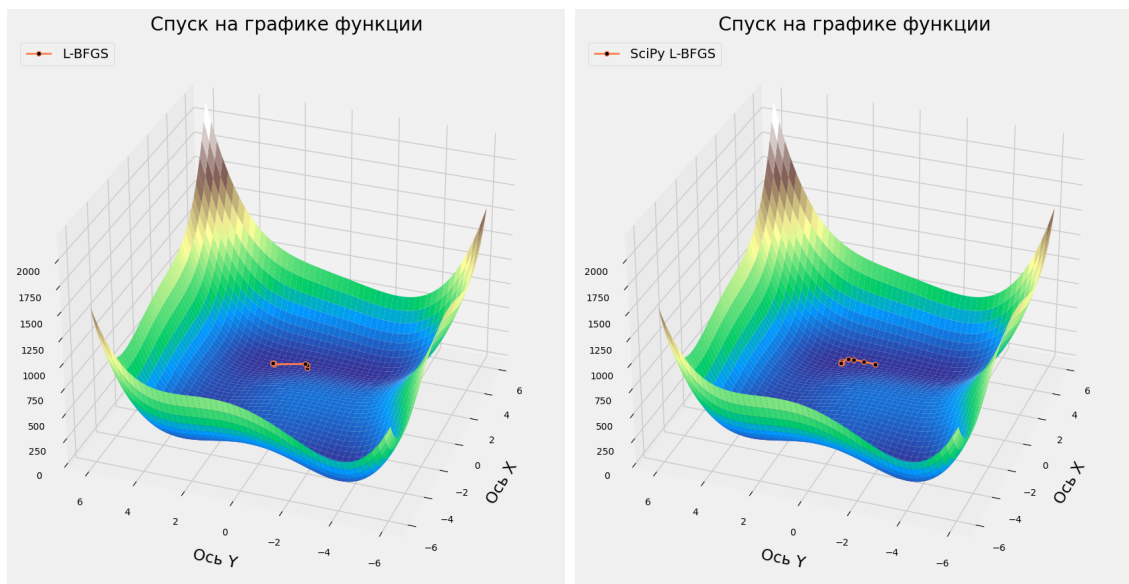


L-BFGS vs. SciPy's on first function (levels)

Данные, полученные после схождения:

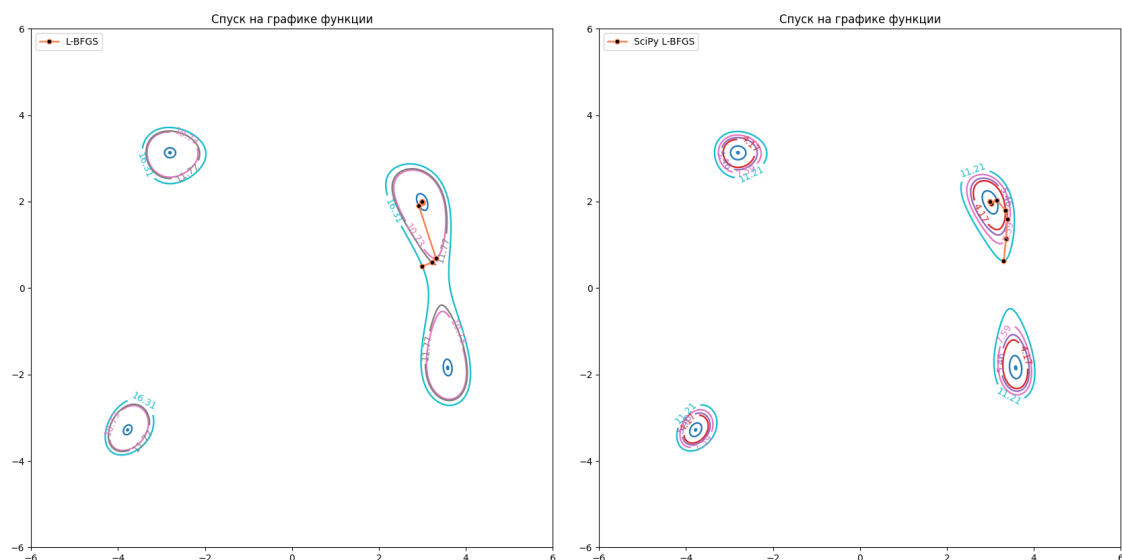
Испытуемый	Количество шагов	$\langle x, y \rangle$	$z$
Собственная реализация	10	$\langle -0.822934, -0.499453 \rangle$	-0.041983
SciPy	8	$\langle -0.822922, -0.499418 \rangle$	-0.041983

