МИНИСТЕРСТВО НАУКИ II ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»

Высшая школа электроники и компьютерных наук
Кафедра вычислительной математики и высокопроизводительных вычислений

РАБОТА ПРОВЕРЕНА
Рецензент,
_____/ФИО
« » _____2020 г.

ДОПУСТИТЬ К ЗАЩИТЕ
Заведующий кафедрой, к.ф.-м.н., доцент
_____/Н.М. Япарова
« ___» _____ 2020 г.

«Масштабируемый алгоритм решения задачи линейного программирования с изменяющимися исходными данными»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ
ЮУрГУ-090401.2020.677 ИЗ ВКР

Руководитель работы, к. ф.-м. н., доцент
_____/И.М. Соколинская
«____»_____2020 г.

Автор работы
Студент группы КЭ-230

_____/Д.Г. Васёв
«____»_____ 2020 г.

Нормоконтролер, к. ф.-м. н., доцент
_____/С.У. Турлакова
«____»_____2020 г.

Челябинск 2020

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Целью выпускной квалификационной работы является исследование масштабируемого алгоритма решения задачи линейного программирования с изменяющимися входными данными Non-Stationary Linear Programming (NSLP).

Для достижения поставленной задачи необходимо, прежде всего, произвести обзор литературных источников, связанных с теоретической базой алгоритма: работ по теории нахождения решений систем линейных неравенств, фейеровским методам в оптимизации, основам линейного программирования.

Также необходимо проанализировать состояние и статус проблемы путём обзора научных публикаций в области разработки методов решения нестационарных задач линейного программирования. Кроме того, необходимо выявить основные особенности таких задач и сложности, возникающие при попытке применения классических методов при их решении. Помимо прочего, в перечень подлежащих разработке вопросов входит проведение анализа основной идеи, лежащей в основе алгоритма NSLP, а также описание его основных фаз. Фаза Quest (поиск) предназначена для поиска решения системы неравенств, задающей ограничения нестационарной задачи, а фаза Targeting (позиционирование) - для формирования следящей области, позволяющей с заданной точностью определить решение.

После этого необходимо написать программу, реализующую алгоритм NSLP, протестировать её на примере модельной задачи различных размерностей (смоделировав нестационарность исходных данных) и провести ряд вычислительных экспериментов по исследованию сходимости алгоритма в зависимости от параметров алгоритма и интенсивности изменения исходных данных.

Таким образом, предметом исследования является предлагающийся в качестве алгоритма решения нестационарных задач алгоритм NSLP. В силу того, что классические методы решения задач линейного программирования при

нестационарном характере входных данных не дают оптимального решения, имеет место необходимость разработки и реализации альтернативных алгоритмов. Следовательно, исследование рассматриваемого в выпускной квалификационной работе алгоритма NSLP является актуальным направлением научной деятельности.

В качестве метода исследования используется метод решения нестационарных задач линейного программирования, основанный на применении фейеровских отображений.

1 ОБЗОР ЛИТЕРАТУРНЫХ ИСТОЧНИКОВ

Основой решения задач линейного программирования является теория нахождения решения систем линейных неравенств. Базовый материал по данной теме приведён в [14, 26]. Кроме того, важную роль в нашей работе играет общая теория линейного программирования. На применении методов линейного программирования основано решение многочисленных практических задач; теоретические основы данных методов и примеры подобных задач рассмотрены в [13]. Также большое количество алгоритмов и методов решения оптимизационных задач и задач поиска экстремума приведено в [19].

Являющийся предметом нашего исследования алгоритм решения нестационарных задач линейного программирования рассмотрен в [23, 24, 30-33]. В качестве метода исследования используется метод решения нестационарных задач линейного программирования, основанный на применении фейеровских отображений. В этой связи отдельного упоминания стоит [10], где автор уделяет таким методам значительное внимание; также изучению подлежит и [П], где описываются основные понятия, связанные с теорией фейеровских отображений и основы фейеровских методов для задач линейной оптимизации.

Выводы по разделу один

В данном разделе был осуществлён обзор использованной в рамках выполнения выпускной квалификационной работы и оформления пояснительной записки по ней литературы.

2 ТЕОРИЯ НАХОЖДЕНИЯ РЕШЕНИЯ СИСТЕМЫ ЛИНЕЙНЫХ НЕРАВЕНСТВ

Линейным неравенством называют такое неравенство, которое содержит линейные функции и один из символов неравенства.

В общем виде для вещественного конечномерного линейного пространства $М^{п}$ размерности n линейное неравенство задаётся следующим образом:

$$f\{x) < Ь, \qquad (1)$$

где $f(x)$ - линейная форма;

$x = (x_1; x_2,..., x_п)$ - вектор неизвестных переменных; b

- постоянный коэффициент.

Более конкретно можно записать выражение (1) с использованием вектора постоянных коэффициентов неравенства $a = (a_1; a_2,..., a_п)$.

$$a_г x_г + a_2 x_2 \text{ Н---- 1- } a_n x_n < b \qquad (2)$$

Решением неравенств вида (1) и (2) называют вектор таких значений х, при котором неравенство удовлетворяется.

Выражения (1) и (2) задают вид линейного неравенства для случая «меньше либо равно». Аналогичным образом задаётся вид выражения для других (строгих и нестрогих) знаков неравенства.

Остановимся отдельно на определении линейного неравенства для двумерного линейного пространства. Двумерное линейное неравенство, как частный случай для выражения (2) задаётся выражением (3).

$$a_1 x_1 + a_2 x_2 < b \qquad (3)$$

Множество решений такого неравенства можно графически представить как полуплоскость евклидовой плоскости (при неравенстве нулю коэффициентов a_2 и a_2; в данном случае решением можно считать либо пустое множество, либо всю евклидову плоскость). Иными словами, графической интерпретацией решения неравенства вида (3) является множество всех точек, лежащих по одну сторону от прямой, задающей границу множества решений (в уравнение такой прямой вырождается выражение (3) при замене в нём знака неравенства на знак равенства) (рисунок 1).

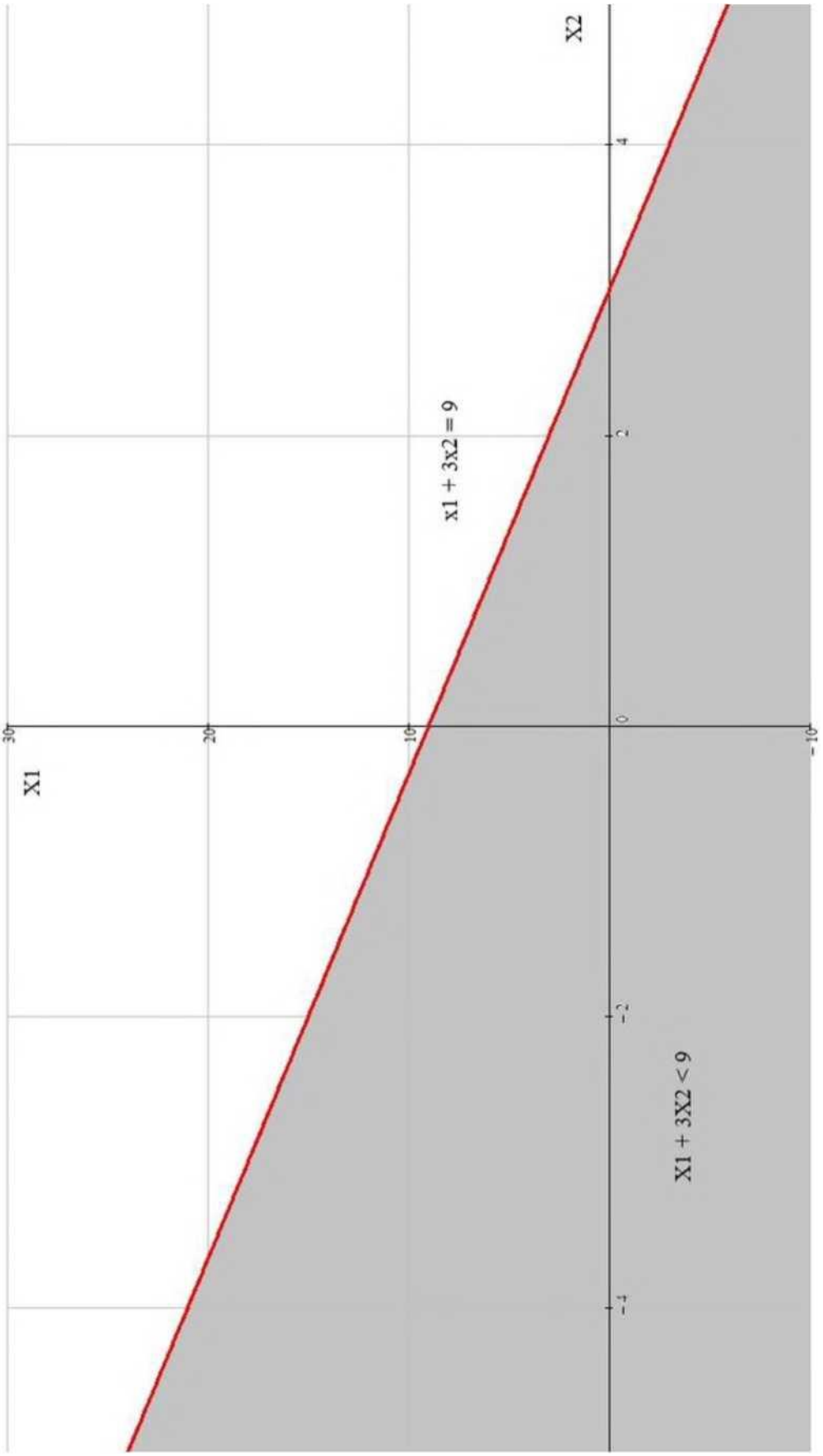

Рисунок 1 - Графическая интерпретация решения линейного неравенства для двумерного линейного пространства

На рисунке 1 изображено графическое представление решения линейного неравенства $x_{\pm} + 3x_2 <9$ (множество всех решений выделено серым цветом).

Под конечной системой линейных неравенств понимают систему вида (4) [10, с. 25].

$$(cL_{Jt}x) < bj, \qquad (4)$$

где $j = 1, m$ (m - количество неравенств системы).

То есть, система линейных неравенств представляет собой совокупность линейных неравенств вида (1) с одним вектором переменных x. Решением системы (4) является такой вектор x_0, который удовлетворяет всем её неравенствам. При наличии у системы (4) хотя бы одного решения она называется совместной.

В [10, с. 26] показано, что при совместности системы (4) множество её решений является выпуклым. Соответственно, решение такой системы называют также выпуклым полиэдральным множеством (или говорят, что система имеет многогранник решений).

Частным случаем системы (4) является случай системы неравенств для двумерного линейного пространства (рисунок 2).

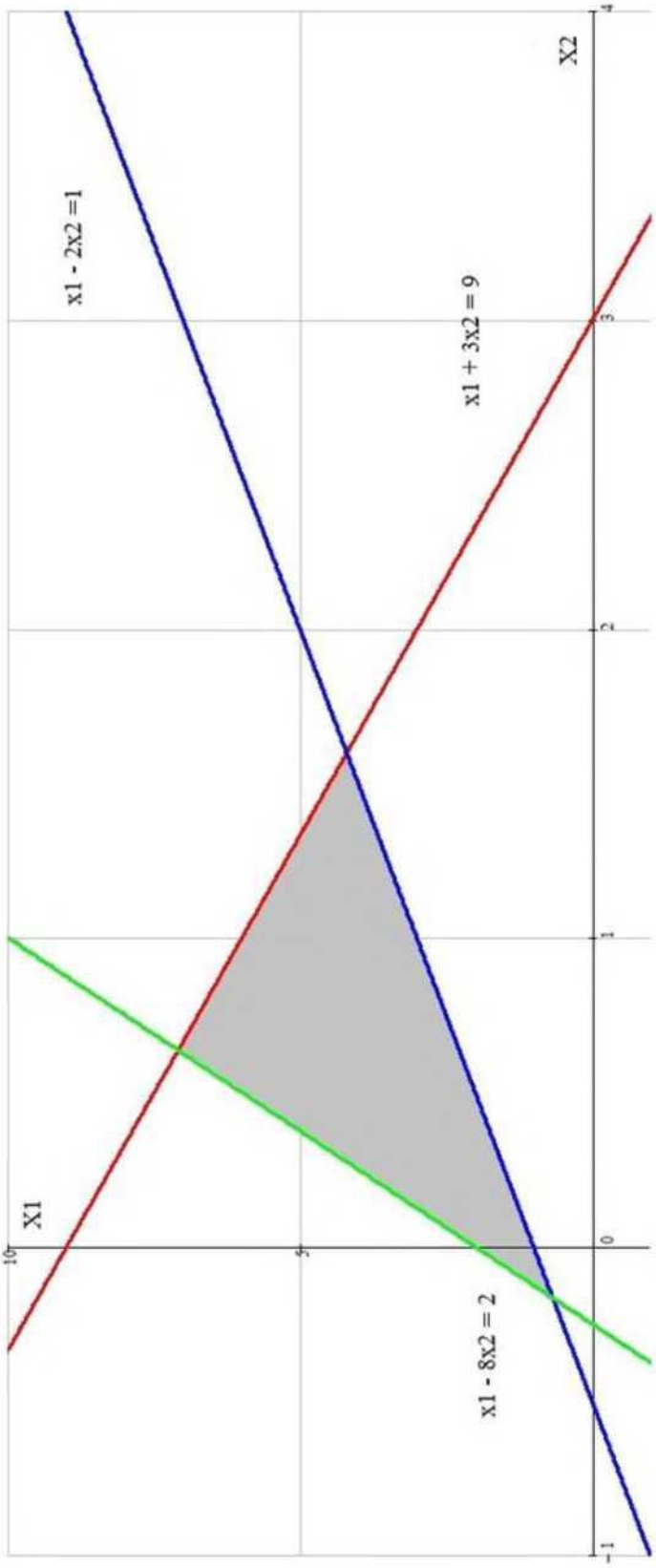

Рисунок 2 - Графическая интерпретация решения системы линейных неравенств в двумерном пространстве

Изображённый на рисунке 2 пример отражает графическое представление решения следующей системы линейных неравенств:

$$x_г + 3x_2 < 9$$
$$x_г — 2X_2 > 1$$
$$^x i \sim 8x_2 < 2$$

Действительно, для нахождения решения системы (то есть, множества таких значений неизвестных переменных, которые будут удовлетворять каждому из неравенств системы) необходимо построить выпуклый многогранник. Уравнения, получаемые из неравенств системы путём замены знака неравенства знаком равенства, представляют собой границы многогранника. Совокупность внутренних точек данного многогранника есть множество решений системы. В зависимости от того, нестрогими или строгими являются неравенства системы, точки границ могут входить или не входить в решение системы (в представленном примере - не входят).

Выводы по разделу два

Данный раздел посвящён краткому обзору основных понятий, касающихся теории нахождения решения систем линейных неравенств. Введены основные определения в данной области, рассмотрены фундаментальные понятия, касающиеся методов решения систем, на примере двухмерного пространства проанализирована графическая интерпретация решений.

3 ПОНЯТИЕ ФЕЙЕРОВСКИХ ОТОБРАЖЕНИЙ

Пусть ϕ £ {$Ж^n$ -> 2^{Rn}] и M = (x : x = <p(x)} - множество неподвижных точек оператора *(p*. Тогда отображение *(p* называется М-фейеровским, если:

$$||z-y|| <||x-y||, \forall z \in (p\{x), \forall y e M, \forall x £ M,$$

где z - некоторый фиксированный элемент из M^n [11].

Оператор *(p* является итерационным. Под итерационным понимается такой оператор, который (при произвольном начальном элементе x_0) обеспечивает сходимость процесса, порождаемого рекуррентным соотношением (5).

$$x_{t+}i = (p(pct), \quad t = \text{ОД},... \tag{5}$$

Другими словами, для последовательности (x_t) (которая порождена рекуррентно включением x_{t+1}<p(x_t)) при некотором заданном начальном значении x_0 G *Ж* М-фейеровское отображение при тех или иных ограничениях обеспечивает свойство ограниченности.

