Второе задание по методам оптимизации

Иван Ермаков

Мне кажется, что в presubmit_tests.py есть некоторые некорректные тесты. Про тесты для бонусной части будет сказано дальше, но там еще есть тест, который проверяет линейный поиск для условия Армихо, запуская его в точке минимума. При этом подходит любой шаг, и в таком случае, как мне кажется, правильный ответ — это удвоенный предыдущий шаг, если он определен. Но тесты ожидают, что вернется ответ, равный предыдущему шагу. В optimization.py для этого пришлось убрать увеличение шага, это место отмечено "#FIXME". Сам тест находится на 350 строчке в presubmit_tests.py.

3

Логистическая регрессия

$$\min_{x\in\mathbb{R}^n}\left\{f(x)=\frac{1}{m}\left\langle\ln\left[1+\exp\left(-b\odot(Ax)\right)\right],1_m\right\rangle+\frac{\lambda}{2}\|x\|^2\right\} \tag{1}$$

Здесь b — вектор из b_i, A – матрица $m\times n$, в которой строки являются векторами a_i.

$$\begin{aligned}\mathrm{D}\,f(x)&=\lambda x-\frac{A^T}{m}\left(b\odot\frac{\exp\left(-b\odot(Ax)\right)}{1+\exp\left(-b\odot(Ax)\right)}\right)=\lambda x-\frac{A^T}{m}\left(b\odot u\left[-b\odot(Ax)\right]\right)\\ u(y)&=\frac{e^y}{1+e^y}=\frac{1}{1+e^{-y}}\end{aligned} \tag{2}$$

Здесь использовано много свойств произведения Адамара.

$$\frac{\mathrm{d}u}{\mathrm{d}y}=\frac{e^y(1+e^y)-e^{2y}}{(1+e^y)^2}=\frac{e^y}{(1+e^y)^2} \tag{3}$$

$$\mathrm{D}^2 f[\mathrm{d}x]=\lambda\mathrm{d}x+\frac{A^T}{m}\left(b\odot\frac{e^y}{(1+e^y)^2}\odot b\odot(A\mathrm{d}x)\right) \tag{4}$$

$$\mathrm{D}^2 f=\lambda+\frac{A^T}{m}\left\{\left[\left(b\odot b\odot\frac{e^y}{(1+e^y)^2}[-b\odot(Ax)]\right)1_n^T\right]\odot A\right\} \tag{5}$$

Потратив еще *немного* времени, можно понять, что это выражение можно переписать как

$$\mathrm{D}^2 f=\lambda+\frac{1}{m}\cdot A^T\,\mathrm{diag}\left\{b\odot b\odot v(-b\odot(Ax))\right\}A \tag{6}$$

где

$$v(y)=\frac{e^y}{(1+e^y)^2} \tag{7}$$

Эксперимент: траектория градиентного спуска

Построение графиков выполнено при помощи скрипта grad_trajectory.py.

При постоянном шаге для хорошо обусловленной матрицы поиск останавливается за 109000 шагов, а для плохо обусловленной за 69000. Скорее всего так происходит из-за того, что в хорошо обсусловленной матрице метод постоянно проскакивает оптимальную точку из-за высокой требуемой точности.

Для условия Армихо все встало на свои места: для хорошо обусловленной матрицы метод сходится всего за 18 шагов. Для плохо обусловленной требуется 327 шагов. Видно, что в первом случае довольно

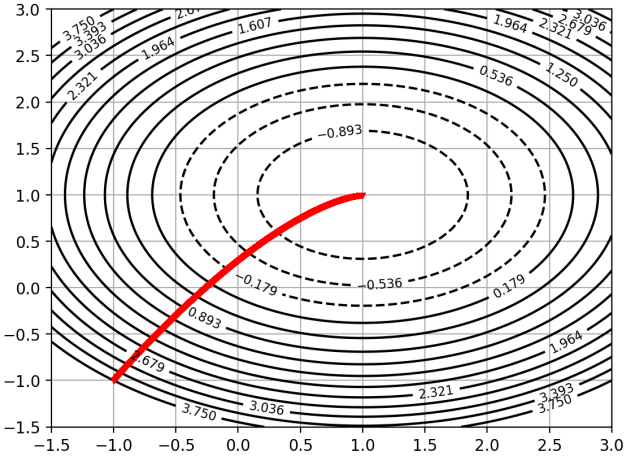

Рис. 1: Постоянный шаг, хорошо обусловленная матрица.

