

Министерство науки и высшего образования Российской Федерации
**Федеральное государственное бюджетное образовательное
учреждение высшего образования**
«Российский технологический университет» (РТУ МИРЭА)

Институт искусственного интеллекта

Отчёт по проекту

Современные методы нелинейной минимизации с ограничениями

Авторы:

Тимлид: Скворцов Владислав Игоревич (vladislavskvo@gmail.com)

Технический директор: Ахмерова Анастасия Алексеевна
(anastasia.akhmerova.03@mail.ru)

Исполнительный директор: Кулаков Никита Алексеевич (niakulaa@mail.ru)

Нургалеев Айнур Рафикович (nurgaleev.a.r@yandex.ru)

Серова Яна Игоревна (yanaserova2006@gmail.com)

Сидорова Елизавета Вячеславовна (2003sev@mail.ru)

Муравьева Александра Алексеевна (muravevaaleksandra9@gmail.com)

Нечаев Иван Александрович (ivannech2002cap@gmail.com)

Группы: КМБО-03-21, КМБО-04-21, КМБО-03-23

Преподаватель: Парfenov Денис Васильевич (promasterden@yandex.ru)

ОГЛАВЛЕНИЕ

1 Введение	2
2 Постановка задачи	3
3 Обзор литературы	4
3.1 Метод расширенного Лагранжиана с неточным степенным дополнением	4
3.2 Метод доверительной области	7
3.3 Метод последовательного квадратичного программирования и выборка значений градиента	15
3.4 Метод барьерного фильтра с линейным поиском	22
3.5 Метод исправленной адаптивной оценки моментов	29
3.6 Ускоренный многоступенчатый градиентный метод	32
4 Обзор фреймворка	41
4.1 Архитектура фреймворка	41
4.2 Логирование	45
4.2.1 Пользовательское логирование	46
4.2.2 Расширенное логирование для разработчиков	50
4.3 Модуль тестирования	54
4.3.1 База целевых функций.	54
4.3.2 Генерация ограничений	56
4.3.3 Сценарий использования (<i>bench.py</i>)	73
4.3.4 Анализ поведения алгоритмов	75
5 Численные результаты	77
5.1 Цель и методология тестирования	77
5.1.1 Описание платформы тестирования	77
5.2 Тестирование алгоритмов	80
5.2.1 Сравнение алгоритмов по качеству решения и анализу сходимости	80
5.2.2 Влияние гиперпараметров	80

1 Введение

Проект по нелинейной оптимизации с ограничениями предполагает разработку и решение задач оптимизации, где целевая функция и/или ограничения являются нелинейными. Задача состоит в нахождении оптимальных значений переменных, которые минимизируют или максимизируют целевую функцию при соблюдении множества ограничений, как равенств, так и неравенств.

В данном проекте рассматривается нелинейная целевая функция. Рассматривается случай мультимодальной целевой функции, это означает, что она имеет несколько локальных экстремумов (минимумов или максимумов) на своем графике, а не только один глобальный экстремум. В случае мультимодальной функции необходимо учитывать, что поиск оптимального решения может быть затруднен из-за наличия множества локальных оптимумов, которые могут быть не теми точками, которые минимизируют целевую функцию на всей области допустимых решений.

Ограничения (равенства и неравенства) также играют ключевую роль в задаче с мультимодальной целевой функцией. Ограничения ограничивают область поиска, однако нелинейные ограничения могут сами быть причиной усложнения задачи.

В данном проекте подразумевается проектирование фреймворка с реализацией существующих эффективных алгоритмов минимизации. В функционал фреймворка входит не только создание модели решателя, но и генерации случайных нелинейных зависимостей целевой функции и ограничений. В качестве платформы для тестирования, предусмотрен набор тестовых функций различной сложности, а также подробный визуализатор траектории решения и рельефа целевой функции. Ключевая ценность проекта заключается в разработке методов, которые не реализованы ни в одном из существующих пакетов для оптимизации.

2 Постановка задачи

Мы рассматриваем задачи нелинейной оптимизации с ограничениями следующего вида:

$$\min_{\mathbf{x} \in \mathbf{R}^n} f(\mathbf{x}) \text{ при условиях:} \quad (2.1)$$

$$c(\mathbf{x}) = 0, \quad (2.2)$$

$$g(\mathbf{x}) \leq 0. \quad (2.3)$$

Задача включает в себя ограничения двух типов:

1. **Ограничения-равенства:** $c(\mathbf{x}) = 0$, где $\mathbf{c} : \mathbf{R}^n \rightarrow \mathbf{R}^m$ – векторная функция, задающая m ограничений.
2. **Ограничения-неравенства:** $g(\mathbf{x}) \leq 0$, где $\mathbf{g} : \mathbf{R}^n \rightarrow \mathbf{R}^p$ – векторная функция, задающая p ограничений.

В некоторых алгоритмах возможно преобразование ограничений-неравенств в ограничения-равенства путем введения дополнительных переменных, называемых фиктивными (англ. slack) переменными. Это может быть выражено следующим образом:

$$g(\mathbf{x}) \leq 0 \iff \begin{cases} g(\mathbf{x}) + \mathbf{s} = 0 \\ \mathbf{s} \geq 0 \end{cases}. \quad (2.4)$$

Таким образом, благодаря неотрицательности переменной \mathbf{s} функция $g(\mathbf{x})$ принимает неположительные значения. Переменные \mathbf{s} "расширяют" исходное пространство \mathbf{R}^n , но при этом не оказывают никакого влияния на значение исследуемой функции $f(\mathbf{x})$.

Особенности задачи:

Мультимодальность целевой функции: Наличие нескольких локальных экстремумов требует применения методов, способных избегать застревания в локальных оптимумах и находить глобальное решение.

Нелинейность ограничений: Ограничения могут быть нелинейными, что усложняет задачу, так как они могут создавать невыпуклые области допустимых решений.

Сложность поиска: Задача может быть осложнена наличием большого числа переменных и ограничений, что требует использования эффективных алгоритмов для нахождения решения.

Разработка и реализация эффективных методов решения задач нелинейной оптимизации с ограничениями, включая методы доверительной области, градиентной выборки и барьерного фильтра с линейным поиском. В рамках работы предполагается создание фреймворка, который позволит генерировать случайные нелинейные целевые функции и ограничения, а также тестировать различные алгоритмы на наборе тестовых функций различной сложности.

3 Обзор литературы

3.1 Метод расширенного Лагранжиана с неточным степенным дополнением

Метод расширенного Лагранжиана с неточным степенным дополнением (англ. inexact power augmented Lagrangian method, IPALM) [13] применим к широкому классу ограниченных, не обязательно выпуклых, задач минимизации, которые включают нелинейные ограничения равенства на выпуклом множестве. Данный алгоритм подходит для задач с ограничениями в виде неравенств, которые могут быть преобразованы в равенства с помощью фиктивных переменных, как указано в (2.4). Поэтому в дальнейшем будем рассматривать задачу (2.1), как задачу с ограничениями типа "равенства", однако также существует возможность задания ограничений-неравенств, так как для этого предусмотрено соответствующее преобразование. Необходимо подчеркнуть, что в контексте данного метода в формуле (2.1) $\mathbf{x} \in X$, где X - простое замкнутое выпуклое множество, а не все векторное пространство. Иными словами, задача решается на выпуклом компакте.

Предыстория и мотивация. Метод IPALM основывается на методах, использующих штрафную функцию [7], которые преобразуют задачу (2.1) к виду:

$$\min_{\mathbf{x} \in X} (f(\mathbf{x}) + \beta \varphi(c(\mathbf{x}))), \quad (3.1.1)$$

где β - штрафной параметр, который обычно выбирается положительным ($\beta > 0$) и может изменяться в процессе итераций; φ – штрафная функция; $c(\mathbf{x})$ - вектор всех ограничений задачи (2.1), включая исходные ограничения-равенства и ограничения-неравенства после их преобразования в формат равенств с помощью фиктивных переменных (2.4).

Штрафная функция φ представляет собой меру нарушения ограничений, накладываемых на переменные \mathbf{x} . В большинстве случаев φ выбирается как квадрат Евклидовой нормы, то есть $\varphi(c(\mathbf{x})) = \frac{1}{2} \|c(\mathbf{x})\|^2$ [13]. Это позволяет штрафовать решения, которые не удовлетворяют ограничениям, тем самым направляя процесс оптимизации к допустимым решениям с помощью контроля величины штрафного параметра β . Детальное объяснение концепции штрафной функции для текущего алгоритма будет представлено в формуле (3.1.5).

Далее поэтапно описывается процесс построения и решения двойственной задачи для решения задачи (3.1.1).

Расширенный. Термин "расширенный" подразумевает добавление к целевой функции задачи (3.1.1) дополнительного члена $\langle \mathbf{y}, c(\mathbf{x}) \rangle$, в результате чего получаем функцию - расширенный Лагранжиан:

$$L_\beta(\mathbf{x}, \mathbf{y}) := f(\mathbf{x}) + \langle \mathbf{y}, c(\mathbf{x}) \rangle + \beta \varphi(c(\mathbf{x})), \quad (3.1.2)$$

где $\mathbf{y} \in \mathbb{R}^m$ – двойственные переменные (множители), $\langle \mathbf{y}, c(\mathbf{x}) \rangle$ – скалярное произведение y на $c(\mathbf{x})$.

Множители \mathbf{y} играют ключевую роль в методах оптимизации, особенно в контексте метода Лагранжа. Они позволяют учитывать ограничения, накладываемые на переменные \mathbf{x} , и помогают формализовать задачу оптимизации с учетом этих ограничений. В частности, множители \mathbf{y} могут интерпретироваться как "ценовые" параметры, которые показывают, как изменение ограничения $c(\mathbf{x})$ влияет на оптимальное значение целевой функции $f(\mathbf{x})$. Они служат связующим звеном между целевой функцией и ограничениями, позволяя находить оптимальные решения в условиях ограничений.

Использование расширенного лагранжиана имеет следующие преимущества:

1. Учет ограничений: расширенный лагранжиан позволяет явно включать ограничения в задачу оптимизации, что делает процесс поиска оптимального решения более структурированным и понятным.
2. Анализ чувствительности: множители \mathbf{y} предоставляют информацию о том, как изменения в ограничениях влияют на оптимальное значение целевой функции.
3. Упрощение вычислений: включение ограничений в лагранжиан может упростить вычисления, так как позволяет использовать методы, которые работают непосредственно с функцией, включающей ограничения, вместо того чтобы решать их отдельно.

Неточный. Каждая классическая итерация метода расширенного лагранжиана (англ. Augmented Lagrangian Method, ALM) [19] обновляет пару первично-двойственных переменных $(\mathbf{x}_k, \mathbf{y}_k)$ в два действия:

- 1) минимизируя $L_\beta(\mathbf{x}, \mathbf{y})$ относительно первичной переменной:

$$\mathbf{x}_{k+1} \in \arg \min_{\mathbf{x} \in X} L_\beta(\mathbf{x}, \mathbf{y}_k), \quad (3.1.3)$$

- 2) выполняя двойственный шаг σ ($\sigma \geq 0$) с целью увеличения значения $L_\beta(\mathbf{x}, \mathbf{y})$:

$$\mathbf{y}_{k+1} = \mathbf{y}_k + \sigma \mathbf{x}_{k+1}. \quad (3.1.4)$$

Двойственный шаг σ используется в методе расширенного Лагранжиана для обновления множителей Лагранжа. Он управляет скоростью адаптации двойственных переменных к выполнению ограничений.

Здесь k - номер текущего шага алгоритма. Начальное приближение $\mathbf{x}_0, \mathbf{y}_0$ выбирается заранее, например, как решение какой-либо вспомогательной задачи или по эвристике. Затем k увеличивается на каждой итерации, пока не выполнены критерии остановки (достаточно малый прирост решения или функции, выполнение условий стационарности или сходимости к допустимому решению).

Для решения первой подзадачи используется специальный "неточный" алгоритм вычисления \mathbf{x}_{k+1} , так как такой подход обеспечивает доказанную сходимость метода к стационарным точкам (см. главу 1.3 статьи [13]). Несмотря на то, что теоретически этот алгоритм имеет лучшую сходимость при худших условиях, на практике (см. главу 4 статьи [13]) обнаруживается, что более простые методы, вроде метода ускоренного ближайшего градиента (англ. Alternating Proximal Gradient Method, APGM) [20], не уступают ему, и более того, зачастую оказываются эффективнее.

Изучение данного "неточного" алгоритма и его сравнение с более простыми аналогами станет одной из ключевых задач в дальнейшем.

Со степенным дополнением. В методе IPALM "степенное дополнение" обеспечивается обобщением штрафной функции φ . Она принимает вид:

$$\varphi(\mathbf{x}) = \frac{1}{\nu + 1} \|\mathbf{x}\|^{\nu+1}, \nu \in (0, 1]. \quad (3.1.5)$$

где ν - степенной параметр, определяющий форму функции φ (см. главу 1.1 статьи [13]). Эта функция используется в штрафном члене $\beta\varphi(c(\mathbf{x}))$ расширенного Лагранжиана (3.1.2), что позволяет параметру ν управлять тем, как метод учитывает нарушения ограничений $c(\mathbf{x})$: при $\nu = 1$ получается классический метод неточного расширенного лагранжиана (англ. inexact augmented Lagrangian method, iALM) (см. главу 1.1 статьи [13]), при любых других значениях степени ν название метода принято дополнять описанием "power". Приведенные в [13] численные эксперименты доказывают, что значение параметра ν оказывает влияние на сходимость метода.

\mathbf{x} — вектор переменных задачи оптимизации, содержащий значения искомых параметров. Его размерность соответствует числу оптимизируемых параметров. В контексте метода IPALM, \mathbf{x} представляет текущее приближение к оптимальному решению. $\|\mathbf{x}\|$ - корень из суммы квадратов координат вектора \mathbf{x} .

На Рис. 1, заимствованном из [13], представлена зависимость неоптимальности решения (см. главу 4.3 статьи [13]) некоторой задачи от количества градиентных вызовов, которые требуются IPALM для достижения стационарной точки при различных значениях степени. Под неоптимальностью решения в данном контексте понимается абсолютное отклонение значения целевой функции в текущей точке \mathbf{x} от оптимального значения f^* , т.е. $|f(\mathbf{x}) - f^*|$. Рисунок 1 подтверждает, что для некоторых задач (см. главу 4.3 статьи [13]) неоптимальность решения неуклонно уменьшается с уменьшением значения степени.

Оценка погрешности и выбор числа количества итераций. При выполнении некоторых предположений (см. главу 1.3-4 статьи [13]) x_{k+1} является $\left(\frac{Q_f}{\beta_1 \omega^{k-1}}, \frac{Q_c}{\beta_1^{\frac{1}{\nu}} \omega^{\frac{k-1}{\nu}}} \right)$ – стационарной точкой, где

$$Q_f := \boldsymbol{\lambda} + J_{cmax} \sigma_1 \frac{\nabla f_{max} + J_{cmax} y_{max} + \varepsilon_1}{\nu}, \quad (3.1.6.1)$$

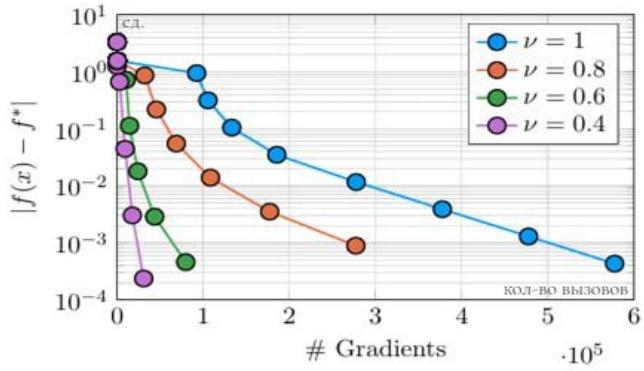


Рис. 1: Зависимость неоптимальности решения задачи на собственные значения (англ. Generalized Eigenvalue Problem, GEVP) от количества необходимых градиентных вызовов при разных значениях ν (см. главу 4 статьи [13]).

$$Q_c := \left(\frac{\nabla f_{max} + J_{cmax} y_{max} + \varepsilon_1}{\nu} \right)^{\frac{1}{\nu}}. \quad (3.1.6.2)$$

Здесь Q_f – оценка первичного остатка оптимизации, отражающий отклонение целевой функции, Q_c – оценка на двойственный остаток, связанный с выполнением ограничений, λ – текущие множители Лагранжа, σ_1 – параметр двойственного шага, ε_1 – допустимая точность при вычислении двойственных переменных. σ_1, λ задаются заранее, как гиперпараметры метода.

В формулах (3.1.6.1) и (3.1.6.2) используются следующие выражения:

$$J_{cmax} := \max_{\mathbf{x} \in X} \|J_c(\mathbf{x})\| < \infty, \quad (3.1.7.1)$$

$$\nabla f_{max} := \max_{\mathbf{x} \in X} \|\nabla f(\mathbf{x})\| < \infty, \quad (3.1.7.2)$$

где $J_c(\mathbf{x})$ – матрица Якоби отображения $c(\mathbf{x})$, k – количество итераций, ω – скорость увеличения штрафного параметра β на каждой итерации.

Таким образом, на каждой итерации можно вычислить неоптимальность задачи и величину нарушения ограничений, которые используются для анализа сходимости алгоритма к стационарной точке первого порядка (степень сходимости и её доказательство см. в [13], теорема 2).

3.2 Метод доверительной области

Методы доверительных областей (англ. Trust-Region Method) [2], [3] широко применяются для решения задач нелинейной оптимизации, включая задачи с ограничениями. Их ключевая идея заключается в построении квадратичной модели целевой функции в пределах локальной области, называемой доверительной областью. Эта область динамически регулируется на каждой итерации в зависимости от успеха шага, что делает методы доверительных областей устойчивыми даже для задач со сложной структурой ограничений.

Задачи с ограничениями-равенствами требуют поиска решений в подпространстве, строго удовлетворяющем равенствам, что достигается через проекцию шагов в допустимое под-

пространство. Для задач с ограничениями-неравенствами необходимо контролировать положительность фиктивных переменных (2.4) и управлять границами области при помощи барьерных методов.

Ограничения-равенства

Оптимизируемая задача записывается в виде (2.1) с нелинейными ограничениями вида (2.2). Алгоритм решает задачу для случая $n \geq m$.

Лагранжиан задачи в общем виде записывается следующим образом:

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) - \boldsymbol{\lambda}^T \mathbf{c}(\mathbf{x}), \quad (3.2.1)$$

где $\boldsymbol{\lambda}$ — вектор множителей Лагранжа.

Для начала введем определение доверительной области. Доверительная область — это ограниченная область в окрестности текущей точки \mathbf{x}_k , внутри которой приближенная модель целевой функции считается достаточно точной для поиска следующего шага.

$$\mathcal{B}_k = \{\mathbf{d} \in \mathbb{R}^n : \|\mathbf{d}\| \leq \Delta_k\},$$

где \mathbf{d} — пробный шаг, $\|\cdot\|$ — евклидова норма (или другая выбранная норма, но в данном алгоритме она именно евклидова), $\Delta_k > 0$ — радиус доверительной области на итерации k .

Центром доверительной области является текущая точка \mathbf{x}_k , в которой строится приближенная модель функции. Шаг \mathbf{d} определяется относительно этой точки и ограничивается областью доверия \mathcal{B}_k , в пределах которой мы "доверяем" построенной модели.

Основной алгоритм базируется на методе доверительных областей Берда [8] и Омоджокуна [9]. Его суть заключается в разбиении общей задачи на две подзадачи: вертикальную и горизонтальную, которые решаются итерационно.

Цель **вертикальной подзадачи** заключается в нахождении такого шага \mathbf{v}_k , который минимизирует нарушение ограничений (следует отметить, что шаг \mathbf{v}_k должен принадлежать образу A_k^T , что обеспечивает его ортогональность горизонтальному шагу, определенному в ядре A_k):

$$\min_{\mathbf{v} \in \mathbb{R}^n} \left(\mathbf{c}_k^T A_k^T \mathbf{v} + \frac{1}{2} \mathbf{v}^T A_k A_k^T \mathbf{v} \right), \quad \text{при } \|\mathbf{v}\| \leq \zeta \Delta_k, \quad (3.2.2)$$

где A_k — матрица градиентов ограничений, \mathbf{c}_k — текущие значения ограничений, Δ_k — радиус доверительной области, $\zeta \in (0, 1)$ — параметр релаксации, который подбирается эмпирически из диапазона $(0, 1)$ и обычно задается как константа.

Для решения вертикальной подзадачи используются метод додлега (см. раздел 3 в [2]) или метод Стейхауга [21]. Метод додлега применяется, когда матрица Гессе $B_k = A_k A_k^T$ положительно определена, метод Стейхауга используется, когда матрица Гессе B_k не положительно определена, то есть содержит отрицательную кривизну. Оба метода гарантируют нахождение вертикального шага, ортогональному горизонтальному шагу (подробнее см. раздел 4.1 в [2]).

- **Метод дogleга.**

Метод дogleга основан на поиске оптимального шага вдоль определенной траектории, называемой дogleг-путем (англ. dogleg path), которая состоит из комбинации двух направлений:

1. От начальной точки $\mathbf{v} = 0$ до точки Коши (\mathbf{v}_{cp}), которая соответствует направлению наилучшего уменьшения нарушения ограничений

$$\mathbf{v}_{cp} = -\alpha A_k \mathbf{c}_k, \quad \alpha = \min \left(\frac{\zeta \Delta_k}{\|A_k \mathbf{c}_k\|}, \frac{\|A_k \mathbf{c}_k\|^2}{(A_k \mathbf{c}_k)^T A_k A_k^T (A_k \mathbf{c}_k)} \right).$$

Здесь параметр α определяет длину шага в противоположном направлении градиента по ограничению. Этот параметр выбирается таким образом, чтобы шаг не выходил за пределы доверительной области и обеспечивал прогресс в минимизации.

Приведем подробный вывод формулы точки Коши.

Градиент функции (3.2.2):

$$A_k \mathbf{c}_k + A_k A_k^T \mathbf{v}.$$

В точке $\mathbf{v} = 0$ он равен $A_k \mathbf{c}_k$. Мы минимизируем (3.2.2) вдоль направления наискорейшего убывания, то есть вдоль направления $-A_k \mathbf{c}_k$, подставим $\mathbf{v} = -\alpha A_k \mathbf{c}_k$ в (3.2.2), тогда получим следующее:

$$-\alpha \mathbf{c}_k^T A_k^T A_k \mathbf{c}_k + \frac{1}{2} \alpha^2 A_k^T \mathbf{c}_k^T A_k A_k^T A_k \mathbf{c}_k = -\alpha \|A_k \mathbf{c}_k\|^2 + \frac{1}{2} \alpha^2 (A_k \mathbf{c}_k)^T A_k A_k^T (A_k \mathbf{c}_k).$$

Минимизируем по $\alpha > 0$ (найдем производную по α и приравняем ее к 0):

$$-\|A_k \mathbf{c}_k\|^2 + \alpha (A_k \mathbf{c}_k)^T A_k A_k^T (A_k \mathbf{c}_k) = 0.$$

Отсюда:

$$\alpha_* = \frac{\|A_k \mathbf{c}_k\|^2}{(A_k \mathbf{c}_k)^T A_k A_k^T (A_k \mathbf{c}_k)}.$$

Тогда с учетом ограничения (3.2.2) получим:

$$\alpha = \begin{cases} \frac{\|A_k \mathbf{c}_k\|^2}{(A_k \mathbf{c}_k)^T A_k A_k^T (A_k \mathbf{c}_k)}, & \text{если } (A_k \mathbf{c}_k)^T A_k A_k^T (A_k \mathbf{c}_k) > 0, \frac{\|A_k \mathbf{c}_k\|^3}{(A_k \mathbf{c}_k)^T A_k A_k^T (A_k \mathbf{c}_k)} \leq \zeta \Delta_k, \\ \frac{\zeta \Delta_k}{\|A_k \mathbf{c}_k\|} & \text{иначе.} \end{cases}$$

Если $\|A_k \mathbf{c}_k\| = 0$, то берем $\mathbf{v}_{cp} = 0$.

2. Ньютоновский шаг (\mathbf{v}_n), который минимизирует квадратичную модель, но может выходить за пределы доверительной области

$$\mathbf{v}_n = -A_k (A_k^T A_k)^{-1} \mathbf{c}_k.$$

Он получается путем приравнивания градиента функции (3.2.2) к нулю:

$$A_k \mathbf{c}_k + A_k A_k^T \mathbf{v} = 0.$$

Решение данного уравнения и есть шаг Ньютона.

Итоговый шаг комбинирует \mathbf{v}_{cp} и \mathbf{v}_n в пределах доверительной области.

Далее приведен псевдокод алгоритма:

Algorithm 1 Метод доглела

Require: Матрица градиентов ограничений A_k , текущие значения ограничений \mathbf{c}_k , радиус доверительной области Δ_k , параметр релаксации $\zeta \in (0, 1)$;

1: Вычислить точку Коши:

$$\mathbf{v}_{cp} = -\alpha A_k \mathbf{c}_k, \quad \text{где } \alpha = \min \left(\frac{\zeta \Delta_k}{\|A_k \mathbf{c}_k\|}, \frac{\|A_k \mathbf{c}_k\|^2}{(A_k \mathbf{c}_k)^T A_k A_k^T (A_k \mathbf{c}_k)} \right);$$

2: Вычислить шаг Ньютона:

$$\mathbf{v}_n = -A_k (A_k^T A_k)^{-1} \mathbf{c}_k;$$

3: **if** $\|\mathbf{v}_n\| \leq \zeta \Delta_k$ **then**

4: $\mathbf{v}_{\text{dogleg}} = \mathbf{v}_n$;

5: **else**

6: Найти точку пересечения отрезка $[v_{cp}, v_n]$ с границей доверительной области:

$$\mathbf{v}_{\text{dogleg}} = \mathbf{v}_{cp} + \tau \cdot (\mathbf{v}_n - \mathbf{v}_{cp}),$$

где $\tau \in (0, 1)$ — решение уравнения:

$$\|\mathbf{v}_{cp} + \tau \cdot (\mathbf{v}_n - \mathbf{v}_{cp})\| = \zeta \Delta_k.$$

7: **end if**

8: **return** $\mathbf{v}_{\text{dogleg}}$.

Рис. 2: Псевдокод метода доглела

• **Метод Стейхауга.**

Метод Стейхауга представляет собой адаптацию метода сопряжённых градиентов (англ. conjugate gradient, CG) [11] для решения задачи доверительной области, возникающей при оптимизации с ограничениями равенства. Его ключевая идея заключается в нахождении шага, который минимизирует квадратичную модель целевой функции, оставаясь внутри доверительной области радиуса Δ_k . Итерации метода выполняются до тех пор, пока одно из следующих условий не будет выполнено:

1. Достигение границы доверительной области: шаг ограничивается сферой радиуса $\zeta \Delta_k$. Это ограничение гарантирует, что шаг остаётся внутри доверительной области:

$$\|\mathbf{v}\| = \zeta \Delta_k.$$

2. Отрицательная кривизна: если направление поиска \mathbf{p}_k становится некорректным из-за отрицательной кривизны (то есть $\mathbf{p}_k^T B_k \mathbf{p}_k \leq 0$, где $B_k = A_k A_k^T$), то итерации останавливаются, и шаг корректируется вдоль текущего направления \mathbf{p}_k до пересечения с границей доверительной области.

3. Сходимость по остатку: если норма остатка модели \mathbf{r}_k становится достаточно малой ($\|\mathbf{r}_k\| < \varepsilon$), то достигнут минимум квадратичной модели, и алгоритм завершает работу.