Под фейеровским процессом, порождаемым отображением *(p* при произвольном начальном приближении x_0 £ 1^n, будем понимать последовательность {«рДхо)}^ [23, с .2]. Известно, что указанный процесс сходится к точке, принадлежащей множеству М:

$$\{<P^s(*o)\}£o \quad * \mathbf{e} M \tag{6}$$

Обозначим выражение (6) как Нгщ^оо <$p^s(x_0)$

Под (^-проектированием (псевдопроектированием) точки х **е** 1^n на множество *М* понимается отображение 7г^(х) = Инт^о, <$p^s(x_0)$.

Выводы по разделу три

В данном разделе был произведён обзор важнейших понятий из области фейеровских методов, были введены ключевые положения теории фейеровских отображений.

4 ОСНОВЫ ТЕОРИИ ЛИНЕЙНОГО ПРОГРАММИРОВАНИЯ

Более общим для понятия линейного программирования является понятие математического программирования. Математическое программирование представляет собой математическую дисциплину, которая занимается изучением экстремальных задач и разработкой методов их решения [1, с. 5].

Задача математического программирования в общем виде ставится следующим образом:

$$/(x)\text{-> extr, при условиях } ^(x) < Ь,\ i = 1, m \qquad (7)$$

Другими словами, задачами математического программирования называют задачи, которые сводятся к отысканию значений параметров, обеспечивающих экстремум функции при наличии ограничений, наложенных на аргументы [7, с. 52]. При этом /(x) (функция, максимум или минимум которой требуется определить) называется целевой функцией.

Анализом и разработкой методов решения экстремальных задач занимается раздел математики, называемый методами оптимизации. Такие задачи встречаются в самых различных сферах человеческой деятельности. Основными направлениями теории экстремальных задач, находящимися в настоящее время в центре внимания математиков являются большие оптимизационные задачи, негладкие экстремальные задачи и задачи оптимизации в условиях неопределённости и в условиях динамически изменяющихся исходных данных [16, 23]. Большой конечномерная экстремальная задача называется тогда, когда количества её ограничений и переменных представляют собой большие числа.

При решении задач особых классов чаще всего применяются следующие подходы: совершенствование общих методов решения и разработка методов для специальных подклассов задач, специфицированных под особую, ярко выраженную структуру математических моделей таких задач. В теории методов

оптимизации классическим примером реализации последнего подхода можно считать методы решения, так называемых, транспортных задач, широко распространённых в приложениях математико-экономического моделирования.

Создание же методов негладкого анализа сводится, в сущности, к разработке методов поиска экстремальных значений функций, у которых отсутствуют непрерывные производные. Наконец, научным проблемам методологии нахождения решений задач с изменяющимися входными данными посвящена основная часть данной работы (частично данная тема затронута в конце данного раздела; центральное же место она занимает в последующих пятом, шестом и седьмом разделах).

Задачи математического программирования, в которых функции /(x) и $g_L(x)$ являются линейными, называются, соответственно, линейными задачами (в противном случае - нелинейными). Линейное программирование представляет собой наиболее изученный раздел теории математического программирования. Разработки в рамках данного раздела положили начало современной теории экстремальных задач, а для решения задач линейного программирования разработан целый ряд эффективных методов, алгоритмов и программ.

Из формулировки (7) видно, что система ограничений, накладываемых на неизвестные переменные, представляет собой (в случае задачи линейного программирования) систему линейных неравенств. Из изложенного выше во втором разделе следует, что нахождение области допустимых решений - множества всех точек, координаты которых удовлетворяют всем ограничениям системы - сводится к нахождению решения системы линейных неравенств.

Для удобства рассмотрим частный случай задачи линейного программирования в двумерном линейном пространстве. В этом случае постановка задачи имеет следующий вид.

$$f(x_1, x_2) = c_1 x_1 + c_2 x_2 \rightarrow extr$$

$$\begin{cases} a_{11} x_1 + a_{12} x_2 < b_1 \\ a_{21} x_1 + a_{22} x_2 < b_2 \\ \dots \\ a_{m1} x_1 + a_{m2} x_2 < b_m \end{cases}$$

Кроме того, на переменные часто накладывается дополнительное ограничение неотрицательности.

$$x_i \geq 0$$

$$x_2 > 0$$

В случае размерности, равной двум, возможно нахождение решения такой задачи с применением графического метода. Каждое из неравенств системы ограничений геометрически определяет полуплоскость, границей которой является прямая. Граница полуплоскости задаётся уравнением, получаемым из неравенства путём замены знака неравенства знаком равенства. Так как полуплоскость представляет собой, строго говоря, выпуклое множество, а пересечение любого числа выпуклых множеств также является выпуклым множеством, то и область допустимых значений есть выпуклое множество [17, с. 12].

Различные численные значения целевой функции задают линии уровня в двумерном пространстве. В целях нахождения максимального (минимального) значения целевой функции, рассматривают направление её возрастания (убывания). Это направление определяется градиентом целевой функции. Для линейной целевой функции градиент является вектором нормали к прямой линии уровня. Поэтому, построив градиент функции с помощью линий уровня, можно определить крайние точки области допустимых решений, в которых функция и будет достигать экстремума [17, с. 12].

Продемонстрируем описанный метод на следующем примере:

$$/(\text{x}) = 6x_{\pm} + x_2 \text{ -> } max$$

$$x_{\text{г}} + 3x_2 < 9\ x_{\text{г}} — 2X_2 > 1\ ^{x}i \sim 8\text{X}_2 < 2$$

$$x_{\pm} > 0, x_2 > 0$$

Г радиентцелевой функции, определяющий направление её возрастания, равен:

$$grad(f) = \{6; 1\}$$

Графическое представление области допустимых решений и направления вектора градиента целевой функции приведено на рисунке 3.

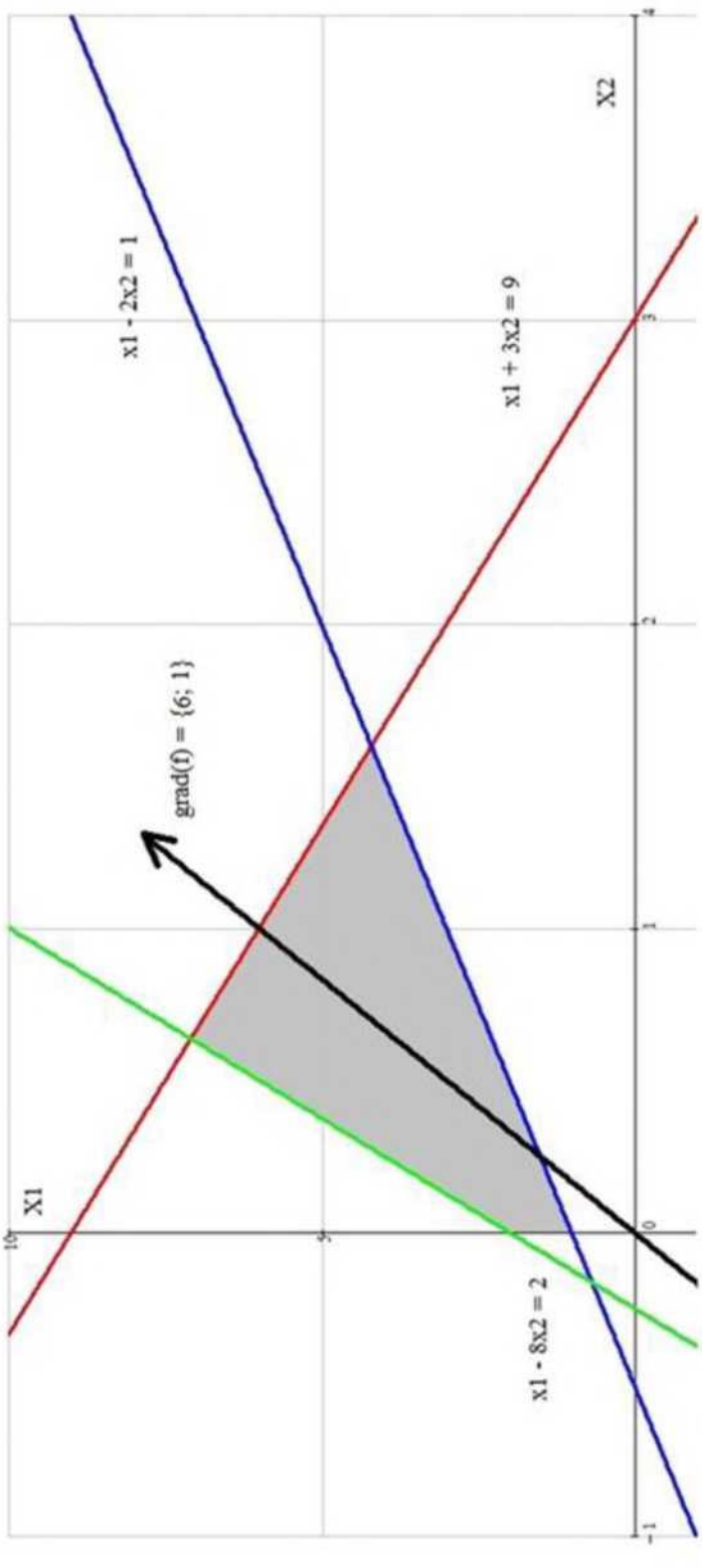

Рисунок 3 - Градиент целевой функции и область допустимых значений её аргументов для задачи линейного программирования в двумерном пространстве

Как видно из рисунка 3, искомый экстремум целевой функции достигается в точке, соответствующей левому верхнему углу области допустимых значений, то есть, на пересечении прямых, задаваемых уравнениями $x_{\pm} — 8x_2 = 2$ и $x_{\pm} + 3x_2 = 9$. Зная это, можно получить точку их пересечения:

$$x_0 = \left(\frac{78}{11}, \frac{7}{11}\right)$$

Подставив данное значение в целевую функцию, получим значение её максимума:

$$fmax\ /\text{Oo})^{,\ ч\ 78}_{7\quad 475} \quad 6\ ^{1}\ \text{YY} + \text{YY}\ \text{"YY"} — 43{,}18$$

Резюмируя вышесказанное, следует ещё раз подчеркнуть, что нахождение решения задачи линейного программирования сводится, фактически, к нахождению определённой точки области допустимых решений, представляющей собой выпуклый многогранник, в соответствии с направлением возрастания (убывания) функции. Данный многогранник (в общем случае п-мерный; двумерный случай был приведён исключительно из соображений визуальной наглядности) является представлением системы ограничений задачи. В этой связи методы нахождения решения системы линейных неравенств (описанные в разделе два) занимают центральное место в теории линейной оптимизации.

Методы решения задач линейного программирования широко применяются в практических приложениях. К задачам линейного программирования сводятся такие оптимизационные задачи, как поиск наиболее эффективного плана производства, поиск оптимального плана перевозок при ресурсных ограничениях и требованием обеспечения максимального дохода и т.п.

Однако существуют области, которым свойственен динамический характер исходных данных задачи. Это означает, что коэффициенты при переменных в

системе ограничений более не являются постоянными и могут в таких случаях с течением времени изменять свои значения. Как следствие, возникает необходимость нахождения экстремумов целевой функции в условиях непостоянности ограничений на переменные задачи. Очевидно, что при каждом изменении входных данных искомый экстремум будет отличаться от значения при предыдущем состоянии системы ограничений. Такие задачи линейного программирования носят название нестационарных.

Важно отметить, что применение наиболее распространённых и классических алгоритмов (к примеру, таких методов, как метод Гомори, метод внутренней точки, симплекс-метод, метод северо-западного угла и т.д.) при изменяющихся с течением времени исходных данных теряет свою эффективность и не обеспечивает получение оптимального решения задачи [24]. Данное обстоятельство обуславливает следующий вывод: разработка, реализация и исследование методов решения нестационарных задач представляет собой актуальное направление научной деятельности.

В качестве примеров областей, в которых возникают нестационарные задачи, можно выделить алгоритмическую торговлю, задачи оптимального управления пассивами и активами и т.п. При моделировании подобных процессов осуществляется конструирование нестационарных задач линейного программирования, причём таких, что период изменения входных данных системы ограничений может составлять сотые или даже тысячные доли секунды.

Выводы по разделу четыре

В данном разделе осуществлён обзор центральных понятий линейного программирования. На примере двумерной задачи был рассмотрен графический метод, а также сделан вывод о роли описанной в первом разделе теории линейных неравенств. Приведены выявленные по результатам анализа предметной области приложения теории линейного программирования. Также частично затронут вопрос о возможности динамического характера исходных данных задач линейного программирования.

5 АЛГОРИТМ NON-STATIONARY LINEAR PROGRAMMING

Как следует из изложенных выше соображений, проверенные и признанные научным сообществом классическими методы решения задач линейного программирования демонстрируют крайне слабую устойчивость к изменениям в исходных данных. Отсутствие эффективных методик нахождения решения при нестационарном характере коэффициентов системы ограничений привела к необходимости разработки усовершенствованных алгоритмов. Данной проблеме посвящена публикация [33], в которой описан алгоритм Non-Stationary Linear Programming (далее - NSLP), который предназначен для поиска решений нестационарных задач линейного программирования сверхбольших размерностей.

Теоретической основой алгоритма NSLP является предложенный авторами для преодоления нестационарности в работах [30, 32] подход, базирующийся на применении (описанных в третьем разделе) фейеровских отображений. Главная идея метода заключается в использовании фейеровских процессов для выполнения операции псевдопроектирования на выпуклое ограниченное множество.

Для дальнейшего рассмотрения алгоритма переформулируем общую постановку задачи линейного программирования следующим образом:

$$\max\{(c_t, x) \mid A_t x < b_t, x > 0), \qquad (8)$$

где матрица A_t имеет m строк.

Нестационарный характер задачи сводится к тому, что значения коэффициентов матрицы A_t векторов b_t, c_t зависят от момента времени t G M>$_0$.

Полагается при этом, что значение t = 0 соответствует начальному моменту времени:

$$A_0 = A, Ь_0 = Ь, c_0 = c$$

Обозначим через M_t задаваемый ограничениями нестационарной задачи линейного программирования (8) выпуклый многогранник.

Рассматриваемый нами в данной работе алгоритм NSLP включает в себя две фазы: Quest (поиск) и Targeting (позиционирование). Рассмотрим подробнее каждую из фаз в отдельности.

5.1 Фаза Quest

Подзадача фазы Quest заключается в поиске решения системы неравенств, задающих систему ограничений задачи линейного программирования в условиях динамического изменения исходных данных. Результатом выполнения фазы является нахождение точки на многограннике M_t.

После выбора произвольной точки z_0 G $M>_0$ в качестве начального приближения решения задачи (8) в целях решения поставленной подзадачи организуется фейеровский процесс вида (6).

Определим отображение $(p_t \backslash R^n$ -> M^nследующим образом:

$$\varphi_t(x) = x - \frac{\lambda}{m}\sum_{i=1}^{m}\frac{\max\{\langle a_{ti}, x\rangle - b_{ti}, 0\}}{\|a_{ti}\|^2}\cdot a_{ti}, \qquad (9)$$

где a_{ti} - i-ая строка матрица A_t;

$b_{t1},\ldots b_{tm}$ - элементы столбца b_t.

Обозначим:

$$\varphi(x) = \varphi_0(x) = x - \frac{\lambda}{m}\sum_{i=1}^{m}\frac{\max\{\langle a_i, x\rangle - b_i, 0\}}{\|a_i\|^2}\cdot a_i \qquad (10)$$

При вычислениях в ходе организованного на фазе Quest фейеровского процесса выполняется последовательное построение фейеровских приближений по формуле отображения (9). Данные приближения предназначены для вычисления точки псевдопроекции, расположенной на многограннике M_t. В

соответствии с нестационарным характером задачи (8) многогранник M_t в процессе вычисления псевдопроекции может менять свои форму и положение в пространстве.