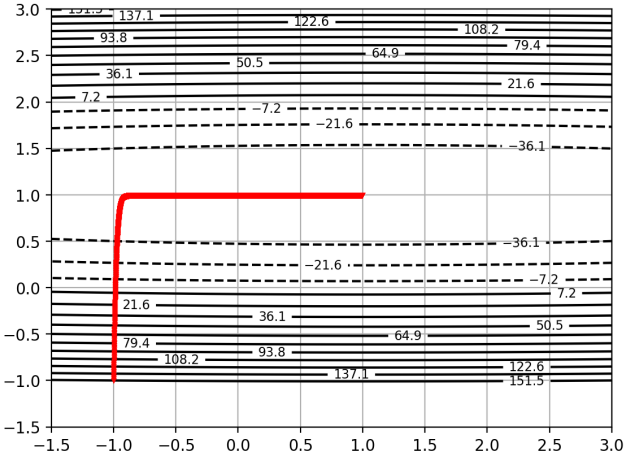

Рис. 2: Постоянный шаг, плохо обусловленная матрица.

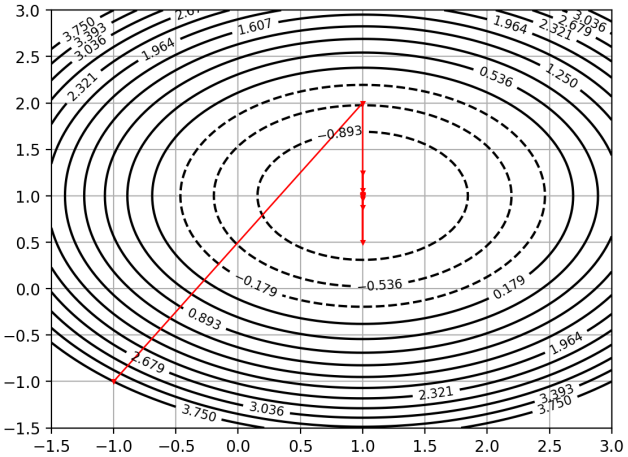

Рис. 3: Условие Армихо, хорошо обусловленная матрица.

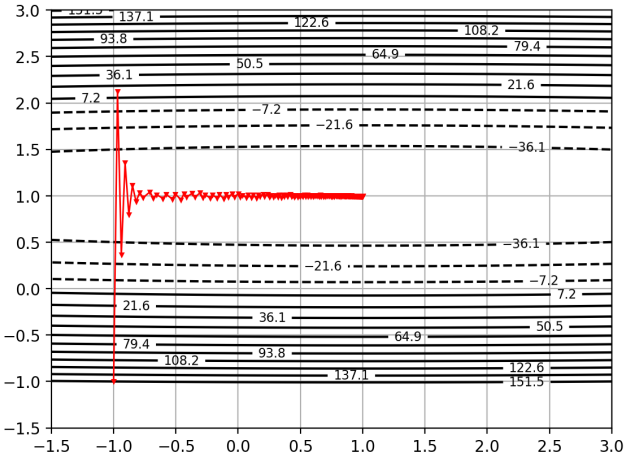

Рис. 4: Условие Армихо, плохо обусловленная матрица.

быстро тракетория становится прямой, а во втором случае она все время колеблется вокруг оптимального пути и совершает больше шагов, чем нужно.