Далее приведен псевдокод алгоритма:

Algorithm 2 Метод Стейхауга

Require: Матрица градиентов ограничений A_k , текущие значения ограничений \mathbf{c}_k , радиус доверительной области Δ_k , параметр релаксации $\zeta \in (0, 1)$, точность $\varepsilon > 0$;

Инициализация: $\mathbf{v}_0 = \mathbf{0}$, $\mathbf{r}_0 = -A_k \mathbf{c}_k$, $\mathbf{p}_0 = \mathbf{r}_0$;

2: **if** $\|\mathbf{r}_0\| < \varepsilon$ **then return** \mathbf{v}_0 ;

end if

4: **for** $j = 0, 1, 2, \dots$: **do**

if $\mathbf{p}_j^T A_k A_k^T \mathbf{p}_j \leq 0$ **then**

 Найти τ , чтобы $\mathbf{v} = \mathbf{v}_j + \tau \mathbf{p}_j$ минимизировал $\phi(\mathbf{v})$ и удовлетворял $\|\mathbf{v}\| = \zeta \Delta_k$;

return \mathbf{v} ;

end if

 Вычислить шаг:

$$\alpha_j = \frac{\mathbf{r}_j^T \mathbf{r}_j}{\mathbf{p}_j^T A_k A_k^T \mathbf{p}_j};$$

10: Обновить шаг:

$$\mathbf{v}_{j+1} = \mathbf{v}_j + \alpha_j \mathbf{p}_j;$$

if $\|\mathbf{v}_{j+1}\| \geq \zeta \Delta_k$ **then**

 Найти $\tau \geq 0$, чтобы $\|\mathbf{v}_j + \tau \mathbf{p}_j\| = \zeta \Delta_k$;

return $\mathbf{v}_j + \tau \mathbf{p}_j$;

end if

 Обновить остаток:

$$\mathbf{r}_{j+1} = \mathbf{r}_j - \alpha_j A_k A_k^T \mathbf{p}_j;$$

16: **if** $\|\mathbf{r}_{j+1}\| / \|\mathbf{r}_0\| < \varepsilon$ **then return** \mathbf{v}_{j+1} ;

end if

18: Обновить направление:

$$\beta_{j+1} = \frac{\mathbf{r}_{j+1}^T \mathbf{r}_{j+1}}{\mathbf{r}_j^T \mathbf{r}_j}, \quad \mathbf{p}_{j+1} = \mathbf{r}_{j+1} + \beta_{j+1} \mathbf{p}_j;$$

end for

Рис. 3: Псевдокод метода Стейхауга

Цель **горизонтальной подзадачи** заключается в нахождении шага \mathbf{u}_k для оптимизации целевой функции $f(\mathbf{x})$, оставаясь в пространстве, ортогональном ограничениям:

$$\min_{\mathbf{u} \in \mathbb{R}^{n-m}} (\mathbf{t}_k + W_k \mathbf{v}_k)^T Z_k \mathbf{u} + \frac{1}{2} \mathbf{u}^T Z_k^T W_k Z_k \mathbf{u}, \quad \|Z_k \mathbf{u}\| \leq \sqrt{\Delta_k^2 - \|\mathbf{v}_k\|^2}, \quad (3.2.3)$$

где $\mathbf{t}_k = \nabla_x \mathcal{L}(\mathbf{x}_k, \boldsymbol{\lambda}_k)$ — градиент лагранжиана, $W_k = \nabla_{xx}^2 \mathcal{L}(\mathbf{x}_k, \boldsymbol{\lambda}_k)$ — гессиан лагранжиана или его аппроксимация, Z_k — базис касательного подпространства, удовлетворяющий $A_k^T Z_k = 0$.

Если вычисление второй производной лагранжиана невозможно или слишком затратно, вместо матрицы Гессе строится приближенный гессиан при помощи метода Бродена — Флетчера — Гольдфарба — Шанно с ограниченной памятью (англ. limited memory Broyden

— Fletcher — Goldfarb — Shanno algorithm, L-BFGS) [10]:

$$W_{k+1} = W_k - \frac{W_k \mathbf{d}_k \mathbf{d}_k^T W_k}{\mathbf{d}_k^T W_k \mathbf{d}_k} + \frac{\mathbf{y}_k \mathbf{y}_k^T}{\mathbf{y}_k^T \mathbf{d}_k}, \quad (3.2.4)$$

где $\mathbf{d}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$, $\mathbf{y}_k = (\mathbf{t}_{k+1} - A_{k+1} \boldsymbol{\lambda}_{k+1}) - (\mathbf{t}_k - A_k \boldsymbol{\lambda}_k)$.

В методе L-BFGS полная матрица W_k не хранится. Вместо этого используется компактное представление на основе последних t пар векторов $(\mathbf{d}_k, \mathbf{y}_k)$, что существенно снижает требования к памяти. Более подробно метод описан в [10].

Итоговый шаг \mathbf{d}_k после вертикального (3.2.2) и горизонтального (3.2.3) шага:

$$\mathbf{d}_k = \mathbf{v}_k + Z_k \mathbf{u}_k. \quad (3.2.5)$$

Функция оценки объединяет целевую функцию $f(\mathbf{x})$ и меру выполнения ограничений $\mathbf{c}(\mathbf{x}) = 0$, она помогает определить, насколько успешен шаг \mathbf{d}_k , предложенный для следующей итерации. Общий вид функции:

$$\phi(\mathbf{x}, \mu) = f(\mathbf{x}) + \mu \|\mathbf{c}(\mathbf{x})\|,$$

где $\mu > 0$ — параметр штрафа, который изначально выбирается эмпирически (обычно $\mu = 1$), далее параметр обновляется по правилу $\mu_{k+1} = \mu^+$, где μ^+ задается формулой (3.2.7).

Шаг \mathbf{d}_k принимается, если он улучшает функцию оценки:

$$\phi(\mathbf{x}_k + \mathbf{d}_k, \mu) < \phi(\mathbf{x}_k, \mu).$$

Размер доверительной области Δ_k обновляется в зависимости от отношения фактического и прогнозируемого уменьшения целевой функции:

$$\rho_k = \frac{\text{Фактическое уменьшение}}{\text{Прогнозируемое уменьшение}} = \frac{f(\mathbf{x}_k) - f(\mathbf{x}_k + \mathbf{d}_k) + \mu^+(\|\mathbf{c}(\mathbf{x}_k)\| - \|\mathbf{c}(\mathbf{x}_k + \mathbf{d}_k)\|)}{-\mathbf{d}_k^T \mathbf{t}_k - \frac{1}{2} \mathbf{d}_k^T W_k \mathbf{d}_k + \mu(\|\mathbf{c}_k\| - \|A_k^T \mathbf{v}_k + \mathbf{c}_k\|)}, \quad (3.2.6)$$

где μ^+ определяется как:

$$\mu^+ = \max \left(\mu_k, 0.1 + \frac{\mathbf{d}_k^T \mathbf{t}_k + \frac{1}{2} \mathbf{d}_k^T W_k \mathbf{d}_k}{\|\mathbf{c}_k\| - \|A_k^T \mathbf{v}_k + \mathbf{c}_k\|} \right). \quad (3.2.7)$$

Если $\rho \geq 0.9$, увеличиваем Δ_k : $\Delta_{k+1} = \max(10\|\mathbf{d}_k\|, \Delta_k)$.

Если $0.3 \leq \rho < 0.9$, слабо увеличиваем Δ_k : $\Delta_{k+1} = \max(2\|\mathbf{d}_k\|, \Delta_k)$.

Если $\rho < 0.3$, уменьшаем: $\Delta_{k+1} = 0.5\|\mathbf{d}_k\|$.

Ограничения-неравенства

Оптимизируемая задача записывается в виде (2.1) с нелинейными ограничениями вида (2.3). Алгоритм основан на **барьерном методе**, где каждую задачу преобразуют с использованием барьерной функции, которая гарантирует наличие решений в допустимой области:

$$\min_{\mathbf{x}, \mathbf{s}} f(\mathbf{x}) - \mu \sum_{i=1}^m \ln(s_i), \quad \text{при условиях } \mathbf{c}(\mathbf{x}) = 0, \mathbf{g}(\mathbf{x}) + \mathbf{s} = 0, \quad (3.2.8)$$

где $s_i > 0$ — фиктивные переменные (slack variables), а μ — параметр барьера, который изначально обычно задается достаточно большим числом, далее стремится к 0, в нашем подходе $\mu_{k+1} = \theta\mu_k$, $\theta \in (0, 1)$.

Фиктивные переменные вводятся в задачах оптимизации с ограничениями типа неравенств, чтобы преобразовать их в эквивалентные ограничения-равенства (2.4).

Таким образом, с учетом барьерной функции лагранжиан для задачи принимает следующий вид:

$$\mathcal{L}(\mathbf{x}, \mathbf{s}, \boldsymbol{\lambda}_c, \boldsymbol{\lambda}_g) = f(\mathbf{x}) - \mu \sum_{i=1}^m \ln(s_i) + \boldsymbol{\lambda}_c^T \mathbf{c}(\mathbf{x}) + \boldsymbol{\lambda}_g^T (\mathbf{g}(\mathbf{x}) + \mathbf{s}), \quad (3.2.9)$$

где $\boldsymbol{\lambda}_c, \boldsymbol{\lambda}_g$ - множители, связанные с ограничениями-равенствами и неравенствами.

Условия Каруша-Куна-Таккера (ККТ) используются для проверки оптимальности решения задачи (см. раздел 3.1 [3]), они формализуют требования к точке, которая может быть локальным экстремумом, обеспечивая согласованность между градиентами целевой функции и ограничений:

$$\nabla f(\mathbf{x}) + A_c(\mathbf{x})\boldsymbol{\lambda}_c + A_g(\mathbf{x})\boldsymbol{\lambda}_g = 0, \quad (3.2.10a)$$

$$S\boldsymbol{\lambda}_g - \mu\mathbf{e} = 0, \quad (3.2.10b)$$

$$\mathbf{c}(\mathbf{x}) = 0, \quad (3.2.10c)$$

$$\mathbf{g}(\mathbf{x}) + \mathbf{s} = 0, \quad (3.2.10d)$$

где $S = \text{diag}(s^1, \dots, s^m)$, $\mathbf{e} = [1, \dots, 1]^T$,

$A_c(\mathbf{x}) = [\nabla c^1(\mathbf{x}), \dots, \nabla c^t(\mathbf{x})]$, $A_g(\mathbf{x}) = [\nabla g^1(\mathbf{x}), \dots, \nabla g^m(\mathbf{x})]$.

При помощи **вертикального шага** находим направление, которое минимизирует нарушения ограничений:

$$\mathbf{v} = (\mathbf{v}_x, \mathbf{v}_s) = \min_{\mathbf{v}_x, \mathbf{v}_s} \|A_c^T \mathbf{v}_x + \mathbf{c}\|_2^2 + \|A_g^T \mathbf{v}_x + \mathbf{v}_s + \mathbf{g} + \mathbf{s}\|_2^2, \quad (3.2.11)$$

при условиях:

$$\|\mathbf{v}_x, S^{-1} \mathbf{v}_s\|_2 \leq \zeta \Delta, \quad \mathbf{v}_s \geq -\tau \mathbf{s},$$

где $\zeta \in (0, 1)$ — параметр релаксации, который подбирается эмпирически из диапазона $(0, 1)$ и обычно задается как константа, $\tau \in (0, 1)$ - также константа, которая подбирается эмпирически.

Вертикальный шаг вычисляется также при помощи методов додлега (см. раздел 3.2 в [3]) или Стейхауга [21], основная идея которых описана ранее (см. Рис. 2, Рис. 3). В случае ограничений типа неравенств для вычисления вертикального шага предпочтительнее использовать метод додлега, так как он лучше справляется с плохо обусловленными системами, тем не менее метод Стейхауга также может быть использован. В методе додлега иначе вычисляются точка Коши и шаг Ньютона:

$$\tilde{\mathbf{v}}^{CP} = \begin{bmatrix} \mathbf{v}_x^{CP} \\ \tilde{\mathbf{v}}_s^{CP} \end{bmatrix} = -\alpha \hat{A} \begin{bmatrix} \mathbf{c} \\ \mathbf{g} + \mathbf{s} \end{bmatrix}, \quad (3.2.12)$$

где:

$$\alpha = \frac{\left\| \hat{A} \begin{bmatrix} \mathbf{c} \\ \mathbf{g} + \mathbf{s} \end{bmatrix} \right\|_2^2}{[\mathbf{c}^T \mathbf{g}^T + \mathbf{s}^T] (\hat{A}^T \hat{A})^2 \begin{bmatrix} \mathbf{c} \\ \mathbf{g} + \mathbf{s} \end{bmatrix}}.$$

$$\tilde{\mathbf{v}}^N = \begin{bmatrix} \mathbf{v}_x^N \\ \tilde{\mathbf{v}}_s^N \end{bmatrix} = -\hat{A} (\hat{A}^T \hat{A})^{-1} \begin{bmatrix} \mathbf{h} \\ \mathbf{g} + \mathbf{s} \end{bmatrix}. \quad (3.2.13)$$

При помощи **горизонтального шага** оптимизируем функцию, учитывая, что направление находится в касательной плоскости к ограничениям:

$$W = \min_{W_x, W_s} \nabla f^T W_x + -\mu \mathbf{e}^T S^{-1} W_s + \frac{1}{2} \left(W_x^T \nabla_{xx}^2 \mathcal{L} W_x + W_s^T \sum W_s \right), \quad (3.2.14)$$

при условиях:

$$A_c^T W_x = A_c^T \mathbf{v}_x, \quad A_g^T W_x + W_s = A_g^T \mathbf{v}_x + \mathbf{v}_s, \quad \|W_x, S^{-1} W_s\| \leq \Delta, \quad W_s \geq -\tau \mathbf{s}.$$

Итоговый шаг \mathbf{d}_k после вертикального (3.2.11) и горизонтального (3.2.14) шага:

$$\mathbf{d}_k = \mathbf{v}_k + W_k. \quad (3.2.15)$$

Если вертикальная составляющая мала по сравнению с горизонтальной составляющей шага, выполняется коррекция второго порядка (англ. second order correction, SOC). Далее приведен псевдокод алгоритма:

Algorithm 3 Коррекция второго порядка

1: **if** $\|\mathbf{v}_k\| \leq 0.1 \|\mathbf{w}_k\|$ **then**

2: Вычислить \mathbf{y}_k как:

$$\mathbf{y}_k = \hat{A} (\hat{A}^T \hat{A})^{-1} \begin{bmatrix} h(\mathbf{x}_k + d\mathbf{x}) \\ g(\mathbf{x}_k + d\mathbf{x}) + \mathbf{s}_k + d\mathbf{s} \end{bmatrix}, \quad \hat{A}_k = \begin{bmatrix} A_h(\mathbf{x}_k) & A_g(\mathbf{x}_k) \\ 0 & S_k \end{bmatrix};$$

3: **else**

4: Установить $\mathbf{y}_k = \mathbf{0}$;

5: **end if**

Рис. 4: Псевдокод коррекции второго порядка [3]

В качестве **функции оценки** используется штрафная функция:

$$\phi(\mathbf{x}, \mathbf{s}, \nu) = f(\mathbf{x}) - \mu \sum_{i=1}^m \ln(s_i) + \nu \left\| \begin{bmatrix} \mathbf{c}(\mathbf{x}) \\ \mathbf{g}(\mathbf{x}) + \mathbf{s} \end{bmatrix} \right\|_2, \quad (3.2.16)$$

где ν — коэффициент штрафа (см. раздел 3.3 в [3]).

Если шаг улучшает эту функцию, он принимается, иначе радиус доверительной области уменьшается вдвое.

Критерием остановки алгоритма служит мера оптимальности, которая используется для оценки того, насколько текущее решение (\mathbf{x}, \mathbf{s}) близко к выполнению условий ККТ:

$$E(\mathbf{x}, \mathbf{s}, \mu) = \max \{ \|\nabla f(\mathbf{x}) + A_c(\mathbf{x})\boldsymbol{\lambda}_c + A_g(\mathbf{x})\boldsymbol{\lambda}_g\|_\infty, \|S\boldsymbol{\lambda}_g - \mu\mathbf{e}\|_\infty, \|\mathbf{c}(\mathbf{x})\|_\infty, \|\mathbf{g}(\mathbf{x}) + \mathbf{s}\|_\infty \}, \quad (3.2.17)$$

если $E(\mathbf{x}, \mathbf{s}, \mu) \leq \epsilon_\mu$, алгоритм останавливается, если условие не выполнено, параметры обновляются (см. Algorithm I [3]).

3.3 Метод последовательного квадратичного программирования и выборка значений градиента

Предложенный метод последовательного квадратичного программирования (англ. Sequential quadratic programming, SQP) [12] предназначен для решения задач нелинейного программирования функций типа (2.1) с ограничениями вида (2.2). Основная идея: аппроксимировать исходную нелинейную задачу последовательностью квадратичных подзадач, которые проще решать.

На каждом шаге формируется задача квадратичного программирования:

$$\min_d \left(\frac{1}{2} \mathbf{d}^T H_k \mathbf{d} + \nabla f(\mathbf{x}_k)^T \mathbf{d} \right). \quad (3.3.1)$$

Важно отметить, что данная формула - это ничто иное как разложение целевой функции рядом Тейлора до второго порядка без учета константного значения $f(\mathbf{x}_k)$, поскольку оно никак не влияет на точку минимума:

$$f(\mathbf{x}_k + \mathbf{d}) \approx f(\mathbf{x}_k) + \nabla f(\mathbf{x}_k)^T \mathbf{d} + \frac{1}{2} \mathbf{d}^T H_k \mathbf{d}. \quad (3.3.1a)$$

Для учета ограничений вводится функция Лагранжа:

$$L(\mathbf{x}, \boldsymbol{\lambda}, \mu) = f(\mathbf{x}) + \boldsymbol{\lambda}^T \mathbf{c}(\mathbf{x}) + \mu^T \mathbf{h}(\mathbf{x}), \quad (3.3.2)$$

далее на каждой итерации вычисляется гессиан Лагранжа H_k :

$$H_k = \nabla^2 f(\mathbf{x}_k) + \sum_{i=1}^m \boldsymbol{\lambda}_{k,i} \nabla^2 \mathbf{c}_i(\mathbf{x}_k) + \sum_{i=1}^q \mu_{k,i} \nabla^2 \mathbf{h}_i(\mathbf{x}_k). \quad (3.3.2a)$$

Это выражение берется из второго дифференциала функции Лагранжа. Оно является «ядром» метода, поскольку учитывает влияние ограничений на искомое направление оптимизации. После решения задачи (3.3.1) обновляется текущая точка:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{d}_k. \quad (3.3.3)$$

Алгоритм последовательного квадратичного программирования состоит из следующих шагов:

1. Инициализация: задать начальную точку x_0 , начальную аппроксимацию H_0 , параметры регуляризации и точности.

2. Решение задачи квадратичного программирования: на текущей итерации k решить задачу квадратичного программирования (3.3.1).
3. Обновление точки: обновить текущее приближение (3.3.3) и обновить множители Лагранжа (решение задачи ККТ) [3].
4. Обновление аппроксимации Гессе: использовать метод BFGS [10] или его модификацию для обновления H_k , если вычисление точного гессиана затруднительно (в случае большого кол-ва переменных, ограничений):

$$H_{k+1} = H_k + \Delta H_k. \quad (3.3.4)$$

5. Проверка условий остановки: проверить условия:

$$\|\nabla f(\mathbf{x}_{k+1}) + \boldsymbol{\lambda}_k^T \nabla c(\mathbf{x}_{k+1}) + \boldsymbol{\mu}_k^T \nabla h(\mathbf{x}_{k+1})\| \leq \epsilon, \quad (3.3.5)$$

где $\boldsymbol{\lambda}_k$ и $\boldsymbol{\mu}_k$ — множители Лагранжа, обновляются на каждой итерации.

6. Если условия выполнены завершить алгоритм или вернуться к шагу 2.

Algorithm 4 SQP

1: **Инициализация:**2: Задать начальное приближение \mathbf{x}_0 , начальную аппроксимацию гессиана H_0 , начальные множители Лагранжа $\boldsymbol{\lambda}_0$ и $\boldsymbol{\mu}_0$, параметры регуляризации и точности.3: Пусть $k = 0$ и ϵ — заданная точность.4: **while** не выполнено условие остановки **do**5: **Решение задачи QR:**

6: Решить задачу квадратичного программирования:

$$\min_{\mathbf{d}} \left(\frac{1}{2} \mathbf{d}^T H_k \mathbf{d} + \nabla f(\mathbf{x}_k)^T \mathbf{d} \right).$$

7: **Обновление точки:**8: $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{d}_k$.9: **Обновление гессиана:**10: $H_{k+1} = H_k + \Delta H_k$, используя метод BFGS или модификацию.11: **Обновление множителей Лагранжа:**12: $\boldsymbol{\lambda}_{k+1} = \boldsymbol{\lambda}_k + \Delta \boldsymbol{\lambda}_k$,13: $\boldsymbol{\mu}_{k+1} = \boldsymbol{\mu}_k + \Delta \boldsymbol{\mu}_k$,14: **Проверка условия остановки:**

15: Вычислить:

$$\|\nabla f(\mathbf{x}_{k+1}) + \boldsymbol{\lambda}_{k+1}^T \nabla c(\mathbf{x}_{k+1}) + \boldsymbol{\mu}_{k+1}^T \nabla h(\mathbf{x}_{k+1})\|.$$

16: **if** условие сходимости выполнено **then**

17: Завершить алгоритм.

18: **else**19: Увеличить k на 1 и перейти к следующей итерации.20: **end if**21: **end while**

Рис. 5: Псевдокод метода последовательного квадратичного программирования

Алгоритм SQP обладает следующими свойствами:

- Для задач с *выпуклыми* ограничениями и *выпуклой* функцией алгоритм сходится к глобальному минимуму (по теореме о выпуклых функциях локальный минимум является абсолютным) [12].

В таких задачах метод SQP и другие методы оптимизации гарантированно сходятся к глобальному минимуму, при условии, что начальная точка находится внутри области допустимых решений и выполнены условия сходимости.

- Для невыпуклых задач возможно нахождение локального минимума.

В случае *невыпуклых задач* функция $f(\mathbf{x})$ или ограничения могут быть такими, что задача имеет несколько локальных минимумов. Алгоритм SQP, как и другие методы оптимизации, может столкнуться с тем, что:

- при отсутствии выпуклости функция может иметь несколько локальных минимумов.
- в этом случае алгоритм может сойтись к *локальному минимуму*, если начальная точка или шаги метода попадут в такую область пространства.

Это естественно, поскольку для невыпуклых задач оптимизация не может гарантировать нахождение глобального минимума, и результат может зависеть от начальной точки и шагов алгоритма.

Метод выборки значений градиента (англ. Gradient Sampling, GS) [4] применяется для задач оптимизации функций, которые могут быть недифференцируемы в некоторых точках. Основная идея заключается в аппроксимации субградиента функции в заданной точке путём вычисления направленных градиентов в окрестности этой точки, то есть смотрим изменения функции по всем направлениям.

Пусть $f(\mathbf{x})$ - заданная целевая функция, которая может быть недифференцируема. На каждом шаге алгоритма для текущей точки \mathbf{x}_k формируется множество выборок градиентов, используемых для аппроксимации субградиента функции:

- Генерируется множество направлений:

$$\{\mathbf{u}_i \mid i = 1, \dots, p\}, \quad \text{где } \|\mathbf{u}_i\| = 1, \mathbf{u}_i \in \mathbb{R}^n. \quad (3.3.6)$$

- Формируется множество точек выборки вокруг текущей точки \mathbf{x}_k :

$$B_k := \{\mathbf{x}_k + \epsilon \mathbf{u}_i \mid i = 1, \dots, p\}, \quad (3.3.7)$$

где $\epsilon > 0$ - радиус выборки, а $p \geq n + 1$ - количество направлений.

- Для каждой точки $\mathbf{x}_k + \epsilon \mathbf{u}_i$ из множества B_k вычисляется приближённый направленный градиент (важно умножить на \mathbf{u}_i - единичный вектор направления, так как градиент - вектор):

$$\nabla \mathbf{f}_k = \frac{f(\mathbf{x}_k + \epsilon \mathbf{u}_i) - f(\mathbf{x}_k)}{\epsilon} * \mathbf{u}_i, \quad i = 1, \dots, p. \quad (3.3.8)$$

Эти значения используются для построения аппроксимации субградиента в точке \mathbf{x}_k .

- Далее формируется множество G_k из приближенных градиентов, вычисленных на прошлом шаге:

$$G_k := \{\nabla \mathbf{f}_i \mid i = 1, \dots, p\}. \quad (3.3.9)$$

Аппроксимированный субградиент вычисляется как среднее направление приближенных градиентов из множества G_k , вычисленных в окрестности точки \mathbf{x}_k :

$$\tilde{\nabla} \mathbf{f}_k = \frac{1}{p} \sum_{i=1}^p \nabla \mathbf{f}_i. \quad (3.3.10)$$

Метод выборки значений градиента состоит из следующих шагов:

1. Инициализация: задать начальную точку x_0 , параметры радиуса выборки $\epsilon > 0$, количество направлений \mathbf{p} , а также точность $\delta > 0$.
2. Выбор направлений: сгенерировать \mathbf{p} случайных направлений $\{\mathbf{u}_i \mid i = 1, \dots, p\}$ на единичной сфере.
3. Формирование множества точек: построить множество B_k , используя для своего формирования \mathbf{u}_i случайные направления, подробнее в (3.3.7).
4. Вычисление приближённых градиентов: для каждой точки из B_k , используя для каждого направления \mathbf{u}_i вычислить $\nabla \mathbf{f}_i$ (3.3.8).
5. Обновление точки: используя средний градиент (3.3.10) обновить текущую точку

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \tilde{\nabla} \mathbf{f}_k, \quad (3.3.11)$$

где $\eta > 0$ - шаг градиентного спуска. Для выбора шага η можно использовать следующие методы:

- Фиксированное значение шага: шаг η задается заранее как константа. Например, $\eta = 0.1$. Этот метод прост, но может не быть оптимальным для всех типов задач.
- Линейный поиск: шаг η подбирается с помощью линейного поиска, чтобы гарантировать уменьшение целевой функции в направлении отрицательного градиента:

$$\eta_k = \arg \min_{\eta} \left[f(\mathbf{x}_k - \eta \tilde{\nabla} \mathbf{f}_k) \right], \quad (3.3.12)$$

где минимизируется целевая функция $f(\mathbf{x})$ вдоль направления $-\tilde{\nabla} \mathbf{f}_k$.

6. Проверка остановки: если $\|\tilde{\nabla} \mathbf{f}_k\| \leq \delta$, алгоритм завершается. Иначе, перейти к шагу 2.

Algorithm 5 GS

Требуется: Начальная точка \mathbf{x}_0 , радиус выборки $\varepsilon > 0$, количество направлений p , точность $\delta > 0$, шаг η

- 1: Инициализация: $x_0, \varepsilon, p, \delta$.
- 2: $k = 0$
- 3: **while** true **do**
- 4: Сгенерировать p случайных направлений $\{\mathbf{u}_i \mid i = 1, \dots, p\}$ на единичной сфере.
- 5: Построить множество точек: $B_k = \{\mathbf{x}_k + \varepsilon \mathbf{u}_i \mid i = 1, \dots, p\}$.
- 6: **for** $i = 1$ to p **do**
- 7: Для каждой точки $\mathbf{x}_k + \varepsilon \mathbf{u}_i$ вычислить приближённый градиент:
$$\nabla f_i = \frac{f(\mathbf{x}_k + \varepsilon \mathbf{u}_i) - f(\mathbf{x}_k)}{\varepsilon} \cdot \mathbf{u}_i, \quad i = 1, \dots, p.$$
- 8: **end for**
- 9: Построить множество градиентов: $G_k = \{\nabla f_1, \nabla f_2, \dots, \nabla f_p\}$.
- 10: Вычислить аппроксимированный субградиент:
$$\tilde{\nabla} f_k = \frac{1}{p} \sum_{i=1}^p \nabla f_i.$$
- 11: Обновить точку:
$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \tilde{\nabla} f_k.$$
- 12: **if** $\|\tilde{\nabla} f_k\| \leq \delta$ **then**
- 13: **Return** \mathbf{x}_{k+1} ▷ Алгоритм завершён
- 14: **end if**
- 15: $k = k + 1$
- 16: **end while**

Рис. 6: Псевдокод метода GS

Метод выборки градиентов обладает следующими свойствами [4]:

- если $f(\mathbf{x})$ локально липшицева, то есть для каждой точки \mathbf{x}_k существует окрестность и константа $L > 0$, такие что

$$|f(\mathbf{x}) - f(\mathbf{y})| \leq L \|\mathbf{x} - \mathbf{y}\|, \text{ для всех } \mathbf{x}, \mathbf{y} \text{ в окрестности } \mathbf{x}_k,$$

то аппроксимированный субградиент $\tilde{\nabla} f_k$ сходится к истинному субградиенту $\nabla f(\mathbf{x}_k)$ при уменьшении $\epsilon \rightarrow 0$.

- для выпуклых функций алгоритм гарантированно сходится к глобальному минимуму.
- для невыпуклых функций метод сходится к стационарной точке.

Объединение методов последовательного квадратичного программирования и выборки градиентов (англ. Sequential quadratic programming - Gradient Sampling, SQP-GS) [12] применяется для решения задач нелинейной оптимизации с ограничениями. В данной задаче рассматривается целевая функция $f(\mathbf{x})$, которая может быть как гладкой, так и

негладкой (например, иметь разрывы или точки, где производная не определена). Алгоритм SQP-GS использует штрафную функцию для учета нарушений ограничений.