Фаза Quest алгоритма NSLP на данном этапе исследования имеет некоторые ограничения. При его использовании рассматривается простейший случай нестационарности, представляющий собой параллельный перенос многогранника $M = M_0$ с фиксированной периодичностью на фиксированный вектор d $ЕЖ^n$. В этом случае $A_t = A$, $c_t =$ с, и нестационарная задача (8) имеет вид:

$$\text{тах}\{(\text{c,x}) \mid A(x — td) < b,\ x > 0),$$

что равносильно:

$$\text{тах}\{(\text{c, x}) \mid Ax < b + Atd,\ x > 0\} \qquad (11)$$

При сравнении (8) и (11) видно, что $b_t = b + Atd$. Тогда $M_г$фейеровское отображение (9) преобразуется к виду:

$$<\text{p}_t(\text{x}) = \text{x}\ \text{у}\ m\ \sum_{i=1}^{m} \max\{(\text{aj, x}) — (b_t + \{a_ь\ td)),\ 0)\ a_{a,-}\ V$$

что равносильно:

$$(p_t\{x) = x\sim — \text{Я}\sum_{i=i}^{m} \max\{(\text{aj, x} — td) — Ь_ь\ 0)\ a,- \qquad \text{[12]}$$

В описанной в [33] версии алгоритма нестационарность задачи смоделирована таким образом, что корректировка исходных данных на фазе Quest происходит каждые L итераций, где L - некоторое фиксированное целое положительное

число, являющееся параметром алгоритма. Тогда введём обозначения t_0, t_t,..., $t_{k)}$… - последовательные моменты времени, соответствующие корректировке исходных данных нестационарной задачи.

Положим:

$$t_0 = 0, t\backslash — L,\ t_2 = 2L, \ldots ,t_k = kL \qquad (13)$$

Пусть многогранник M_t принимает в моменты времени (13) формы и положения, обозначаемые как $M_0,\ M_{\pm},\ \ldots, M_к, \ldots$. Тогда $(p_0, (p_1, \ldots, (p_к, \ldots$ - фейеровские отображения, определяемые формулой (9) с учётом изменения данных задачи (8) в моменты времени (13).

Данный математический аппарат является основой для разработанного следящего алгоритма, способного находить решения нестационарных задач линейного программирования. М-фейеровское отображение характеризуется тем, что точка $(p\{pc)$ находится к задаваемому системой ограничений многограннику M ближе, чем исходная точка. Многократное применение фейеровского отображения, начиная с исходной точки z_0, задаёт итерационный процесс, заключающийся в вычислении последовательности точек псевдопроекций. Завершение данного итерационного процесса определяется некоторым положительным вещественным числом £, являющимся параметром алгоритма. При этом ключевым моментом является то, что фейеровский процесс сходится к точке, лежащей на границе многогранника M. Сходимость итерационного процесса доказана в [24]. Итерационным процесс называется по той причине, что вычисление каждой последующей точки по формуле (12) осуществляется на основе предыдущей. Псевдопроекцией же вычисляемая точка называется потому, что, в отличие от операции проектирования, траектория, по которой точки «приближаются» к многограннику, является криволинейной, однако, в конечном счёте, процесс сходится к его границе (поэтому можно считать, что вычислена

«проекция» особого рода - псевдопроекция - из начальной точки на многогранник).

Таким образом, операция псевдопроектирования в ходе итерационного процесса с помощью фейеровских отображений является «самонаводящейся»: при изменении системы неравенств, задающих многогранник M, фейеровское отображение изменяет направление своего движения. Также отметим, что фейеровский процесс для описанного случая сходится быстрее, чем изменяет своё положение перемещающийся в и-мерном пространстве многогранник нестационарной задачи линейного программирования *M*. Данный процесс проиллюстрирован на рисунке 4.

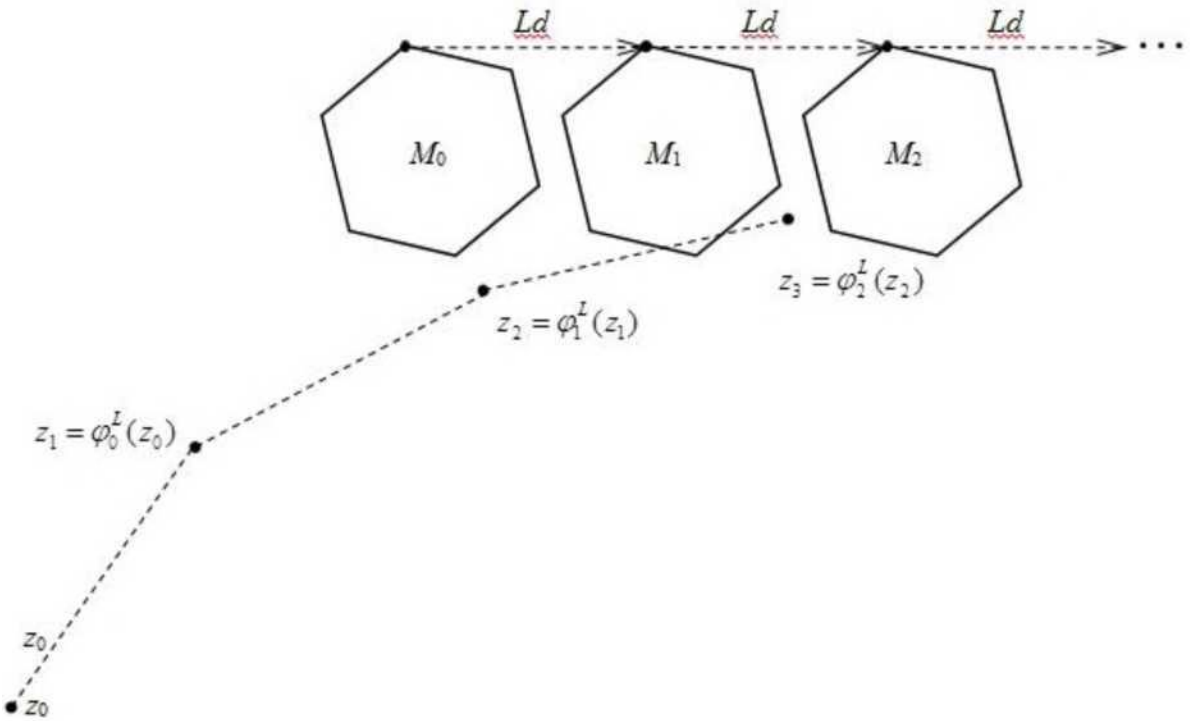

Рисунок 4 - Итерационный процесс на фазе Quest для нестационарной задачи линейного программирования на примере двумерного пространства

В приведённом на рисунке 4 вычисляемые в ходе фейеровского процесса псевдопроекции:

$$\{Zi = \langle pJ(z_0), Z_2 = (p|(z \quad = Pfc\text{-}iOfc\text{-}i), \ldots\}$$

5.2 Фаза Targeting

Фаза Targeting следует непосредственно после фазы Quest и предназначена для вычисления экстремального значения целевой функции, удовлетворяющего найденному на фазе Quest решению системы линейных неравенств. То есть результатом выполнения фазы является приближённое решение задачи (8), в то время как на фазе Quest главной задачей алгоритма было попадание в область допустимых значений переменных задачи (в задаваемый ограничениями задачи многогранник).

На фазе Targeting осуществляется формирование специальной системы точек, имеющей форму осесимметричного n-мерного креста. Из данных точек выбирается одна точка, называемая центральной. Данная область перемещается и масштабируется в n-мерном пространстве, причём таким образом, чтобы обеспечивалось нахождение решения задачи (8) в £-окрестности центральной точки креста.

Построение крестообразной области подчиняется ряду правил и имеет ряд особенностей. Приведём их описание.

Формирование крестообразной осесимметричной фигуры осуществляется вокруг центральной ячейки; рёбра каждой ячейки должны быть сонаправлены с осями координат. Параметрами крестообразной области являются количество ячеек по одному измерению K длина ребра ячейки s и координаты центральной ячейки. Таким образом, общее количество ячеек P в следящей области в n-мерном пространстве вычисляется по формуле (14).

$$P = n\{K - 1) + 1 \qquad (14)$$

Каждая ячейка следящей области характеризуется следующим набором параметров:

1) Уникальный номер a £ (О, …, P — 1}. Значение a, в сущности, выступает в роли индекса или порядкового номера ячейки.

2) Маркер (/,77). Маркером называется пара целых чисел / и ц, обладающих следующими свойствами:

$$0 < x < Л$$
$$\backslash г] \backslash < (К\text{-}1)/2$$

С неформальной точки зрения данные числа несут в себе следующий смысл:

- x называется измерением и задаёт сонаправленный с координатной осью с индексом x столбец ячеек;
- $г]$ называется индексом и задаёт порядковый (по отношению к центральной ячейке) номер ячейки в столбце.

Центральная ячейка считается относящейся к нулевому измерению, то есть, имеет значение измерения x — 0. Значения элементов маркера (x, rf) могут быть рассчитаны с использованием порядкового номера a и количества ячеек по одному измерению $К$ по формулам (15) и (16) соответственно [31, с. 216].

$$\begin{cases} г\ \mathbf{0}, if\ a = \mathbf{0}, \\ \wedge\ \ |(\text{а} — 1)\text{-н}\ (К — 1), if\ a > 0 \end{cases} \tag{15}$$

$$V = \begin{cases} (\\ {}^{1})mod\ \text{I} \quad if\ a = 0, \\ /К\text{-}1\backslash К\ \text{-}\ 1, (\text{«}\ \text{-} \\ )\text{-------}—, \quad if\ 0 < (a — 1)mod(K — 1) < \dfrac{К\text{-}1}{К\text{-}1} \\ \wedge\ \ (a\ \text{-}\ i)mod\ (—\text{-}—1 + 1, \end{cases} \tag{16}$$

Также можно совершить обратное преобразование и рассчитать значение индекса a исходя из значений элементов маркера $(x, г/)$ по формуле (17) [31, с. 216].

$$a = \begin{cases} г \quad 0, if\ г] = 0, \\ К — 1 \\ <г] + \quad + x(К\text{-}1) + 1, if\ Л < 0, \\ К — 1 \\ г] + \quad + x(К\text{-}1), if\ Ц > 0 \end{cases} \tag{17}$$

Изображения следящей области для размерности $n = 2$ и количеством ячеек по одному измерению $K = 7$ с маркерной нумерацией (при помощи маркера $ix>^{r}i))^{и\,с}$ соответствующей линейной нумерацией (при помощи порядкового номера *а)* приведены на рисунках 5 и 6 соответственно.

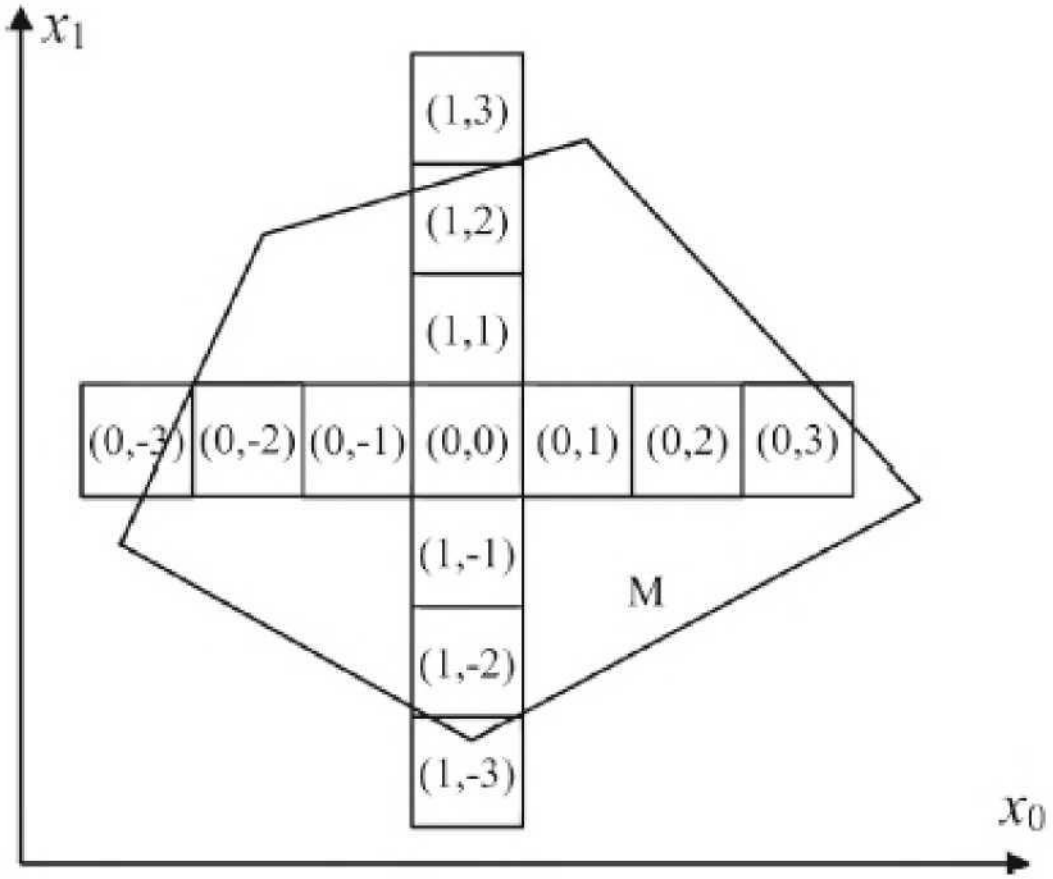

Рисунок 5 - Крестообразная следящая область для размерности $n = 2$ с нумерацией ячеек с помощью маркера *(j*, ц)

3) Координаты нулевой вершины. Нулевой вершиной кубической (имеется в виду в общем случае п-мерный куб) ячейки называют такую её вершину, которая является ближайшей (при сравнении со всеми другими её вершинами) к началу координат, у'-ая координата у; нулевой вершины ячейки с измерением $x^{и}$ индексом *г]* рассчитывается по формуле (18) [31, с. 215].

$$y_j = \begin{cases} \partial_x + r]s, \; if \; j = x, \\ 9j > \; if j * X \end{cases}, \tag{18}$$

где $j = \overline{0, n - 1}$;

g_j - j-ая координата нулевой вершины центральной ячейки.

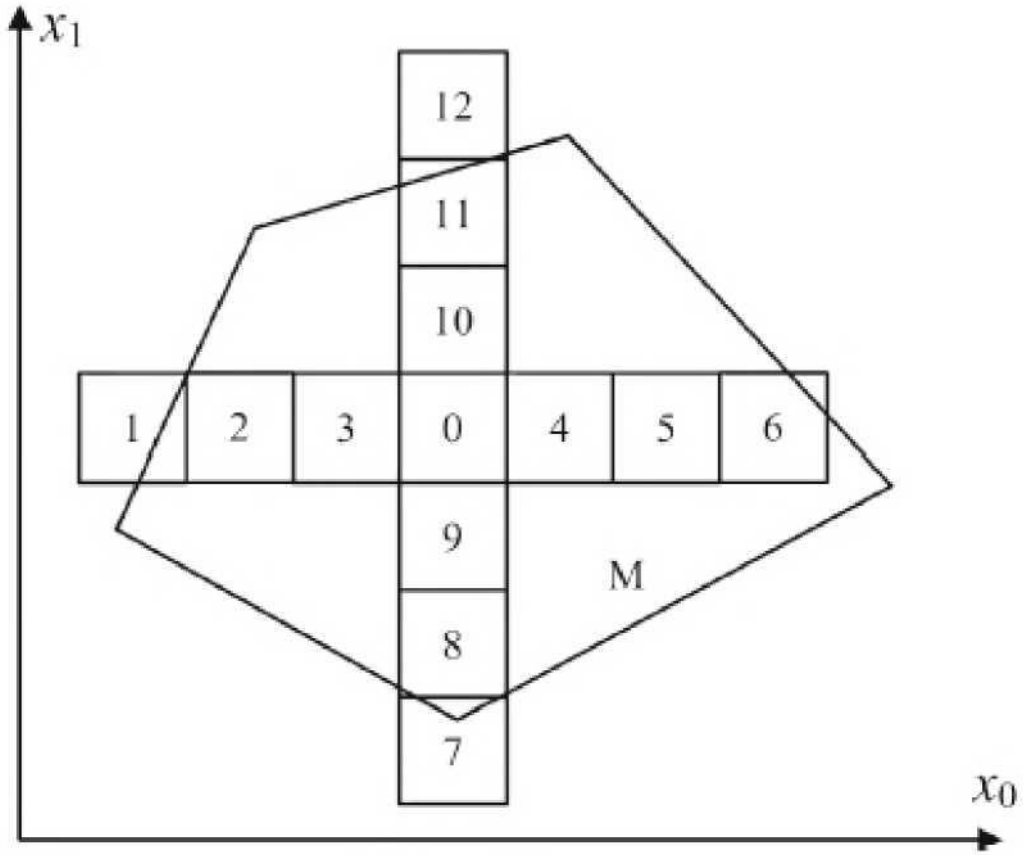

Рисунок 6 - Крестообразная следящая область для размерности $n = 2$ с линейной нумерацией ячеек

Опишем алгоритм вычислений при позиционировании на фазе Targeting.