Для условия Вольфа ситуация аналогичная первому случаю: для хорошо обусловленной матрицы метод работает “слишком хорошо” и проскакивает точку экстремума. Метод сходится за 18 шагов. Для плохо обусловленной матрицы требуется всего 4 шага. Понятно, что в общем случае для хорошо обусловленой матрицы метод Вольфа тоже работает хорошо, и некоторое $O(1)$ количество операций в конце, связанное с проскакиванием, роли не играет.

При этом, конечно, чем ближе прямая градиента к оптимальной точке, тем быстрее будут работать все методы, потому что траектория будет ближе к прямой.

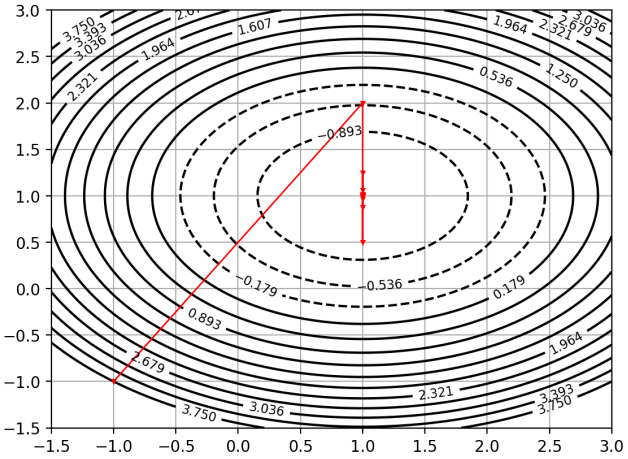

Рис. 5: Условие Вольфа, хорошо обусловленная матрица.

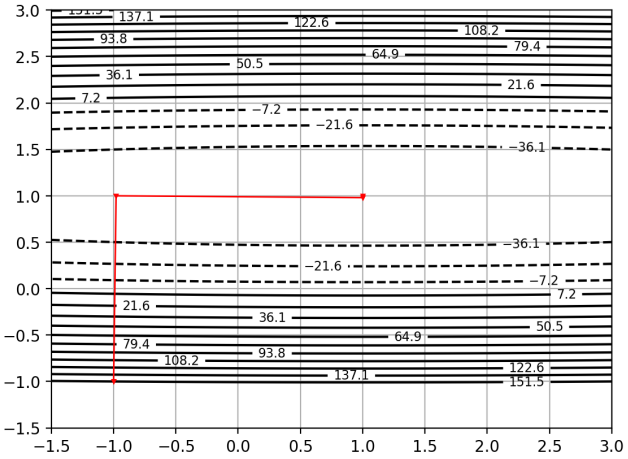

Рис. 6: Условие Вольфа, плохо обусловленная матрица.

Эксперимент: зависимость числа итераций итераций градиентного спуска от числа обусловленности и размерности пространства

Выборка генерируется при помощи скрипта perf_plots.py так, как было рекомендовано в задании: генерируется диагональная матрица, далее у нее фиксируется число обусловленности. Потом случайным образом выбираются x_0 и b. Получившиеся семейства кривых изображены на рисунке 7