Штрафная функция $\phi_\rho(\mathbf{x})$ определяется как комбинация исходной целевой функции $f(\mathbf{x})$ и штрафов за нарушение ограничений:

$$\phi_\rho(\mathbf{x}) = f(\mathbf{x}) + \rho \sum_{i=1}^m \max(0, g_i(\mathbf{x}))^2. \quad (3.3.13)$$

где $\rho > 0$ - это штрафной параметр, который контролирует влияние нарушений ограничений, \mathbf{g}_i - ограничения. Если $g_i(\mathbf{x}) \leq 0$, то штраф за это ограничение равен нулю, в противном случае добавляется штраф $\rho \cdot \max(0, g_i(\mathbf{x}))^2$.

Для оптимизации $f(\mathbf{x})$ с учетом ограничений $g_i(\mathbf{x})$ используется следующая задача квадратичного программирования на каждой итерации:

$$\min_{\mathbf{d}} q_\rho(\mathbf{d}; \mathbf{x}_k, B_k, H_k) = f(\mathbf{x}_k) + \nabla f(\mathbf{x}_k)^T \mathbf{d} + \frac{1}{2} \mathbf{d}^T H_k \mathbf{d} + \rho \sum_{i=1}^m \max(0, g_i(\mathbf{x}_k + \mathbf{d}))^2. \quad (3.3.14)$$

где \mathbf{x}_k - это текущая точка и B_k (3.3.7) - набор точек для аппроксимации градиента и гессиана, H_k (3.3.4) - аппроксимация гессиана, а также направление поиска \mathbf{d}_k .

Алгоритм SQP-GS состоит из нескольких ключевых шагов, включая генерирование случайных точек для аппроксимации функции и градиентов, вычисление направления поиска и обновление параметров штрафа:

1. Выбираются начальные параметры:

- радиус выборки $\epsilon > 0$,
- штрафной параметр $\rho > 0$,
- порог нарушения ограничений $\theta > 0$,
- размер выборки $p \geq n + 1$,
- коэффициенты для сжатия радиуса выборки β , штрафного параметра β_ρ , и порога нарушения ограничений β_θ ,
- параметры сходимости для стационарности $\nu > 0$,
- выбирается начальная точка $x_0 \in D$.

2. Генерация выборки: генерируются случайные точки $B_k = \{x_{k,1}, x_{k,2}, \dots, x_{k,p}\}$, где $p \geq n + 1$ - это количество точек в выборке, и все точки выбираются из окрестности текущей точки x_k . Эти точки используются для оценки значений целевой функции и градиентов:

$$f(\mathbf{x}_k + \mathbf{d}) \approx \frac{1}{p} \sum_{i=1}^p f(\mathbf{x}_k + \delta_i), \quad (3.3.15)$$

где δ_i - случайные значения, которые используются для аппроксимации градиента. Суть метода выборки градиентов: поскольку целевая функция может быть негладкой, градиент не всегда можно вычислить напрямую. Метод позволяет аппроксимировать градиент и гессиан на основе значений функции в случайных точках.

3. Вычисление направления поиска: направление поиска \mathbf{d}_k рассчитывается как решение задачи квадратичного программирования (3.3.14), которое направлено на минимизацию стоимости с учетом штрафа за нарушения ограничений.
4. Обновление параметров: если уменьшение функции $\Delta q_\rho(\mathbf{d}_k)$ достаточно велико (то есть изменения в модели функции достаточно заметны), то радиус выборки ϵ уменьшается, и штрафной параметр ρ также изменяется для улучшения точности решения. Параметры обновляются следующим образом:

$$\epsilon \leftarrow \beta\epsilon, \quad \rho \leftarrow \beta_\rho\rho, \quad \theta \leftarrow \beta_\theta\theta. \quad (3.3.16)$$

5. Линейный поиск: для гарантии достаточного уменьшения функции в направлении d_k , выполняется линейный поиск, который удовлетворяет условию достаточного уменьшения:

$$\phi_\rho(\mathbf{x}_{k+1}) \leq \phi_\rho(\mathbf{x}_k) - \eta\alpha_k\Delta q_\rho(\mathbf{d}_k; \mathbf{x}_k, B_k, H_k), \quad (3.3.17)$$

где α_k - это шаг, который подбирается с помощью линейного поиска, η -это коэффициент шага, определяющий, насколько сильно будет уменьшаться функция на текущей итерации. Если найдено подходящее α_k , то обновляется точка $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{d}_k$.

6. Итерация: переход к следующей итерации: $k \leftarrow k + 1$, и повторение шагов алгоритма.

Таким образом, алгоритм SQP-GS строит приближение решения задачи оптимизации с ограничениями, применяя штрафную функцию и выборку градиентов для аппроксимации модели задачи. На каждом шаге выбираются новые точки для оценки функции и её градиентов, затем вычисляется направление поиска, и параметры обновляются для улучшения точности решения. Этот алгоритм обладает хорошими свойствами сходимости, так как штрафной параметр и радиус выборки уменьшаются по мере приближения к оптимальному решению.

3.4 Метод барьерного фильтра с линейным поиском

Метод барьерного фильтра с линейным поиском (англ. A Line-Search Filter Method) [1] был предложен как альтернатива традиционным функциям штрафа и критериям Армихо (см. 2.3 [1]) для улучшения сходимости в методах внутренней точки. Основная идея заключается в том, чтобы рассматривать задачу минимизации как двойственную: улучшать либо значение целевой функции, либо уменьшить нарушение ограничений (меру несоблюдения условий). Этот подход был впервые предложен Флетчером и Лейфером в 2002 году [5] и с тех пор активно развивался (см. статьи [34], [35], [36]).

Классические методы используют функции штрафа или оценочные функции, комбинируя целевую функцию $f(\mathbf{x})$ и нарушения ограничений $c(\mathbf{x})$. Однако такие методы могут приводить к медленной сходимости из-за противоречий между уменьшением целевой функции $f(\mathbf{x})$ и соблюдением ограничений. Подробно о проблеме в [6].

Метод фильтра решает эту проблему, вводя альтернативный критерий приемлемости шага:

- Шаг \mathbf{x}_{trial} (пробный шаг) считается допустимым, если он **улучшает** либо значение целевой функции, либо уменьшает нарушение ограничений $\theta(\mathbf{x}) = \|c(\mathbf{x})\|$, где $\theta(\mathbf{x})$ – это мера нарушения ограничений.

Таким образом, фильтр предотвращает нежелательное поведение, когда метод ”циклически” либо улучшает целевую функцию и увеличивает нарушение ограничений, либо наоборот.

Для решения ставится **двойственная формулировка задачи**. Рассматривается задача минимизации с нелинейными ограничениями вида (2.1). Для обеспечения глобальной сходимости к решению используется барьерная функция вида:

$$\varphi_\mu(\mathbf{x}) = f(\mathbf{x}) - \mu \sum_{i=1}^n \ln(\mathbf{x}^{(i)}), \quad (3.4.1)$$

где $\ln(\mathbf{x}^{(i)})$ – логарифмический барьерный член, который ”штрафует” приближение переменных $\mathbf{x}^{(i)}$ к нулю, μ – барьерный параметр, который начинается с начального значения вычисляемого на основе начальных данных задачи, и уменьшается в процессе оптимизации, стремясь к нулю. Этот подход основан на классических работах Фиакко и Маккорника [7], где барьерные функции используются для преобразования задачи с ограничениями в последовательность задач без ограничений.

Рассмотрим критерий допустимости шага. Метод фильтра рассматривает задачу как двухкритериальную оптимизацию:

- $\varphi_\mu(\mathbf{x})$ – значения барьерной функции;
- $\theta(\mathbf{x}) = \|c(\mathbf{x})\|$ – нарушений ограничений.

При поиске новой точки $\mathbf{x}_k(\alpha)$, где α – размер шага, выполняется одно из условий:

1. Уменьшение нарушения ограничений:

$$\theta(\mathbf{x}_k(\alpha)) \leq (1 - \gamma_\theta)\theta(\mathbf{x}_k), \quad (3.4.2a)$$

где $\gamma_\theta \in (0, 1)$ – фиксированный параметр;

2. Уменьшение барьерной функции (если нарушение ограничений достаточно мало):

$$\varphi_\mu(\mathbf{x}_k(\alpha)) \leq \varphi_\mu(\mathbf{x}_k) - \gamma_\varphi \theta(\mathbf{x}_k), \quad (3.4.2b)$$

где $\gamma_\varphi \in (0, 1)$.

Значение γ_θ выбирается в зависимости от специфики решаемой задачи и служит для контроля уменьшения нарушения ограничений при поиске следующей точки. Этот параметр должен быть достаточно малым, чтобы гарантировать постепенное сокращение нарушения ограничений без резких изменений, но в то же время достаточным для обеспечения сходимости алгоритма.

Таким образом, точка $\mathbf{x}_k(\alpha)$ считается допустимой для фильтра, если она улучшает либо значение нарушения ограничений $\theta(\mathbf{x})$, либо значение барьерной функции $\varphi_\mu(\mathbf{x})$. Однако если текущее нарушение ограничений $\theta(\mathbf{x}_k)$ мало: меньше фиксированного порога θ_{min} , который задаётся заранее и определяет, насколько малым должно быть нарушение ограничений, чтобы переключиться с фильтра на критерий Армихо по целевой функции (Обычно он выбирается эмпирически, например 10^{-4}); то фильтр переключается на критерий Армихо (см. раздел 2.3 [1]) для целевой функции.

Критерий Армихо.

$$\varphi_\mu(\mathbf{x}_k(\alpha)) \leq \varphi_\mu(\mathbf{x}_k) + \eta_\varphi \alpha \nabla \varphi_\mu(\mathbf{x}_k)^T \mathbf{d}_k, \quad (3.4.3)$$

где $\eta_\varphi \in (0, \frac{1}{2})$ - константа, \mathbf{d}_k - направление поиска. Выбор η_φ обычно основан на численных экспериментах, стандартное значение часто берут около 0.1. Параметр определяет, насколько консервативным будет условие Армихо. Направление \mathbf{d}_k определяется решением линейной системы, зависящей от текущего приближения и структуры задачи.

Механизм фильтра. Фильтр F_k - это множество пар значений $(\boldsymbol{\theta}, \varphi_\mu)$, которые соответствуют точкам, не обеспечивающим достаточного улучшения либо по нарушению ограничений, либо по значению барьерной функции (см. раздел 2.3 [1]). Это позволяет избежать "циклического" поведения, когда алгоритм чередует улучшение целевой функции и ухудшение ограничений (или наоборот). Точка $\mathbf{x}_k(\alpha)$ не принимается, если:

$$(\theta(\mathbf{x}_k(\alpha)), \varphi_\mu(\mathbf{x}_k(\alpha))) \notin F_k. \quad (3.4.4)$$

Фильтр на начальном этапе инициализируется следующим образом:

$$F_0 = \{(\boldsymbol{\theta}, \varphi) : \boldsymbol{\theta} \geq \boldsymbol{\theta}_{max}\},$$

где $\boldsymbol{\theta}_{max}$ ограничивает максимальное допустимое нарушение ограничений.

Далее фильтр обновляется после каждого успешного шага:

$$F_{k+1} = F_k \cup \{(\boldsymbol{\theta}, \varphi) : \boldsymbol{\theta} \geq (1 - \gamma_\theta)\theta(\mathbf{x}_k), \varphi \geq \varphi_\mu(\mathbf{x}_k) - \gamma_\varphi \theta(\mathbf{x}_k)\}. \quad (3.4.5)$$

Линейный поиск и корректировка шага. Метод использует стандартный линейный поиск с возвратом (англ. *backtracking line-search*) [5]. При этом шаг уменьшается на каждой итерации:

$$\alpha_k^l = 2^{-l} \alpha_k^{max}, \quad l = 0, 1, 2, \dots, \quad (3.4.6)$$

где α_k^{max} определяется "правилом доли до границы" (англ. *fraction-to-the-boundary*) для сохранения положительности переменных:

$$\alpha_k^{max} = \max \alpha \in (0, 1] : \mathbf{x}_k + \alpha \mathbf{d}_k \geq (1 - \tau_j) \mathbf{x}_k. \quad (3.4.7)$$

Здесь $\tau_j \in (0, 1)$ - параметр, который определяется как $\tau_j = \max\{\tau_{min}, 1 - \mu_j\}$, где τ_{min} - ее минимальное значение, μ_j — барьерный параметр, определяющий величину приближения к

границе допустимой области. Чем меньше μ_j , тем ближе точка может подойти к границе ограничений. Параметр τ_j служит для контроля допустимости шага α_k , обеспечивая сохранение положительности переменных. Он определяет, насколько близко новая точка $\mathbf{x}_k + \alpha \mathbf{d}_k$ может подойти к границе области допустимых значений.

Если ни один шаг α не удовлетворяет критериям фильтра, то алгоритм переходит в фазу восстановления допустимости. (см. раздел 3.3 [1])

Когда линейный поиск не может найти допустимый шаг, то есть когда ни одно из условий приемлемости шага не выполняется для всех пробных значений (3.4.6). Это происходит, если для всех l выполняется:

$$\alpha_k^l < \alpha_k^{min}, \quad (3.4.8)$$

где α_k^{min} – минимально допустимый размер шага, определяемый следующим образом:

1. Если $\nabla \varphi_{\mu_j}(\mathbf{x}_k)^T \mathbf{d}_k^x <$ и $\theta(\mathbf{x}_k) \leq \boldsymbol{\theta}^{min}$, то:

$$\alpha_k^{min} = \gamma_\alpha \cdot \min\left\{\gamma_\theta, -\frac{\gamma_\theta \theta(\mathbf{x}_k)}{-\nabla \varphi_{\mu_j}(\mathbf{x}_k)^T \mathbf{d}_k^x}, [-\nabla \varphi_{\mu_j}(\mathbf{x}_k)^T \mathbf{d}_k^x]^{\gamma_\theta}\right\};$$

2. Если $\nabla \varphi_{\mu_j}(\mathbf{x}_k)^T \mathbf{d}_k^x <$ и $\theta(\mathbf{x}_k) > \boldsymbol{\theta}^{min}$, то:

$$\alpha_k^{min} = \gamma_\alpha \cdot \min\left\{\gamma_\theta, -\frac{\gamma_\theta \theta(\mathbf{x}_k)}{-\nabla \varphi_{\mu_j}(\mathbf{x}_k)^T \mathbf{d}_k^x}\right\};$$

3. В остальных случаях:

$$\alpha_k^{min} = \gamma_\alpha \cdot \gamma_\theta.$$

Здесь $\gamma_\alpha \in (0, 1]$ – ”фактор безопасности”, масштабирует минимально допустимый размер шага α_k^{min} . Если α_k^l становится меньше α_k^{min} , то алгоритм переходит в фазу восстановления допустимости (см. раздел 3.3 [1]). $\gamma_\theta \in (0, 1)$ – фиксированный параметр, устанавливает порог для достаточного снижения нарушения ограничений $\theta(\mathbf{x})$ для принятия шага. $\boldsymbol{\theta}^{min}$ – пороговое значение нарушения ограничений, которое определяет переход алгоритма на критерий Армихо (3.4.8), когда $\theta(\mathbf{x}_k)$ становится достаточно малым (см. раздел 2.3 [1]).

Минимальный допустимый шаг α_k^{min} определяется таким образом, чтобы обеспечить достаточно уменьшение нарушения ограничений или функции, особенно если все предыдущие шаги не удовлетворяют критериям фильтра. Формула выведена из условий Армихо и условий фильтра, и гарантирует устойчивость метода (см. [1]). Если условие (3.4.8) выполняется для всех пробных шагов, алгоритм переходит к фазе восстановления допустимости. Цель этой фазы – найти новую точку \mathbf{x}_{k+1} , которая удовлетворяет следующим условиям:

1. Уменьшение нарушения ограничений:

$$\theta(\mathbf{x}_{k+1}) \leq \kappa_{resto} \theta(\mathbf{x}_k), \quad (3.4.9)$$

где $\kappa_{resto} \in (0, 1)$ – константа, обычно равная 0.9 (см. раздел 3.3.1 [1]). Это пороговое значение, которое определяет минимально необходимое уменьшение нарушения ограничений, обеспечивая прогресс алгоритма в фазе восстановления.

2. Допустимость для фильтра:

$$(\theta(\mathbf{x}_{k+1}), \varphi_{\mu_j}(\mathbf{x}_{k+1})) \notin F_k. \quad (3.4.10)$$

Подробности о реализованной фазе восстановления представлены в (Раздел 3.3 [1]).

Алгоритм успешно завершает работу при выполнении одного из следующих условий (см. разделы 2.1, 2.4 и 3.3 [1]):

1. **Уменьшение нарушения ограничений.** Алгоритм завершает работу, если текущая точка $(\mathbf{x}_k, \boldsymbol{\lambda}_k, \mathbf{z}_k)$ удовлетворяет критериям сходимости для исходной задачи. Это означает, что ошибка оптимальности $E_0(\mathbf{x}_k, \boldsymbol{\lambda}_k, \mathbf{z}_k)$ становится меньше или равна заданной точности ϵ_{tol} ($E_0(\mathbf{x}_k, \boldsymbol{\lambda}_k, \mathbf{z}_k) \leq \epsilon_{tol}$). Ошибка оптимальности E_0 включает в себя три компоненты:

a) **Прямое условие оптимальности.** Оно проверяет, насколько градиент Лагранжиана $\nabla L(\mathbf{x}, \boldsymbol{\lambda}, \mathbf{z})$ приближен к нулю. Формула имеет вид:

$$\frac{\|\nabla f(\mathbf{x}_k) + \nabla c(\mathbf{x}_k)\boldsymbol{\lambda}_k - \mathbf{z}_k\|}{s_d},$$

где $\nabla f(\mathbf{x}_k)$ — градиент целевой функции, $\nabla c(\mathbf{x}_k)$ — матрица Якоби ограничений, \mathbf{z}_k — переменные двойственности для условий неравенств, а s_d — масштабирующий параметр, предотвращающий проблемы с большими значениями множителей Лагранжа.

б) **Выполнение условий равенств.** Это проверка, насколько текущая точка удовлетворяет равенствам:

$$\|c(\mathbf{x}_k)\|_\infty,$$

где $c(\mathbf{x}_k)$ — вектор ограничений равенства. Норма $\|\cdot\|_\infty$ измеряет максимальное нарушение среди всех условий.

в) **Выполнение условий комплементарности.** Этот компонент проверяет выполнение условий комплементарности для барьерного метода:

$$\frac{\|Z_k X_k \mathbf{e}\|_\infty}{s_c},$$

где $X_k = \text{diag}(\mathbf{x}_k)$ и $Z_k = \text{diag}(\mathbf{z}_k)$ — диагональные матрицы, соответствующие текущим значениям переменных и их двойственных множителей. Здесь e — вектор из единиц, а s_c — масштабирующий параметр для комплементарности.

Таким образом, полная формула ошибки оптимальности принимает вид:

$$E_0(\mathbf{x}_k, \boldsymbol{\lambda}_k, \mathbf{z}_k) = \max \left\{ \frac{\|\nabla f(\mathbf{x}_k) + \nabla c(\mathbf{x}_k)\boldsymbol{\lambda}_k - \mathbf{z}_k\|}{s_d}, \|c(\mathbf{x}_k)\|_\infty, \frac{\|Z_k X_k \mathbf{e}\|_\infty}{s_c} \right\}.$$

Здесь s_d и s_c — масштабирующие параметры, которые адаптируются к величине множителей Лагранжа $\boldsymbol{\lambda}_k$ и \mathbf{z}_k , $Z_k = \text{diag}(\mathbf{z}_k)$, $X_k = \text{diag}(\mathbf{x}_k)$, \mathbf{e} — вектор из единиц.

2. **Условие сходимости для барьерной задачи.** Если текущая точка $(\mathbf{x}_k, \lambda_k, \mathbf{z}_k)$ удовлетворяет условию сходимости для барьерной задачи с текущим значением барьерного параметра μ_j , то есть если:

$$E_{\mu_j}(\mathbf{x}_k, \lambda_k, \mathbf{z}_k) \leq \kappa_\epsilon \mu_j,$$

где $\kappa_\epsilon > 0$ - константа, то барьерный параметр уменьшается, и алгоритм продолжает работу с новым значением μ_{j+1} , вычисляемым по формуле:

$$\mu_{j+1} = \max\left\{\frac{\epsilon_{tol}}{10}, \min\{\kappa_\mu \mu_j, \mu_j^{\theta_\mu}\}\right\},$$

где $\kappa_\epsilon \in (0, 1)$ и $\theta_\mu \in (1, 2)$ — параметры, управляющие скоростью уменьшения μ_j .

Algorithm 6 Метод барьерного фильтра с линейным поиском

Require: Начальная точка (x_0, λ_0, z_0) , параметры: $\mu_0 > 0$, $\varepsilon_{tol} > 0$, $\kappa_\varepsilon > 0$, $\kappa_\mu \in (0, 1)$, $\theta_\mu \in (1, 2)$, $\tau_{min} \in (0, 1)$, $\kappa_\sigma > 1$, $\theta_{max} > \theta(x_0)$, $\gamma_\theta, \gamma_\phi \in (0, 1)$, $\eta_\phi \in (0, \frac{1}{2})$, $\kappa_{SOC} \in (0, 1)$, $p_{max} \in \mathbb{N}$;

Инициализация: $j \leftarrow 0$, $k \leftarrow 0$, $F_0 \leftarrow \{(\theta, \phi) : \theta \geq \theta_{max}\}$;

Вычислить $\tau_0 = \max\{\tau_{min}, 1 - \mu_0\}$;

while $E_0(x_k, \lambda_k, z_k) > \varepsilon_{tol}$ **do** ▷ Основной цикл

4: **if** $E_{\mu_j}(x_k, \lambda_k, z_k) \leq \kappa_\varepsilon \mu_j$ **then** ▷ Проверка сходимости

 Обновить параметры:

$$\mu_{j+1} = \max\left(\frac{\varepsilon_{tol}}{10}, \min(\kappa_\mu \mu_j, \mu_j^{\theta_\mu})\right), \quad \tau_{j+1} = \max(\tau_{min}, 1 - \mu_{j+1});$$

$j \leftarrow j + 1$;

 Сбросить фильтр: $F_k \leftarrow \{(\theta, \phi) : \theta \geq \theta_{max}\}$;

8: Продолжить.

end if

 Вычислить направление поиска $(d_k^x, d_k^\lambda, d_k^z)$, решив систему:

$$\begin{bmatrix} W_k + \Sigma_k & A_k^T \\ A_k & -\Delta_k \end{bmatrix} \begin{bmatrix} d_k^x \\ d_k^\lambda \end{bmatrix} = - \begin{bmatrix} \nabla \phi_{\mu_j}(x_k) + A_k^T \lambda_k \\ c(x_k) \end{bmatrix}.$$

Найти шаги α_k и α_k^z , чтобы гарантировать положительность переменных.

12: **while** Шаг α_k не принят **do**

if $\theta(x_k(\alpha_k)) \leq \theta_{min}$ **then**

 Проверить условие Армихо (3.4.3)

else

16: Проверить прогресс по ограничениям (3.4.2а-б)

end if

if шаг α_k не принят **then**

if применяется коррекция второго порядка (SOC) **then**

20: Вычислить поправку $d_k^{x,SOC}$:

$$d_k^{x,SOC} = d_k^x + \Delta d_k^x, \quad \text{где } A_k \Delta d_k^x + c(x_k + d_k^x) = 0;$$

 Проверить фильтр для скорректированной точки.

else

24: Уменьшить шаг α_k и продолжить.

end if

end if

end while

 Принять шаг: $x_{k+1} = x_k + \alpha_k d_k^x$;

28: Обновить фильтр, если точка не удовлетворяет условиям переключения.

if шаг слишком мал **then**

 Перейти в фазу восстановления и найти x_{k+1} , уменьшающую $\theta(x)$;

end if

32: $k \leftarrow k + 1$;

end while

Рис. 7: Псевдокод метода барьерного фильтра с линейным поиском [1]

Таким образом, метод барьерного фильтра с линейным поиском позволяет гарантировать глобальную сходимость, благодаря использованию фильтра и фаз восстановления, и эффективно сочетает уменьшение целевой функции с устранением нарушений ограничений. Он использует фильтр для определения допустимости точек и избегает недостатков функций штрафа. Метод является мощным инструментом для решения широкого класса задач, требующих точной и надежной оптимизации. Однако, для получения наилучших результатов требуется тщательный выбор параметров, таких как ε_{tol} , κ_μ , и θ_μ , в зависимости от конкретной задачи.

3.5 Метод исправленной адаптивной оценки моментов

Метод исправленной адаптивной оценки моментов (англ. Rectified Adaptive Moment Estimation, RAdam) [14] был предложен как альтернатива методу адаптивной оценки моментов (англ. Adaptive Moment Estimation, Adam) [15] для улучшения сходимости последнего, особенно на начальных этапах вычисления адаптивного параметра обучения (англ. adaptive learning rate) (см. (1) в [14]). Основная проблема Adam заключается в высокой дисперсии оценок второго момента градиентов на начальных этапах обучения [14], что ведёт к нестабильным шагам при обновлении параметров (см. 3 в [14]) и сильной зависимости от выбора начального шага обучения (см. Figure 7 в [14]). Метод RAdam решает эту проблему, вводя на начальных этапах работы алгоритма (см. 4.2 в [14]) поправку (англ. rectification term) (см. Theorem 1, 4.1, 4.2 в [14]) для вычисления адаптивного параметра обучения, что стабилизирует шаг обновления на начальных итерациях (см. 4.2, 4.3 в [14]). После стабилизации дисперсии оценок моментов, используется адаптивный параметр обучения, предложенный оптимизатором Adam (см. строку 14 Algorithm 2 в [14] и Algorithm 1 в [15]). Таким образом, RAdam представляет собой более устойчивый метод решения оптимизационных задач без ограничений и имеет более быструю сходимость к решению (см. 5.1 в [14]).

Алгоритм RAdam состоит из следующих шагов:

1. Инициализация: задать шаг обучения (англ. learning rate) α , начальную точку θ_0 , начальные скорости затухания (англ. decay rates) для оценок моментов (см. главу 2 в [15]), целевую функцию для оптимизации $f(\theta)$ и количество итераций T .
2. Определение дальнейшего шага обучения: на текущей итерации t оценить дисперсию оценок моментов и выбрать необходимый шаг: шаг метода Adam или шаг метода Adam с поправкой (см. строки 8-12 Algorithm 2 в [14]).
3. Обновление точки: сделать шаг в выбранном направлении (обновить текущее приближение θ_t).
4. Условие остановки: если количество итераций $t = T$, то метод завершается, иначе шаги 1-3 повторяются.

Algorithm 7 RAdam (Rectified Adam)

```

1: Input:  $\theta_0 \in \mathbb{R}^d$ ,  $\beta_1, \beta_2 \in [0, 1]$ ,  $\alpha, \epsilon$ 
2: Output:  $\theta_T$  - решение оптимизационно задачи (2.1) без ограничений
3:  $m_0 \leftarrow 0$ 
4:  $v_0 \leftarrow 0$ 
5: Вычислить асимптотическое значение для коррекции дисперсии:  $\rho_\infty \leftarrow \frac{2}{1-\beta_2} - 1$ 
6: for  $t = 1$  to  $T$  do
7:   Вычислить градиент:  $g_t \leftarrow \nabla_{\theta} f(\theta_{t-1})$ 
8:   Обновить первый момент:  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ 
9:   Обновить второй момент:  $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ 
10:  Скорректировать первый момент:  $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$ 
11:  Оценка степеней свободы на итерации:  $\rho_t \leftarrow \rho_\infty - \frac{2t\beta_2^t}{1-\beta_2^t}$ 
12:  if  $\rho_t > 4$  then
13:     $l_t \leftarrow \sqrt{\frac{1-\beta_2^t}{v_t}}$ 
14:     $r_t \leftarrow \sqrt{\frac{(\rho_t-4)(\rho_t-2)\rho_\infty}{(\rho_\infty-4)(\rho_\infty-2)\rho_t}}$ 
15:    Обновление параметров с шагом Adam с поправкой:  $\theta_t \leftarrow \theta_{t-1} - \alpha r_t \hat{m}_t l_t$ 
16:  else
17:    Обновление параметров с шагом шагом Adam:  $\theta_t \leftarrow \theta_{t-1} - \alpha \hat{m}_t$ 
18:  end if
19: end for
20: return:  $\theta_T$ 

```

Рис. 8: Псевдокод метода исправленной адаптивной оценки моментов [14]

Для использования RAdam для решения оптимизационных задач вида (2.1) используются метод барьерных функций (англ. Barrier Method) [16] в сочетании с методом введения штрафных функций (англ. Penalty Method) [17].

Метод барьерных функций преобразует задачу с ограничениями типа неравенства в задачу без ограничений путём добавления барьерных членов к целевой функции (см. формулу (3.5.1)). Степень аппроксимации границы области, задаваемой ограничениями типа "неравенства", определяется барьерным параметром μ . Уменьшая параметр барьера, мы лучше аппроксимируем границу области, задаваемой ограничениями типа "неравенства" [16].