Теперь перейдем к второй функции. Пути сходимости алгоритмов:



L-BFGS vs. SciPy's on second function

Линии уровней путей сходимости:



L-BFGS vs. SciPy's on first function (levels)

Наконец, полученные после схождения данные:

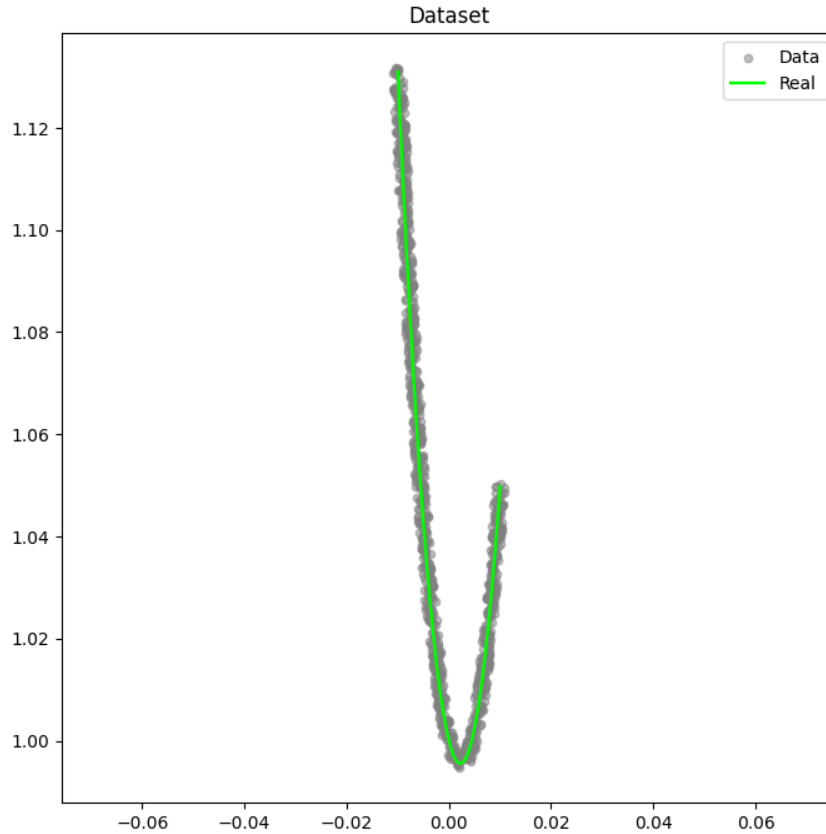
Испытуемый	Количество шагов	$\langle x, y \rangle$	$z$
Собственная реализация	11	$\langle 3.000000, 2.000000 \rangle$	0.000000
SciPy	9	$\langle 3.000000, 2.000000 \rangle$	0.000000

vs. `scipy.optimize.least_squares`

## Градиентный спуск PyTorch

Итак, поймем, что мы хотим от любого спуска к минимуму: чтобы градиент, который является неотъемлемой частью почти любого известного нам алгоритма минимизации значения, был более менее правдоподобен. Но здесь мы встречаемся лицом к лицу с проблемой, связанной с возможностями нашей системы и математики: мы не можем бесконечно долго манипулировать и что-либо другого делать с не целочисленными типами данных, они будут всё хуже и хуже деградировать от одной математической операции к другой. Поэтому, мы ставим задачу получить как можно лучший градиент и значение в этой точке более точно.

Для примера мы рассмотрим функцию Розенброка и три метода получения градиента: из библиотеки `torch`, из `numdifftool`, и методом «по определению» в буквальном смысле. В качестве функции минимизации мы возьмем модификацию стохастического градиентного спуска Adam (из коллекции собственных реализаций), в качестве начальной точки  $\langle p_{x_0}, p_{y_0} \rangle \leftarrow \langle -1.0, 0.1 \rangle$ , для всех разных методов получения градиента мы положим в качестве начального шага `initial_w = 0.5` и, наконец, сгенерируем набор данных для исследования градиентов.