1) В качестве начального приближения выбирается точка z на многограннике M, полученная в результате вычислений на фазе Quest. Первоначально формируется крестообразная следящая область с нулевой вершиной центральной ячейки в точке z;

2) Из точки z осуществляется вычисление псевдопроекций на пересечения каждой из ячеек следящей области с многогранником M ограничений задачи; при этом вычисления производятся при перемещающемся многограннике М, то есть, в условиях динамического изменения данных задачи. Ячейки, имеющие пустое пересечение с многогранником, отбрасываются.

Рассмотрим более детально вычисление пересечения многогранника M с ячейкой. Область внутри ячейки a (включая границы) задаётся независимо с

помощью системы из $2n$ **неравенств** (19).

$$
\left|\begin{array}{lll}
\mathbf{1}\ \mathbf{0} & < & -\text{Уо} \\
-x_{\pm} & < & \sim Y1 \\
 & < & \\
\mathbf{-*-71-1} & < & \\
x_0 & < & \sim Yn\text{-}1\ Yo + s\ ' \\
\mathbf{*1} & < & y\,i + s \\
 & < & \\
\mathbf{\bullet*\bullet 71\text{-}1} & < & Yn\text{-}1 + {}^{s}
\end{array}\right. \qquad (19)
$$

где у, - j-ая координата нулевой вершины ячейки a.

Запишем систему (19) в матричном виде.

$$A_a x < Ь_a, \qquad (20)$$

где A_a - матрица коэффициентов при переменных системы (19);
$Ь_a$ - столбец свободных членов системы (19).

В качестве примера распишем матрицу A_a и столбец $Ь_a$ для размерности $n = 3$:

$$A(X \quad \begin{matrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix} \quad \begin{matrix} \backslash \\ \\ \\ \\ / \\ \\ \end{matrix}$$

$$/ \begin{matrix} -Уо \\ -У1 \\ \sim У2 \end{matrix} \backslash$$

$$'a \quad \begin{matrix} Уо + 5 \\ У_2 + * \\ < У_2 + \end{matrix}$$

Ведём понятия расширенной матрицы A и расширенного столбца свободных членов $Ъ\backslash$ задаваемые формулами (21) и (22) соответственно.

$$A' = \begin{bmatrix} A \\ A_\alpha \end{bmatrix}, \tag{21}$$

где A' имеет размерность $(m + 2n)$ X n.

$$Ь' = \begin{bmatrix} b \\ b_\alpha \end{bmatrix}, \tag{22}$$

где $Ь'$ имеет размерность $(m + 2n)$ X 1.

Таким образом, пересечение многогранника M с ячейкой a задаётся в матричной форме при помощи системы неравенств (23) (при задании пересечения используются введённые выше понятия расширенной матрицы A' и расширенного столбца свободных членов $Ь')$.

$$A'x < Ь' \tag{23}$$

У расширенного столбца $Ь'$ (в соответствии с формулой (22)) имеется вариативная часть $Ь_a$, которая зависит как от параметров ячейки a, пересечение

многогранника M с которой требуется найти (координаты её нулевой вершины (y_0,... , y_n-i)), так и от параметров самой следящей области (длина ребра ячейки s), и инвариативную часть Ь, не зависящую ни от первого, ни от последнего. Элементы же расширенной матрицы A', в свою очередь, целиком не зависят от координат нулевой вершины ячейки а и от длины ребра ячейки следящей области s.

Таким образом, для получения псевдопроекции на пересечение многогранника M и ячейки a необходимо организовать итерационный процесс из точки z с вычислениями на каждой итерации по формуле (12), где в качестве значений a_t и b_L используются соответственно элементы расширенной матрицы A' и расширенного столбца $Ь'$.

При этом пересечение многогранника M с ячейкой a является пустым, если точка псевдопроекции из точки z на это пересечение не принадлежит многограннику М;

3) Если по результатам выполнения шага 2) ячейки, имеющие непустое пересечение с многогранником М, отсутствуют, то длина ребра ячейки s увеличивается в w раз (w является параметром алгоритма; назовём w коэффициентом масштабирования или масштабирующим коэффициентом) и алгоритм переходит снова на шаг 2);

4) При получении на шаге 2) непустого множества псевдопроекций осуществляется вычисление значения целевой функции для каждой из (неотброшенных на шаге 2)) ячеек. Затем для каждого измерения определяется максимум значения целевой функции. После этого вычисляется центр масс полученных n точек и происходит параллельный перенос следящей области, таким образом, чтобы нулевая вершина новой центральной ячейки z совпала с найденным центром масс;

5) При необходимости происходит масштабирование следящей области путём корректировки длины ребра её кубических ячеек в соответствии с расстоянием, на которое на шаге 4) был осуществлён её параллельный перенос. Так, если

расстояние между центром масс и нулевой вершиной центральной ячейки составляет меньше, чем ^ s, то размер ребра ячейки s уменьшается в два раза (то есть, при слишком незначительном перемещении креста его размер необходимо уменьшить). Если же расстояние между центром масс и нулевой вершиной центральной ячейки составляет больше, чем то s увеличивается в полтора раза (то есть, напротив, более значительное перемещение креста свидетельствует о необходимости увеличения его размера);

6) Переход на шаг 2).

Перемещение и масштабирование следящей области при выполнении шагов 1 - 6) происходит в условиях перемещающегося многогранника *M*.

Выводы по разделу пять

Данный раздел посвящён рассмотрению масштабируемого алгоритма решения задачи линейного программирования с изменяющимися входными данными (алгоритма NSLP). Произведены анализ особенностей таких задач и обзор научных источников, содержащих последние наработки в данной области. Раздел содержит главным образом описание структуры и основных понятий алгоритма NSLP, а также его текстовое описание. Детально были разобраны фазы алгоритма (Quest и Targeting). Также в данном разделе были рассмотрены фундаментальная теоретическая основа, особенности и ограничения применения алгоритма.

6 РЕАЛИЗАЦИЯ АЛГОРИТМА NON-STATIONARY LINEAR PROGRAMMING

Алгоритм NSLP был в ходе выполнения выпускной квалификационной работы реализован в виде программы на языке Python версии 3.7. Листинг программы приведён в Приложениях 1-4.

Особенностью нашей реализации, отличающей её от описанной в [30-33] версии алгоритма является то, что нестационарность в нашей работе смоделирована таким образом, что параллельный перенос многогранника M происходит каждые t секунд, в то время как в существующих реализациях перенос происходит каждые L итераций. Фактически, в работах [30-33] в реализациях авторов нестационарность задачи привязывается к итерациям выполнения программы, в то время как наша реализация подразумевает привязанное к реальному времени изменение исходных данных независимо от вычислений программы с заданной заранее периодичностью t (период изменения исходных данных задаётся как параметр задачи).

Реализованная на текущий момент версия программы предназначена только для решения модельных задач. Для тестирования и отладки программы использовалась модельная задача Model-n. Model-n представляет собой шаблон для конструирования имеющих решение задач линейного программирования любой размерности. Вид задачи Model-n произвольной размерности n представлен формулой (24).

$$
\begin{cases}
(*0 & & & < & 200 \\
 & Xi & & < & 200 \\
 & & X_{JI-л} & < & 200 \\
\circ\circ\ + & Xi & л & < 200(\text{n} > & \\
\ + & & + X_{11}^{3} & & \\
 & Xi & + X_{31}^{л} & & - 1) + 100\ 100 \\
x_0 & & & > & 0 \\
 & Xi & & > & 0 \\
 & & X_{JI-л} & > & 0
\end{cases}
\qquad (24)
$$

$$
Q_{max}\ (\text{x}) = 2x_0 + 2x_{\pm} + \text{—Ь } 2x_{n\_z} + x_{n\_1}
$$

С помощью разработанной программы помимо нестационарных задач можно решать также и стационарные задачи линейного программирования.

При разработке использовались библиотеки NumPy для матричных вычислений, Threading для реализации счётчика времени, а также Datetime для фиксации времени выполнения программы (более подробно о проведении вычислительных экспериментов написано в седьмом разделе).

При реализации алгоритма была разработана архитектура, предусматривающая следующие классы:

1) Класс Counter. Предназначен для моделирования счётчика времени, в соответствии с отсчётами которого происходит параллельный перенос многогранника *M*. Объекту класса Counter при инициализации передаётся период изменения исходных данных в секундах. При старте счётчика происходит инициализация объекта класса Timer (из библиотеки Threading). При создании класса также инициализируется специальный атрибут t. Смысл данной переменной аналогичен смыслу переменной t, используемой в подразделе 5.1 как счётчик моментов времени. Значение данного атрибута увеличивается с фиксированной периодичностью в отдельно организованном вычислительном потоке. При этом инкрементируемое параллельно главному потому вычислений значение t обновляется при вычислениях по формуле (12) в реальном времени. Подробнее об упомянутых вычислениях в пункте 3);

2) Класс **Model Problem.** Предназначен для формирования модельной задачи линейного программирования для тестирования алгоритма. Объекту класса **Model Problem** при инициализации передаются: размерность задачи; координаты вектора, на который осуществляется параллельный перенос многогранника; булева переменная, являющаяся флагом стационарности/нестационарности задачи (при передаче логической единицы - нестационарна); период изменения исходных данных в секундах. Также при вызове конструктора класса инициализируется атрибут, представляющий собой экземпляр класса Counter.

Код классов **Model Problem** и Counter содержится в модуле **model_problem. py** и приведён в Приложении **1;**

3) Класс **Quest.** Предназначен для реализации фазы **Quest** алгоритма. Объекту класса **Quest** при инициализации передаются экземпляр класса **Model Problem** и коэффициент релаксации Я, использующийся в расчёте точек псевдопроекций с помощью фейеровского отображения. Отметим, что в классе **Quest,** помимо прочих, предусмотрен метод **_fejer_mapping,** реализующий вычисления (для нестационарных задач) по формуле (12). Организация фейеровского итерационного процесса также реализована на уровне данного класса и имплементирована при помощи публичного метода **find_solution_of_inequalities_system.** Данный метод имеет, в числе прочих, булев аргумент **time it** и предусматривает фиксацию времени вычисления и количества итераций при установке данного аргумента в логическую единицу. Критерий останова реализованного в методе фейеровского процесса предусмотрен следующий: вычисление прекращается тогда, когда расстояние между двумя последними точками становится меньше некоторого заданного заранее числа (пересчитываемого из задаваемой при запуске программы порядка точности вычислений на фазе Quest).

Как было сказано выше, программа предусматривает нахождение решение и стационарных задач. Для этого в классе **Quest** предусмотрена реализация фейеровского процесса для стационарного случая и, как следствие, неподвижного

многогранника (то есть, фактически, решающая обыкновенную систему линейных неравенств как в классических задачах линейного программирования). Если система ограничений не изменяется во времени, то коэффициенты матрицы A и столбца свободных членов b остаются постоянными. В сущности, стационарность системы означает, что вектор, на который осуществляется перенос многогранника, является нулевым: $d = 0$. Это, в свою очередь, означает, что формула (12) вычисления псевдопроекции с помощью фейеровского отображения вырождается в формулу (25).

$$\varphi_t(x) = x - \frac{\lambda}{m}\sum_{i=1}^{m}\frac{\max\{\langle a_i, x\rangle - b_i, 0\}}{\|a_i\|^2}\cdot a_i \qquad (25)$$

Для стационарного случая в методе **_fejer_mapping** класса **Quest** вычисление псевдопроекции осуществляется по формуле (25).

Код класса **Quest** содержится в модуле **quest.py** и приведён в Приложении 2;

4) Класс **Cross.** Предназначен для формирования крестообразной следящей области на фазе Targeting. Объекту класса **Cross** при инициализации передаются координаты нулевой вершины центральной ячейки, размерность задачи, количество ячеек по одному измерению и длина ребра ячейки. В классе предусмотрен метод **_form_points** возвращает JSON структуру, содержащую для каждой ячейки значения порядкового номера a, измерения x, индекса ц, а также координаты нулевой вершины. Код класса **Cross** содержится в модуле **cross. py** и приведён в Приложении 3;

5) Класс NSLP. Предназначен для реализации алгоритма NSLP. В сущности, класс реализует функциональность алгоритма на глобальном уровне и является, по сути, основным во всей архитектуре, в то время как классы 1) - 4) выполняют, скорее, вспомогательные функции и предоставляют базу для всей программы.

Объекту класса NSLP при инициализации передаются:

- размерность задачи;
- точность вычислений на фазе Targeting;
- координаты вектора, на который осуществляется параллельный перенос многогранника;
- булева переменная, являющаяся флагом стационарности/нестационарности задачи (при передаче логической единицы - нестационарна);
- период изменения исходных данных в секундах;
- коэффициент релаксации Я;
- координаты начальной точки для фазы Quest;
- порядок точности вычислений на фазе Quest;
- количество ячеек крестообразной следящей области по одному измерению;
- первоначальное значение длины ребра кубической ячейки следящей области;
- коэффициент масштабирования длины ребра ячейки при пустом пересечении многогранника и следящей области.

Класс имеет публичный метод **solve**, при вызове которого осуществляется поиск решения модельной задачи линейного программирования. Метод имеет один аргумент - **time it**, являющийся булевой переменной и позволяющий (при установке его в логическую единицу) фиксировать общее время вычислений и число итераций фазы Targeting. При передаче аргумента **timeit** = **False**, метод возвращает только приближённое решение задачи линейного программирования.

Метод расчёта центра масс точек **_calc_mass_center** реализован в виде функции расчёта покоординатного среднего арифметического. Итерационное вычисление решения (позиционирование и масштабирование следящей области) осуществляется пока расстояние между двумя последними вычисленными точками больше заданной точности (некоторое фиксированное число, задаваемое при запуске программы).

При инициализации экземпляра класса **NSLP** осуществляется проверка значения K на нечётность (нечётное число ячеек по одному измерению обусловлено осесимметричностью следящей области).

Также в программе реализовано специальное исключение **TargetingDoesNotConvergeError** (класс, наследующий базовый класс исключений **Exception** в Python). Данный объект предназначен для предотвращения «зависания» программы в бесконечном цикле при вычислении псевдопроекций на пересечения многогранника M с ячейками следящей области. Как указано в пункте 3) подраздела 5.2 при описании фазы Targeting, при пустом множестве пересечений осуществляется масштабирование следящей области и повторный расчёт псевдопроекций. Наша реализация предусматривает генерацию исключения **TargetingDoesNotConvergeError** и, соответственно, экстренный выход из программы в случае, если с десяти попыток пересечения с многогранником так и не находятся. Данное свойство программы используется при поиске значения масштабирующего коэффициента, обеспечивающего сходимость алгоритма (седьмой раздел).

Код класса **NSLP** содержится в модуле **nslp. py** и приведён в Приложении **4;**

О примерах использования программной реализации алгоритма NSLP говорится в седьмом разделе, посвящённом вычислительным экспериментам над программой.

Выводы по разделу шесть

В данном разделе был осуществлён краткий обзор реализации алгоритма NSLP. Приведены описание классов, формирующих архитектуру программы, основные инструменты, использовавшиеся при разработке, а также некоторые особенности программной реализации алгоритма.

7 ПРОВЕДЕНИЕ ВЫЧИСЛИТЕЛЬНЫХ ЭКСПЕРИМЕНТОВ

Исследование алгоритма в ходе выполнения работы было выполнено путём проведения ряда вычислительных экспериментов. Все эксперименты проводились на примере модельных задач Model-n (формула (24)) трёх размерностей: 10, 20 и 30. Для исследования использовались библиотеки Tqdm для отслеживания хода экспериментов, **Pickle** для сохранения их результатов и **Matplotlib** для построения графиков. Построение графиков осуществлялось в интерактивной среде **Dupyter Notebook**, а для вывода на экран графиков использовалась специально написанная для этой цели функция **plot_results**, представленная в модуле **plotting, py.** Листинг модуля **plotting, py** приведён в Приложении **5.**

Рассмотрим проведение экспериментов более подробно.