В общем виде метод барьерных функций преобразует целевую функцию к виду:

$$\min_{\mathbf{x} \in \mathbb{R}^n} \left(f(\mathbf{x}) + \mu \sum_{i=1}^m \phi(h_i(\mathbf{x})) \right), \quad (3.5.1)$$

где:

- $\mu > 0$ — барьерный параметр;
- $\phi(h_i(\mathbf{x}))$ — барьерная функция.

Метод штрафных функций, напротив, решает задачу с ограничениями типа равенства (хотя, в общем случае, может решать задачи с любыми ограничениями, см. 1 в [17]), сводя

исходную задачу вида (2.1) к задаче без ограничений, через введение специальных штрафных членов (англ. penalty terms) (см. (1) в [17]). Таким образом, комбинируя метод барьерных функций с методом штрафных функций, можем решать задачи оптимизации с ограничениями путём сведения их к задаче оптимизации функции без ограничений.

Для случая работы только с ограничениями типа "равенства" метод штрафных функций выглядит следующим образом,

$$P(\mathbf{x}, r) = f(\mathbf{x}) + h_\tau(c(\mathbf{x})), \quad (3.5.2)$$

где:

- h_τ — функция штрафа (см. (1.2) в [17]);
- τ - параметр штрафа (см. 1 в [17]);
- $c(\mathbf{x})$ — заданные ограничения типа "равенства" (см. (2.2)).

Тогда общая оптимационная задача (2.1) сводится к виду:

$$\min_{\mathbf{x} \in \mathbb{R}^n} \left(f(\mathbf{x}) + \mu \sum_{i=1}^m \phi(h_i(\mathbf{x})) + h_\tau(c(\mathbf{x})) \right), \quad (3.5.3)$$

после чего подаётся на вход для оптимизации в RAdam.

Для адаптивного подбора барьерного параметра используется концепция центрального пути (англ. Central Path) (см. 11.2 в [18]). Для определённости, рассмотрим задачу (3.5.3) в следующем виде:

$$\min_{\mathbf{x} \in \mathbb{R}^n} \left(f(\mathbf{x}) - \frac{1}{t} \sum_{i=1}^m \log(-h_i(\mathbf{x})) + \tau \sum_{j=1}^p (c_j(\mathbf{x}))^2 \right). \quad (3.5.4)$$

Принцип концепции центрального пути заключается в следующем:

1. Инициализация: выбрать начальную точку $\mathbf{x}_k = \mathbf{x}_0$, удовлетворяющую условиям (2.3), небольшой параметр t_0 (о способе его выбора см. Figure 11.1 и 11.2.1 в [18]), параметр ε - точность приближения к границе области, задаваемой неравенствами (2.3).
2. Барьерная задача: решить задачу оптимизации без ограничений (3.5.4) в точке \mathbf{x}_k .
3. Обновление параметров: обновить начальную точку \mathbf{x}_k на точку, полученную в результате решения барьерной задачи на этапе 2.
4. Условие остановки: при выполнении условий $\frac{1}{t} \geq \varepsilon$ и $\|\mathbf{x}_{k+1} - \mathbf{x}_k\| \leq \varepsilon$ алгоритм завершается, иначе повторяем шаги 2-3.

Algorithm 8 Barrier Method c Central Path concept

```

1: Input:  $x_0$  - точка, удовлетворяющая неравенствам (2.3),  $t_0$  - начальное значение барьерного параметра,  $\alpha > 1$  - параметр для обновления барьерного параметра,  $\varepsilon > 0$  - заданная точность вычислений.
2: Output:  $x_k$  - решение оптимизационной задачи (2.1)
3:  $x_k \leftarrow x_0$ ,  $x_{prev} \leftarrow x_0$ 
4:  $t \leftarrow t_0$ 
5: while True do
6:   Решить барьерную задачу (3.5.4) при  $t = t_0$  в точке  $x_k$ 
7:   Обновить начальную точку:  $x_{k+1} \leftarrow x_k^*$ 
8:   Обновить барьерный параметр:  $t^{k+1} \leftarrow \alpha t^k$ 
9:   if  $\frac{1}{t} \geq \varepsilon$  и  $\|x_{k+1} - x_k\| \leq \varepsilon$  then                                 $\triangleright$  Условие остановки
10:    return:  $x_{k+1}$ 
11:   end if
12:   Сохранить предыдущее значение:  $x_{prev} = x_k$ 
13: end while

```

Рис. 9: Псевдокод барьерного метода с концепцией центрального пути (см. 11.2 в [18])

3.6 Ускоренный многоступенчатый градиентный метод

Рассмотрим ускоренный многоступенчатый градиентный метод Нестерова (англ. Accelerated Multistep Gradient Scheme, AMGS) [22] для аппроксимации глобального минимума выпуклой целевой функции:

$$\phi = f(\mathbf{x}) + \Psi(\mathbf{x}), \quad (3.6.1)$$

где

- $f(\mathbf{x})$ – дифференцируемая выпуклая функция, заданная в виде "черного ящика" (англ. *Black-box*) [22]. Это означает, что функция не имеет явного аналитического представления, но для неё можно вычислить значение и градиент в любой заданной точке. Такой подход часто используется, когда внутренняя структура функции неизвестна или слишком сложна для анализа.
- $\Psi(\mathbf{x})$ – «простая» функция будем говорить, что с известной проксимальной для неё операцией (см. [26]), что позволяет эффективно находить замкнутые решения для её минимизации (см. [22]).

Примеры исходной задачи:

- Пусть Q замкнутое выпуклое множество. Определим Ψ как индикаторную функцию множества Q

$$\Psi(\mathbf{x}) = \begin{cases} 0, & \text{если } \mathbf{x} \in Q \\ \infty, & \text{иначе} \end{cases}. \quad (3.6.2)$$

Тогда неограниченная минимизация составной функции (3.6.1) эквивалентна минимизации f на заданном Q . Оказывается, что поскольку Ψ простая, то мы сможем находить в замкнутом виде евклидову проекцию произвольной точки на Q .

2. Представление ограниченного множества через барьер

Предположим, что целевая функция задачи минимизации с выпуклыми ограничениями

$$\text{find } f^* = \min_{\mathbf{x} \in Q} f(\mathbf{x}), \quad (3.6.3)$$

Функция задается через механизм запросов (так называемый "оракул" – англ. *oracle*), работающий по принципу чёрного ящика. При этом множество Q описывается с помощью самосогласованного барьера $F(\mathbf{x})$ (см. раздел 4.1 в [23]).

Определим функцию $\Psi(\mathbf{x}) = \frac{\epsilon}{\nu} F(\mathbf{x})$, где ϵ - параметр точности, контролирующий близость решения к границе допустимой области, ν - параметр самосогласованности барьера, определяющий скорость роста барьерной функции. Поясним, что выпуклая функция f является самосогласованной, если для неё выполняется $|f'''(\mathbf{x})| \leq 2(f''(\mathbf{x}))^{3/2}$ для любого f . Также многомерная функция f называется самосогласованной, если одномерная функция $\phi(t) = f(\mathbf{x} + \mathbf{h}t)$ является самосогласованной для любых $\mathbf{x}, \mathbf{h} \in R^n$, также перечислим свойства самосогласованной функции: 1) что сумма самосогласованных функций будет самосогласованной функцией $f_1 + f_2 + \dots + f_n = f$, 2) что при умножении на число она также не перестает быть самосогласованной $\alpha f_1 = f_2$, 3) что композиция самосогласованной функции с аффинной функцией также будет оставаться самосогласованной $f_1 \circ \zeta$, где ζ - аффинная функция. Показано, что для самосогласованной функции может быть введен самосогласованный барьер и какому условию он вместе с самой функцией должен удовлетворять для установления границ ограничений целевой функции (см. раздел 2 [22]).

3. Разреженный метод наименьших квадратов (см. формула 1 [28]).

Для характеристики решения задачи определим конус возможных направлений и двойственный ему – нормальный:

$$\begin{aligned} \mathcal{F}(\mathbf{y}) &= \{\mathbf{u} = \tau \cdot (\mathbf{x} - \mathbf{y}), \mathbf{x} \in Q, \tau \geq 0\} \subseteq E, \\ \mathcal{N}(\mathbf{y}) &= \{s : \langle s, \mathbf{x} - \mathbf{y} \rangle \geq 0, \mathbf{x} \in Q\} \subseteq E^*, \mathbf{y} \in Q, \end{aligned} \quad (3.6.4)$$

где $\mathcal{F}(\mathbf{y})$ обозначает множество точек \mathbf{x}, \mathbf{y} , которые определяются как возможные вектора для поиска минимальных точек \mathbf{x}, \mathbf{y} , а $\mathcal{N}(\mathbf{y})$ обозначает множество в котором определяется градиент в точке, τ – длина вектора между точками \mathbf{x}, \mathbf{y} , Q – ограниченное множество при задаче минимизации, E – векторное пространство, E^* – двойственное пространство, образованное всеми линейными функциями на E .

Соответственно, необходимые условия оптимальности 1-го порядка локального минимума x^* примут вид:

$$\langle \phi', \mathbf{u} \rangle \geq 0 \quad \forall \mathbf{u} \in \mathcal{F}(x^*). \quad (3.6.5)$$

Поскольку в задаче с ограничениями ситуация несколько иная чем без ограничений, то градиент целевой функции следует трактовать иначе (подробнее см. 2.2.3 в [23]):

Введем составное градиентное отображение (composite gradient mapping)

$$m_L(\mathbf{y}; \mathbf{x}) = f(\mathbf{y}) + \langle \nabla f(\mathbf{y}), \mathbf{x} - \mathbf{y} \rangle + \frac{L}{2} \|\mathbf{x} - \mathbf{y}\|^2 + \Psi(\mathbf{x}),$$

$$\mathbf{T}_L(\mathbf{y}) = \arg \min_{\mathbf{x} \in \mathbb{Q}} m_L(\mathbf{y}; \mathbf{x}), \quad (3.6.6)$$

где L – оценка константы Липшица, которая выбирается положительным начальным значением $L_0 > 0$. То, что функция Ψ является ”простой”, позволяет нам легко находить проекцию точки на множество Q (этую проекцию мы обозначили как $\mathbf{T}_L(\mathbf{y})$). Благодаря этому мы можем определить специальный вектор направления $g_L(\mathbf{y})$, который играет роль градиента в ограниченной области (см. раздел 2 [22]):

$$\mathbf{g}_L(\mathbf{y}) = L \cdot B(\mathbf{y} - \mathbf{T}_L(\mathbf{y})) \in E^*, \quad (3.6.7)$$

где B в нашем случае векторного пространства – единичная матрица, y – текущая точка оптимизации.

Также перечислим важнейшие свойства составного градиентного отображения $\mathbf{g}_L(\mathbf{y})$:

1. 1-е свойство (непрерывный градиент по Липшицу):

$$\|\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})\|_* \leq L_f \|\mathbf{x} - \mathbf{y}\|, \quad \mathbf{x}, \mathbf{y} \in Q, \quad (3.6.8)$$

2. 2-ое свойство (подробнее см. 2.2.3 в [23]):

$$\langle f'(\mathbf{x}), \mathbf{x} - \mathbf{x}^* \rangle \geq \frac{1}{L} \|f'(\mathbf{x})\|^2. \quad (3.6.9)$$

Ускоренная схема и ее скорость сходимости

В задаче (3.6.1) функция f непрерывная по Липшицу (3.6.8), Ψ замкнута и сильно выпуклая с параметром выпуклости $\mu_\Psi \geq 0$, предполагается, что этот параметр известен, а, если $\mu_\Psi = 0$, то имеем дело с выпуклой функцией Ψ .

Для обоснования скорости сходимости ускоренной схемы по аналогии как и в статье Юрия Нестерова про ускоренную кубическую регуляризацию будем применять механизм последовательности оценочных функций:

Необходимо поддерживать рекурсивно следующие последовательности:

- Минимизирующая последовательность $\{\mathbf{x}_{k=0}^\infty\}$.
- Последовательность масштабирующих коэффициентов $\{A_k\}_{k=0}^\infty$:

$$A_0 = 0, \quad A_k \stackrel{\text{def}}{=} A_{k-1} + a_k, \quad k \geq 1.$$

- Последовательность оценочных функций

$$\psi_k(\mathbf{x}) = l_k(\mathbf{x}) + A_k \Psi(\mathbf{x}) + \frac{1}{2} \|\mathbf{x} - \mathbf{x}_0\|^2 \quad k \geq 0, \quad (3.6.13)$$

где $\mathbf{x}_0 \in \text{dom} \Psi$ – стартовая точка, l_k – линейная функция для $\mathbf{x} \in E$.

Мы должны ввести условие $L_0 < L_k \leq L_f$ (L_f – константа Липшица), при котором оценка этой константы будет в пределах данного условия (подробнее см. формула 3.2 в [22]), параметры, корректирующие оценку L_k примем такими: $\gamma_u > 1$, $\gamma_d \geq 1$.

для поддержания всех выше описанных условий, согласно механизму оценочных функций, прийдем к следующим выражениям:

$$\begin{aligned} \mathcal{R}_k^1 : A_k \phi(\mathbf{x}_k) &\leq \psi_k^* \equiv \mathbf{T}_L(\mathbf{y}) = \min_{\mathbf{x} \in \mathbb{Q}} \psi_k(\mathbf{x}), \\ \mathcal{R}_k^2 : \psi_k(\mathbf{x}) &\leq A_k \phi(\mathbf{x}) + \frac{1}{2} \|\mathbf{x} - \mathbf{x}_0\|^2, \forall \mathbf{x} \in E, \end{aligned} \quad (3.6.14)$$

где $k \geq 0$.

Теперь определим следующее выражение:

$$\mathbf{v}_k = \arg \min_{\mathbf{x} \in \mathbb{E}} \psi_k(\mathbf{x}), \quad (3.6.15)$$

принимая $\mu_{\psi_k} \geq 1$, для любого $\mathbf{x} \in E$ имеем следующее:

$$A_k \phi(\mathbf{x}_k) + \frac{1}{2} \|\mathbf{x} - \mathbf{v}_k\|^2 \stackrel{\mathcal{R}_k^1}{\leq} A_k \psi_k^* + \frac{1}{2} \|\mathbf{x} - \mathbf{v}_k\|^2 \leq \psi_k(\mathbf{x}) \stackrel{\mathcal{R}_k^2}{\leq} A_k \phi(\mathbf{x}) + \frac{1}{2} \|\mathbf{x} - \mathbf{x}_0\|^2, \quad (3.6.16)$$

следовательно, получим выражение для оценки скорости сходимости, но пока, что неокончательное:

$$\phi(\mathbf{x}_k) - \phi(\mathbf{x}^*) \leq \frac{\|\mathbf{x} - \mathbf{x}^*\|^2}{2A_k}, \quad k \geq 1. \quad (3.6.16)$$

Далее перейдем непосредственно к самому псевдокоду для этой ускоренной схемы.

Псевдокод AMGS

$\mathbf{x}_0 \in \text{dom} \Psi$, нижняя оценка L_0 константы Липшица L_f и нижняя оценка $\mu \in [0, \mu_\Psi]$ для параметра выпуклости μ_Ψ . Начальные значения параметров для корректировки L_k Юрий Нестеров в оригинальной статье предлагает взять следующие для достижения более простой оценки сходимости для данной схемы $\gamma_d = 2$, $\gamma_u = 2$ (см. формула 4.13[22])

Ниже представлен сам псевдокод алгоритма AMGS:

Algorithm 9 AMGS

Require: x_0 - начальная точка, $L_0 > 0$ - начальная оценка константы Липшица, параметр выпуклости $\mu_\Psi \geq 0$, $\gamma_u > 1$, $\gamma_d \geq 1$, $\psi_0(x) = \frac{1}{2}\|x - x_0\|^2$ - оценочная функция на нелувом шаге;

- 1: Инициализация: $x_0, L_0, \mu_\Psi, \gamma_u, \gamma_d, \psi_0(x), A_0 = 0$.
- 2: **for** $k \geq 0$ **do**
- 3: $L := L_k$
- 4: **while** true **do**
- 5: найти a из квадратного уравнения $\frac{a^2}{A_k+a} = 2\frac{1+\mu A_k}{L}$ $\triangleright *$.
- 6: Set $y = \frac{A_k x_k + a v_k}{A_k + a}$, вычислить $T_L(y)$
- 7: **if** $\langle \psi'(T_L(y)), y - T_L(y) \rangle \geq \frac{1}{L} \|\psi'(T_L(y))\|_*^2$ **then** $\triangleright **$
- 8: break
- 9: **end if**
- 10: $L := L \gamma_u$
- 11: $y_k := y, M_k = L, a_{k+1} := a,$
- 12: $L_{k+1} := M_k / \gamma_d, x_{k+1} := T_{M_k}(y_k),$
- 13: $\psi_{k+1}(x) := \psi_k(x) + a_{k+1} [f(x_{k+1}) + \langle \nabla f(x_{k+1}), x - x_{k+1} \rangle + \Psi(x)]$
- 14: вычислить v_{k+1}
- 15: **end while**

Рис. 10: Псевдокод ускоренного многоступенчатого градиентного метода [22]

Оценочная функция $\psi_{k+1}(x)$ исходя из этого псевдокода из оригинальной статьи будет иметь вид:

$$\psi_{k+1}(x) = \psi_k(x) + a_{k+1} [f(x_{k+1}) + \langle \nabla f(x_{k+1}), x - x_{k+1} \rangle + \Psi(x)]. \quad (3.6.17)$$

При $k = 0$ оценочная функция выбрана следующей:

$$\psi_0(x) = \frac{1}{2}\|x - x_0\|^2. \quad (3.6.18)$$

Лемма 6 из оригинальной статьи, показывает, что условие $**$ в псевдокоде выполняется для любого $k \geq 0$ (подробнее см. Лемма 6 в [22]). А также зафиксируем главный результат при доказательстве этой леммы (подробнее см. доказательство Леммы 6 в [22]). Запишем следующее выражение для коэффициента A_k :

$$A_{k+1} \equiv A_k + a_{k+1} = \frac{M_k a_{k+1}^2}{2(1 + \mu A_k)}. \quad (3.6.19)$$

Далее сформулируем и докажем другую Лемму (подробнее см. Лемма 7 в [22]), для оценки сходимости последовательности A_k , поскольку это необходимо в силу механизма оценочных функций чтобы потом подставив в (обосн.) вывести скорость сходимости самого ускоренного метода (подробнее см. доказательство Леммы 2.2.1 в [23])

Лемма*. Для любого $\mu \geq 0$, последовательность коэффициентов масштабирования растет следующим образом:

$$A_k \geq \frac{k^2}{2\gamma_u L_f}, \quad k \geq 0. \quad (3.6.20)$$

Для $\mu > 0$ скорость роста линейна:

$$A_k \geq \frac{1}{\gamma_u L_f} \cdot \left[1 + \sqrt{\frac{\mu}{2\gamma_u L_f}} \right]^{2(k-1)}, \quad k \geq 1.$$

Доказательство.

Действительно, для выражения (*) из псевдокода мы имеем следующее:

$$\begin{aligned} A_{k+1} &\leq A_{k+1}(1 + \mu A_k) = \frac{M_k}{2}(A_{k+1} - A_k)^2 = \frac{M_k}{2} \left[A_{k+1}^{1/2} - A_k^{1/2} \right] \left[A_{k+1}^{1/2} + A_k^{1/2} \right] \\ &\leq 2A_{k+1} M_k \left[A_{k+1}^{1/2} - A_k^{1/2} \right]^2 \leq 2A_{k+1} \gamma_u L_f \left[A_{k+1}^{1/2} - A_k^{1/2} \right]^2. \end{aligned}$$

Таким образом для любого $k \geq 0$ имеем $A_k^{1/2} \geq \frac{k}{\sqrt{2\gamma_u L_f}}$. Если $\mu > 0$, то по тем же причинам, что и выше получаем следующее:

$$\mu A_k A_{k+1} < A_{k+1}(1 + \mu A_k) \leq 2A_{k+1} \gamma_u L_f \left[A_{k+1}^{1/2} - A_k^{1/2} \right].$$

Следовательно, $A_{k+1}^{1/2} \geq A_k^{1/2} \left[1 + \sqrt{\frac{\mu}{2\gamma_u L_f}} \right]$. $A_1 = \frac{1}{M_0} \geq \frac{1}{\gamma_u L_f}$.

Наконец, подставляя вывод леммы* в выражение (3.6.16) мы получим скорость сходимости равную $O(\frac{1}{k^2})$ (подробнее см. Теорема 6 в [22]).

Далее необходимо получить, то каким образом вычисляются \mathbf{T}_{L_k} и $\mathbf{v}_k(\mathbf{y}_k)$. Уже ранее для \mathbf{T}_{L_k} мы имели выражение (3.6.6), но ровно также как и для \mathbf{v}_k согласно (3.6.15). Показано, что мы можем воспользоваться для их явного нахождения следующими выражениями (подробнее см. раздел 2.2 и Лемма 5 [27]):

$$\begin{aligned} \mathbf{T}_{L_k}(\mathbf{y}_k) &= \mathbf{y}_k - \frac{1}{L_k} \nabla f(\mathbf{y}_k), \\ \mathbf{v}_k(\mathbf{y}_k) &= \mathbf{v}_{k-1}(\mathbf{y}_k) - a_k \nabla f(\mathbf{T}_{L_k}(\mathbf{y}_k)), \end{aligned} \quad (3.6.21)$$

Минимизация l_1 регуляризованной функции

$$\phi = f(\mathbf{x}) + \Psi(\mathbf{x}) = \frac{1}{2} \|A\mathbf{x} - \mathbf{b}\|_2^2 + \lambda \|\mathbf{x}\|_1, \quad (3.6.22)$$

здесь первое слагаемое - это функция потерь $f(x) = \frac{1}{2} \|A\mathbf{x} - \mathbf{b}\|_2^2$, штрафующая за отклонения предсказания Ax от целевых значений b , второе же - L_1 -регуляризация $\Psi(x)$, которая "сжимает" коэффициенты (обнуляет те из них, которые малы по модулю), чтобы модель стала проще (см. формула 1 [28]).

Параметр λ называется параметром регуляризации, а для вычисления проекций элементов $\mathcal{P}_\delta^{(j)}(\mathbf{T}_L(\mathbf{y}))$ и $\mathcal{P}_\delta^{(j)}(\mathbf{v}_k)$ будем использовать оператор проектирования (или оператор усадки)

$\mathcal{P}_\delta^{(j)}$ (см. формулу 37 [29]). Проясним смысл этой операции. Если $|\mathbf{T}_\delta(\mathbf{y})^{(j)}| < \delta$, то оператор обнуляет эту компоненту: $\mathcal{P}_\delta^{(j)}(\mathbf{T}) = 0$. Таким образом, оператор усадки исключает малые компоненты вектора.

Условие (***) в предыдущем псевдокоде можно изменить так, чтобы контролировать длину шага (подробнее см. раздел 3.2 [27]), условие (3.6.8) также должно выполняться для любого $L \geq L_f$, перепишем данное условие по-другому для наглядности:

$$f(\mathbf{T}_{L_k}(\mathbf{y}_k)) \leq f(\mathbf{y}_k) + \langle \nabla f(\mathbf{y}_k), \mathbf{T}_{L_k}(\mathbf{y}_k) - \mathbf{y}_k \rangle + \frac{L}{2} \|\mathbf{T}_{L_k}(\mathbf{y}_k) - \mathbf{y}_k\|^2. \quad (3.6.23)$$

Также в работе [25] предлагается ввести "градиентное условие рестарта", которое ускоряет сходимость алгоритма за счет предотвращения зациклования в неоптимальных точках, а также повышает устойчивость алгоритма, обеспечивая согласованность последовательных шагов (подробнее см. раздел 4.2 [27]):

$$\langle \mathbf{y}_k - \mathbf{T}_{L_k}(\mathbf{y}_k), T_{L_{k-1}}(\mathbf{y}_k) - \mathbf{x}_k \rangle > 0. \quad (3.6.24)$$

На Рис. 11 представлен псевдокод ускоренного градиентного метода с модификацией, отличающейся от предыдущего рис.10 в добавлении «оператора усадки». Здесь параметр выпуклости μ_Ψ полагается равным нулю (и поэтому опускается), поскольку $\|\mathbf{x}\|_1$ – выпуклая функция.

Algorithm 10 AMGS modified

Require: x_0 - начальная точка , $L_0 > 0$ - начальная оценка константы Липшица, $\gamma_u > 1$,
 $\gamma_d \geq 1$, $\tilde{v}_0(x) := x_0$, $A_0 := 0$, $\lambda \geq 0$

- 1: Инициализация: $x_0, L_0, \gamma_u, \gamma_d, \tilde{v}_0, A_0, \lambda$.
- 2: **for** $k \geq 0$ **do**
- 3: $v_k := \mathcal{P}_{\lambda A_k}(\tilde{v}_k)$
- 4: **while** true **do**
- 5: найти a из квадратного уравнения $\frac{a^2}{2(A_k+a)} = \frac{1}{L}$ ▷ *
- 6: $y_k = \frac{A_k x_k + a_k v_k}{A_k + a_k}$, $T_L(y_k) = \mathcal{P}_{\lambda \frac{1}{L}}(y_k - \frac{1}{L} \nabla f(y_k))$
- 7: **if** $f(T_{L_k}(y)) \leq f(y_k) + \langle \nabla f(y_k), T_{L_k}(y_k) - y_k \rangle + \frac{L_k}{2} \|T_{L_k}(y_k) - y_k\|^2$ **then** ▷ ***
- 8: break $L := L \gamma_u$
- 9: **end if**
- 10: $L := L \gamma_u$
- 11: $\tilde{v}_{k+1} := \tilde{v}_k - a_{k+1} \nabla f(T_{L_k}(y_k))$,
- 12: $A_{k+1} := A_k + a_{k+1}$ $L_{k+1} := L_k / \gamma_d$
- 13: **if** $\langle y_k - T_{L_k}(y), T_{L_k}(y) - T_{L_{k-1}} \rangle > 0$ **then** ▷ рестарт
- 14: $T_{L_k} := T_{L_{k-1}}$, $\tilde{v}_k := T_{L_{k-1}}$, $A_{k+1} := 0$
- 15: **end if**
- 16: **end while**

Рис. 11: Псевдокод ускоренного градиентного метода с модификацией[27]

Графики, иллюстрирующие оценку константы Липшица и поиска решения на каждом шаге

На следующих графиках убывания значения функции ϕ мы учитываем только точки этой функции во внешнем цикле, а для графиков изменения оценки константы Липшица L_f используются все итерации с учетом внутреннего цикла:

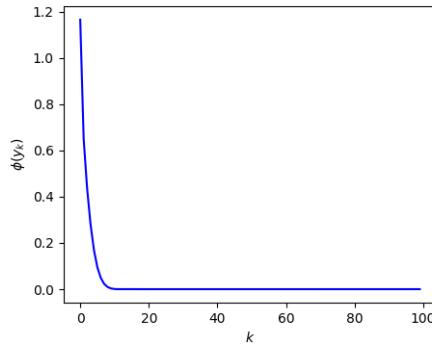


Рис. 12: убывание ϕ , при
 $L_0 = 0.001, \gamma_u = 2, \gamma_d = 2$

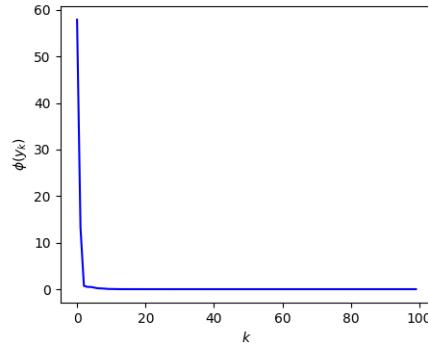


Рис. 13: убывание ϕ , при
 $L_0 = 0.001, \gamma_u = 8, \gamma_d = 2$

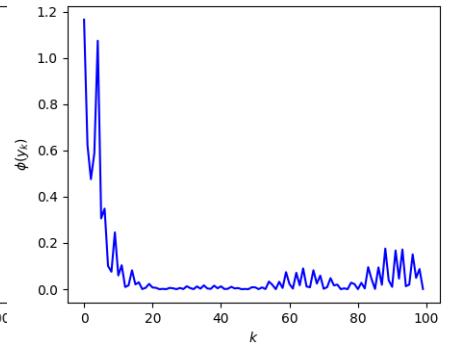


Рис. 14: убывание ϕ , при
 $L_0 = 0.001, \gamma_u = 4, \gamma_d = 3$

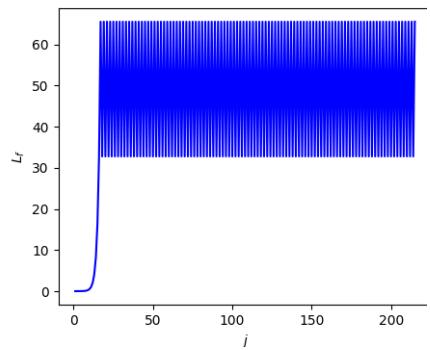


Рис. 15: оценка L_f при $L_0 = 0.001, \gamma_u = 2, \gamma_d = 2$

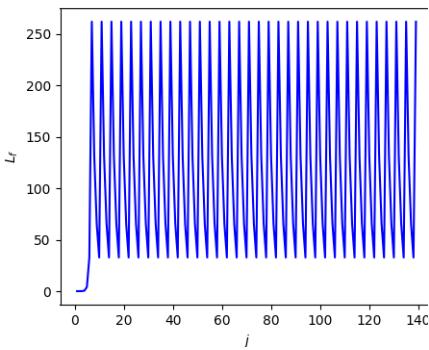


Рис. 16: оценка L_f при $L_0 = 0.001, \gamma_u = 8, \gamma_d = 2$

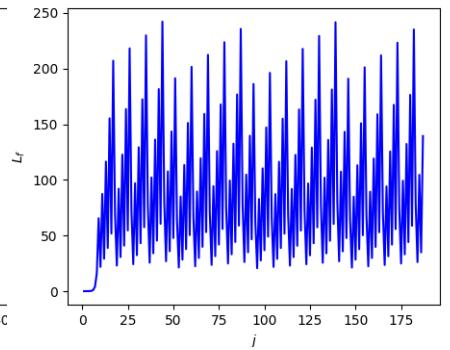


Рис. 17: параметры $L_0 = 0.001, \gamma_u = 4, \gamma_d = 3$

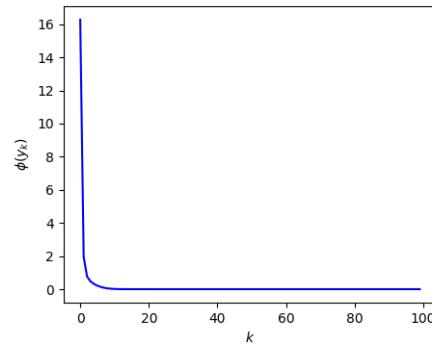


Рис. 18: убывание ϕ , при
 $L_0 = 0.1, \gamma_u = 2, \gamma_d = 2$

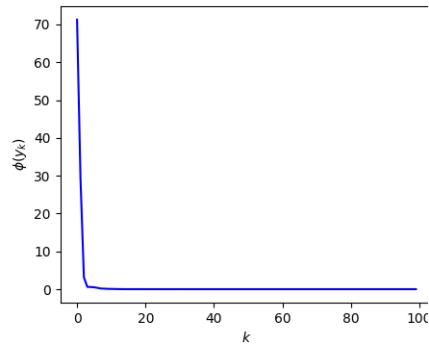


Рис. 19: убывание ϕ , при
 $L_0 = 0.1, \gamma_u = 8, \gamma_d = 2$

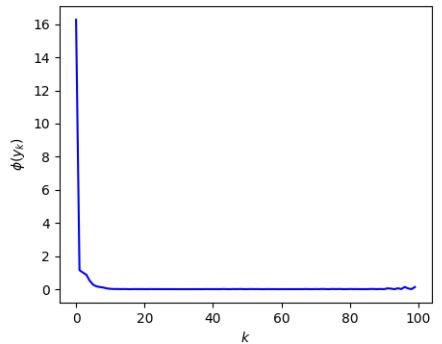


Рис. 20: убывание ϕ , при
 $L_0 = 0.1, \gamma_u = 4, \gamma_d = 3$

4 Обзор фреймворка

4.1 Архитектура фреймворка

Проект представляет собой модульный фреймворк для решения задач оптимизации. В статье описаны основные компоненты структуры, функционал, алгоритмы, а также возможности логирования и дифференцирования, представленные в текущей версии фреймворка. Проект организован следующим образом. Папка Framework/ содержит основные модули проекта. Документация находится в docs/. Функциональные тесты в папке tests/.