Generated dataset for Rosen

Запустим наших зверей и посмотрим вот на какие результаты:

Выбранный метод градиента	Количество шагов до схождения
«По определению»	51
<code>torch</code>	51
<code>numdifftool</code>	51

Замечаем интересную находку: все использованные методы сходятся за одинаковое число шагов, умоляя общности среднего подсчета, что нам явно гарантирует, что мы можем использовать любой из них и, при любом раскладе, выиграть. Теперь же мы посмотрим на разницу отношения текущей приближенной точки алгоритма и её градиента для всех трех методов. Зададим в качестве обозначений  $\langle p_x, p_y \rangle$  - точка,  $\langle t_x, t_y \rangle$  - градиент из `torch`,  $\langle n_x, n_y \rangle$  - градиент из `numdifftool` и  $\langle d_x, d_y \rangle$  - градиент по определению. Полученные результаты для `torch`, здесь и далее будут выведены первые и последние четыре измерения.:

$\langle p_x, p_y \rangle$	$\langle t_x, t_y \rangle$
$\langle -1.0, 0.1 \rangle$	$\langle -0.00146706559993418353, -0.00014721156217214419 \rangle$
$\langle -0.58826355, 0.14131537 \rangle$	$\langle -0.00131440381006264718, -0.00019862055015740611 \rangle$
$\langle -0.14727034, 0.19673504 \rangle$	$\langle -0.00115017868324561500, -0.00026529536789049484 \rangle$
$\langle 0.29978107, 0.2682456 \rangle$	$\langle -0.00098270545906175904, -0.00034802171851757964 \rangle$
$\vdots$	$\vdots$
$\langle 1.79226672, 3.02522473 \rangle$	$\langle -0.00008879931707961148, 0.00002868576216914081 \rangle$
$\langle 1.81834509, 3.03876149 \rangle$	$\langle -0.00007504682716224205, 0.00008702729437093851 \rangle$
$\langle 1.8468342, 3.04139108 \rangle$	$\langle -0.00006181034719901721, 0.00012621921750756307 \rangle$
$\langle 1.87666167, 3.03296134 \rangle$	$\langle -0.00005016024349737770, 0.00013933684458654905 \rangle$

Полученные результаты для `numdifftool`:

$\langle p_x, p_y \rangle$	$\langle n_x, n_y \rangle$
$\langle -1.0, 0.1 \rangle$	$\langle -0.00146706559993418201, -0.00014721156217214299 \rangle$
$\langle -0.58826355, 0.14131537 \rangle$	$\langle -0.00131440381006265347, -0.00019862055015741075 \rangle$
$\langle -0.14727034, 0.19673504 \rangle$	$\langle -0.00115017868324561912, -0.00026529536789049452 \rangle$
$\langle 0.29978107, 0.2682456 \rangle$	$\langle -0.00098270545906175687, -0.00034802171851757129 \rangle$
$\vdots$	$\vdots$
$\langle 1.79226672, 3.02522473 \rangle$	$\langle -0.00008879931707961108, 0.00002868576216914129 \rangle$
$\langle 1.81834509, 3.03876149 \rangle$	$\langle -0.00007504682716224151, 0.00008702729437093872 \rangle$
$\langle 1.84683428, 3.04139108 \rangle$	$\langle -0.00006181034719901730, 0.00012621921750756380 \rangle$
$\langle 1.87666167, 3.03296134 \rangle$	$\langle -0.00005016024349737770, 0.00013933684458654840 \rangle$

Полученные результаты для собственного градиента:

$\langle p_x, p_y \rangle$	$\langle d_x, d_y \rangle$
$\langle -1.0, 0.1 \rangle$	$\langle -0.00146706560007897613, -0.00014721156195093843 \rangle$
$\langle -0.58826355, 0.14131537 \rangle$	$\langle -0.00131440380931810430, -0.00019862055056218964 \rangle$
$\langle -0.14727034, 0.19673504 \rangle$	$\langle -0.00115017868319855499, -0.00026529536813896248 \rangle$
$\langle 0.29978107, 0.2682456 \rangle$	$\langle -0.00098270545867418235, -0.00034802171864842246 \rangle$
$\vdots$	$\vdots$
$\langle 1.79226672, 3.02522473 \rangle$	$\langle -0.00008879931708840553, 0.00002868576212612835 \rangle$
$\langle 1.81834509, 3.03876149 \rangle$	$\langle -0.00007504682706977080, 0.00008702729432008473 \rangle$
$\langle 1.84683428, 3.04139108 \rangle$	$\langle -0.00006181034720512124, 0.00012621921753218010 \rangle$
$\langle 1.87666167, 3.03296134 \rangle$	$\langle -0.00005016024348539292, 0.00013933684456614911 \rangle$