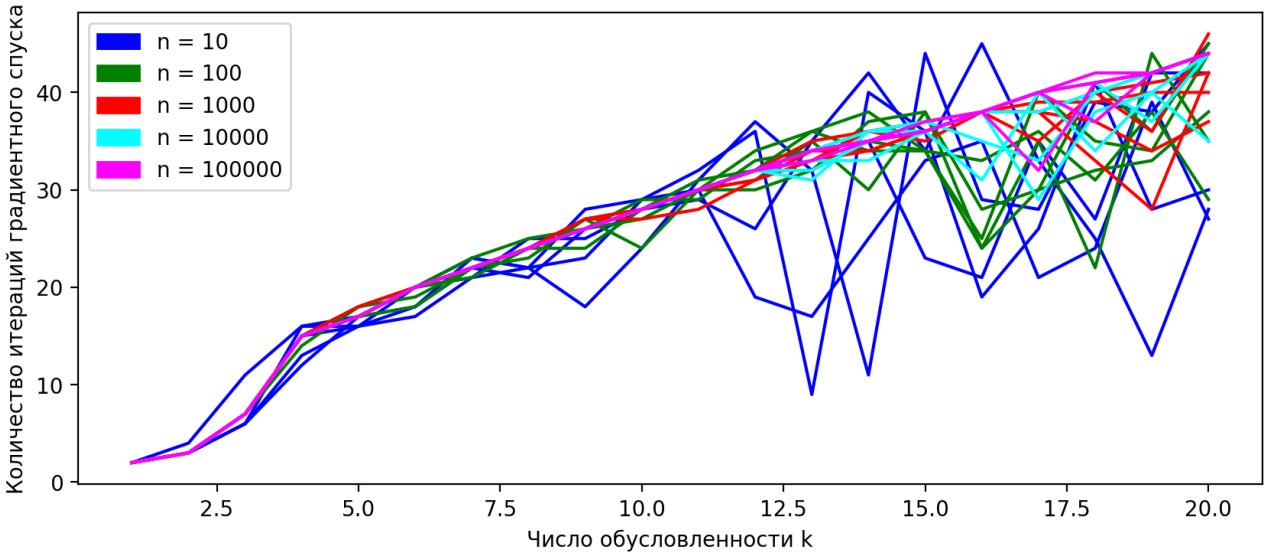

Рис. 7: Зависимость числа итераций от числа обусловленности и размерности пространства

Видно, что от размерности пространства количество итераций не зависит. А вот с ростом числа обусловленности количество итераций растет. Причем рост, хотя это и сложно определить, похож на линейный. Линейный рост можно объяснить так:

$$f(x_k) - f^* \leq \frac{L}{2}\left(\frac{\kappa - 1}{\kappa + 1}\right)^{2k} \|x_0 - x^*\|^2 \tag{8}$$

Отсюда

$$k = \frac{\ln \alpha}{2 \ln \left(\frac{\kappa - 1}{\kappa + 1} \right)} \approx -\kappa \ln \alpha \tag{9}$$

из Тейлоровского разложение при $\kappa \gg 1$.

Эксперимент: Сравнение методов градиентного спуска и Ньютона на реальной задаче логистической регрессии

На одну итерацию градиентному спуску требуется $O(n)$ памяти, поскольку хранятся только векторы x и ∇f. Матрица, конечно, тоже хранится, но она разреженная и не занимает много памяти. Подсчет градиента требует $O(nm)$ времени для умножения матрицы на вектор. Линейный поиск также требует $O(nm)$ операций, поскольку для вычисления значения функции тоже нужно умножать матрицу. Для сложения координат нужно $O(n)$ операций. В итоге требуется $O(nm)$ времени для одного шага градиентного спуска.

Метод Ньютона требует $O(n^2)$ памяти на каждом шаге, поскольку разложение Холецкого из scipy не поддерживает разреженные матрицы, и внутри неявным образом она приводится к обычной плотной матрице. Для обращения гессиана(или, эквивалентно, для решения СЛАУ) требуется $O(n^3)$ операций. Для последующего линейного поиска требуется $O(nm)$ операций. В итоге получается сложность операции $O(nm + n^3)$.

У меня получилось выполнить на своем компьютере с 16 Гб оперативной памяти все вычисления, кроме метода Ньютона для датасета real-sim. Матрица из этого датасета, если хранить ее как плотную матрицу с числами с плавающей точкой одинарной точности, занимает в памяти 5.6 Гб. Разреженная матрица занимает в памяти намного меньше места. Однако, как было отмечено выше, разложение Холецкого неявно превращает матрицу в плотную. Более того, видимо, из-за особенностей аллокации numpy, на второй итерации метода количество памяти еще возрастает. Поэтому тесты для этого датасета были выполнены на кластере ЦОД МФТИ используя 32 Гб оперативной памяти. Для возможности сравнивать время обработки разных датасетов все вычисления в итоге были выполнены на кластере.