7.1 Исследование стационарного варианта фазы Quest

Как было сказано в шестом разделе, программная реализация алгоритма предусматривает решение стационарных задач, а, следовательно, и организацию фейеровского процесса для неподвижного многогранника с вычислением псевдопроекции по формуле (25).

В ходе эксперимента исследовались зависимости времени вычисления и количества итераций фейеровского процесса от коэффициента релаксации Я, использующегося в формуле фейеровского отображения (формуле (25) для стационарного случая). При анализе формулы (25) видно, что увеличение значения Я приводит к тому, что при расчёте каждой последующей точки в ходе фейеровского процесса из предыдущей точки вычитается большая сумма. С неформальной точки зрения это означает, что расстояние между двумя соседними точками тем меньше, чем меньше коэффициент релаксации. Соответственно, одно и то же расстояние фейеровский процесс с низким Я «преодолеет» медленнее и за большее число итераций, чем с большим Я. А значит, увеличение значения коэффициента релаксации Я должно приводить к более быстрой сходимости.

Значение Я при проведении эксперимента варьировалось от 0.1 до 10 с шагом 0.1. В качестве начальной точки взята точка начала координат. Порядок точности - три (то есть, решение системы неравенств ищется с точностью, равной 0.001).

Листинг кода эксперимента приведён в Приложении 6. Полученные в результате графики представлены на рисунках 7 и 8.

Приведённые на рисунках 7 и 8 графики иллюстрируют нелинейный убывающий характер зависимости времени вычисления и количества итераций стационарного варианта фазы Quest при увеличении коэффициента релаксации Я. Это подтверждает основанную на интуитивном предположении вышеупомянутую гипотезу о влиянии значения Я на сходимость алгоритма.

При этом при фиксированном значении Я время и число итераций ожидаемо тем больше, чем больше размерность задачи. Однако при достаточно больших значениях Я разница в скорости решения для разных размерностей становится всё менее существенной. Следовательно, для обеспечения более быстрой сходимости необходимо выбирать значение Я = 10.

Отметим, что проведению аналогичного описанному исследования посвящена работа [6].

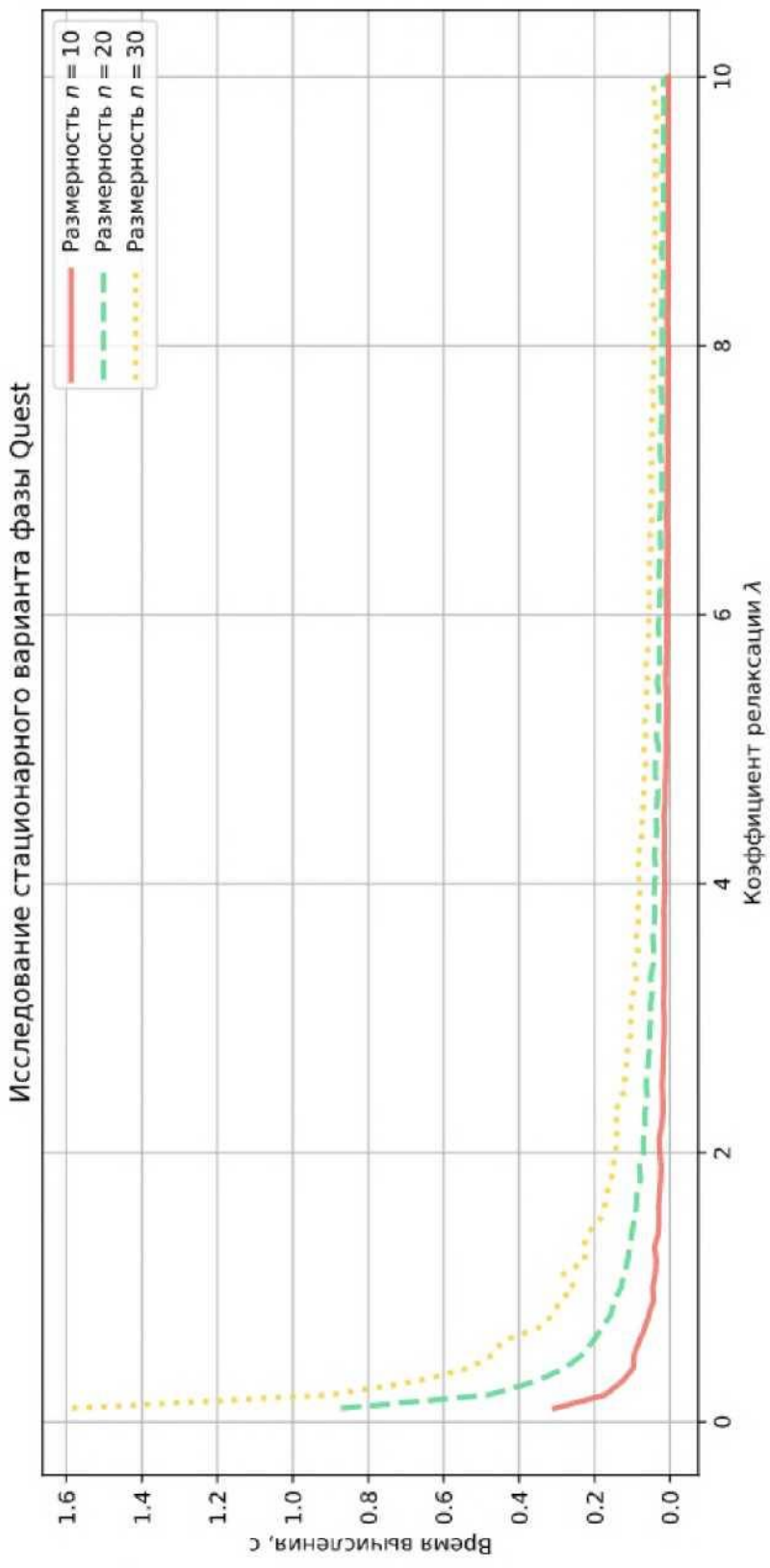

Рисунок 7 - Зависимость времени вычисления стационарного варианта фазы Quest алгоритма NSLP от коэффициента релаксации *Л* для размерностей *n* = 10,20,30

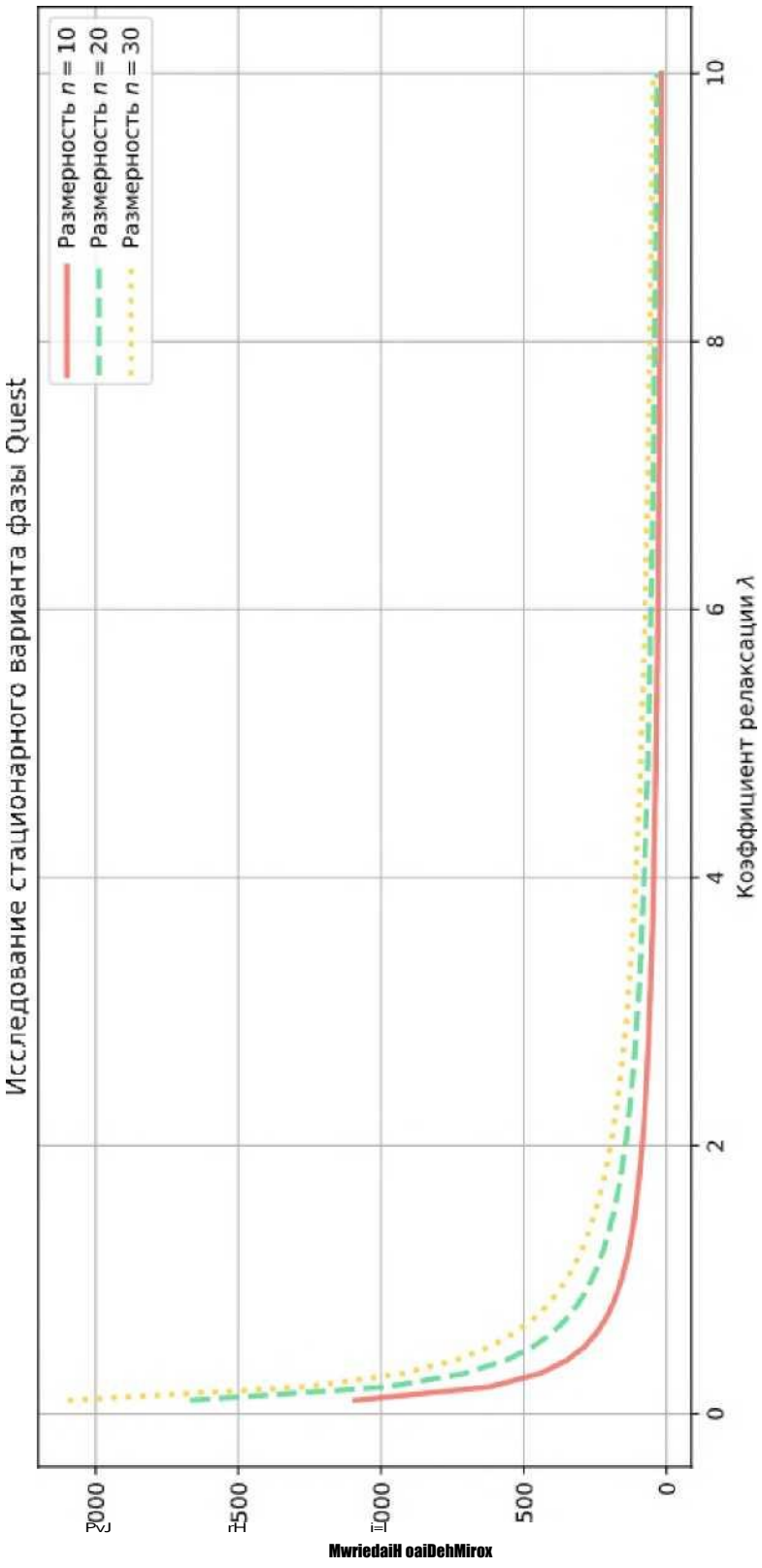

Рисунок 8 - Зависимость количества итераций стационарного варианта фазы Quest алгоритма NSLP от коэффициента релаксации *Л* для размерностей *n* = 10,20,30

7.2 Исследование нестационарного варианта фазы Quest

Так как в ходе исследований для примера использовалась нестационарная задача (с перемещающимся многогранником ограничений), вычисление фейеровского отображения осуществлялось по формуле (12).

В ходе экспериментов исследовались зависимости времени вычисления и количества итераций фейеровского процесса от модуля вектора, на который осуществляется параллельный перенос многогранника задачи. При проведении экспериментов все координаты вектора увеличивались на одну и ту же величину. К примеру, при решении модельной задачи размерности и = 10 координаты вектора варьировались следующим образом:

$$d = (1,1,1,1,1,1,1,1,1,1\} \rightarrow$$
$$\rightarrow d = (100,100,100,100,100,100,100,100,100,100) \rightarrow$$
$$\rightarrow d = (200, 200,200,200,200,200,200,200, 200,200) \rightarrow$$
$$\rightarrow d = (300, 300,300,300,300,300,300,300, 300,300) \rightarrow$$
$$\rightarrow d = (400,400,400,400,400,400,400,400,400,400,) \rightarrow$$
$$\rightarrow d = (500, 500,500,500,500,500,500,500, 500,500) \rightarrow$$
$$\rightarrow d = (600, 600,600,600,600,600,600,600, 600,600) \rightarrow$$
$$\rightarrow d = (700, 700,700,700,700,700,700,700, 700,700) \rightarrow$$
$$\rightarrow d = (800,800,800,800,800,800,800,800,800,800) \rightarrow$$
$$d = (900,900,900,900,900,900,900,900,900,900) \rightarrow$$
$$\rightarrow d = (1000,1000,1000,1000,1000,1000,1000,1000,1000,1000)$$

Аналогичным образом изменялось и значение вектора сдвига для задач размерностей я = 20 и я = 30 - координаты вектора *d* принимали аналогичные значения, однако число координат, очевидно, соответствовало размерности задачи. В качестве начальной точки взята точка начала координат. Порядок точности - три. Коэффициент релаксации был взят равным Я = 10 (для обеспечения более быстрой сходимости - в соответствии с выводами подраздела

7.1 и при предположении о том, что влияние параметра Я на сходимость процесса одинаково как для стационарной (формула (25)), так и для нестационарной (формула (12)) задач). Период изменения исходных данных был взят равным 1.3 секунды - такое значение было подобрано экспериментальным путём (выбран такой период, чтобы сходимость фазы Quest гарантированно обеспечивалась для задач размерностей $n = 10,20,30$).

Листинг кода эксперимента приведён в Приложении 7. Полученные в результате графики представлены на рисунках 9 и 10.

Приведённые на рисунке 9 графики иллюстрируют отсутствие зависимости времени вычисления фазы Quest от модуля вектора, на который осуществляется параллельный перенос многогранника. В значениях времени вычисления наблюдаются (в рамках одной размерности) незначительные колебания при изменении координат вектора d, однако этими отклонениями можно пренебречь и считать время постоянной величиной, не зависящей от модуля вектора d. При этом при фиксированном значении координат вектора d время вычисления, закономерным образом, увеличивается с увеличением размерности задачи. Таким образом, «интенсивность» (которую с неформальной точки зрения воплощает величина координат вектора d) смещения многогранника нестационарной задачи линейного программирования не влияет на время, которое требуется для нахождения точки на данном многограннике с помощью фейеровского процесса на фазе Quest.