Фреймворк реализован на языке Python и включает в себя следующие основные компоненты:

1. Модули оптимизационной модели (Framework/core/optimization_model),
2. Алгоритмы оптимизации (Framework/core/optimization_algorithm),
3. Модуль логирования и отладки (Framework/logger),
4. Модули тестирования алгоритмов (Framework/benchmarks).

В модель оптимизации (см. Рисунок 12) входит несколько суб-модулей: `_constraints.py` (ограничения), `_objective.py` (целевая функция), `_differentiate_wrapper.py` (обертка для дифференцирования), `model.py` (оптимизационная модель) и `norms.py` (подсчет норм). Модуль `_objective.py` содержит класс *Objective*, который представляет собой обертку для работы с функциями произвольной размерности. Класс *Objective* разработан для работы с целевой функцией и ограничениями. Для этого класса определены методы подсчета функции, градиента и гессиана в точке (модуль `_differentiate_wrapper.py`). Класс *Constraints* представляет собой коллекцию ограничений разных типов, таких как равенство, неравенство и простых ограничений на переменные. `norms.py` — это модуль, определяющий различные типы норм для векторов и матриц. Он поддерживает входные данные в виде *NumPy* массивов, *SymPy* матриц или списков. Здесь находятся функции вычисляющие манхэттенскую норму (l_1), евклидову норму (l_2), бесконечную (максимум) норму, норму Фробениуса для матрицы, р-норму вектора, ядерную норму матрицы, трейс-норму (сумму диагональных элементов) матрицы и спектральную норму матрицы.

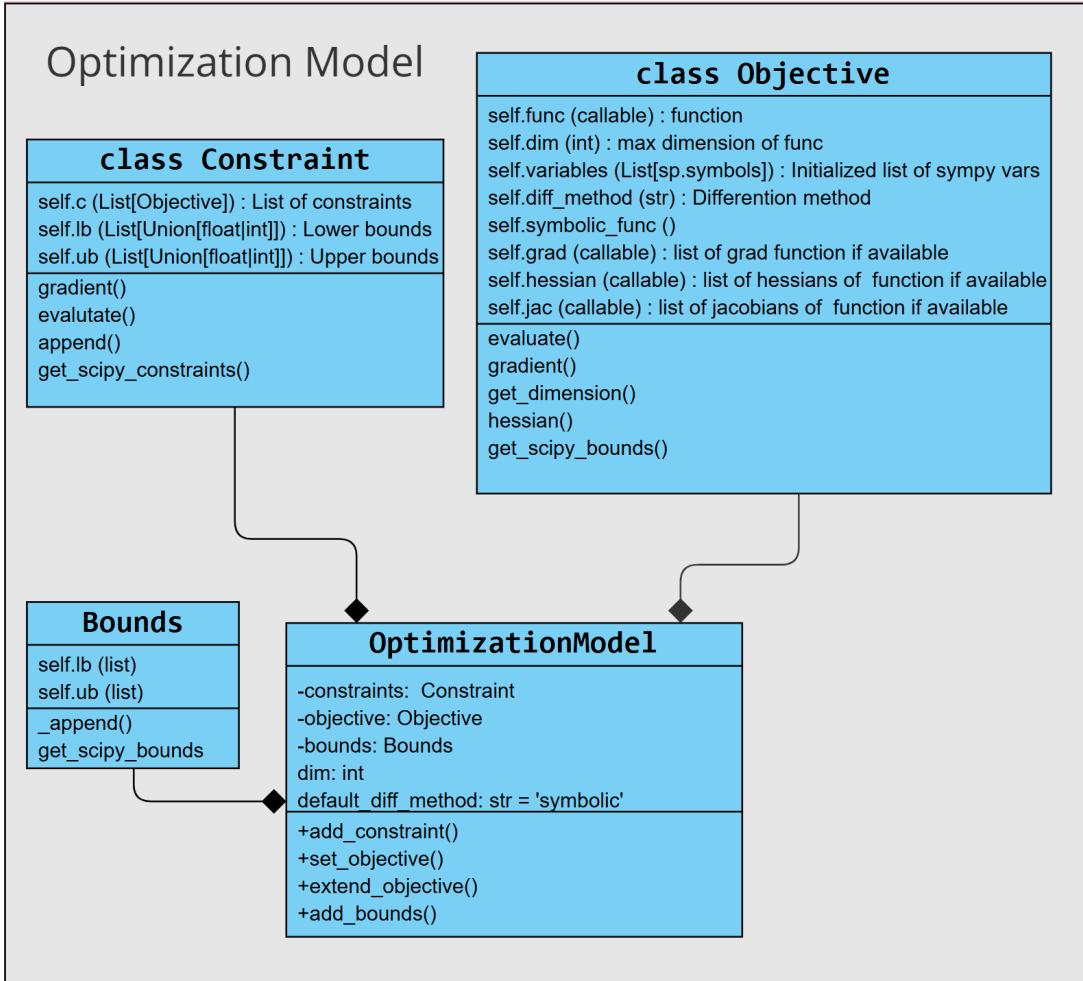


Рис. 21: Схема модели оптимизации

Оптимизационная задача формулируется путем задания целевой функции и ограничений (файл **model.py**). Затем пользователь выбирает метод решения задачи, и вызывает метод **solve()**. При выборе метода требуется указать сопутствующие параметры алгоритма. Типичный пользовательский путь представлен в Листинге 1 на примере *dummy* алгоритма. Файл **dummy.py** представляет собой заглушку алгоритма оптимизации. Он иллюстрирует структуру работы фреймворка, не выполняя реальной оптимизации, но сохраняя ключевые этапы.

```

1 import numpy as np
2 from typing import List
3
4 from framework.core.optimization_algorithm.optimization_algorithm import
5     OptimizationAlgorithm
6 from framework.core.optimization_model.model import OptimizationModel
7 from framework.benchmarks.test_objectives import rosen
8 from framework.core.optimization_model.norms import frobenius_norm
9 from framework.logger.Logger import *
10
11 # Setup loggers
12 loggers = setup_loggers()
13 debug_logger = loggers["debug_logger"]
14 timing_logger = loggers["timing_logger"]
15 stats_logger = loggers["stats_logger"]
16
17 class NonlinearOptimizer(OptimizationAlgorithm):
18     ...
19
20 # Define and Run Optimization
21 n = NonlinearOptimizer(learning_rate=0.01)
22
23
24 def eq1(x):
25     return x[0] + x[1]
26
27 def ineq1(x):
28     return x[0] ** 2 + x[1] ** 3
29
30 model = OptimizationModel()
31 model.set_objective(func=rosen, dim=2, diff_method='numerical')
32 model.add_constraint(func=[eq1], dim=2)
33 model.add_constraint(func=[ineq1], dim=2, lb=[0], ub=[9])
34
35 res = n.solve(model, np.array([2, 1]))
36
37 print(res)

```

Листинг 1: Пример пользовательского пути

Здесь пользовательский путь показывает:

- 1) Как подключить алгоритм в проекте.
- 2) Как задать целевую функцию и ограничения.
- 3) Как работает логирование и диагностика.
- 4) Как проходит процесс оптимизации: вычисление градиента, обновление переменных, контроль ограничений.

В фреймворке представлено несколько алгоритмов оптимизации (см. Рисунок 22):

- Папка IPALM : Inexact power augmented lagrangian method (3.1),
- Папка trust_region: Trust-region (см. 3.2),
- Папка gradient_sampling: SQP-GS (см. 3.3),
- Папка IPOPT: A Line-Search Filter Method (см. 3.4),
- Папка RAdamConstrained - RAdam с добавлением барьерных и штрафных техник (см. 3.5),
- Папка AMGS - ускоренный многоступенчатый градиентный метод Нестерова (см. 3.6).

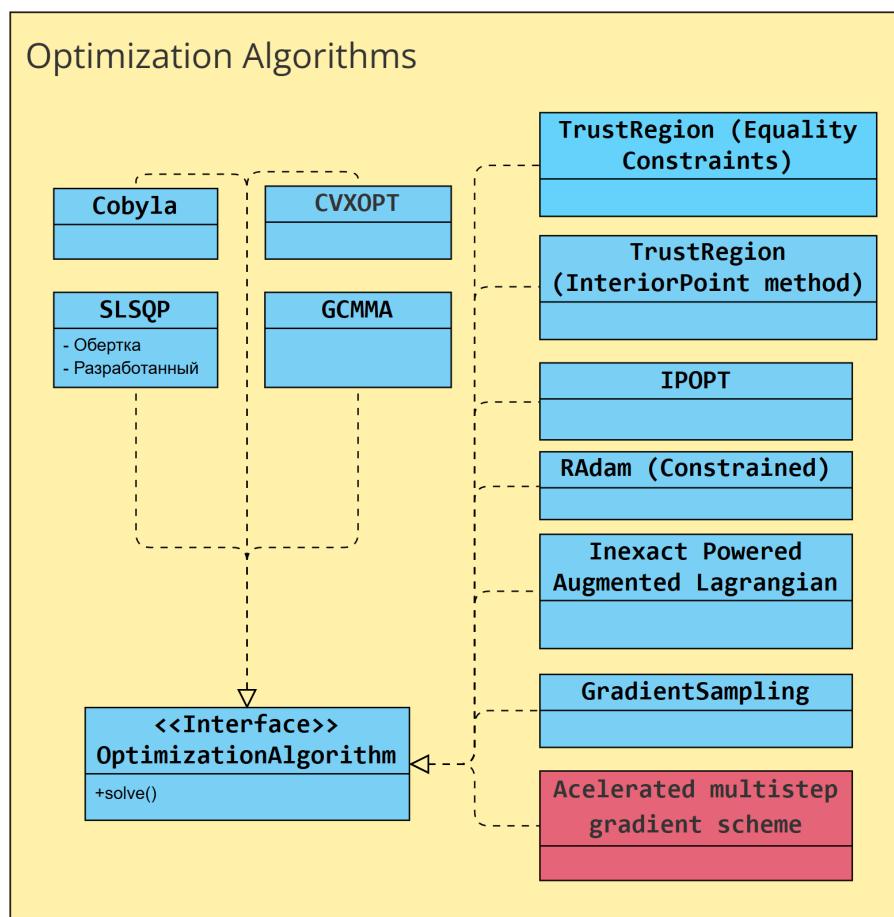


Рис. 22: Алгоритмы в фреймворке

4.2 Логирование

Логирование во фреймворке реализовано с учетом двух логических уровней (см. Рисунок 23):

1. **Пользовательское логирование** – направлено на регистрацию хода оптимизации: постановка задачи, промежуточные итерации, финальные результаты. Оно предназначено для анализа работы метода с позиции прикладного пользователя, без углубления в детали реализации. Реализуется на фоне обёртки методов, реализованных в расширенном логировании (см. пункт 2.). Логи записываются в структурированном (JSON) и читаемом формате, а также реализован интерфейс считывания логов.
2. **Расширенное логирование для разработчиков** – охватывает технические аспекты исполнения кода: входы и выходы функций, время исполнения, структура модели, возникающие исключения. Оно формируется через *debug_logger* и *timing_logger* и используется преимущественно для отладки и профилирования.

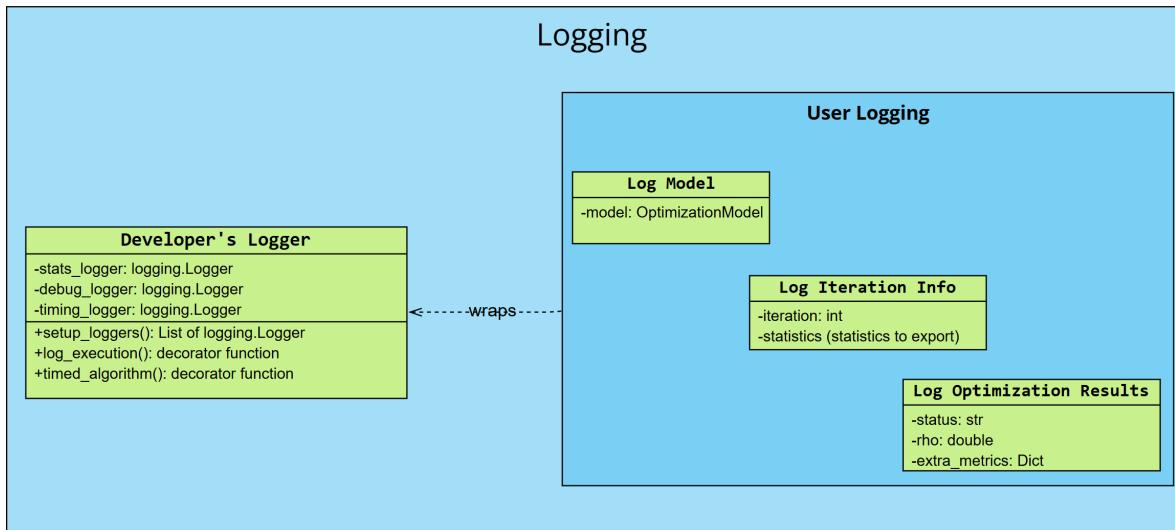


Рис. 23: Логическая структура логирования

4.2.1 Пользовательское логирование

Для регистрации ключевых этапов оптимизационного процесса реализованы три метода пользовательского логирования: *log_model*, *log_iteration_info* и *log_optimization_results*. Важно отметить, что все эти функции реализованы как обёртки над функциями расширенного логирования для разработчиков (см. главу 4.2.2), обработкой и экспортом их результатов занимается *stats_logger* (подробнее см. в главе 4.2.2), который, в свою очередь, логирует данные в файл с расширением *.json* (в JSON-формат) – делается это для удобства их последующего считывания (см. главу 5). Важно отметить, что функции образуют унифицированный интерфейс вывода, одинаково применимый ко всем реализованным алгоритмам во фреймворке. Это означает, что независимо от выбранного метода оптимизации (градиентный спуск, IPALM, барьерный метод и т.д.), логирование задач, итераций и результатов осуществляется через одни и те же функции и в едином формате.

Функция **log_model(...)** (тип логируемого события *OPTIMIZATION_TASK*) логирует начальную постановку задачи. Автоматически извлекаются из переданного *OptimizationModel* (см. Рис. 21) целевая функция (*objective*), ограничения (*constraints*), границы и область определения (*bounds, domain*).

Демонстрация логирования оптимизационной модели показана на Рисунке 24. На изображении показан пример вызова функции внутри метода *solve(...)* (см. Рис. 24a). В результате логирования выводится информация о модели в JSON-формате (см. Рис. 24б). Здесь видно время логирования события типа *OPTIMIZATION_TASK*, целевую функцию, ограничения, границы переменных. Такой вывод позволяет в одном месте увидеть всю информацию о задаче, в том числе сложные составные ограничения. Метод *log_model(...)* используется во всех алгоритмах фреймворка и не зависит от метода оптимизации.

```

def solve(self, model: OptimizationModel, x0: List[float]):
    """
    Main optimization loop.
    Demonstrates full logging usage:
        - Logs the start and end of optimization.
        - Uses Timer to measure overall optimization time.
        - Uses StatsContext to collect step and objective statistics.
        - Logs iterations, constraint violations, and convergence detail
    """
    Logger.log_model(model)
    obj = model.objective

```

(a) Пример вызова в коде

```
{
    "type": "OPTIMIZATION_TASK",
    "time": "01:15:15.604778",
    "objective": "Function (x1 - 1)**2 + 100*(x1**2 - x2)**2",
    "domain": null,
    "bounds": "Bounds [-inf, inf] and [inf, 0]",
    "constraints": [
        "Constraint=c=[\nFunction [x1 + x2 - 1]\n, \nFunction [x1**2 + x2**2 - 4]\n]"
    ]
}
```

(б) Вывод в JSON-формате

Рис. 24: Работа метода `log_model(...)`

`log_iteration_info(...)` (тип логируемого события - *ITERATION_INFO*) используется для логирования данных на каждой итерации оптимизации. Поддерживаются следующие "стандартные" метрики, такие как

1. номер итерации (*iteration*),
2. значение целевой функции (*objective_value*),
3. норма градиента (*gradient_norm*),
4. величина нарушения ограничений (*constraintViolation*),
5. длина шага (*step_size*),
6. значение параметра штрафа (*penalty_parameter*),
7. радиус доверительной области (*trust_region_radius*),
8. количество вызовов функции/градиента (*n_function_evals/n_gradient_evals*) и другие (см. документацию логирования в файле *Logger.py* модуля *Logger* фреймворка).

Так же реализована возможность логирования пользовательских метрик (*extra_metrics*) - реализуется это через передачу новых аргументов функции `log_iteration_info(...)` - пример добавления пользовательской метрики показан на рисунке 28.

```

Logger.log_iteration_info(
    iteration=i,
    objective_value=model.objective.evaluate(x),
    gradient_norm=frobenius_norm(model.objective.gradient(x)),
    constraintViolation=0.0,
    step_size=0.1 * i,
    penalty_parameter=optimizer.mu,
    key="DemoExtra",

    # User-defined extra metric
    extra_metric= i * 42
)

```

(a) Пример вызова в коде

```
{
  "type": "ITERATION_INFO",
  "time": "01:55:44.416742",
  "iter": 6,
  "objective": "90025",
  "||gradient||": 72259.53293277798,
  "constraintViolation": 0,
  "step_size": 0.600000000000001,
  "penalty_parameter": 33.75,
  "trust_region_radius": null,
  "primal_feasibility": null,
  "dual_feasibility": null,
  "n_function_evals": null,
  "n_gradient_evals": null,
  "key": "DemoExtra",
  "extra_metric": 252
}
```

(б) Вывод в JSON-формате

Рис. 25: Пример логирования пользовательской метрики *extra_metric*

Функция *log_iteration_info(...)* поддерживает несколько режимов логирования

1. режим *"full"* - позволяет логировать расширенный набор характеристик (количество вызовов функции, градиента функции, параметр барьера, радиус доверительной области и другие, более подробно см. документацию логирования). По умолчанию режим логирования - *"full"*.
2. режим *"short"* — логирование базовых параметров (номер итерации, значение целевой функции на текущей итерации, норма градиента целевой функции, величина нарушения ограничений и длина шага алгоритма).

Важно отметить, что режим *"full"* добавляет описанные выше характеристики к характеристикам режима *"short"*, тем самым обеспечивая более полную картину собираемой информации.

Приведём пример работы пользовательского логирования на основе простейшего оптимизационного алгоритма фреймворка (см. файл *dummy.py* фреймворка). На рисунке 26 показан вызов функции *log_iteration_info(...)* внутри цикла решения оптимизационной задачи вида (2.1) в методе *solve(...)* класса *NonlinearOptimizer*. Перед вызовом собираются данные для экспорта:

1. *x_new* – значение новой точки;
2. *fval* – значение целевой функции в новой точке;
3. *grad_norm* – норма градиента (вычисляется через норму фробениуса);
4. *maxViolation* – максимальное нарушение ограничений;
5. *step_size* – длина шага;

В нижней части изображения приведены примеры сообщений, сгенерированных для итераций 1 и 2.

The screenshot shows a code editor with Python code for a `NonlinearOptimizer` class. The code implements a gradient descent-like algorithm with trust-region logic. It includes a `log_iteration_info` method that prints JSON-formatted optimization statistics to a `debug_logger`. Below the code, two log entries are shown:

```

21:31:58.557456 [INFO] debug_logger:
Iteration 1
{
    objective: 3.1325326458840515e+20,
    ||gradient||: 2.978392965334161e+16,
    constraintViolation: [1.77082409e+09],
    step_size: 42103.01557107671,
    penalty_parameter: 15.0,
    trust_region_radius: None,
    primal_feasibility: None,
    dual_feasibility: None,
    mu: None,
    n_function_evals: None,
    n_gradient_evals: None,
}
21:31:58.557456 [INFO] debug_logger:
Iteration 2
{
    objective: 7.869153000955188e+59,
    ||gradient||: 0.0,
    constraintViolation: [8.87082465e+28],
    step_size: 297839296533416.06,
    penalty_parameter: 15.0,
    trust_region_radius: None,
    primal_feasibility: None,
    dual_feasibility: None,
    mu: None,
    n_function_evals: None,
    n_gradient_evals: None,
}

```

Рис. 26: Работа метода `log_iteration_info(...)`

Функция `log_optimization_results(...)` (тип события - *OPTIMIZATION_RESULTS*) логирует результаты оптимизационного процесса - её информация позволяет ответить на вопросы: завершена ли оптимизация алгоритмом (флаг *finished*)? Почему завершена (*condition*) (ошибка, превысили максимально допустимое число итераций, сошлись с заданной точностью и т.д.)? Каков результат оптимизации, полученный на последней итерации (значение целевой функции, число потребовавшихся итераций, значение штрафа и др.)?

На фрагменте кода (Рисунок 27) показан вызов функции внутри метода `solve()` пользовательского класса `NonlinearOptimizer`. Здесь:

1. *finished* – логическое значение, полученное в ходе итерационного цикла;
2. *condition* – строка с причиной завершения работы;
3. *final_result* – вектор оптимального решения в списковом формате;
4. *rho* – мера изменения на последней итерации (используется как индикатор сходимости).

Запись оформляется в виде читаемого блока в `debug_logger`, структурированного JSON через `stats_logger`.

```

class NonlinearOptimizer(OptimizationAlgorithm):
    def solve(self, model: OptimizationModel, x0: List[float]):
        with StatsContext("Value of penalized function", iteration=iteration, f_val=penalized_value):
            pass

        #debug_logger.debug(f"Iteration {iteration}: x_new = {x_new.tolist()}, penalized_value = {penalized_value}")

        # Convergence check using the Frobenius norm
        if frobenius_norm(x_new - x_current) < 1e-6:
            #debug_logger.info(f"Convergence reached at iteration {iteration}.")
            finished = True
            x_current = x_new
            break

        x_current = x_new
    else:
        finished = False
    #debug_logger.info(f"Optimization completed. Final solution: {x_current.tolist()}")
    Logger.print_optimization_results(
        finished=finished,
        condition="converged" if finished else "max_iter",
        final_result=x_current.tolist(),
        rho=frobenius_norm(x_new - x_current)
    )
return x_current

23:22:56.190020 [INFO] debug_logger:
[OPTIMIZATION_RESULTS]
{
    finished: True,
    condition: converged,
    final_result: [-297839296469929.8, 3571712962.7607975],
    rho: 0.0,
}
{"type": "OPTIMIZATION_RESULTS", "finished": true, "condition": "converged", "final_result": [-297839296469929.8, 3571712962.7607975], "rho": 0.0}
Final solution: [-297839296469929.8, 3571712962.7607975]

```

Рис. 27: Работа метода `print_optimization_results(...)`

Важно отметить, что при работе инструмента логирования при итеративном процессе, получается множество статистик, укомплектованных построчно (см. Рис. 28а), что может усложнять их последующее чтение пользователем – для этого реализован скрипт `json_print_prettifier.py`, на вход которого подаётся файл формата JSON с собранными статистиками, он выводит его в удобочитаемом для пользователя виде (см. Рис).