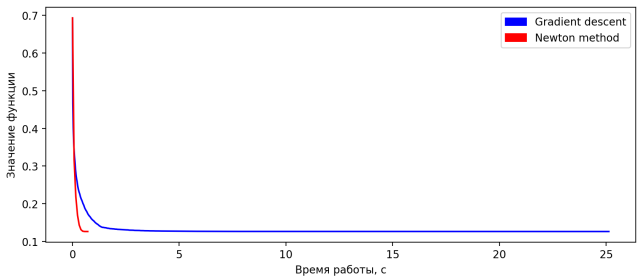

Рис. 8: Датасет w8a, значения функции

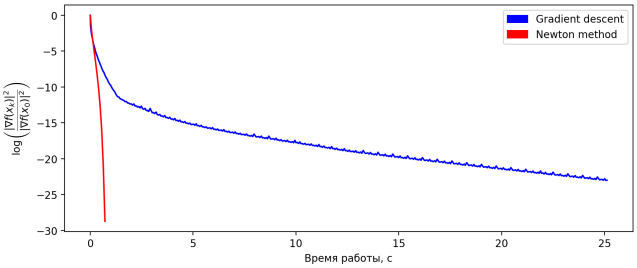

Рис. 9: Датасет w8a, значения градиента

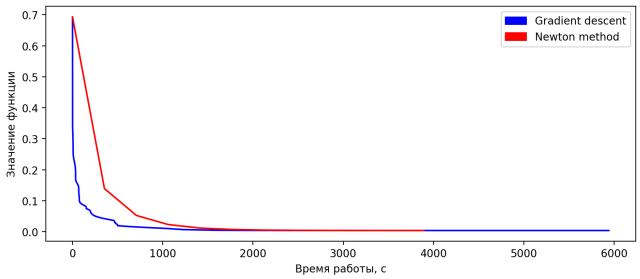

Рис. 10: Датасет gisette, значения функции

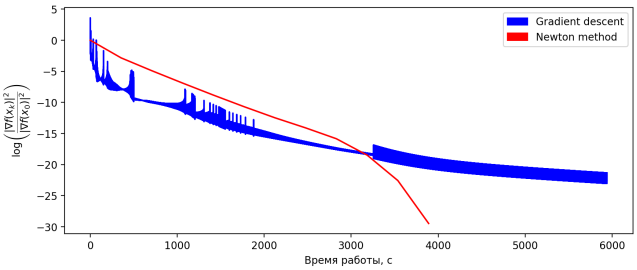

Рис. 11: Датасет gisette, значения градиента

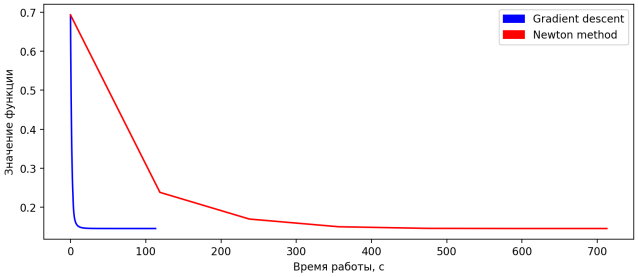

Рис. 12: Датасет real-sim, значения функции

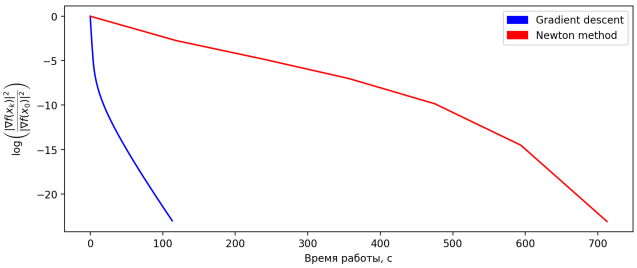

Рис. 13: Датасет real-sim, значения градиента

Выводы

Если посмотреть на графики, становится видно, что метод Ньютона для большинства датасетов работает быстрее из-за сверхлинейной скорости сходимости. Однако для больших n градиентный спуск начинает выигрывать, потому что слагаемое n^3 в сложности шага метода Ньютона начинает доминировать. При этом количество итераций градиентного спуска и метода Ньютона не зависит от размерностей.

Также интересно отметить колебания градиента функции в градиентном спуске. Это, по видимости, связано с тем, что линейный поиск выбирает достаточно большую длину шага для того, чтобы на ней градиент заметно изменился. Можно предположить, что происходят осцилляции в направлениях, заметно отличающихся от направления к минимуму, то есть задача в данных точках плохо обусловлена.