## Эксперименты с SciPy

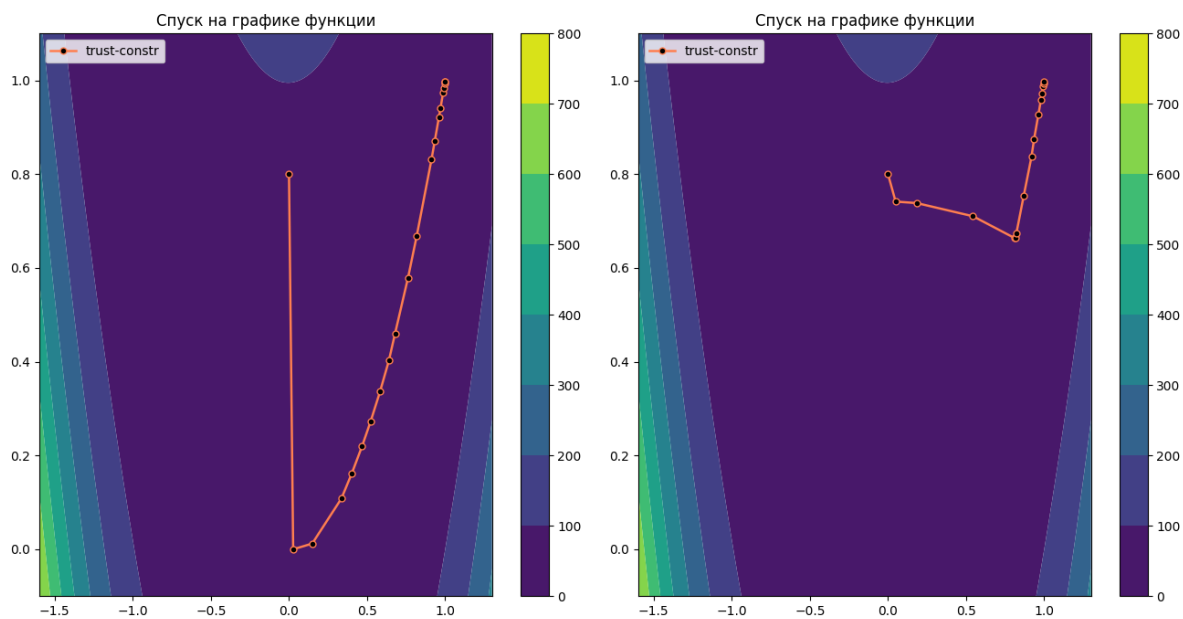
## Линейные и нелинейные ограничения

Помимо всех рассмотренных ранее прелестей от библиотеки SciPy, вроде минимизации функции совершенно разными и некоторыми нерассмотренными на курсе методами или получения минимальной суммы квадратов, поддерживаются также возможности ограничения функции. Это бывает крайне полезно, когда почти-аналитически мы можем кое-что сказать про саму функцию и, возможно, таким образом, избавиться от лишнего числа хождений «не туда», куда надо. А как мы могли бы реализовать ограничения? Через запрет хождения функции. Поставим некую стену, через которую нельзя пройти, и таким образом, если наш алгоритм сделал шаг не туда, в сторону стены, то нам, пользователю, может повести, если он «отскочит» в нужную сторону, к минимуму.