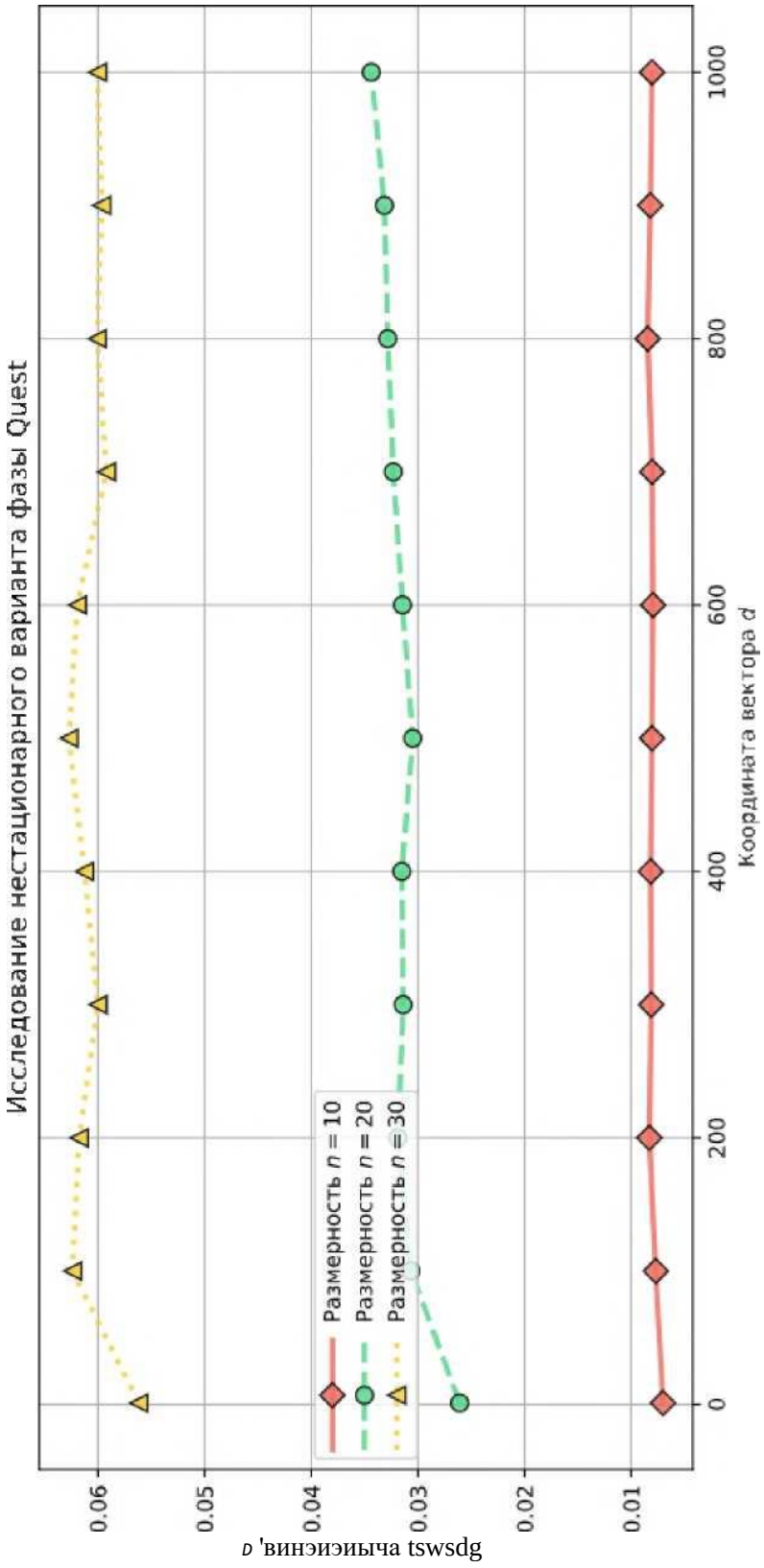

Рисунок 9 - Зависимость времени вычисления нестационарного варианта фазы Quest алгоритма NSLP от модуля вектора *d* параллельного переноса многогранника для размерностей *n* = 10,20,30

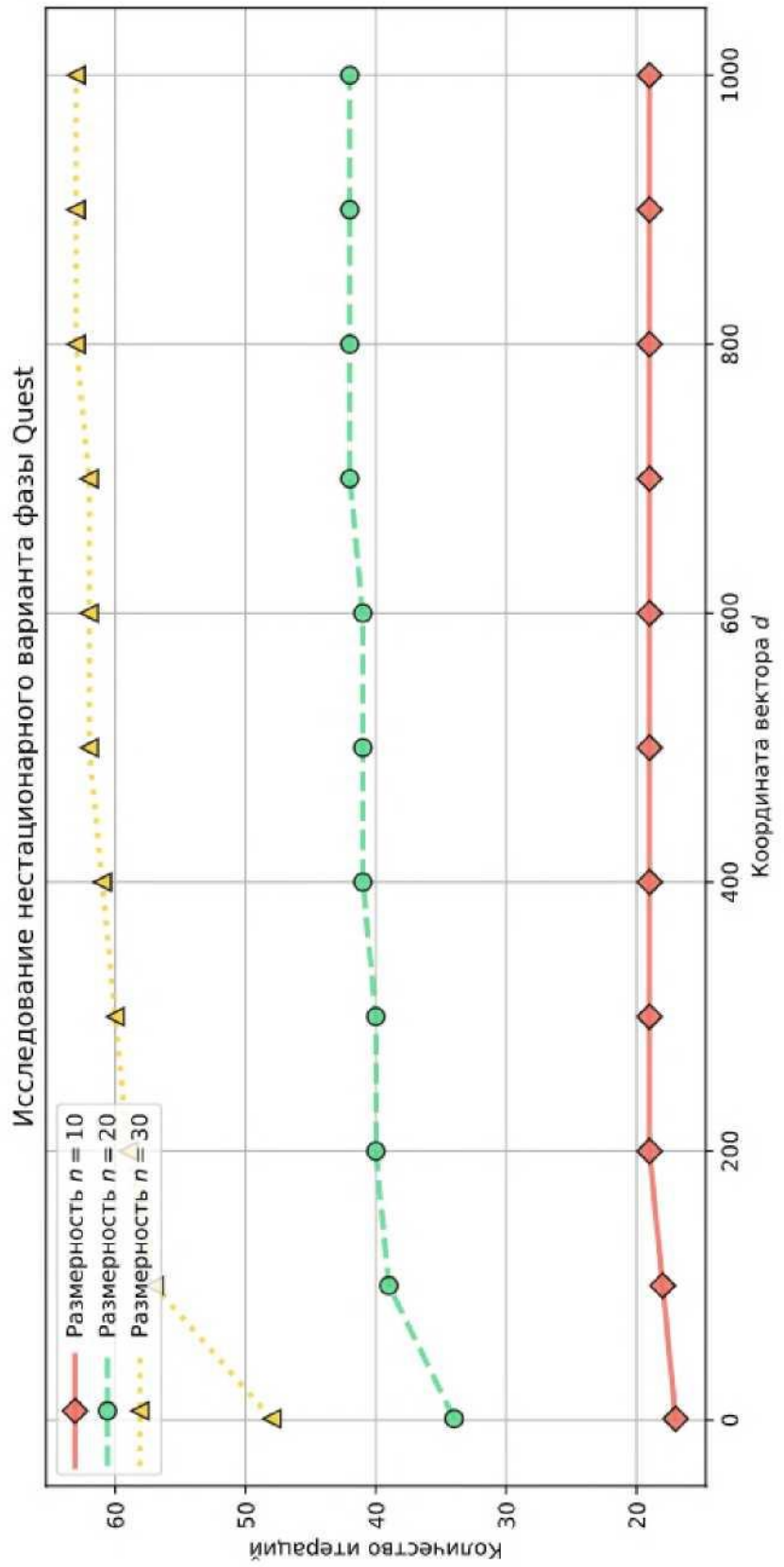

Рисунок 10 - Зависимость количества итераций нестационарного варианта фазы Quest алгоритма NSLP от модуля вектора d параллельного переноса многогранника для размерностей $n = 10,20,30$

Приведённые на рисунке 10 графики иллюстрируют наличие слабой зависимости между количеством итераций фазы Quest от модуля вектора d. С увеличением величины координаты наблюдается незначительное увеличение числа итераций, причём с увеличением размерности эта зависимость становится всё более ярко выраженной. При этом аналогично графикам на рисунке 9 увеличение размерности при фиксированном модуле вектора переноса многогранника способствует ожидаемому увеличению числа итераций алгоритма. Таким образом, чем дальше сдвигается многогранник ограничений задачи в каждый период изменения исходных данных, тем больше итераций фейеровского процесса требуется для решения системы неравенств задачи на фазе Quest. Однако при этом, как было сказано выше, на времени выполнения фазы Quest это сказывается несущественно.

7.3 Исследование стационарного варианта алгоритма NSLP

По аналогии с исследованием фазы Quest при решении стационарных задач, были также проведены исследования стационарного варианта всего алгоритма NSLP. В ходе эксперимента исследовались зависимости времени вычисления и количества итераций фазы Targeting от количества ячеек K по одному измерению формируемой на фазе Targeting крестообразной следящей области. Значение K при этом варьировалось от 11 до 1011 с шагом 100. В качестве начальной точки взята точка начала координат. Порядок точности фейеровского процесса - три. В соответствии с полученными в подразделе 7.1 выводами, коэффициент релаксации для лучшей сходимости был взят равным $Я = 10$ (при этом мы исходим из предположения, что формулы (12) и (25) (для нестационарного и стационарного случаев соответственно) аналогичны в отношении влияния $Я$ на сходимость процесса). Координата вектора параллельного переноса многогранника - единица. Начальное значение длины ребра ячейки следящей области было взято равным 0.01. Коэффициент масштабирования w длины ребра ячейки (применяемого при отсутствии пересечений следящей области с многогранником) был взято равным двум. Точность итерационного процесса

фазы Targeting также была взята равной двум. Такой выбор обусловлен тем, что при меньших значениях (особенно при решении задач больших размерностей) не обеспечивалась сходимость фазы Targeting - следящая область продолжала перемещаться, но критерий останова алгоритма не выполнялся.

Листинг кода эксперимента приведён в Приложении 8. Полученные в результате графики представлены на рисунке 11.

Приведённые на рисунке 11 графики иллюстрируют возрастающий характер зависимости времени вычисления алгоритма NSLP от количества ячеек следящей области по одному измерению: при увеличении K увеличивается и время, за которое алгоритм получает решение задачи. Это можно объяснить тем, что при увеличении количества ячеек следящей области увеличивается и количество псевдопроекций на пересечения этих ячеек с многогранником, которые необходимо вычислять на каждой итерации фазы Targeting. Отметим также, что увеличение размерности задачи при одном и том же значении K способствует закономерному замедлению получения решения (возрастание n, так же как и возрастание K, приводит к увеличению общего числа P ячеек креста, см. формулу (14)). Пренебрегая незначительными колебаниями исследуемой величины, зависимость времени вычисления от K можно считать линейной, при этом коэффициент наклона прямой тем больше, чем больше размерность задачи.

График зависимости от K количества итераций фазы Targeting приводить нецелесообразно в силу того, что во всех экспериментах итерационный процесс позиционирования следящей области завершался за одну итерацию. Это связано с выбором большого значения точности решения задачи (который обусловлен, как было отмечено выше, необходимостью обеспечить сходимость алгоритма как таковую).

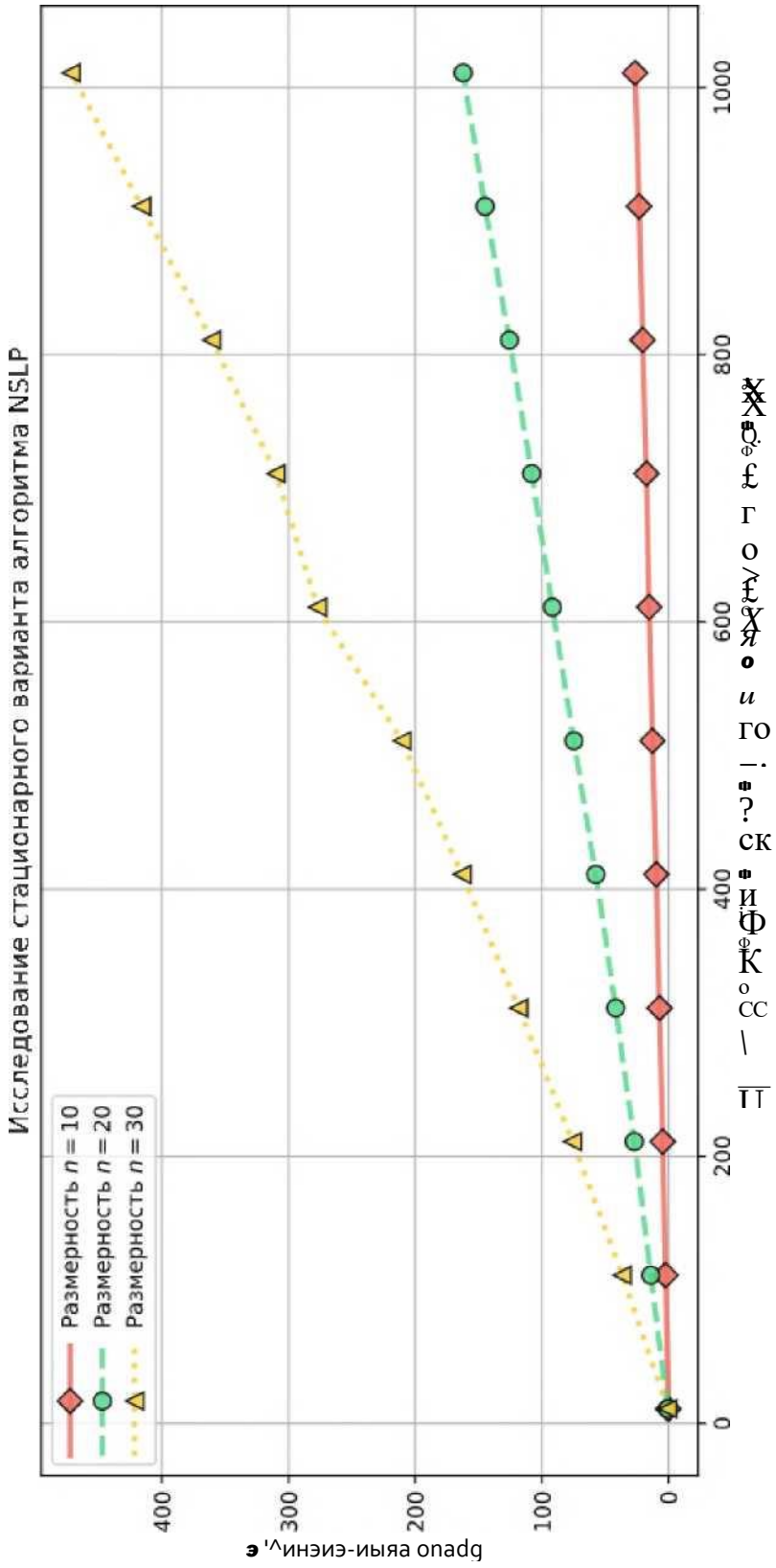

Рисунок 11 - Зависимость времени вычисления стационарного варианта алгоритма NSLP от количества ячеек *K* по одному измерению формируемой на фазе Targeting крестообразной следящей области для размерностей *n* = 10,20,30

7.4 Исследование нестационарного варианта алгоритма NSLP

7.4.1 Исследование зависимости сходимости от количества ячеек следящей области по одному измерению

В ходе эксперимента исследовались зависимости времени вычисления и количества итераций фазы Targeting от количества ячеек K по одному измерению формируемой на фазе Targeting крестообразной следящей области при решении уже нестационарных задач. Значение K при этом, аналогично исследованиям в подразделе 7.3 варьировалось от 11 до 1011 с шагом 100. Период изменения исходных данных взят 1.5 секунды; он был подобран так, чтобы при решении задач любой из исследуемых размерностей алгоритм сходился. Остальные же (неизменяемые в ходе экспериментов) параметры аналогичны зафиксированным в подразделе 7.3.

Листинг кода эксперимента приведён в Приложении 9. Полученные в результате графики представлены на рисунке 12.

Как и при исследовании зависимости сходимости алгоритма на стационарных задачах, результаты экспериментов с нестационарными задачами демонстрируют возрастающий характер времени вычисления при увеличении числа ячеек по одному измерению K. Для одного и того же значения параметра K увеличение n приводит также к увеличению времени получения решения. Однако в данном случае гипотеза о линейной зависимости между K и временем решения подтверждается только для размерностей $n = 10$ и $n = 20$. Для задачи же размерности $n = 30$ при значении $K = 1000$ наблюдается резкий скачок кривой, опровергающий предположение о линейном характере исследуемой зависимости.

Кроме того, как и при исследовании стационарного варианта NSLP в подразделе 7.3, алгоритм по всех экспериментах находил решение за одну итерацию фазы Targeting. Поэтому график количества итераций в данном исследовании не приводится.