Градиентный спуск лучше для очень большого количества признаков, то есть для большого числа n. При этом желательно, чтобы в данных не было выбросов, так как это потенциально может привести к плохой обусловленности задачи. Метод Ньютона хорош для достижения высокой точности из-за сверхлинейной скорости сходимости, однако он требует большого количества памяти и не подходит для очень больших n.

Я попробовал использовать пакет scikit-sparse, который поддерживает разложение Холецкого для разреженных матриц. По итогам экспериментов оказалось, что использование этого пакета действительно сильно снижает использование памяти. Все тесты стало возможно выполнить на 16 Гб оперативной памяти. Код, использующий scikit-sparse расположен в другой ветке репозитория, поэтому в прилагаемые файлы не вошел.

Бонусная часть: оптимизированный оракул для логистической регрессии

Я не очень разобрался в тестах для оптимизированного оракула. Мы можем оптимизировать только вычисление Ax, поскольку вычисление $A^T v$ будет вестись не в x, $x + \alpha d$, ..., а в других точках, связанных нелинейным образом. В тестах, кажется, занижены допустимые значения количества вызовов $A^T x$, так как считается, что его мы должны пересчитывать, но, поскольку это невозможно, исходные тесты программа не проходит. Я прикладываю модифицированные в бонусном разделе тесты, в которых исправлено количество вызовов $A^T x$.

Графики тестов

Код для генерации графиков находится в perf_plot_additional.py.