Начнем с уже ни раз используемой функции Розенброка, а в качестве одного из методов, поддерживающий те или иные ограничения, возьмем `trust-constr`. Зададим начальную точку  $\langle x_0, y_0 \rangle = \langle 0.0, 0.8 \rangle$ , а также посмотрим сначала на нелинейное ограничение: поставим в некоторой точке центр, смещенного эллипса, заданный следующей формулой:

$$f(x, y) = \frac{(x^2)}{3} + \frac{(y^2)}{0.5} - 0.6$$

Само же ограничение будет определено на луче  $[0.5, +\infty)$ . Итак, посмотрим же на график без ограничений и с ними:



Rosen with no constraint and with non-linear

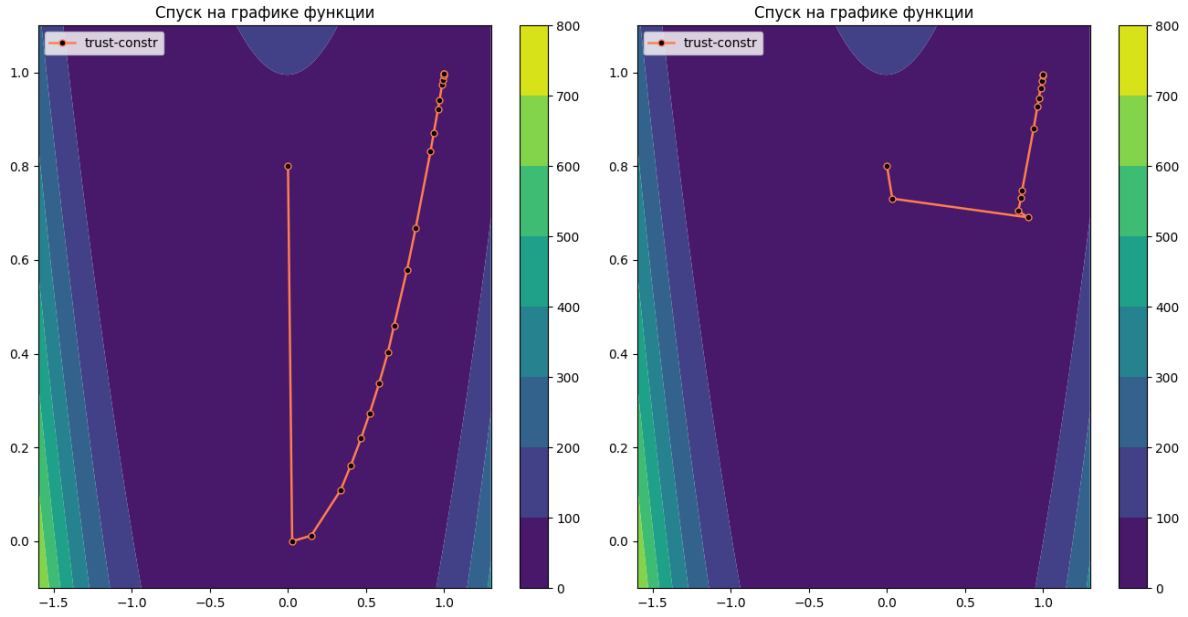
Уже без каких-либо таблиц с результатами схождения мы можем увидеть, что расстояние и, соответственно, количество шагов сильно уменьшилось при добавленном эллипсе. Мы можем заметить небольшую «горку» в шагах алгоритма, которые и приводят к месту, где обитает минимум. Полученные табличные результаты:

Испытуемый	Число шагов	$\langle x, y \rangle$	$z$
No any constraints	58	$\langle 0.998788, 0.997573 \rangle$	0.000001
With non-linear constraint	44	$\langle 0.998787, 0.997571 \rangle$	0.000001

Теперь сделаем тоже самое, но уже только с линейным ограничением. Положим в качестве матрицы  $A$  – матрица, определяющая ограничение, – следующее:

$$A = \begin{pmatrix} 0.1 & 1 \end{pmatrix}$$

Также, само ограничение будет определено на луче  $[0.78, +\infty)$ . Итак, посмотрим же на график без ограничений и с ними:



Rosen with no constraint and with linear

Опять же, мы видим, что довольно-таки сильный срез в пути, а именно, в правильном направлении, по которому следует идти, позволил ограничить число шагов до сходимости в разы. Полученные табличные результаты:

Испытуемый	Число шагов	$\langle x, y \rangle$	$z$
No any constraints	58	$\langle 0.998788, 0.997573 \rangle$	0.000001
With linear constraint	35	$\langle 0.997298, 0.994596 \rangle$	0.000007

Только вот на сей раз вышло немного забавно: мы хоть и дошли до минимума быстрее, но ошиблись на смешной  $\epsilon$ .