4.2.2 Расширенное логирование для разработчиков

Для обеспечения мониторинга, отладки и сбора статистики в фреймворке реализован модуль логирования `Logger.py` в папке **Logger**. Модуль реализует трёхконтурную систему логирования:

- *debug*-контур – для регистрации хода выполнения прикладного кода: вызовов функций, переданных аргументов, возвращаемых значений и обращений к данным;
- *timing*-контур – для замеров времени выполнения операций;
- *stats*-контур – для записи статистики хода алгоритма (итерации, метрики, показатели сходимости).

```

[{"type": "ITERATION_INFO", "time": "01:15:15.606808", "iter": 0, "objective": 3.2000000000000006, "||gradient||": 3.577708763999664, "constraintViolation": 0.0, "step_size": 0.447213595499958, "penalty_parameter": 1.0, "trust_region_radius": null, "primal_feasibility": null, "dual_feasibility": null, "n_function_evals": null, "n_gradient_evals": null, "key": "Dummy"}, {"type": "ITERATION_INFO", "time": "01:15:15.608803", "iter": 1, "objective": 2.048, "||gradient||": 2.862167011199731, "constraintViolation": 0.0, "step_size": 0.3577708763999664, "penalty_parameter": 1.0, "trust_region_radius": null, "primal_feasibility": null, "dual_feasibility": null, "n_function_evals": null, "n_gradient_evals": null, "key": "Dummy"}, {"type": "ITERATION_INFO", "time": "01:15:15.610815", "iter": 2, "objective": 1.3107199999999999, "||gradient||": 2.2897336089597844, "constraintViolation": 0.0, "step_size": 0.28621670111997305, "penalty_parameter": 1.0, "trust_region_radius": null, "primal_feasibility": null, "dual_feasibility": null, "n_function_evals": null, "n_gradient_evals": null, "key": "Dummy"}, {"type": "ITERATION_INFO", "time": "01:15:15.612782", "iter": 3, "objective": 0.8388608, "||gradient||": 1.8317868871678278, "constraintViolation": 0.0, "step_size": 0.22897336089597847, "penalty_parameter": 1.0, "trust_region_radius": null, "primal_feasibility": null, "dual_feasibility": null, "n_function_evals": null, "n_gradient_evals": null, "key": "Dummy"}, {"type": "ITERATION_INFO", "time": "01:15:15.614402", "iter": 4, "objective": 0.5368709120000001, "||gradient||": 1.4654295097342624, "constraintViolation": 0.0, "step_size": 0.1831786887167828, "penalty_parameter": 1.0, "trust_region_radius": null, "primal_feasibility": null, "dual_feasibility": null, "n_function_evals": null, "n_gradient_evals": null, "key": "Dummy"}, {"type": "ITERATION_INFO", "time": "01:15:15.616423", "iter": 5, "objective": 0.3435973836800015, "||gradient||": 1.1723430778741, "constraintViolation": 0.0, "step_size": 0.14654295097342626, "penalty_parameter": 1.0, "trust_region_radius": null, "primal_feasibility": null, "dual_feasibility": null, "n_function_evals": null, "n_gradient_evals": null, "key": "Dummy"}, {"type": "ITERATION_INFO", "time": "01:15:15.618396", "iter": 6, "objective": 0.2199823255520012, "||gradient||": 0.9378748862299281, "constraintViolation": 0.0, "step_size": 0.11723436077874101, "penalty_parameter": 1.0, "trust_region_radius": null, "primal_feasibility": null, "dual_feasibility": null, "n_function_evals": null, "n_gradient_evals": null, "key": "Dummy"}, {"type": "ITERATION_INFO", "time": "01:15:15.620411", "iter": 7, "objective": 0.14073748835532804, "||gradient||": 0.758299988939424, "constraintViolation": 0.0, "step_size": 0.09378748862299281, "penalty_parameter": 1.0, "trust_region_radius": null, "primal_feasibility": null, "dual_feasibility": null, "n_function_evals": null, "n_gradient_evals": null, "key": "Dummy"}]

```

(а) Пример логирования итеративного процесса

```

{
    "type": "ITERATION_INFO",
    "time": "01:15:15.606808",
    "iter": 0,
    "objective": 3.2000000000000006,
    "||gradient||": 3.577708763999664,
    "constraintViolation": 0.0,
    "step_size": 0.447213595499958,
    "penalty_parameter": 1.0,
    "trust_region_radius": null,
    "primal_feasibility": null,
    "dual_feasibility": null,
    "n_function_evals": null,
    "n_gradient_evals": null,
    "key": "Dummy"
}

```

(б) Пример работы скрипта

Рис. 28: Пример вывода *json_print_prettifier.py*

Для реализации этих уровней логирования используются как стандартные средства библиотеки *logging*, так и сторонние расширения, такие как *python-json-logger* и *decorator*, что позволяет поддерживать формат JSON и автоматическое измерение времени выполнения. Основные компоненты модуля:

1. Средства форматирования (*CustomFormatter*, *CustomJsonFormatter*)

Определяют формат вывода сообщений логов. Поддерживают высокоточные метки времени (вплоть до микросекунд) с помощью переопределения метода *formatTime()* на основе стандартного поведения базового класса *logging.Formatter*. Это критически важно при работе с короткими итерациями или высокочастотными событиями в оптимизационных методах.

2. Функция настройки логеров (*setup_loggers*)

Основная точка входа для инициализации системы логирования. Возвращает три логера с разным назначением (отладка, тайминг, метрики). Все сообщения могут записываться как в консоль, так и в файл.

3. Декораторы (*log_execution*, *timed_algorithm*)

Упрощают добавление логирования в существующие функции. Декораторы позволяют автоматически фиксировать вызов функции, её аргументы, результат и время выполнения.

4. Контекстные менеджеры (*Timer*, *StatsContext*)

Обеспечивают удобный способ измерения времени и логирования статистики для про-

извольных блоков кода, что особенно полезно при анализе сходимости или мониторинге качества решения на разных этапах оптимизации.

Центральной функцией является `setup_loggers()`, которая создаёт и настраивает три независимых логера: `debug_logger` для логирования действий на прикладном уровне, `timing_logger` для измерения времени выполнения, `stats_logger` для сбора статистики в формате JSON. Каждый логер настраивается с отдельными обработчиками для вывода на консоль (`stdout`, `stderr`) и записи в единый лог-файл. Её параметрами являются:

- `debug_level`, `timing_level`, `stats_level`

Уровни логирования для каждого логера (по умолчанию — DEBUG для отладки и тайминга, INFO для статистики).

- `stats_file_path`

Путь к лог-файлу, куда будут записываться все сообщения. Может быть переопределён через переменную окружения `STATS_LOG_FILE`.

- `time_format`

Формат временных меток (например, `%H:%M:%S.%f`), задаёт точность времени (до миллисекунд или микросекунд).

Реализовано две степени глубины логирования: первая степень логирует основные события обращения к данным, вызова функций, события использования других логеров (`stats_logger`, `timing_logger`), вторая степень логирует более подробно сам итерационный процесс алгоритма. На рисунке 29 показаны разные уровни логирования.

```
00:21:46.155213 [INFO] debug_logger: Entering function solve args=(<__main__.Dummy object at 0x000001A6A079C140>,
<__main__.MiniModel object at 0x000001A6A104BAA0>,
[2.0, 1.0]), kwargs={}
00:21:46.155213 [INFO] debug_logger: [OPTIMIZATION_TASK] Exporting optimization model to JSON (stats_logger working)
00:21:46.156186 [INFO] debug_logger: Starting optimization, x0 = [2.0, 1.0]
00:21:46.156186 [TIMING] Entering context: Overall Optimization Time
00:21:46.158217 [INFO] debug_logger: [ITERATION_INFO] entered iterative process (stats_logger working)
00:21:46.194344 [TIMING] Exiting context: overall Optimization Time (elapsed: 0.0380 seconds)
00:21:46.194344 [INFO] debug_logger: [OPTIMIZATION_RESULTS] exporting results to JSON (stats_logger working)
00:21:46.194344 [TIMING] solve executed in 0.0392 seconds
00:21:46.195344 [INFO] debug_logger: Exiting function solve with result=[0.02305843 0.01152922]
```

(a) Логирование при `debug_level=1`

```
00:39:06.587309 [INFO] debug_logger: Entering function solve args=(<__main__.Dummy object at 0x000002F735294E60>,
<__main__.MiniModel object at 0x000002F735186CC0>,
[2.0, 1.0]), kwargs={}
00:39:06.610822 [INFO] debug_logger: [OPTIMIZATION_TASK] Exporting optimization model to JSON (stats_logger working)
00:39:06.610822 [INFO] debug_logger: Starting optimization, x0 = [2.0, 1.0]
00:39:06.610822 [TIMING] Entering context: Overall Optimization Time
00:39:06.708597 [INFO] debug_logger: [ITERATION_INFO] entered iterative process (stats_logger working)
00:39:06.708597 [INFO] debug_logger: [ITERATION_INFO] Exporting iteration 0 results to JSON (stats_logger working)
00:39:06.710717 [INFO] debug_logger: [ITERATION_INFO] Exporting iteration 1 results to JSON (stats_logger working)
00:39:06.713264 [INFO] debug_logger: [ITERATION_INFO] Exporting iteration 2 results to JSON (stats_logger working)
00:39:06.715123 [INFO] debug_logger: [ITERATION_INFO] Exporting iteration 3 results to JSON (stats_logger working)
00:39:06.717248 [INFO] debug_logger: [ITERATION_INFO] Exporting iteration 4 results to JSON (stats_logger working)
00:39:06.719241 [INFO] debug_logger: [ITERATION_INFO] Exporting iteration 5 results to JSON (stats_logger working)
00:39:06.721239 [INFO] debug_logger: [ITERATION_INFO] Exporting iteration 6 results to JSON (stats_logger working)
00:39:06.723232 [INFO] debug_logger: [ITERATION_INFO] Exporting iteration 7 results to JSON (stats_logger working)
00:39:06.725229 [INFO] debug_logger: [ITERATION_INFO] Exporting iteration 8 results to JSON (stats_logger working)
00:39:06.728119 [INFO] debug_logger: [ITERATION_INFO] Exporting iteration 9 results to JSON (stats_logger working)
00:39:06.729673 [INFO] debug_logger: [ITERATION_INFO] Exporting iteration 10 results to JSON (stats_logger working)
00:39:06.731708 [INFO] debug_logger: [ITERATION_INFO] Exporting iteration 11 results to JSON (stats_logger working)
00:39:06.734332 [INFO] debug_logger: [ITERATION_INFO] Exporting iteration 12 results to JSON (stats_logger working)
00:39:06.736267 [INFO] debug_logger: [ITERATION_INFO] Exporting iteration 13 results to JSON (stats_logger working)
00:39:06.737803 [INFO] debug_logger: [ITERATION_INFO] Exporting iteration 14 results to JSON (stats_logger working)
00:39:06.739834 [INFO] debug_logger: [ITERATION_INFO] Exporting iteration 15 results to JSON (stats_logger working)
00:39:06.741853 [INFO] debug_logger: [ITERATION_INFO] Exporting iteration 16 results to JSON (stats_logger working)
00:39:06.743792 [INFO] debug_logger: [ITERATION_INFO] Exporting iteration 17 results to JSON (stats_logger working)
00:39:06.745574 [INFO] debug_logger: [ITERATION_INFO] Exporting iteration 18 results to JSON (stats_logger working)
00:39:06.747640 [INFO] debug_logger: [ITERATION_INFO] Exporting iteration 19 results to JSON (stats_logger working)
00:39:06.748569 [TIMING] Exiting context: overall optimization Time (elapsed: 0.1374 seconds)
00:39:06.748569 [INFO] debug_logger: [OPTIMIZATION_RESULTS] exporting results to JSON (stats_logger working)
00:39:06.748569 [TIMING] solve executed in 0.1389 seconds
00:39:06.841953 [INFO] debug_logger: Exiting function solve with result=[0.02305843 0.01152922]
Final solution: [0.02305843 0.01152922]
```

(б) Логирование при `debug_level=2`

Рис. 29: Вывод `debug_logger` при различных значениях параметра `debug_level`

Для обеспечения гибкой и высокоточной регистрации событий логирования в модуле реализованы два пользовательских формировщика логов:

1. **CustomFormatter**

Предназначен для форматирования логов, выводимых в консоль и текстовые файлы. Поддерживает произвольные шаблоны времени, включая микросекунды (через %f), что особенно важно при анализе быстродействия итеративных алгоритмов. Форматтер расширяет стандартный *logging.Formatter*, переопределяя метод *formatTime*;

2. **CustomJsonFormatter**

Используется для генерации структурированных JSON-логов, преимущественно через *stats_logger*. Наследуется от *pythonjsonlogger.JsonFormatter* и аналогично переопределяет метод *formatTime*. JSON-структура логов, созданная с этим форматтером, легко поддаётся парсингу и анализу средствами Python (например, *pandas.read_json(...)*), что удобно для хранения статистики оптимизационного процесса (итерации, значения функции, штрафы и др.).

Для автоматизации логирования во время выполнения функций реализованы два декоратора. Они позволяют добавить отслеживание вызовов и времени исполнения без вмешательства в основной код функций. *log_execution* логирует вход в функцию и выход из неё, включая имя функции, переданные аргументы и возвращаемое значение. Он необходим, чтобы видеть, какие функции вызываются, с какими параметрами и что они возвращают. Используется преимущественно с *debug_logger* для отслеживания ключевых вызовов и отладки алгоритма. Поддерживает настройку логгера и уровня сообщений. *timed_algorithm* измеряет и логирует время выполнения функции, чтобы видеть сколько времени заняла функция. Лог сообщения оформляется через *timing_logger*. Если функция возвращает словарь, то декоратор автоматически добавляет в него поле '*runtimes*' с продолжительностью выполнения в секундах.

Контекстные менеджеры предоставляют инструмент для логирования характеристик произвольных блоков кода без необходимости выносить их в отдельные функции. Это удобно, когда необходимо собрать статистику или измерить время внутри сложных функций или циклов.

Timer – простой контекстный менеджер для измерения времени выполнения кода. При входе и выходе из блока записывает соответствующие сообщения в *timing_logger*, включая затраченное время. *StatsContext* – более универсальный инструмент, предназначенный для регистрации структурированных метрик. На вход принимает название блока (*label*), метаданные в виде именованных аргументов (*kwargs*), флаг *json_export* (если включён, то лог записывается в JSON-формате через *stats_logger*). При выходе из контекста логируются как время, так и переданные метрики. Также фиксируется исключение, если оно возникло в блоке (оно попадает в лог).

4.3 Модуль тестирования

Для системной оценки эффективности различных алгоритмов оптимизации во фреймворке реализован модуль тестирования **benchmarks** (Рис. 30). Он обеспечивает автоматизированную генерацию задач, запуск тестов, логгирование результатов, визуализацию и анализ поведения методов.

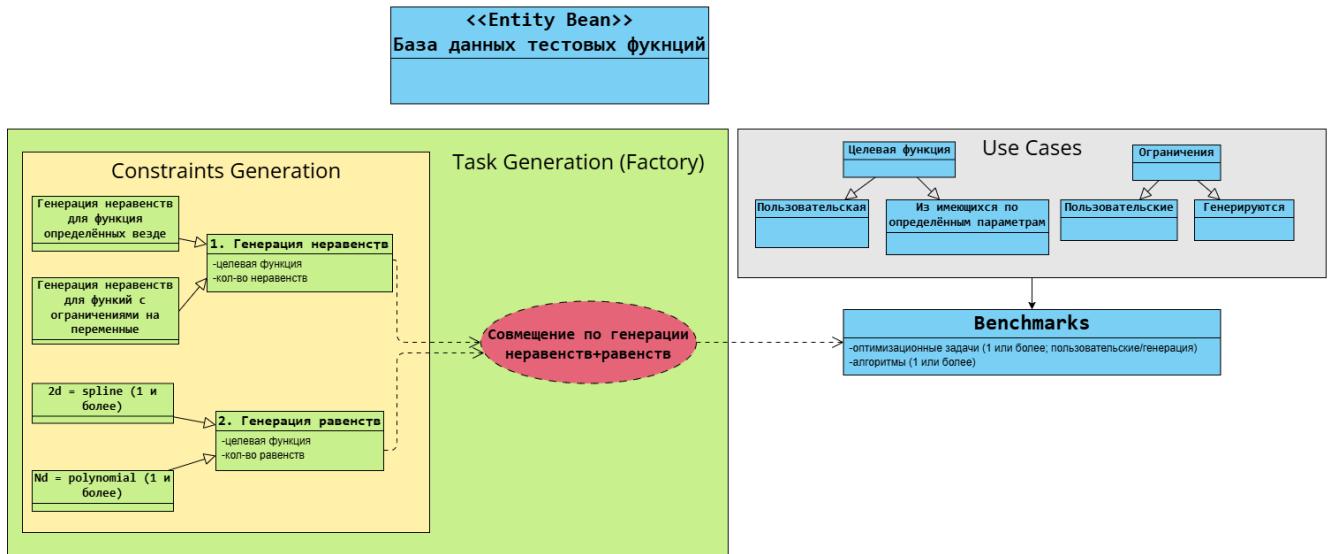


Рис. 30: Структура **benchmarks**

4.3.1 База целевых функций.

База функций (*test_objectives.py*) включает более 70 известных тестовых задач, различающихся по структуре, сложности и аналитическим свойствам. Это позволяет проверять алгоритмы на устойчивость к особенностям ландшафта, размерности и условиям задачи. Каждая функция описывается:

1. названием (ключ в словаре *Tests().tests*),
2. callable-объектом с реализацией `__call__`,
3. метками характеристик (*labels*):
например, "Scalable", "Multimodal", "Lipschitz", "Strictly_convex", и др.,
4. в некоторых случаях — аналитически заданными точками глобального и локального минимума.

Это позволяет проводить выборку по классам задач и контролировать масштабируемость и сложность экспериментов. На текущий момент реализованы следующие классы функций:

- 1) **Выпуклые** (например, *Quadratic*, *BoothFunction*) — обладают глобальным минимумом и полезны для базовой проверки корректности;
- 2) **Невыпуклые** (*Rosenbrock*, *Himmelblau*) — имеют несколько экстремумов, что важно для тестирования глобальных методов;

- 3) **Мультимодальные** (*Eggholder, Ackley*) — содержат множество локальных минимумов;
- 4) **Недифференцируемые** — функции с разрывами в производной, проверяют устойчивость к негладкости;
- 5) **Масштабируемые** (*Scalable*) — функции, размерность которых можно произвольно увеличивать (например, *Sphere, Rastrigin*);
- 6) **Стохастические** — (на перспективу) функции с шумом или случайным компонентом;
- 7) Задачи с аналитически известным минимумом — позволяют сравнивать найденное решение с эталонным.

Каждая функция реализуется как класс с методом `__call__`, и может предоставлять градиенты, гессианы, локальные и глобальные минимумы.

Для гибкой фильтрации функций в базе каждая из них сопровождается набором меток (*labels*), отражающих её свойства. Поддерживаются следующие метки:

1. 'Convex', 'Nonconvex' - выпуклая/невыпуклая функция;
2. 'Smooth' - функция непрерывно дифференцируема;
3. 'Nonsmooth' - функция не является всюду гладкой;
4. 'Multimodal', 'Unimodal' - функция содержит множество локальных минимумов/один локальный минимум;
5. 'Scalable' - Функция допускает масштабирование размерности;
6. 'Known_minimum' - Для функции аналитически известны точка и значение глобального минимума;
7. 'Analytical_gradient', 'Analytical_hessian' - У функции задан точный аналитический градиент/гессиан.

Метки хранятся в атрибуте *labels* каждого объекта функции и используются при фильтрации тестов через параметр *test_classes*

Чтобы добавить новую функцию в базу, необходимо создать класс, реализующий интерфейс, зарегистрировать её в словаре *Tests().tests*, и также опционально можно указать известный минимум. (Листинг 2)

```

1 #1)
2 class MyFunction:
3     def __init__(self): ...
4     def __call__(self, x): ...
5     def gradient(self, x): ...
6     def get_labels(self): return [ 'Smooth', 'Convex' ]
7 #2)
8 self.tests["MyFunction"] = {
9     "func": MyFunction(),
10    "label": [ "Smooth", "Convex" ]
11 }
12 #3)
13 "minimum": {
14     "point": [ 0.0, 0.0 ],
15     "value": 0.0
16 }
```

Листинг 2: Пример добавления новой функции

4.3.2 Генерация ограничений

Фреймворк поддерживает динамическую генерацию ограничений, что позволяет создавать обширные и разнообразные тестовые задачи с ограничениями без ручного задания. Генерация реализована в модуле *polynomial_constraint.py* и охватывает как алгебраические ограничения, так и геометрические формы (например, замкнутые кривые).

Основным механизмом генерации ограничений служит функция *generate_polynomial(...)* (её псевдокод представлен на Рис. 31), которая возвращает объект **Constraint**. Ограничение формируется как сравнение значения случайного многочлена с константой по выбранной логической функции ($<$, \leq , $=$, \geq , $>$).

Параметры генерации:

1. *var* — число переменных;
2. *degree* — максимальная степень мономов;
3. *quantity* — количество слагаемых (мономов) в полиноме;
4. *low*, *high* — диапазон случайных коэффициентов;
5. *func* — функция сравнения (если не задана, выбирается случайно);
6. *const* — правая часть сравнения (если не задана, выбирается случайно);
7. *seed* — параметр управления воспроизводимостью.

Генератор также предотвращает дублирование мономов и проверяет допустимость параметров.

Algorithm 11 generate_polynomial(...)

Require: n — число переменных; d, q, s — параметры; low, high — границы коэффициентов;

$\mathbf{p} = \text{None}$ — фиксированная точка

- 1: $\text{rng} \leftarrow s$
- 2: Сгенерировать $\mathbf{a} = [a_1, \dots, a_q] \in [\text{low}, \text{high}]$
- 3: Выбрать q уникальных мономов степени $\leq \mathbf{d}$ с векторами степеней \mathbf{d}_i
- 4: **if** const = None **then**
- 5: Сгенерировать const $\in [\text{low}, \text{high}]$
- 6: **end if**
- 7: $\mathbf{x} \leftarrow (x_0, \dots, x_{n-1})$
- 8: $f(\mathbf{x}) \leftarrow \sum_{i=1}^q a_i \mathbf{x}^{\mathbf{d}_i}$ ▷ построение полинома
- 9: **if** $\mathbf{p} \neq \text{None}$ **then**
- 10: $f(\mathbf{x}) \leftarrow f(\mathbf{x}) - f(\mathbf{p})$
- 11: **end if**
- 12: **if** typ = None **then**
- 13: typ $\leftarrow 1$ ▷ по умолчанию: неравенство
- 14: **end if**
- 15: constraint $\leftarrow \text{Constraint}()$ (см. главу 4.1)
- 16: **if** typ = 0 **then**
- 17: Добавить ограничение $f(\mathbf{x}) = 0$ в constraint
- 18: **else**
- 19: Добавить ограничение $f(\mathbf{x}) \geq 0$ в constraint
- 20: **end if**
- 21: **return** constraint

Рис. 31: Псевдокод функции *generate_polynomial(...)*

Для поддержки генерации ограничений в модуле реализован ряд вспомогательных процедур. Функция *monom_degree(...)* отвечает за генерацию разбиения заданной степени на степени отдельных переменных монома. Например, при степени 3 и трёх переменных возможным результатом будет разбиение $[2, 1, 0]$, соответствующее мономам вида $\mathbf{x}_1^2 \mathbf{x}_2^1 \mathbf{x}_3^0$. Функция *generate_random_points(...)* создаёт набор случайных точек в двумерном пространстве. Такие точки могут использоваться в качестве контрольных при построении геометрических ограничений, основанных на кривых.

Для получения упорядоченной последовательности, формирующей замкнутую фигуру, используется функция *order_points(...)*, которая сортирует сгенерированные точки по углу относительно центра масс. Это позволяет формировать непрерывные контуры, пригодные для дальнейшего построения ограничивающих областей.

Проверка корректности формы кривой осуществляется с помощью *has_self_intersections(...)*, определяющей наличие самопересечений у построенного многоугольного контура.

Генерация ограничений типа равенств. Для построения более сложных тестов написана функция `generate_multiple_polynomials(...)`. Эта функция генерирует разреженные многочлены n переменных, $f_i(\mathbf{x}) = 0$ - задают какую-то гиперповерхность, что пересекаются минимум в одной точке. Это позволяет проверять устойчивость и корректность работы алгоритмов оптимизации в условиях, где множество допустимых решений определяется сложной системой ограничений. Генерация кривых осуществляется по следующему алгоритму:

1. Сначала выбирается фиксированная точка пересечения в заданной области — она будет общей для всех кривых;
2. Для каждой кривой генерируется множество случайных точек, включающее эту общую точку;
3. Точки сортируются по координате \mathbf{x} для обеспечения возможности интерполяции явной функции $\mathbf{y} = f(\mathbf{x})$;
4. Далее, через отсортированные точки строится кривая с помощью кубической сплайн-интерполяции;
5. Полученные кривые сохраняются в виде набора точек и визуализируются для наглядности.

Такой подход позволяет формировать сложные кусочно-заданные ограничения, пригодные как для численного анализа поведения оптимизаторов, так и для визуального контроля корректности пересечения областей.

Наконец, для генерации геометрических ограничений в двумерном пространстве используется `generate_random_curves(...)`, реализованная в модуле `curve_constraint.py`. Она предназначена для построения системы равенств, заданных через кусочно-заданные кубические сплайны. Каждое такое равенство задаётся как выражение $\mathbf{y} - S_i(\mathbf{x}) = 0$, где $S_i(\mathbf{x})$ — кубический сплайн, проходящий через случайно выбранные точки. Ключевая особенность метода `generate_random_curves(...)` — гарантированное наличие общей точки пересечения у всех кривых. На рисунке 32 показан результат работы функции `generate_random_curves(...)`, при генерации которой были использованы параметры: `num_lists=3`, `num_points=5`, `x_range=(100, 100)`, `y_range=(100, 100)`, `fixed_point=(5, 5)`. Все три кривые построены через случайные точки, но гарантированно пересекаются в одной общей фиксированной точке (помечена красной звездой). Каждая кривая задаёт равенство вида $\mathbf{y} = S_i(\mathbf{x})$, где S_i — кубический сплайн, построенный по отсортированным по \mathbf{x} точкам. Такой способ задания ограничений позволяет проверять работу оптимизаторов в условиях тесных и потенциально конфликтующих равенств.

Генерация ограничений типа неравенств. Функции `random_inequalities(...)` и `random_inequalities_with_bound(...)` генерируют систему случайных нелинейных неравенств, чье решение не пусто. Рассмотрим ключевые геометрические объекты, используемые в задаче.

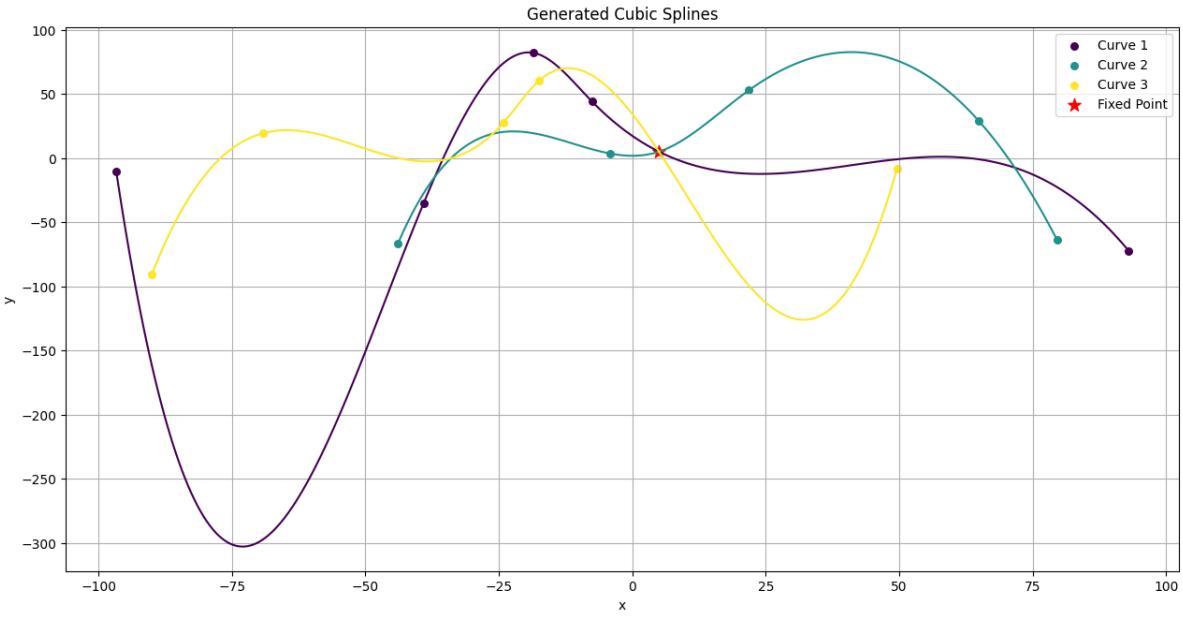


Рис. 32: Генерация системы ограничений в виде сплайнов

n -мерным параллелепипедом называют множество таких точек $\mathbf{x} = (x_1, \dots, x_n)$, которые удовлетворяют следующему условию:

$$\max_{1 \leq i \leq n} \left| \frac{2(x_i - p_i)}{a_i} \right| \leq 1, \quad (4.3.2.1)$$

где p_1, \dots, p_n - центр параллелепипеда, a_1, \dots, a_n - стороны параллелепипеда. Далее под прямоугольником будем подразумевать n -мерный параллелепипед.

р-эллипсом (или псевдоокружностью, если полуоси равны) называют множество таких точек $\mathbf{x} = (x_1, \dots, x_n)$, удовлетворяющих следующему условию:

$$\sqrt[p]{\sum_{i=1}^n \left| \frac{x_i - c_i}{r_i} \right|^p} = 1, \quad (4.3.2.2)$$

где c_1, \dots, c_n - центр окружности, r_1, \dots, r_n - полуоси окружности, $p \geq 1$ - параметр нормы. Иначе говоря, это эллипс в p -норме (при $p = 2$ получаем стандартную окружность в евклидовой норме, для простоты далее будем называть просто окружностью/кругом, при $p = 1$ получаем ромб, см. Рис 33): $\|\frac{\mathbf{x}-\mathbf{c}}{\mathbf{r}}\|_p = 1$.