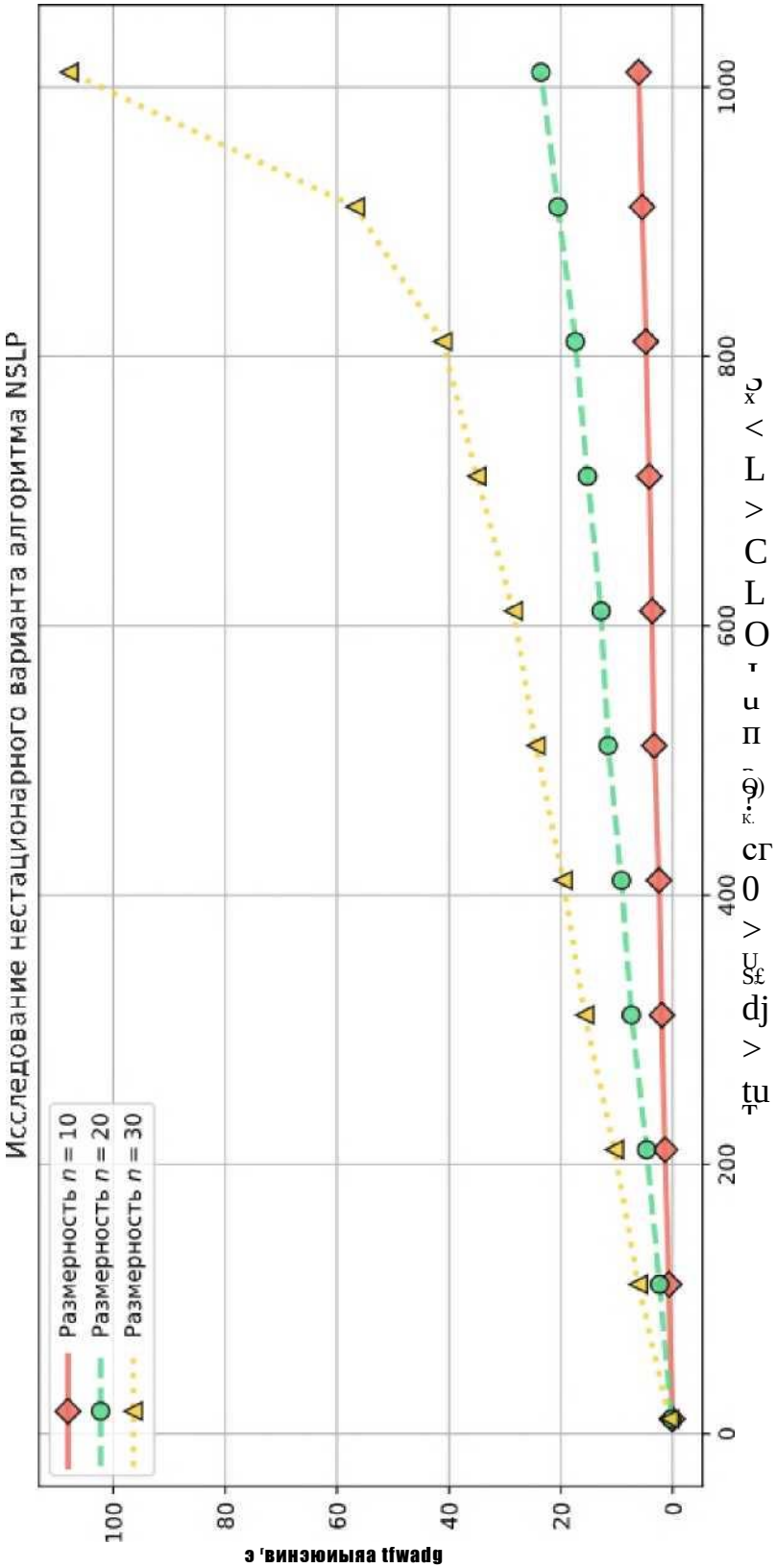

Рисунок 12 - Зависимость времени вычисления нестационарного варианта алгоритма NSLP от количества ячеек *K* по одному измерению формируемой на фазе Targeting крестообразной следящей области для размерностей *n* = 10,20,30

Отметим также и следующее наблюдение: при сравнении графиков на рисунках 11 и 12 видно, что при одном и том же значении n и K время решения стационарной задачи существенно больше времени решения соответствующей нестационарной задачи. Так, решение модельной стационарной задачи с $n = 30$ переменными с помощью алгоритма NSLP с числом ячеек по одному измерению следящей области, равным $K = 500$, занимает около 200 секунд (рисунок 11). Для решения же аналогичной модельной задачи $n = 30$ и при установке параметра алгоритма $K = 500$ требуется на порядок меньше времени: приблизительно 25 секунд (рисунок 12).

7.4.2 Исследование зависимости сходимости от модуля вектора параллельного переноса многогранника

Были проведены эксперименты по исследованию времени вычисления и количества итераций от величины модуля вектора d, на который сдвигается многогранник задачи. Данное исследование аналогично описанному в подразделе 7.2, но в этот раз оно проводилось не только для фазы Quest (то есть, для нахождения решения системы неравенств задачи), а глобально, в масштабе всего алгоритма NSLP, с выполнением как фазы Quest, так и фазы Targeting (то есть, для нахождения решения самой задачи).

Период изменения исходных данных взят 1.5 секунды. В качестве начальной точки взята точка начала координат. Порядок точности фейеровского процесса - три. Коэффициент релаксации был взят равным Я = 10, а количество ячеек следящей области по одному измерению - $K = 11$ (для обеспечения лучшей сходимости в соответствии с результатами экспериментов в подразделе 7.1 и пункте 7.4.1 соответственно). Начальное значение длины ребра ячейки следящей области было взято равным 0.01. Коэффициент масштабирования w длины ребра ячейки (применяемого при отсутствии пересечений следящей области с многогранником) было взято равным двум. Точность итерационного процесса фазы Targeting была также взята равной двум.

Первоначально была предпринята попытка варьирования координат вектора *d* по тому же принципу, что и в подразделе 7.2. Однако уже при решении задачи размерности $n = 30$ и с вектором $d = \{200, 200,200,200,200,200,200,200, 200,200\}$ алгоритм перестал сходиться: на каждой итерации фазы Targeting (а именно, на шаге 2) описанного в подразделе 5.2 алгоритма получалось пустое множество пересечений ячеек с многогранником. Вероятно, это связано с существенным изменением исходных данных, несмотря на относительно низкую скорость этого изменения (полторы секунды, как отмечено выше). Пока проходят вычисления одной итерации фазы Targeting, многогранник слишком далеко «убегает», поэтому ни одна из вычисленных псевдопроекций к этому моменту уже ему не принадлежит.

7.4.3 Исследование по поиску значения масштабирующего коэффициента следящей области

Было выдвинуто предположение, что при достаточно большом значении масштабирующего коэффициента *w* более существенное увеличение размера ячеек креста будет компенсировать большой шаг перемещения многогранника, и алгоритм будет сходиться. Для подтверждения или опровержения данной гипотезы был проведён специальный вычислительный эксперимент.

Значение коэффициента масштабирования *w* следящей области изменялось в геометрической прогрессии с начальным значением, равным двум, и знаменателем два. При каждом значении *w* осуществлялся запуск программы алгоритма (с параметрами, как в пункте 7.4.2 и нестационарной задачей размерности $n = 30$ и вектором d, равным $d = (200, 200,200,200,200,200,200,200, 200,200\}$ - то есть таким, на котором в пункте 7.4.2 алгоритм перестал сходиться). Если при вычислениях на фазе Targeting в течение десяти попыток пересечения следящей области с многогранником *M* отсутствовали, то выполнение текущей программы прерывалось, значение коэффициента *w* увеличивалось в два раза и проводился повторный запуск программы уже с новым значением *w*. Такое поведение

программы обеспечивается благодаря описанному в пункте 5) шестого раздела (посвящённому классу **NSLP**) механизму с использованием исключения **TargetingDoesNotConvergeError**. Листинг кода эксперимента приведён в Приложении 10.

По итогам исследования сходимость алгоритма NSLP при решении нестационарной задачи в 30-мерном пространстве с координатой вектора d, равной 200, так и не была достигнута при последовательном увеличении коэффициента w: при каждом из значений происходил выход из программы. Эксперимент был прекращён, когда значение w достигло значения 64.

Таким образом, можно сделать вывод, что при достаточно существенном переносе (на большой вектор) многогранника ограничений задачи, даже относительно высокий коэффициент масштабирования следящей области не обеспечивает, что область будет иметь пересечение с этим многогранником.

7.4.4 Исследование зависимости сходимости от модуля вектора параллельного переноса многогранника (со скорректированным масштабом изменения координат)

Так как в пункте 7.4.3 было выявлено, что увеличение масштабирующего коэффициента w не влияет на сходимость при больших значениях координат вектора d, было решено сократить масштаб изменения координат вектора d таким образом, чтобы алгоритм сходился даже для размерности $n = 30$. К примеру, при решении модельной задачи размерности $n = 10$ координаты вектора варьировались следующим образом:

$$d = (1,1,1,1,1,1,1,1,1,1\} \rightarrow$$
$$\rightarrow d = \{20, 20,20,20, 20,20,20,20, 20,20\} \rightarrow$$
$$\rightarrow d = (40,40,40,40,40,40,40,40,40,40) \rightarrow$$
$$\rightarrow d = (60, 60,60,60, 60,60,60,60, 60,60) \rightarrow$$
$$\rightarrow d = (80,80,80,80,80,80,80,80,80,80) \rightarrow$$
$$\rightarrow d = (100,100,100,100,100,100,100,100,100,100) \rightarrow$$

-> d = {120,120,120,120,120,120,120,120,120,120} ->

-> d = {140,140,140,140,140,140,140,140,140,140} ->

-> d = (160,160,160,160,160,160,160,160,160,160} ->

-> d = (180,180,180,180,180,180,180,180,180,180}

Аналогичным образом изменялось и значение вектора сдвига для задач размерностей я = 20 и я = 30 - координаты вектора d принимали аналогичные значения, однако число координат, очевидно, соответствовало размерности задачи.

Листинг кода эксперимента приведён в Приложении 11.

Полученные в результате графики представлены на рисунке 13.

Из графиков на рисунке 13 видно, что вне зависимости от значения модуля вектора d время решения задачи фиксированной размерности остаётся примерно одинаковым. При этом чем более высокой размерности решается задача, тем больше времени занимает её решение. Это означает, что для одного и того же я время решения нестационарной задачи линейного программирования с помощью реализованного нами алгоритма NSLP можно (если пренебречь незначительными колебаниями) считать постоянной во времени величиной. Однако, как было сказано выше, дальнейшее увеличение модуля вектора сдвига многогранника приводит к тому, что алгоритм не сходится и бесконечно перемещает и масштабирует следящую область, не находя пересечений ячеек с многогранником.

График зависимости количества итераций от значений координат вектора d также не приводится из соображений нецелесообразности в силу аналогичных приведённым в выводах подраздела 7.3 и пункта 7.4.1 причин.

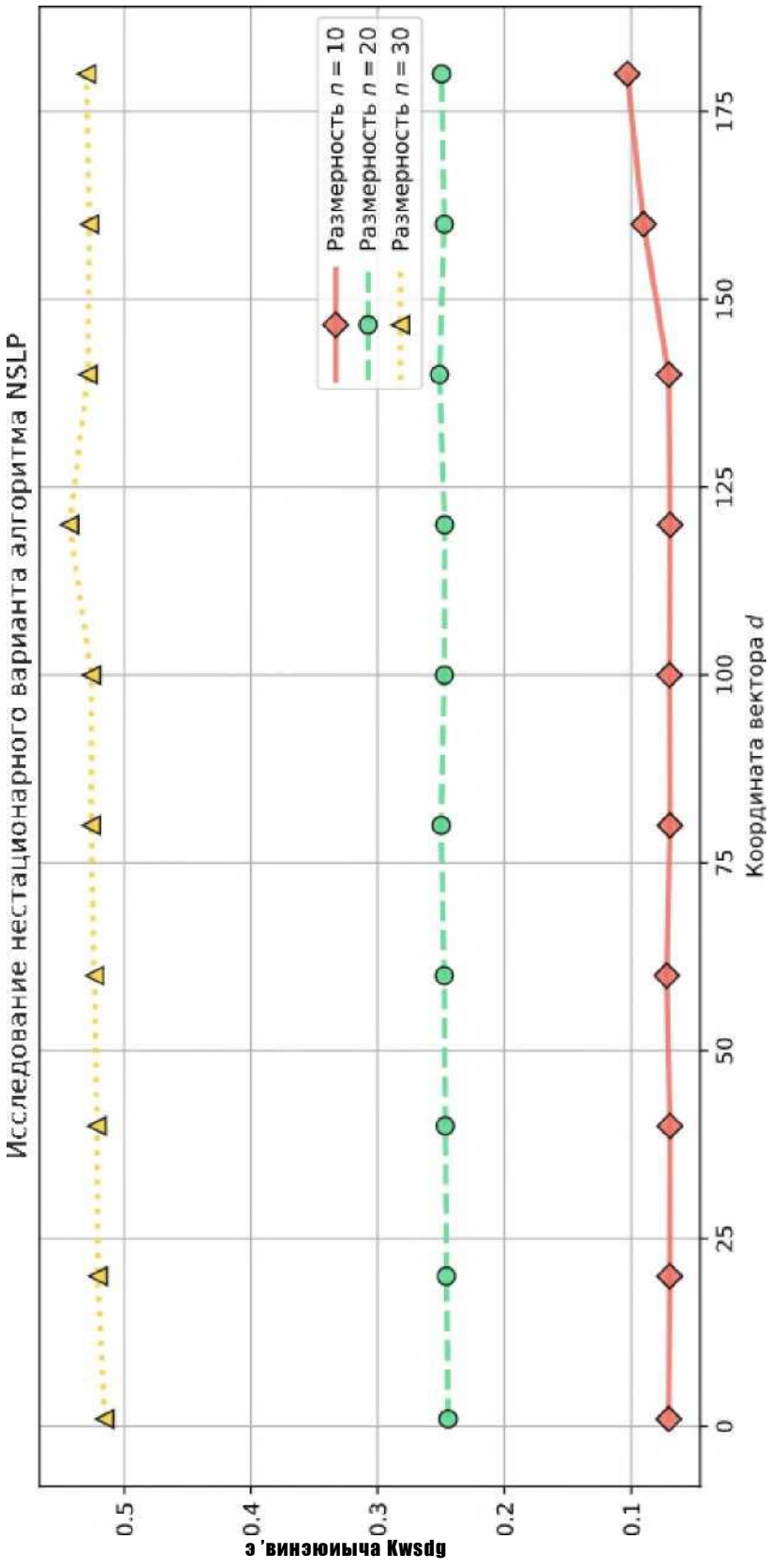

Рисунок 13 - Зависимость времени вычисления нестационарного варианта алгоритма NSLP от модуля вектора d параллельного переноса многогранника для размерностей $n = 10,20,30$

Выводы по разделу семь

Данный раздел посвящён исследованию алгоритма NSLP путём проведения вычислительных экспериментов. Произведено подробное описание экспериментов: предмет исследования, варьируемые параметры (а также пределы их изменения), значения зафиксированных параметров алгоритма. Результаты исследований были визуализированы в виде графиков и проанализированы. По результатам экспериментов был сделан ряд выводов касаемо характера зависимости сходимости программной реализации алгоритма NSLP от различных параметров.

ЗАКЛЮЧЕНИЕ

В рамках выполнения выпускной квалификационной работы был исследован масштабируемый алгоритм решения задачи линейного программирования с изменяющимися исходными данными Non-Stationary Linear Programming (NSLP).

В основе исследования лежит осуществлённый в рамках работы анализ предметной области. Он подразумевает под собой, прежде всего, обзор теоретической базы предложенного алгоритма Non-Stationary Linear Programming, включающей в себя: теорию нахождения решения систем линейных неравенств (второй раздел), теорию фейеровских методов оптимизации (третий раздел) и основ линейного программирования (четвёртый раздел).

Был произведён анализ последних достижений в разработке методов решения нестационарных задач линейного программирования посредством обзора научных публикаций в данной области. После выявления практической ценности разработки алгоритма NSLP, было произведено подробное описание работы его основных фаз (Quest и Targeting) и особенностей его применения (пятый раздел).

После анализа литературных источников и теоретической базы алгоритм NSLP был реализован в виде программы на языке Python. Вкратце описаны архитектура разработанной программы, особенности и инструменты реализации и ограничения её применения (шестой раздел).

Затем с программно реализованным алгоритмом был проведён ряд вычислительных экспериментов, в которых исследовалось влияние различных параметров алгоритма на его работу (седьмой раздел). Из наиболее значимых результатов исследований алгоритма можно выделить следующие:

1) Наиболее быстрая сходимость фейеровского процесса на фазе Quest достигается при относительно больших значениях значении коэффициента релаксации (в наших исследованиях таким значением было Я = 10);

2) Наиболее быстрая сходимость алгоритма NSLP (в сущности, фазы Targeting) достигается при относительно небольшом количестве ячеек следящей

области (в наших исследованиях таким значением было $K = 11$ (число ячеек по одному измерению) при начальной длине ребра ячейки s = 0.01). Такой вывод справедлив как для стационарных, так и для нестационарных задач; замечено при этом, что при сравнении отработки алгоритма при решении нестационарной и стационарных задач при одном и том же значении K, соответствующая нестационарная задачи сходится существенно быстрее;

3) Величина модуля вектора d, на который происходит параллельный перенос многогранника M, не оказывает существенного влияния на скорость сходимости ни фазы Quest, ни всего алгоритма целиком. Однако при этом следует отметить, что при достижении определённого значения модуля d алгоритм NSLP перестаёт сходиться и «застревает» в бесконечном итерационном процессе на фазе Targeting (в наших исследованиях такая ситуация наступала при координатах вектора d, равных двести каждая и периоде изменения исходных данных, равном полутора секундам). Дополнительные исследования показали, что увеличение значения параметра w (коэффициента масштабирования следящей области в случае отсутствия пересечений её ячеек с многогранником) вплоть до 64 всё равно не обеспечивает сходимость алгоритма при больших значениях вектора d.