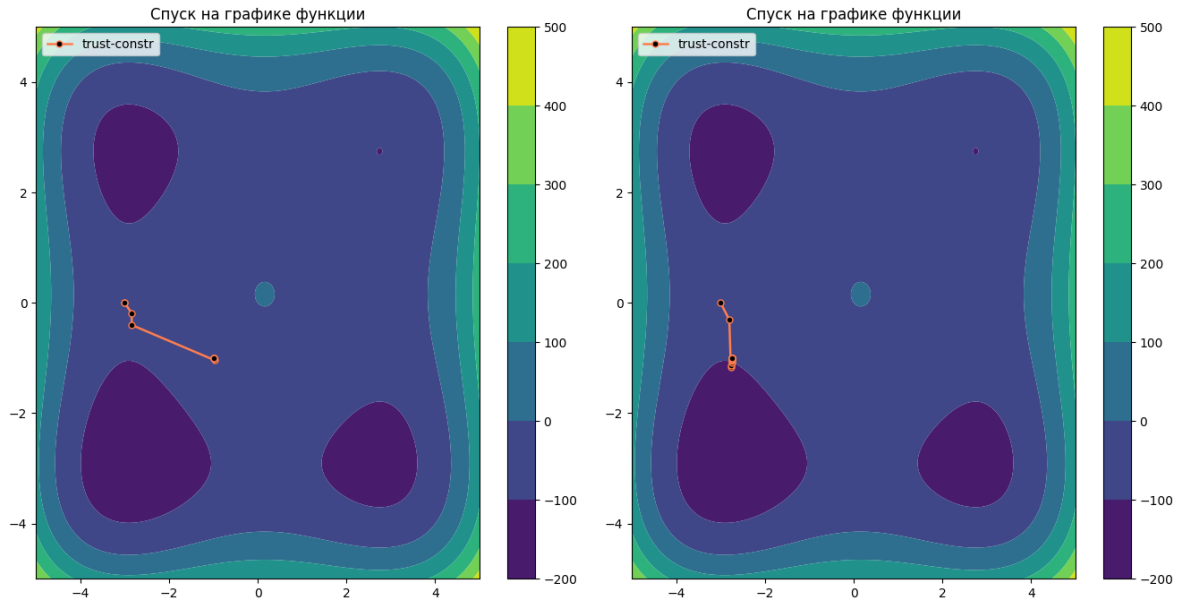
А теперь возьмем функцию посильнее и менее тривиальную в дальнейшей минимизации, а именно – Стыбинского-Танга, где также, как и во многих других неприятных функциях, имеется еще как минимум несколько похожих минимумов, среди которых лишь один является настоящим. Сама функция определяется следующей формулой, здесь  $\vec{x}$  – вектор, так как функция рассчитана на многомерные и многопеременные случаи:

$$f(\vec{x}) = \frac{\sum_{i=1}^n x_i^4 - 16x_i^2 + 5x_i}{2}$$

Зададим начальную точку как  $\langle x_0, y_0 \rangle = \langle -3.0, 0.0 \rangle$ , а также линейное ограничение со следующими параметрами:

$$A = \begin{pmatrix} 1 & 0 \end{pmatrix}, \quad \text{lb} = -4.5,$$

где  $\text{lb}$  – это нижняя граница ограничительного луча  $[\text{lb}, +\infty)$ . Итак, посмотрим же на график без ограничений и с ними:



Tanga with no constraint and with linear

Итак, здесь мы видим, что изначально метод **trust-constr**, сделав первые 10 шагов, остановился и остался стоять на одном месте, то есть, не сошелся. Однако, здесь мы впервые видим, как аналитическое решение приводит к правильному направлению к минимуму: метод хоть и не сошелся хорошо, но на минимуме начал замедляться, как бы пытаясь найти самый-самый, не пропуская ни один из виду. Полученные табличные результаты:

Испытуемый	Число шагов	$\langle x, y \rangle$	$z$
No any constraints	$10 + \infty$	$\langle -0.999878, -0.999878 \rangle$	-39.991961
With linear constraint	419	$\langle -2.750000, -1.000000 \rangle$	-97.558594