Область значений функции — это множество всех возможных значений, которые может принимать функция. Другими словами, это набор входных значений, для которых функция определена. Например, для функции $f(\mathbf{x}) = \sqrt{\mathbf{x}}$ область значений будет ограничен неотрицательными числами.

Лемма 1 (О содержании прямоугольника в окружности). *Пусть прямоугольник, а также стороны ограничения известны заранее. Тогда область допустимых значений центра окружности, что полностью содержит прямоугольник, описывается следующим условием*

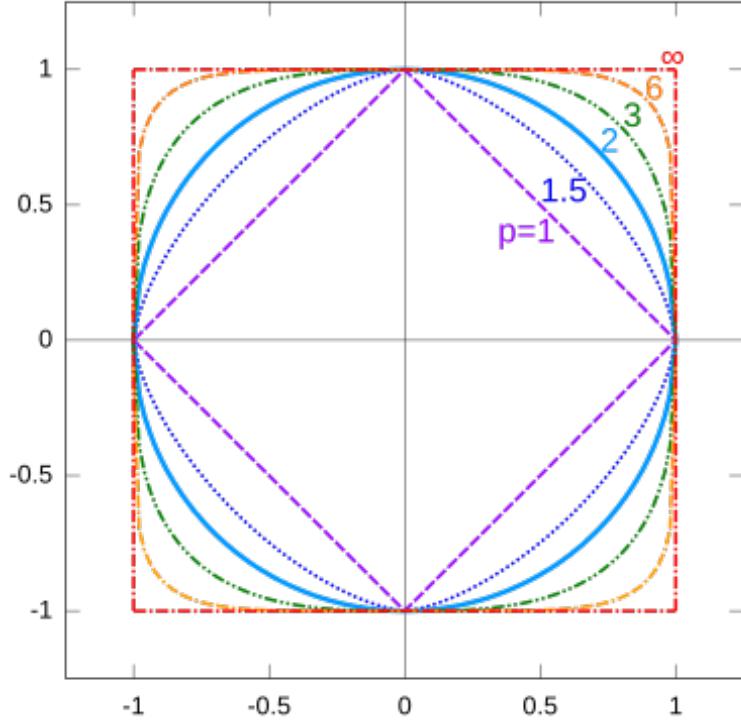


Рис. 33: Поведение единичных ”псевдоокружностей” в \mathbb{R}^2 , основанных на различных p -нормах. В алгоритме используется параметр нормы $p \geq 1$.

ем:

$$\sqrt[p]{\sum_{i=1}^n \left| \frac{p_i - c_i}{r_i} \right|^p} \leq S \Leftrightarrow \sum_{i=1}^n \left| \frac{p_i - c_i}{r_i} \right|^p \leq S^p,$$

$$\text{где } S = 1 - \sqrt[p]{\sum_{i=1}^n \left| \frac{a_i}{2r_i} \right|^p}, \quad \sqrt[p]{\sum_{i=1}^n \left| \frac{a_i}{2r_i} \right|^p} < 1.$$

Стоит отметить, что это условие не описывает все возможные центры ограничений. Оно говорит лишь о том, что если условие выполнено, прямоугольник лежит в круге. Обратное не верно. Если условие нарушается, возможно, что прямоугольник все еще внутри.

Доказательство.

Для любого \mathbf{x} , лежащего в прямоугольнике, справедливо следующее неравенство:

$$|p_i - x_i| \leq \frac{a_i}{2}, \quad \forall i = 1, \dots, n. \quad (4.3.2.3)$$

Учитывая (4.3.2.3), а также свойство p -нормы: $\|\mathbf{x} + \mathbf{y}\|_p \leq \|\mathbf{x}\|_p + \|\mathbf{y}\|_p$, получим:

$$\begin{aligned} \sqrt[p]{\sum_{i=1}^n \left| \frac{x_i - c_i}{r_i} \right|^p} &= \sqrt[p]{\sum_{i=1}^n \left| \frac{x_i - p_i + p_i - c_i}{r_i} \right|^p} \leq \sqrt[p]{\sum_{i=1}^n \left| \frac{x_i - p_i}{r_i} \right|^p} + \sqrt[p]{\sum_{i=1}^n \left| \frac{p_i - c_i}{r_i} \right|^p} \leq \\ &\leq \sqrt[p]{\sum_{i=1}^n \left| \frac{p_i - c_i}{r_i} \right|^p} + \sqrt[p]{\sum_{i=1}^n \left| \frac{a_i}{2r_i} \right|^p} \leq 1 \implies \sqrt[p]{\sum_{i=1}^n \left| \frac{p_i - c_i}{r_i} \right|^p} \leq 1 - \sqrt[p]{\sum_{i=1}^n \left| \frac{a_i}{2r_i} \right|^p}, \end{aligned}$$

$$\text{где } \sqrt[p]{\sum_{i=1}^n \left| \frac{a_i}{2r_i} \right|^p} < 1.$$

Следовательно, для любого x , лежащего в прямоугольнике, при данном условии выполняется

следующее неравенство:

$$\sqrt[p]{\sum_{i=1}^n \left| \frac{x_i - c_i}{r_i} \right|^p} \leq 1.$$

Таким образом, \mathbf{x} лежит и в окружности. \square

Система неравенств будет состоять из окружностей, что содержат определенный прямоугольник (будем называть допустимой областью), чтобы решение было не пусто. Первый алгоритм, *random_inequalities(...)*, должен сгенерировать допустимую область, полуоси и степени ограничений, выбрать центр каждого ограничения, используя Лемму 1. Примеры сгенерированных ограничений и результаты работы функции *random_inequalities(...)* представлены на рисунках 34, 35, 36, 37.

```
Center of parallelepiped: [-61.21879208 32.3677808]
Sides of parallelepiped: [12.09610106 46.32460101]
center: [-60.58226709 33.58468025] semi_axes: [58.22490011 35.18243543] p: 2.09
center: [-60.33523999 95.89370051] semi_axes: [ 18.06639751 249.2501587 ] p: 2.98
center: [-60.53347414 34.52134635] semi_axes: [82.9252703 56.50805707] p: 2.87
center: [-58.32075007 40.52266047] semi_axes: [ 10.31290749 335.23203488] p: 2.47
center: [-55.15215616 51.40787339] semi_axes: [ 71.29554603 121.50337406] p: 1.14
```

Рис. 34: Результат работы функции *random_inequalities(2, 5)*

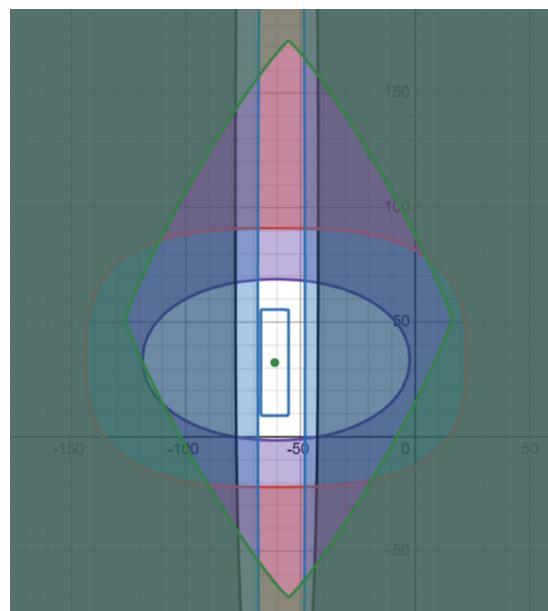


Рис. 35: Сгенерированные при помощи функции *random_inequalities(2, 5)* ограничения. Здесь синий прямоугольник - допустимая область, белая область - вся область, что удовлетворяет ограничениям

```
Center of parallelepiped: [23.74227352 -7.49606214]
Sides of parallelepiped: [75.47469075 98.4281591 ]
center: [26.99470033 2.6312323 ] semi_axes: [121.21189569 98.72152033] p: 2.87
center: [52.44500786 10.77655717] semi_axes: [221.45439703 96.88127728] p: 2.36
center: [1281.20855444 -2.08319198] semi_axes: [22345.098922 111.36673128] p: 1.94
center: [60.4004783 52.94874751] semi_axes: [164.34544425 223.18941465] p: 2.9
center: [29.48492854 0.36212003] semi_axes: [105.94161692 116.16418501] p: 1.95
```

Рис. 36: Результат работы функции *random_inequalities(2, 5)*

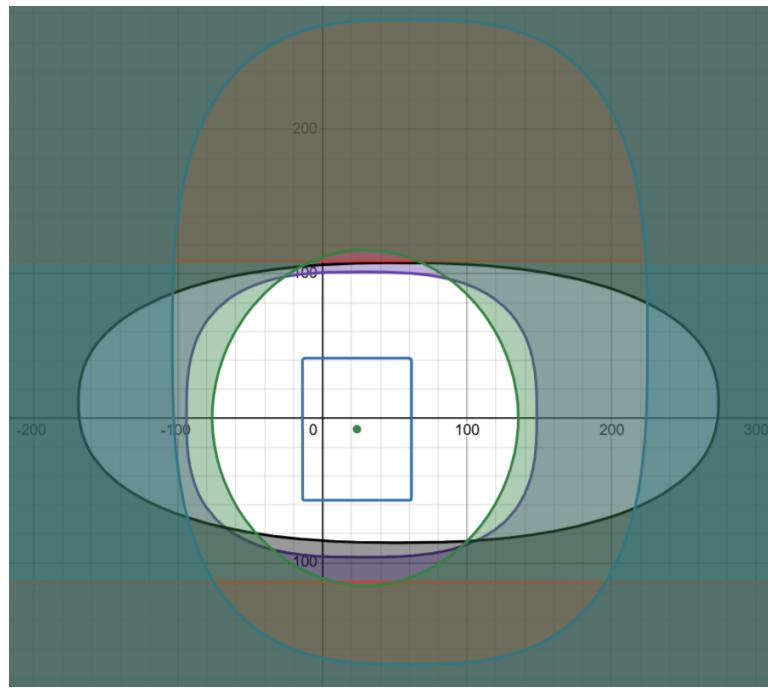


Рис. 37: Сгенерированные при помощи функции `random_inequalities(2, 5)` ограничения. Здесь синий прямоугольник - допустимая область, белая область - вся область, что удовлетворяет ограничениям

Данная конструкция не учитывает область значений функции (см. Рис. 38, для любой точки из области значений ограничения выполняются автоматически).

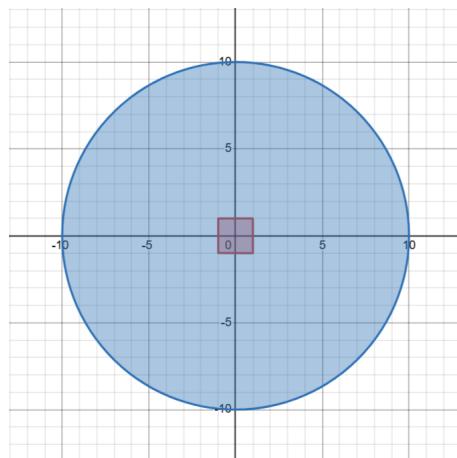


Рис. 38: $\max\{|x|, |y|\} \leq 1$ - область значений (красный квадрат), $x^2 + y^2 \leq 100$ - ограничение (синяя окружность)

Однако требуется обеспечить влияние ограничений на область значений функции. Рассмотрим два случая:

1. Множество, что задается ограничением, настолько малое, что не может содержать область значений (ограничения уже влияют на область значений). Пример такого случая показан на рисунке 39.

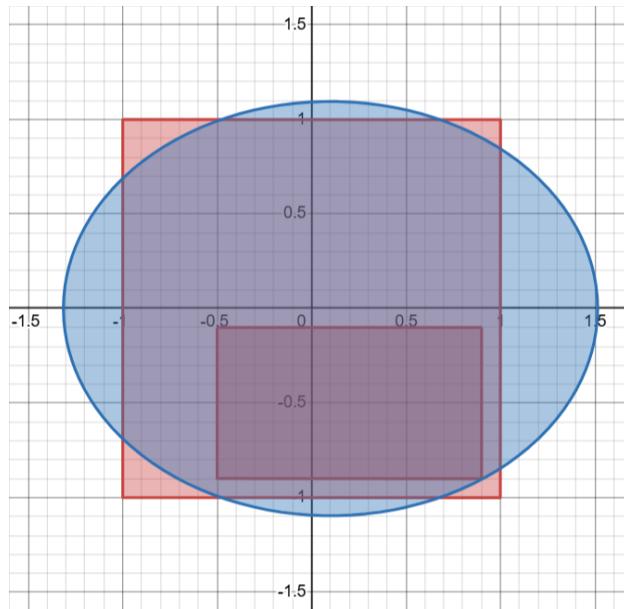


Рис. 39: $\max\{|x|, |y|\} \leq 1$ - прямоугольник (красный квадрат), $\max\{|\frac{x-0.2}{0.7}|, |\frac{y+0.5}{0.4}|\} \leq 1$ - допустимая область (красный прямоугольник), $\frac{(x-0.1)^2}{5} + \frac{y^2}{3} \leq 0.4$ - ограничение (синий эллипс)

2. Полуоси ограничений позволяют выбрать центр так, чтобы заключить область значений внутри.

2.1. Ограничение не влияет на область значений (см. Рис. 40).

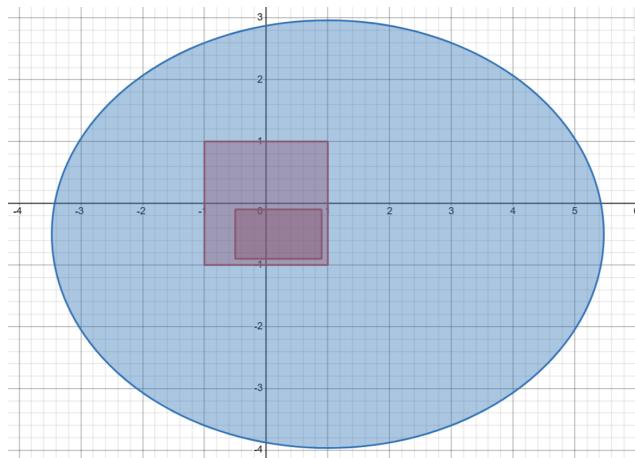


Рис. 40: $\max\{|x|, |y|\} \leq 1$ - прямоугольник (красный квадрат), $\max\{|\frac{x-0.2}{0.7}|, |\frac{y+0.5}{0.4}|\} \leq 1$ - допустимая область (красный прямоугольник), $\frac{(x-1)^2}{5} + \frac{(y+0.5)^2}{3} \leq 4$ - ограничение (синий эллипс)

2.2. Ограничение влияет на область значений (см. Рис. 41).

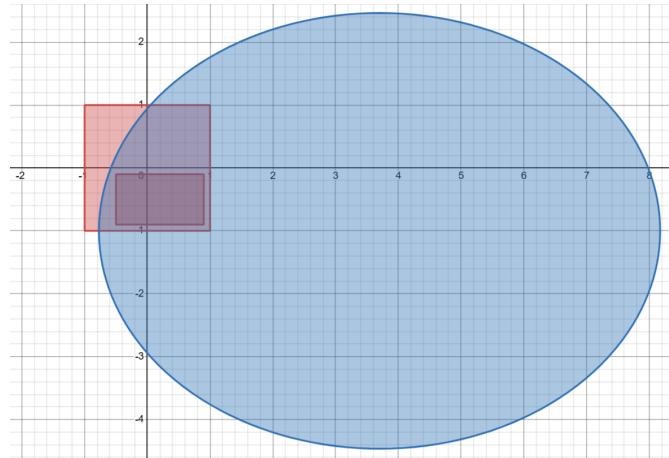


Рис. 41: $\max\{|x|, |y|\} \leq 1$ - прямоугольник (красный квадрат), $\max\{|\frac{x-0.2}{0.7}|, |\frac{y+0.5}{0.4}|\} \leq 1$ - допустимая область (красный прямоугольник), $\frac{(x-3.7)^2}{5} + \frac{(y+1)^2}{3} \leq 4$ - ограничение (синий эллипс)

Таким образом, мы хотим, чтобы область значений функции не всегда удовлетворяла ограничениям, тогда как допустимая область должна полностью удовлетворять им. Согласно этому принципу действует вторая функция генерации ограничений, `random_inequalities_with_bound(...)`, уже с учетом области значений.

Второй алгоритм, `random_inequalities_with_bound(...)`, должен сгенерировать допустимую область внутри области значений, полуоси и степени ограничений. Если полуоси достаточно малы, ограничения уже влияют на область значений. Необходимо, используя Лемму 1, выбрать центр ограничения так, чтобы в нем содержалась допустимая область. Иначе используем отрицание Леммы 1 (чтобы область значений не лежала в ограничении) и саму Лемму 1 (чтобы допустимая область лежала в ограничении). Таким образом, центр ограничения должен удовлетворять уже двум неравенствам.

Разберем второй случай подробнее (см. Рис. 42).

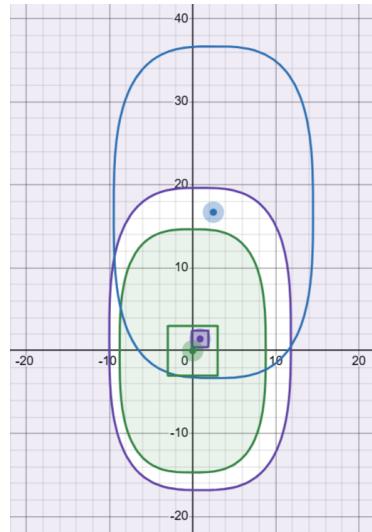


Рис. 42: Синий эллипс - ограничение, зеленый квадрат - область значений, фиолетовый квадрат - допустимая область

Если центр:

- снаружи фиолетового эллипса - ограничение не содержит ни допустимую область, ни область значений (см. Рис. 43).

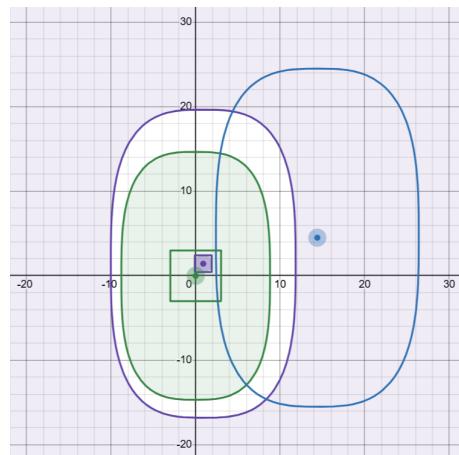


Рис. 43: Синий эллипс - ограничение (центр снаружи фиолетового эллипса), зеленый квадрат - область значений, фиолетовый квадрат - допустимая область

- внутри фиолетового эллипса - ограничение содержит допустимую область, но не обязательно содержит область значений (см. Рис. 44).

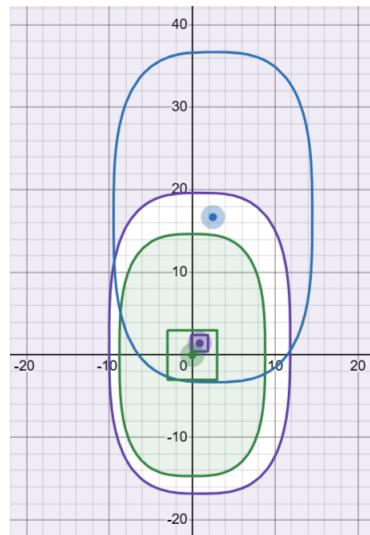


Рис. 44: Синий эллипс - ограничение (центр внутри фиолетового эллипса), зеленый квадрат - область значений, фиолетовый квадрат - допустимая область

- внутри зеленого эллипса - ограничение содержит область значений и допустимую область (см. Рис. 45).

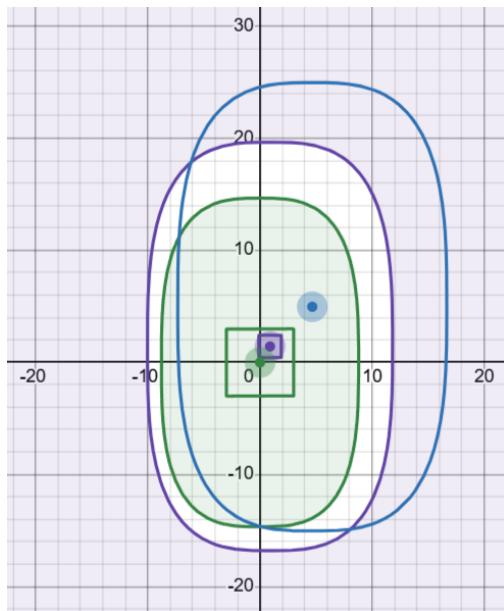


Рис. 45: Синий эллипс - ограничение (центр внутри зеленого эллипса), зеленый квадрат - область значений, фиолетовый квадрат - допустимая область

То есть мы хотим, чтобы центр ограничения лежал в фиолетовом эллипсе, но не в зеленом (в белой области на графике).

Стороны допустимой области a_i генерируются случайно на интервале $[0.2 \cdot A_i; 0.7 \cdot A_i]$, здесь A_i - стороны области значений. Центр допустимой области берется произвольным образом так, что допустимая область полностью лежит в области значений.

Полуоси генерируются следующим образом:

Генерируем $compr$ равновероятно из промежутка $[0.1; 0.9]$ таким образом, что $\sqrt[p]{\sum_{i=1}^n \left| \frac{a_i}{2r_i} \right|^p} =$

compr. Далее из этого соотношения находим r_i . Для этого генерируем случайные положительные величины так (для этого генерируем $n - 1$ точку на интервале $[0; \text{compr}^p]$ и проверяем, что они различны. Расстояние между точками и задает эти величины), что их сумма равна compr^p :

$$\sum_{i=1}^n v_i = \text{compr}^p, \quad v_i \geq 0.$$

r_i находим из соотношения $\left| \frac{a_i}{2r_i} \right|^p = v_i$:

$$\frac{a_i}{2\sqrt[p]{v_i}} = r_i.$$

Если $\sqrt[p]{\sum_{i=1}^n \left| \frac{A_i}{2r_i} \right|^p} \geq 1$, то размер ограничения недостаточен для содержания области значений в ограничении. Следовательно, ограничения влияют на область значений.

Пользуясь Леммой 1, выбираем центр ромба так, чтобы в нем содержалась допустимая область:

$$\sqrt[p]{\sum_{i=1}^n \left| \frac{p_i - c_i}{r_i} \right|^p} \leq 1 - \sqrt[p]{\sum_{i=1}^n \left| \frac{a_i}{2r_i} \right|^p} = S.$$

Для случайного выбора центра поступим следующим образом: сгенерируем случайное число $k \in [0, S]$ такое, что

$$\sqrt[p]{\sum_{i=1}^n \left| \frac{p_i - c_i}{r_i} \right|^p} = k \leq S.$$

Таким образом, для c_i получим следующее:

$$\sum_{i=1}^n v_i = k^p, \quad v_i \geq 0.$$

$$\left| \frac{p_i - c_i}{r_i} \right|^p = v_i.$$

$$c_i = \sqrt[p]{v_i} r_i \pm p_i.$$

Если размеры окружности позволяют заключить область значений внутрь, то нам необходимо выбрать центр ограничения так, чтобы он удовлетворял системе неравенств. То есть окружность не будет полностью содержать область значений, но допустимая область будет содержаться в ограничении полностью:

$$\begin{cases} \sqrt[p]{\sum_{i=1}^n \left| \frac{p_i - c_i}{r_i} \right|^p} \leq 1 - \sqrt[p]{\sum_{i=1}^n \left| \frac{a_i}{2r_i} \right|^p} \\ \sqrt[p]{\sum_{i=1}^n \left| \frac{P_i - c_i}{r_i} \right|^p} > 1 - \sqrt[p]{\sum_{i=1}^n \left| \frac{A_i}{2r_i} \right|^p} \end{cases},$$

где P_i - центр области значений.

То есть мы из первой окружности вычитаем вторую. Эта система разрешима, а множество решений связно, так как полуоси равны, а радиус первой окружности больше, чем радиус второй. Исходя из выбора a_i , получим:

$$a_i \leq 0.7 \cdot A_i \leq A_i.$$

Возвведение в степень $p \geq 1$ - неубывающая функция:

$$\left| \frac{a_i}{2r_i} \right|^p \leq \left| \frac{0.7 \cdot A_i}{2r_i} \right|^p \leq \left| \frac{A_i}{2r_i} \right|^p.$$

Взятие корня $p \geq 1$ - неубывающая функция:

$$\sqrt[p]{\sum_{i=1}^n \left| \frac{a_i}{2r_i} \right|^p} \leq \sqrt[p]{\sum_{i=1}^n \left| \frac{A_i}{2r_i} \right|^p}.$$

Таким образом, получим:

$$1 - \sqrt[p]{\sum_{i=1}^n \left| \frac{A_i}{2r_i} \right|^p} \leq 1 - \sqrt[p]{\sum_{i=1}^n \left| \frac{a_i}{2r_i} \right|^p}.$$

Центр должен выбираться случайным образом из решений системы неравенств. Для решения этой задачи использовалась библиотека PyMC языка программирования python [30] и алгоритм DEMetropolisZ [31]. Классический MCMC [32], методы которого реализованы в PyMC, начинается с произвольной исходной точки в пространстве, и на каждом шаге предлагаёт новую точку случайным образом; она принимается с определенной вероятностью, иначе алгоритм остаётся на прежней точке. Повторяя такие шаги много раз, получают цепь, аппроксимирующую заданное распределение. В DEMetropolisZ же на каждом шаге новое предложение строится так: из истории цепи выбирают две предыдущие точки, вычисляют их разностный вектор, масштабируют его постоянным коэффициентом и добавляют к результату небольшой шум из нормального распределения, а затем принимают с некоторой вероятностью.

Как выбираются начальные точки и СКО для задания нормального распределения из которого генерируется шум:

- Для каждой i -ой переменной пространства вычисляются точки f_i и g_i . Эти точки расположены на i -ой координатной оси, проходящей через центр второго неравенства и лежащей на границе эллипсоидов. При этом выбирается та сторона, где расстояние между точками больше. Если расстояния примерно равны, выбирается случайная сторона.
- СКО i -ой переменной задаётся как половина расстояния между соответствующими точками на границах вложенного и внешнего p -эллипсоидов вдоль i -ой оси. Для генерации начальных точек случайным образом выбираются две координатные оси, вдоль которых строятся отрезки между соответствующими точками. Центры этих отрезков (их середины) используются в качестве начальных точек, лежащих между двумя телами. Пример выборки начальных точек показан на рисунке 46.

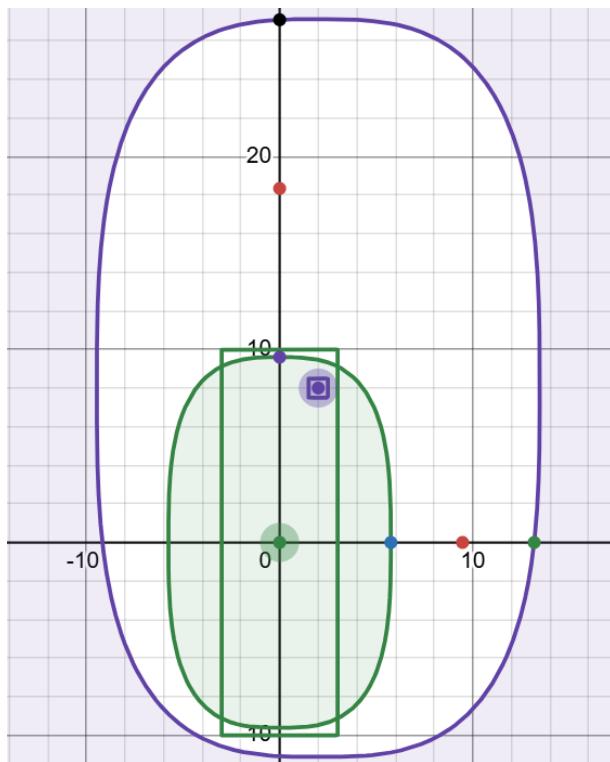


Рис. 46: Наглядный пример выборки начальных точек (они обозначены красным цветом)

Предложенный способ построения начальных точек и оценки разброса прошёл проверку посредством прямых вычислений и показал корректную работу на различных конфигурациях p -эллипсоидов (см. рисунок 47, рисунок 48).