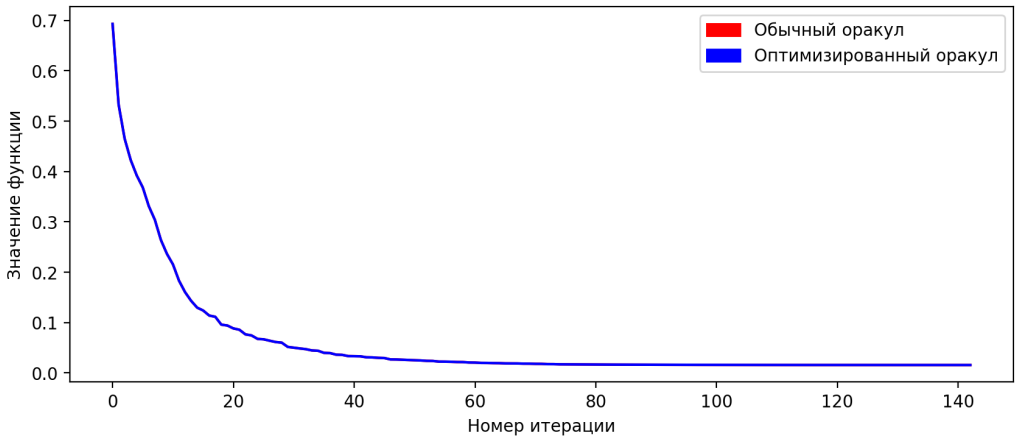

Рис. 14: Зависимость значения функции от количества итераций

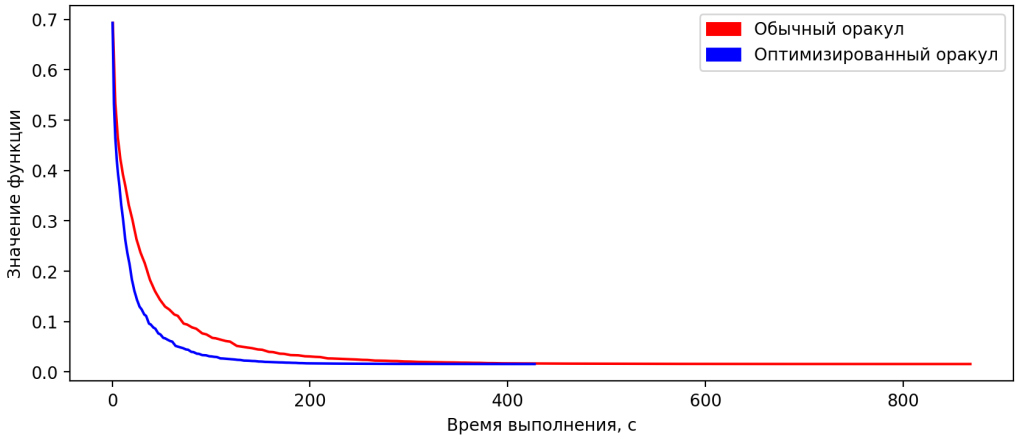

Рис. 15: Зависимость значения функции от времени работы

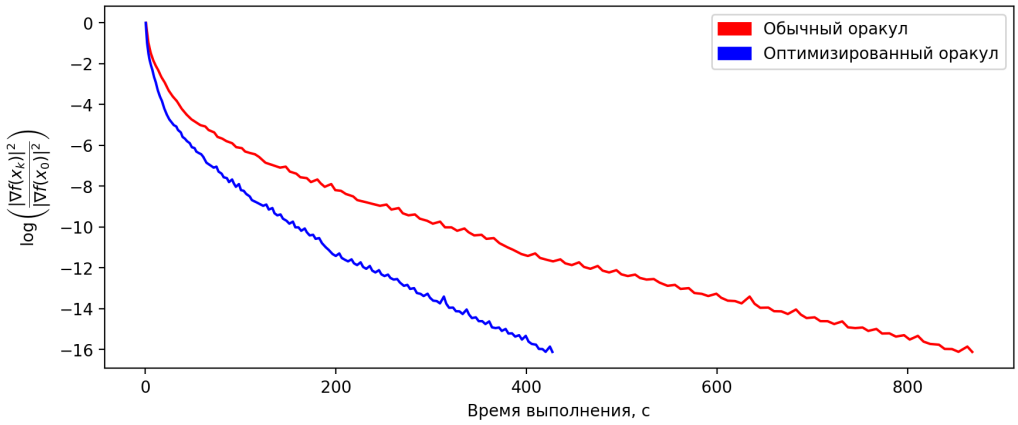

Рис. 16: Зависимость градиента функции от времени работы в логарифмическом масштабе

Вывод

На первом графике функции совпадают, поскольку оптимизация никак не влияет на количество итераций и вычисления, которые в них происходят, она только ускоряет выполнение одной итерации.

Если посмотреть на графики, видно, что алгоритм ускоряется очень значительно, примерно в два раза. Логарифмическая невязка по градиенту убывает заметно круче, чем в неоптимизированном оракуле. Характер графика, конечно, не меняется, поскольку график для оптимизированного оракула представляет из себя график для обычного оракула с растянутой(возможно, неоднородно) осью времени.

В итоге можно сказать, что для достаточно больших матриц такая оптимизация действительно имеет смысл.