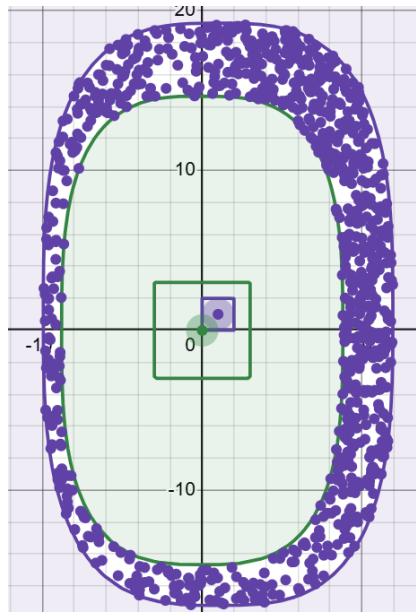


Рис. 47: Проверка-1 способа построения начальных точек

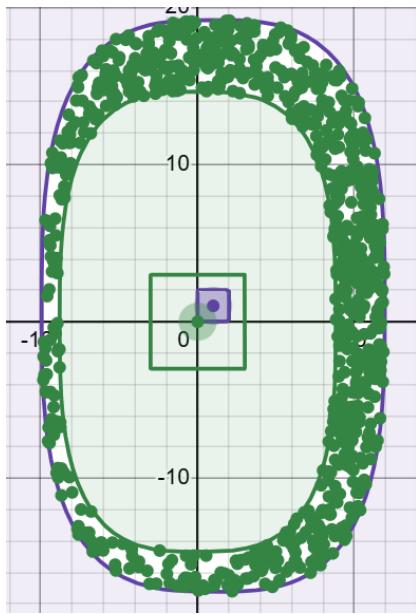


Рис. 48: Проверка-2 способа построения начальных точек

Таким образом мы, получили множество точек, что удовлетворяют системе неравенств. Центр ограничения выбираем произвольным образом из этого множества.

Функция также может возвращать множество точек, что удовлетворяет системе сгенерированных нелинейных ограничений. Начальные точки инициализируются равномерно внутри допустимой области (при помощи последовательностей Соболя из qmc scipy [33]), СКО i -ой переменной - половина длины допустимой области по i -ой координате. Пример работы функции и результаты генерации таких ограничений представлены на рисунках 49, 50, 51.

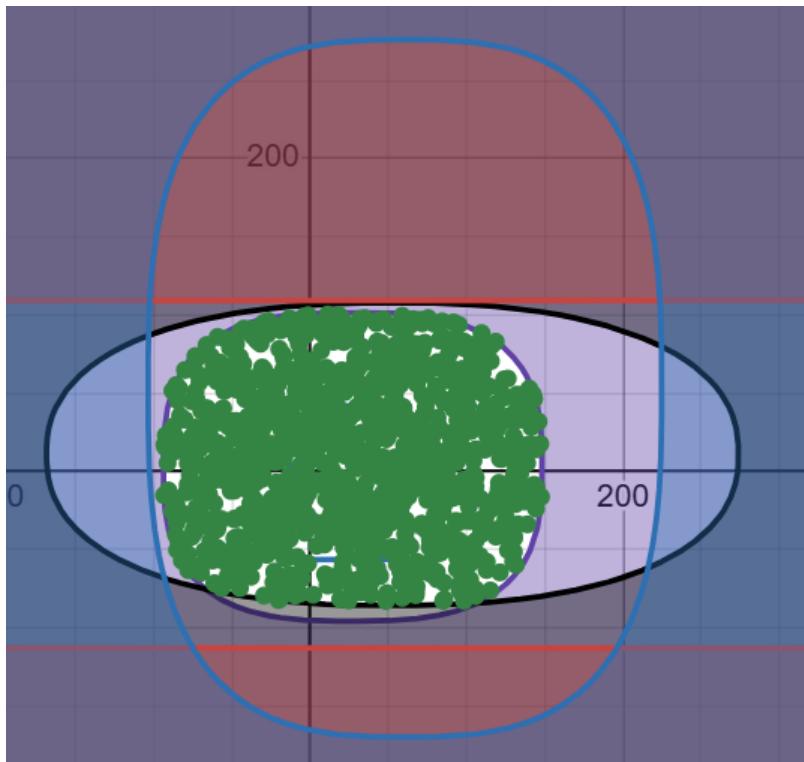


Рис. 49: Все точки удовлетворяют полученным ограничениям

```

center_r: [ 0.77972187 -1.49088416] semi_axes_r: [8.41311397 7.83185028] p: 1.42
center_r: [ 1.35143378 -0.1905663 ] semi_axes_r: [1.05613449 1.70469836] p: 2.57
center_r: [ 1.62372997 -0.08379196] semi_axes_r: [2.815291  1.28310426] p: 2.44
center_r: [ 1.58763725 -0.1496578 ] semi_axes_r: [2.69811368 1.47069211] p: 1.72
center_r: [1.17769565 0.12232181] semi_axes_r: [3.6928165  1.91332825] p: 1.69
Center of parallelepiped: [ 1.04992512 -0.20530153]
Sides of parallelepiped: [1.22000927 1.35774959]
```

Рис. 50: Пример работы функции `random_inequalities_with_bound(2, 5, [0, 0], [4, 4], seed = 456447)`

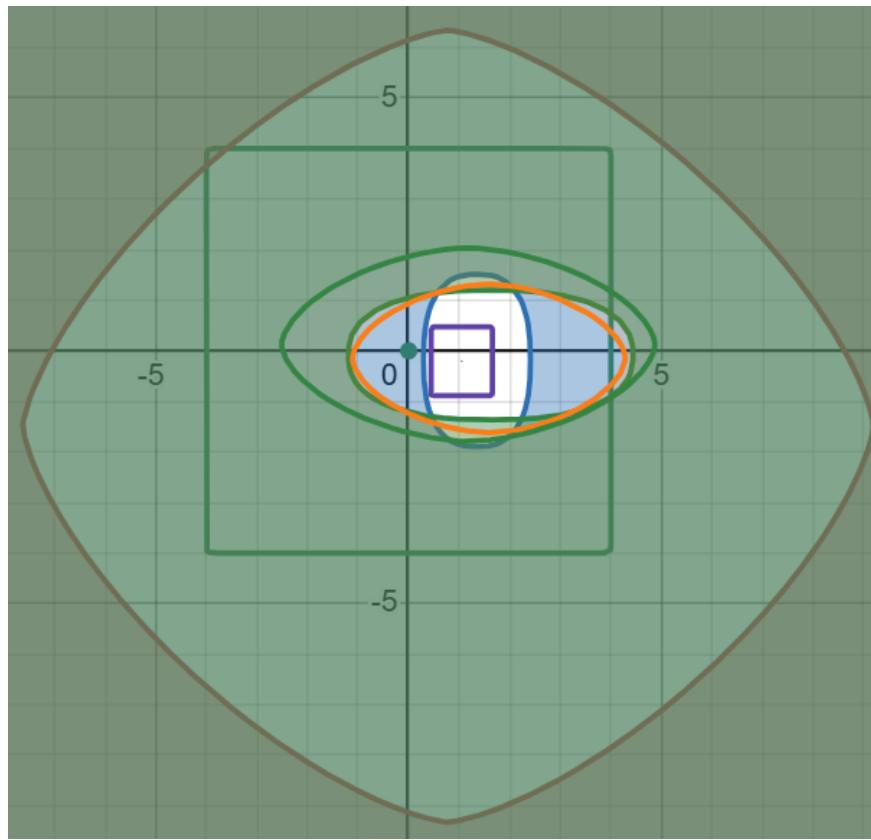


Рис. 51: Наглядный результат

4.3.3 Сценарий использования (*bench.py*)

Функция *Benchmarking(...)* реализует единый интерфейс запуска тестов с параметрами:

1. название эксперимента (*name*),
2. количество оптимизационных задач (*quantity*) при генерации, или их список (*model_list*),
3. список целевых функций (*test_func*),
4. список ограничений для целевых функций (*constraint*),
5. список алгоритмов оптимизации (*algorithms*),
6. стартовая точка (*x0*),
7. параметры для генерации ограничений (*params_polynomial*, *params_spline*, *params_for_inequality*).

Тесты выполняются циклически по заданным функциям и начальному положению, что позволяет варьировать условия и получать устойчивую статистику. На листинге 3 показаны различные сценарии использования функции *Benchmarking(...)* для различных параметров.

```

1 # 1
2 benchmark = Benchmarking(
3     name="manual_run",
4     quantity=2,
5     test_func=[MyTestFunction(), MyTestFunction()],
6     constraint=[MyConstraint(), MyConstraint()],
7     algorithms=[MyAlgorithm(), MyAlgorithm()],
8     x0=[[0.1, 0.2], [0.3, 0.4]])
9
10 # 2
11 test_params = Par_TestObj(...)
12 cons_params = Par_Pol(...)
13 algo_params = Par_Algo(...)
14 benchmark = Benchmarking(
15     name="auto_all",
16     quantity=5,
17     test_func=test_params,
18     params_polynomial=cons_params,
19     algorithms=algo_params,
20     diff_method="jax")
21
22 # 3
23 benchmark = Benchmarking(
24     name='test_problems_only_5',
25     quantity=5,
26     test_problems=True,
27     algorithms=[Dummy(DummyParams())])
28
29 # 4
30 benchmark = Benchmarking(
31     name='test_problems_manual',
32     quantity=1,
33     test_problems=[SumOfDifferentPowersTask(n=2)],
34     algorithms=[Dummy(DummyParams())])
35
36 # 5
37 benchmark = Benchmarking(
38     name='test_problems_all',
39     quantity='all',
40     test_problems=True,
41     algorithms=[Dummy(DummyParams())])
42 }

```

Листинг 3: Пример сценария использования

Листинг 3 демонстрирует различные варианты использования основного интерфейса *Benchmarking* для запуска тестов. Класс поддерживает как ручное, так и автоматическое задание параметров задач и ограничений. В представленных примерах показаны пять типовых

сценариев:

1. Полностью ручной режим;
2. Автоматическая генерация тестовых функций и ограничений;
3. Прогонка 5 случайных задач;
4. Выбор определенной задачи;
5. Прогонка всех тестовых задач из базы.

4.3.4 Анализ поведения алгоритмов

Собранные логи и метрики позволяют проводить анализ по сходимости (число итераций, штрафы, значения цели), группировку по типу функций (например, плохо масштабируемые), сопоставление эффективности разных методов при разных ограничениях.

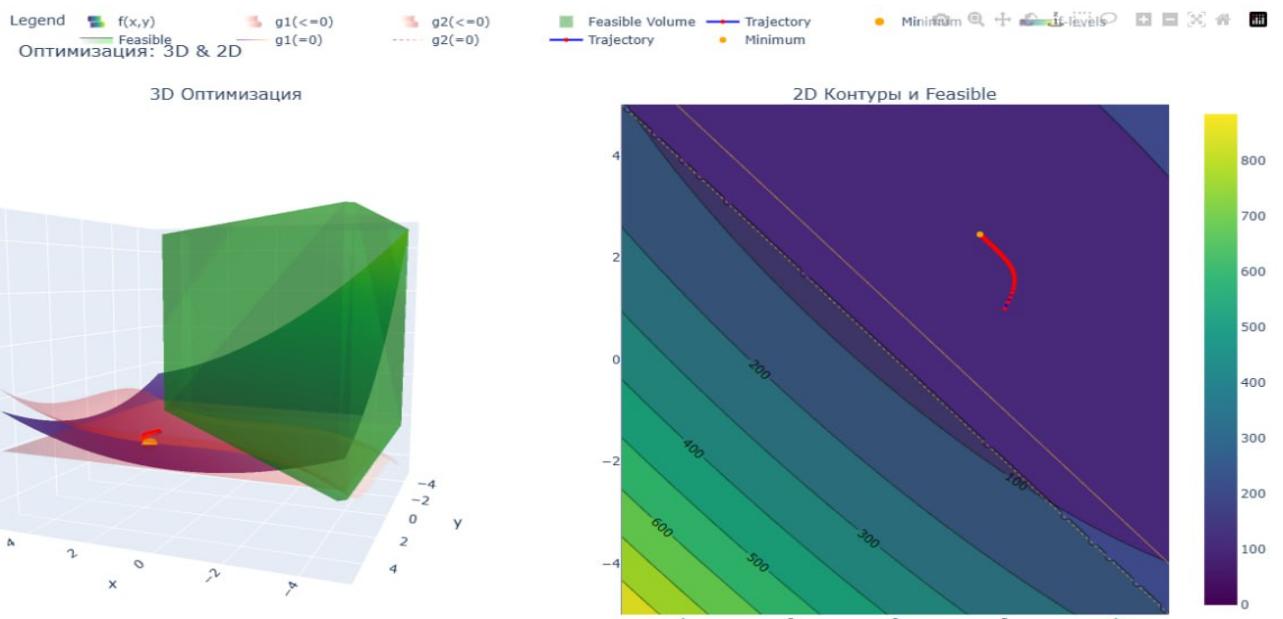


Рис. 52: Пример 3D и 2D визуализации с **Plotly**

На рисунке 52 представлен пример визуализации, который разделен на две части:

- С левой стороны трехмерный график целевой функции $f(\mathbf{x}, \mathbf{y})$ с градиентной палитрой *Viridis* (полупрозрачная). Это позволяет одновременно видеть как форму поверхности, так и внутренние элементы, такие как траектория и ограничения.
- Справа - контурный 2D-график уровней $f(\mathbf{x}, \mathbf{y})$, на который наложена область допустимых решений (*Feasible region*). Контур помогает сопоставить геометрию ограничений с топографией уровней целевой функции.

Основные функциональные элементы:

- 1) **Объёмная область допустимых решений (Feasible Region).** Область, удовлетворяющая всем ограничениям, формируется из точек (\mathbf{x}, \mathbf{y}) , для которых выполняются условия допустимости. Эти точки собираются как на нижнем (ниже значения минимума \mathbf{z}) уровне, так и на верхнем (на уровне максимального значения \mathbf{z}). По собранным точкам строится выпуклая оболочка (*ConvexHull*), которая визуализируется в виде полупрозрачного зелёного объёма (тип графика — *Mesh3d*).
- 2) **Поверхности ограничений.** Каждое ограничение вида $g_i(\mathbf{x}, \mathbf{y}) = 0$ визуализируется в виде красной полупрозрачной поверхности, проходящей через график функции $f(\mathbf{x}, \mathbf{y})$. Эти поверхности отделяют допустимую и недопустимую области в пространстве.
- 3) **Границы ограничений на 2D-контуре.** На правом контурном графике каждая функция-ограничение отображается как линия соответствующего цвета и стиля штриха (сплошная, точечная, пунктирная и комбинированная). Такие линии демонстрируют, где именно активируются (срабатывают) соответствующие ограничения $g_i(\mathbf{x}, \mathbf{y})$.
- 4) **Траектория оптимизации.** Путь алгоритма оптимизации по поверхности целевой функции изображается синей ломаной линией с красными маркерами на промежуточных точках. Конечная точка минимума отмечена оранжевым маркером, что позволяет оценить результат и характер сходимости метода.
- 5) **Интерактивные обозначения.** Над графиками расположена легенда, элементы которой можно включать и выключать кликом. Это предоставляет пользователю возможность управлять отображаемыми компонентами визуализации и фокусироваться на интересующих аспектах (например, скрыть все ограничения и оставить только траекторию и поверхность цели).

5 Численные результаты

5.1 Цель и методология тестирования

Целью исследования является тестирование, сравнение, а также изучение влияния гиперпараметров реализованных алгоритмов (см. главу 3) на различных оптимизационных задачах (см. 4.3.1 и 4.3.2) для установления их различий, преимуществ одних алгоритмов над другими и выяснения влияния различных оптимизационных задач на результаты работы алгоритмов.

В качестве метрик оценки производительности и качества работы алгоритмов используются

1. Точность подхода алгоритма к известному оптимуму (соответственно относительная и абсолютная ошибка работы алгоритма) - известна на каждой итерации.
2. Время работы алгоритма над задачами.
3. Число итераций алгоритма для решения задачи.
4. Конечное значение целевой функции, при завершении алгоритма.
5. Уровень нарушения ограничений - мера отклонения найденного решения от заданных ограничений.

Далее под "статистиками" будет подразумеваться комбинация метрика плюс статистическая функция (среднее значение, медиана, минимальное значение, максимальное значение).

5.1.1 Описание платформы тестирования

Для реализации поставленных целей, используется концепция написания скриптов, состоящая из следующих шагов

1. агрегирования данных. Происходит на уровне формирования дата-фреймов (англ. dataframe) при парсинге логов;
2. выделения всех возможных статистик из агрегированных данных;
3. визуализация собранных статистик в виде таблиц и/или графиков.

Таким образом, общий конвеер (англ. pipeline) работы скрипта тестинга может быть представлен в виде рисунка (см. Рис. 53 ниже).



Рис. 53: Конвеер работы скрипта по сбору и визуализации статистик

- **Сбор статистики**

Поддерживается агрегирование данных двух различных типов

1. сбор итеративных данных при работе алгоритмов;
2. сбор результатов оптимизации.

Такое разделение помогает отдельно работать с данными, полученными во время выполнения алгоритмов, и с данными, полученными в результате завершения работы последних.

На данном этапе поддерживаются следующие статистики при агрегировании итеративных данных

1. конечное значение целевой функции,
2. уровень нарушения ограничений,
3. количество запусков алгоритма,
4. среднее время работы алгоритма,
5. минимальное и максимальное время работы алгоритма,
6. среднее число итераций для решения задачи,
7. минимальное и максимальное число итераций при решении задач.

При агрегировании данных результатов оптимизации выделяются следующие статистики

1. общее количество итераций алгоритма для решения оптимизационной задачи,
2. конечное значение целевой функции,
3. максимальный уровень нарушения ограничений, т.е. максимальное отклонение найденного решения от заданных ограничений.

Таким образом, для оценки скорости работы алгоритмов, скорости их сходимости используются статистики, полученные при первом типе агрегирования данных; для проверки точности и допустимости решения, в смысле меры отклонения от ограничений, - второй тип агрегирования.

На основе такого агрегированного данных возможно извлекать интересующие пользователя статистики, к примеру, такие функции скрипта как `best_algorithms_by_time(...)` и `worst_algorithms_by_time(...)` извлекают N лучших и N худших алгоритмов по среднему времени выполнения соответственно.

Исходя из абзаца выше, важно отметить, что существует возможность добавления своего типа агрегирования данных, статистик - реализуется это на фоне добавление новых sql-запросов в файле `log_parser.py`, - а также добавления новых метрик - более подробно в файле `log_parser.md` фреймворка.

• Визуализация

Для визуализации результатов сбора статистик применяются функции `pivot_mean_times(...)` и `pivot_mean_iters(...)` для вывода информации в виде таблиц. Первая функция выводит среднее время решения оптимизационных задач разными алгоритмами, используется для сравнения скорости решения различных оптимизационных задач различными алгоритмами. Вторая же функция выводит среднее количество итераций, потребовавшихся для решения данной оптимизационной задачи.

Так же в скрипте реализована возможность построения графиков:

- значения целевой функции от номера итерации для данной задачи - реализует это функция `plot_all_traces(...)`, принимающая данные, агрегированный первым типом (см. 5.1.1). Пример графика показан на рис. 54;

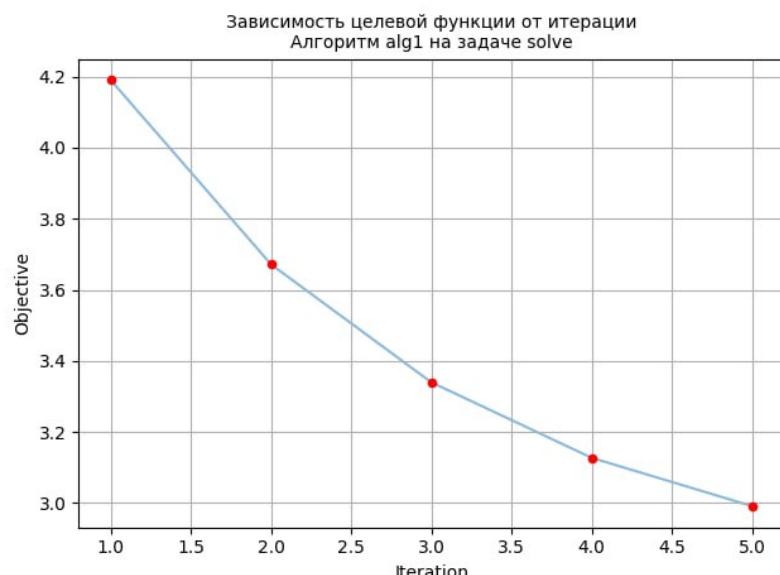


Рис. 54: График зависимости целевой функции от номера итерации

Таким образом, в качестве метода исследования используется автоматизированное тестирование разработанных алгоритмов над различными нелинейными оптимизационными задачами на основе разработанной платформы тестирования.

5.2 Тестирование алгоритмов

5.2.1 Сравнение алгоритмов по качеству решения и анализу сходимости

5.2.2 Влияние гиперпараметров

Список литературы

1. Wächter, A. On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming / A. Wächter, L. T. Biegler // Mathematical Programming. – 2005. – Vol. 106, No. 1. – pp. 3–19. – 17 p. – DOI: 10.1007/s10107-004-0559-y.
2. Lalee, M. On the Implementation of an Algorithm for Large-Scale Equality-Constrained Optimization / M. Lalee, J. Nocedal, T. Plantenga // SIAM Journal on Optimization. – 1998. – Vol. 8, No. 3. – pp. 682–697. – 26 p. – DOI: 10.1137/S1052623493262993.
3. Byrd, R. H. An Interior Point Algorithm for Large-Scale Nonlinear Programming / R. H. Byrd, M. E. Hribar, J. Nocedal // SIAM Journal on Optimization. – 1999. – Vol. 9, No. 4. – pp. 2–19. – 18 p. – DOI: 10.1137/S1052623497325107.
4. Burke, J. V. Gradient Sampling Methods for Nonsmooth Optimization / J. V. Burke, F. E. Curtis, A. S. Lewis, M. L. Overton, L. E. A. Simões // arXiv preprint. – 2018. – arXiv:1804.11003v1. – pp. 1–15. – 15 p.
5. Fletcher, R. Nonlinear Programming Without a Penalty Function / R. Fletcher, S. Leyffer // Mathematical Programming. – 2002. – Vol. 91, No. 2. – pp. 635–659. – 25 p.
6. Conn, A. R. A Primal-Dual Trust-Region Algorithm for Non-Convex Nonlinear Programming / A. R. Conn, N. I. M. Gould, D. Orban, Ph. L. Toint // Mathematical Programming. – 2000. – Vol. 87, No. 2. – pp. 215–249. – 35 p.
7. Fiacco, A. V. Nonlinear Programming: Sequential Unconstrained Minimization Techniques / A. V. Fiacco, G. P. McCormick. – New York, USA: John Wiley, 1968. – Reprinted by SIAM Publications, 1990. – pp. 103–200. – 98 p.
8. Byrd, R. H. Robust trust region methods for constrained optimization / R. H. Byrd. // Proceedings of the Third SIAM Conference on Optimization. – 1987. – pp. 1–23. – 23 p.
9. Omojokun, E. O. Trust Region Algorithms for Optimization with Nonlinear Equality and Inequality Constraints: Ph.D. thesis. // Dept. of Computer Science, University of Colorado. – 1989. – pp. 57–87. – 31 p.
10. Liu, D. C. On the limited memory BFGS method for large scale optimization / D. C. Liu, J. Nocedal // Math. Programming. – 1989. – Volume 45, issue 1–3. – pp. 503–528. – p. 26.
11. Steihaug, T. The conjugate gradient method and trust regions in large scale optimization / T. Steihaug. // SIAM Journal on Numerical Analysis. – 1983. – Volume 20, №3. – pp. 626–637. – p. 12.
12. Curtis, F. E. Sequential Quadratic Programming Algorithm for Nonconvex, Nonsmooth Constrained Optimization / F. E. Curtis, M. L. Overton // SIAM Journal on Optimization. – 2012. – Volume 22, №2. – pp. 474–500. – p. 27.

13. Bodar, A. The inexact power augmented Lagrangian method for constrained nonconvex optimization / A. Bodar, K. Oikonomidis, E. Laude, P. Patrinos. // Mathematical Programming. – 2024. – pp. 1-34. – p. 34. – arXiv:2410.20153v1.
14. Liu, L. On the Variance of the Adaptive Learning Rate and Beyond / L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, J. Han // Computing Research Repository. - 2021. - pp. 1-14. - p.14 — arXiv:1908.03265.
15. Kingma, D.P. A Method for Stochastic Optimization / D.P. Kingma, J. Ba // Computing Research Repository. — 2017. - pp. 1-15. - p.15 — arXiv:1412.6980.
16. Dvurechensky, P. Barrier Algorithms for Constrained Non-Convex Optimization / P. Dvurechensky, M. Staudigl // Mathematical Programming. – 2024. — pp. 1-23. - p.23 - arXiv:2404.18724.
17. Diouane, Y. A nonsmooth exact penalty method for equality-constrained optimization: complexity and implementation / Y. Diouane, M. Gollier, D. Orban // Mathematical Programming. – 2024. - pp. 1-21 - p. 21 — DOI: 10.13140/RG.2.2.16095.47527.
18. Boyd, S. Convex Optimization / S. Boyd, L. Vandenberghe. // Stanford: Stanford University - 2004. - pp. 3-714 - p. 712
19. Hestenes, M. R. Multiplier and gradient methods / M. R. Hestenes // Journal of Optimization Theory and Applications. — 1969. — Vol. 4, No. 5. — pp. 303–320 - p. 18
20. Beck, A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems / A. Seck, M. Teboulle // SIAM Journal on Imaging Sciences. - 2009. - 2(1) - pp. 183–202 - p. 20
21. Hribar, M.E. Large-scale constrained optimization: Ph.D. Dissertation. // Dept. EECS, Northwestern University. - 1996. - pp. 1-28 - p. 28
22. Nesterov, Yu. E. Gradient methods for minimizing composite objective function / Yu. E. Nesterov // CORE. – 2007. – pp. 1-30. -30 p.
23. Nesterov, Yu. E. Introductory lectures on convex optimization: A basic course / Yu. E. Nesterov // Springer. – 2004. – Volume 87. – pp. 1–253. – p. 253. - DOI: 10.1007/978-1-4419-8853-9
24. Nesterov, Yu. E. Accelerating the cubic regularization of Newton’s method on convex problems / Yu. E. Nesterov // Springer. – 2007. – pp. 160–181. – p. 22. - DOI: 10.1007/s10107-006-0089-x
25. O’Donoghue B. Adaptive restart for accelerated gradient schemes / B. O’Donoghue, E. Candès // Foundations of Computational Mathematics. – 2013. – pp. 1–18. – p. 18.
26. Parikh N. Proximal Algorithms / N. Parikh, S. Boyd // Stanford University. – 2014. – pp. 128–224. – p. 97.

27. Rodomanov A. Analysis of the Fast Gradient Nesterov Method for Machine Learning Problems with L1 Regularization / A. Rodomanov // Lomonosov Moscow State University. - 2014. - pp. 1-19. p. 19.
28. Tibshirani R. The Lasso Problem and Uniqueness / R. Tibshirani // Carnegie Mellon University. – 2009. – pp. 1-27. – p. 27.
29. Lin Q. An adaptive accelerated proximal gradient method and its homotopy continuation for sparse optimization / Q. Lin, L. Xiao // Carnegie Mellon University. – 2014. – pp. 73-81. – p. 7.
30. Salvatier J., Wiecki T.V., Fonnesbeck C. Probabilistic programming in Python using PyMC3 / J. Salvatier, T.V. Wiecki, C. Fonnesbeck // PeerJ Computer Science. — 2016. — Vol. 2. — P. e55.
31. ter Braak C.F.J. Differential Evolution Markov Chain with snooker updater and fewer chains / C.F.J. ter Braak // Statistics and Computing. — 2008. — Vol. 18, No. 4. — P. 435–446.
32. Robert C.P., Casella G. Monte Carlo Statistical Methods / C.P. Robert, G. Casella. — 2nd ed. — New York: Springer, 2004. — 645 p.
33. Sobol' I.M. On the distribution of points in a cube and the approximate evaluation of integrals / I.M. Sobol' // USSR Computational Mathematics and Mathematical Physics. — 1967. — Vol. 7, No. 4. — P. 86—112.
34. Benson H.Y., Shanno D.F., Vanderbei R.J. Interior-point methods for nonconvex nonlinear programming: Filter methods and merit functions / H.Y. Benson, D.F. Shanno, R.J. Vanderbei // Operations Research and Financial Engineering, Princeton University. – 2000. – 36 p.
35. Wächter A., Biegler L.T. Line search filter methods for nonlinear programming: Motivation and global convergence / A. Wächter, L.T. Biegler // Department of Chemical Engineering, Carnegie Mellon University. – 2003. – 26 p.
36. Yamashita H. A globally convergent primal-dual interior-point method for constrained optimization / H. Yamashita // Optimization Methods and Software. – 1998. – Vol. 10. – P. 443–469. - 26 p.