Направление дальнейших исследований может быть связано с расширением функциональности алгоритма: например, разработка метода, обеспечивающего нахождение решения нестационарных задач линейного программирования, нестационарность которых имеет более сложный вид, чем параллельный перенос многогранника на фиксированный вектор с постоянной частотой. Кроме того, существующая версия алгоритма реализует решение только модельных задач; соответственно, в будущих исследованиях необходимо провести работу по совершенствованию алгоритма для обеспечения решения более широкого класса задач.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1 Акулич, И.Л. Математическое программирование в примерах и задачах: учебное пособие / И.Л. Акулич - 2-е изд., стер. - СПб.: «Лань», 2011. - 352 с.

2 Ананченко, И.В. Торговые роботы и управление в хаотических средах: обзор и критический анализ / И.В. Ананченко, А.А. Мусаев. // Труды СПИИРАН. - 2014. - № 3(34). - С. 178-203.

3 Банди, Б. Основы линейного программирования / Б. Банди - М.: Радио и связь, 1989. - 176 с.

4 Бобков, А.В. Системы распознавания образов: учебное пособие / А.В. Бобков. - М.: Изд-во МГТУ им Н.Э. Баумана, 2018 - 187 с.

5 Вапник, В.И. Теория распознавания образов (статистические проблемы обучения) / В.Н. Вапник, А.Я. Червоненкис. - М.: «Наука», Главная редакция физико-математической литературы, 1974. - 416 с.

6 Васёв, Д.Г. Алгоритм решения задачи линейного программирования в условиях изменяющихся исходных данных / Д.Г. Васёв, И.М. Соколинская // Тезисы докладов III Всероссийской конференции «Образование магистров: проблемы и перспективы развития» (21-27 ноября 2019 г., г. Челябинск). — 2019. — С. 18-24.

7 Вентцель, Е.С. Исследование операций: задачи, принципы, методология: учебное пособие / Е.С. Вентцель. - 5-е изд., стер. - М.: КНОРУС, 2013, - 192 с.

8 Дж. Ту, Р., Еонсалес. Принципы распознавания образов / Еонсалес Дж. Ту, Р. -М.: «Мир», 1978. -414 с.

9 Дышаев, М.М. Представление торговых сигналов на основе адаптивной скользящей средней Кауфмана в виде системы линейных неравенств / М.М. Дышаев, И.М. Соколинская // Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика. - 2013. - Т. 2, №4. - С. 103-108.

10 Ерёмин, И.И. Теория линейной оптимизации / И.И. Ерёмин. - Екатеринбург: «Екатеринбург», 1999. -312 с.

11 Ерёмин, И.И. Фейеровские методы для задач выпуклой и линейной оптимизации / И.И. Ерёмин. - Челябинск: Изд-во ЮУрЕУ, 2009 - 200 с.

12 Карпелевич, Ф. И. Элементы линейной алгебры и линейного программирования / Ф.И. Карпелевич, Л.Е. Садовский - М.: «Наука», 1967. - 276 с.

13 Линейное программирование в современных задачах оптимизации: учебное пособие / Ю.В. Бородакий, А.М. Загребаев, И.А, Крицын и др. - М.: МИФИ, 2008. - 188 с.

14 Линейные неравенства и смежные вопросы: сборник статей / под ред. Е.У. Куна, А.У. Таккера - М.: «Издательство иностранной литературы», 1959 - 471 с.

15 Лунгу, К.Н. Линейное программирование. Руководство к решению задач / К.Н. Лунгу. -М.: ФИЗМАЛИТ, 2005. - 128 с.

16 Методы оптимизации / Р. Еабасов, Ф.М. Кириллова, В.В. Альсевич и др. - Минск: «Четыре четверти», 2011. - 472 с.

17 Мунасыпов, И.А. Линейное программирование: учебное пособие / Оренбург: ООО «Агентство «Пресса», 2015. - 122 с.

18 Панин, С.Д. Теория принятия решений и распознавание образов / С.Д. Панин. - М.: Изд-во МЕТУ им. Н.Э. Баумана, 2017. - 239 с.

19 Пантелеев, А.В. Методы оптимизации в примерах и задачах: учебное пособие / А.В. Пантелеев, Т.А. Летова. - 2-е изд., исправл. - М.: Высш. шк., 2005. - 544 с.

20 Потапов, А.С. Распознавание образов и машинное восприятие: Общий подход на основе принципа минимальной длины описания / А.С. Потапов. - СПб.: «Политехника», 2007. - 548 с.

21 Раденков, С.П. Автоматизированные торговые системы и их инсталляция в рыночную среду (часть 1) / С.П. Раденков, С.С. Еакрюшин, В.В. Соколянский. // Вопросы экономических наук. - 2015. - № 6 (76). - С. 70-74.

22 Саати, Т. Л. Математические методы исследования операций / Т.Л. Саати. - М.: «Воениздат», 1963. -420 с.

23 Соколинская, И.М. Модифицированный следящий алгоритм для решения нестационарных задач линейного программирования на кластерных вычислительных системах с многоядерными ускорителями / И.М. Соколинская, Л.Б. Соколинский // Суперкомпьютерные дни в России: Труды международной конференции (26-27 сентября 2016 г., г. Москва). - 2016. - С. 294-306.

24 Соколинская, И.М. О решении задачи линейного программирования в эпоху больших данных / И.М. Соколинская, Л.Б. Соколинский // Параллельные вычислительные технологии (ПаВТ 2017). - 2017. - Том Апрель. - С. 471-484.

25 Факторный, дискриминантный и кластерный анализ / Дж.-О. Ким, Ч.У. Мьюллер, У.Р. Клекка и др. - М.: «Финансы и статистика», 1989. - 215 с.

26 Черников, С.И. Линейные неравенства / С.И. Черников - М.: «Наука», 1986. -488 с.

27 Юдин, Д.Б. Задачи и методы линейного программирования / Д.Б. Юдин, Е.Г. Гольштейн. -М.: «Советское радио», 1961. -494 с.

28 Klee, V. How good is the simplex algorithm? / V. Klee, G.J. Minty. // Inequalities III (Proceedings of the Third Symposium on Inequalities held at the University of California, Los Angeles, Calif., September 1-9, 1969, dedicated to the memory of Theodore S. Motzkin). New York-London: Academic Press. - 1972. - P. 159-175.

29 Sodhi M.S. LP modeling for asset-liability management: A survey of choices and simplifications / M.S. Sodhi // Operations Research. - 2005. - V. 53, No. 2. - P. 181196.

30 Sokolinskaya I.M, Sokolinsky L.B. Implementation of Parallel Pursuit Algorithm for Solving Unstable Linear Programming Problems. Bulletin of the South Ural State University. Series: Computational Mathematics and Software Engineering. - 2016. - V. 5. - No. 2. -P. 15-29 (in Russian) DOI: 10.14529/cmsel60202.

31 Sokolinskaya I.M, Sokolinsky L.B. Revised Pursuit Algorithm for Solving Non-Stationary Linear Programming Problems on Modem Computing Clusters with Manycore Accelerators // Proceedings of the RuSCDays 2016. Series: Communications in Computer and Information Science. - 2016. - V.5. - P. 212-223. DOI: 10.1007/978- 3-319-55669-7 17

32 Sokolinskaya I.M, Sokolinsky L.B. Solving unstable linear programming problems of high dimension on cluster computing systems // Proceedings of the 1st Russian Conference on Supercomputing - Supercomputing Days (RuSCDays 2015). Moscow, Russian Federation, September 28-29, 2015. CEUR Workshop Proceedings. - 2015. -V. 1482. -P. 420-427.

33 Sokolinskaya, I.M Scalable Algorithm for Non-stationary Linear Programming Problems Solving / I.M. Sokolinskaia //IEEE. - 2017. - P. 49-53.

ПРИЛОЖЕНИЯ

Приложение 1

Листинг модуля mode 1_pnoblem. py

```
from threading import Timer
import sys

import numpy as np



class Counter:

    def __ init___ (
                self j
                period



        :param period: период изменения исходных данных в секундах.
        it и и
        self.period = period self.t = 0

    def stop(self):
        и и и
        Останавливает счётчик.
        :return:
        и и и

        if hasattr(self, 'timer'): self.timer.cancel() else:


            raise Exception('Таймер не инициализирован.') def

    start(self):
        и и и
        Запускает счётчик.
        :return:
        и и и
        self.timer = Timer( self.period, self.start
        )
        self.timer.start() self.t += 1
        # Выводим текущее значение счётчика t на экран sys.stdout.write(f'\rt =
        {str(self.t)}')



class ModelProblem: def

    _form_A(self):
        и и и


        :return: матрица коэффициентов задачи.
        и и и
```

```
        A = np.vstack((
            np.diagflat(np.ones(self.n)),          # n строк, где =< 200
            np.array(np.ones(self.n)),          # 1 строка, где =< 200 * (n - 1) + 100
            np.array(-np.ones(self.n)),          # 1 строка, где => 100
            np.diagflat(-np.ones(self.n))          # n строк, где => 0


        return A

    def _form_b(self):



        :return: столбец свободных членов задачи.
        ii и и
        b = np.vstack((
            np.full((self.n, 1), 200), # n строк, где =< 200
            np.array([200 * (self.n - 1) + 100]), # 1 строка, где =< 200 * (n - 1) +
100
            np.array([-100]), # 1 строка, где => 100 np.full((self.n,
            1), 0) # n строк, где => 0



        return b

    def _form_c(self):



        :return: вектор коэффициентов целевой функции задачи.
        и и и
        c = np.hstack([
            np.array([2 * np.ones(self.n - 1)]),
            np.array([np.ones(1)])
        ])
        return c

    def __ init___ (
            self,
            in
            d,
            non_stationary,
            period
    ):
        и и и


        :param n: размерность задачи.
        :param d: координаты вектора сдвига многогранника.
        :param non_stationary: флаг стационарности/нестационарности задачи. :param
        period: период изменения исходных данных в секундах.
        и и и
        assert len(d) == n

        self.n = n self.d = d
        self.A = self._form_A() self.b
        =  self._form_b()  self.c  =
        self._form_c()                if
        non_stationary:
```

```
            #   Если задача нестационарна,
            #   то должны быть переданы координаты вектора сдвига многогранника, assert d
            is not None
            #   Инициализируем счётчик как атрибут
            self.counter = Counter(
                  period=period
            )

def _form_variative_A(self):


      :return: вариативную часть матрицы коэффициентов.
      ii и и
      return np.vstack(
            (
                  np.diagflat(-np.ones(self.n)),
                  np.diagflat(np.ones(self.n))
            )
      )

@staticmethod
def _form_variative_b(cell_zero_vertex_coordinates, s):
      и и и


      :param cell_zero_vertex_coordinates: координаты нулевой вершины ячейки, псевдопроекцию на
      пересечение с которой мы вычисляем.
      :param s: длина ребра ячейки.
      :return: вариативную часть столбца свободных членов.
      и и и
      return np.vstack(
            (
                  -cell_zero_vertex_coordinates,
                  cell_zero_vertex_coordinates + s
            )
      )

def _extend_A(self, variative_A):
      и и и
      Соединяет инвариативную и вариативную части матрицы коэффициентов. :param
      variative_A:
      :return:
      и и и
      return np.vstack(
            (
                  self.A, variative_A
            )
      )

def _extend_b(self, variative_b):
      и и и
      Соединяет инвариативную и вариативную части столбца свободных членов. :param
      variative_b:
      :return:
      и и и
      return np.vstack(
            (
```

```
                self.b,
                variative_b
            )
        )

def extend_coefficient_matrices(self, cell_zero_vertex_coordinates, s):
    ii и и
    Формирует расширенные матрицу коэффициентов и столбец свободных членов. :param
    cell_zero_vertex_coordinates:
    :param s:
    :return:
    и и и

    assert self.n == len(cell_zero_vertex_coordinates) variative_A = self._form_variative_A()
    variative_b = self._form_variative_b(cell_zero_vertex_coordinates, s)

    extended_A = self._extend_A(variative_A) extended_b = self._extend_b(variative_b)

    self.extended_A = extended_A self.extended b = extended b
```

Приложение 2

Листинг модуля **quest. py**

```
import numpy as np
import datetime


class Quest:

    def ___init __ (
            self,
            problem,
            1
    ):
        ii и и


        :param problem: экземпляр класса ModelProblem.
        :param 1: коэффициент релаксации.
        и и и
        self.problem = problem self.1 = 1
        self.pseudoprojections = list()

    def _fejer_mapping(
            self,
            x,
            A,
            b,
            d,
            1,
            coeffs_type
    ):
        и и и
        Вычисляет фейеровское отображение.
        :param x: точка, из которой осуществляется отображение.
        :param A: матрица коэффициентов.
        :param b: столбец свободных членов.
        :param d: вектор сдвига многогранника.
        :param 1: коэффициент релаксации.
        :param coeffs_type: тип коэффициентов (расширенные/обычные матрицы).
        :return:
        и и и
        x = np.array(x).reshape(-1, 1) d = np.array(d).reshape(-1, 1)

        m = A.shape[0] # количество неравенств системы _sum = 0

        for i in range(m): a = A[i, :]

            #   Вычисляем сумму, которая вычитается из точки,
            #   для которой считается псевдопроекция

            if hasattr(self.problem, 'counter') and coeffs_type != 'extended_' new_point = x -

            self.problem.counter.t * d else:
```

```
                new_point = x

            potential_ch = (np.dot(a, new_point) - b[i])[0]

            ch = max(potential_ch, 0) zn = np.linalg.norm(a) ** 2 _sum +=
            np.dot((ch / zn), a)

        _sum = _sum.reshape(-1, 1)
        #   Вычитаем из точки получившуюся сумму с коэффициентом 1 / m phi =
        x - (1 / m) * _sum

        return phi

def _append_pseudoprojection(self, coeffs_type):
    ''' '''
        Вычисляет новую псевдопроекцию в ходе фейеровского процесса.
        :param coeffs_type: тип коэффициентов (расширенные/обычные матрицы. :return:
        ''' '''
        #   Удостоверимся, что список псевдопроекций не пуст assert
        self.pseudoprojections

        A = getattr(self.problem, f'{coeffs_type}A') b = getattr(self.problem, f'{coeffs_type}b')

        self.pseudoprojections.append( self._fejer_mapping(
                x=self.pseudoprojections[-1],
                A=A,
                b=b,
                d=self.problem.d, l=self.1,
                coeffs_type=coeffs_type
            )
        )

def _stopping_criterion(self, precision_order):
    ''' '''
        Реализует критерий останова фейеровского процесса.
        :param precision_order: порядок точности.
        :return:
        ''' '''
        #   Пересчёт порядка точности в число eps eps = 10 ** (-precision_order)

        return np.linalg.norm(
            self.pseudoprojections[-1] - self.pseudoprojections[-2]
        ) <= eps

def find_solution_of_inequalities_system( self,
            X0,
            precision_order, coeffs_type='', timeit=False
):
```

```
"""

:рагат x0: координаты начальной точки.
:param precision_order: порядок точности.
:рагат coeffs_type: тип коэффициентов (расширенные/обычные матрицы). :param
timeit: флаг фиксации времени вычисления и кол-ва итераций. Нужно, для
проведения вычислительных экспериментов.
:return: решение системы неравенств.
"""

assert 1еп(x0) == self.problem.n assert coeffs_type in ['', 'extended_']

#   Запускаем счётчик.
#   Если тип коэффициентов "расширенный", то не запускаем,
#   т.к. в этом случае вычисляется псевдопроекция на пересечение
#   многогранника с ячейкой следящей области.
if hasattr(self.problem, 'counter') and coeffs_type != 'extended_': self.problem.counter.start()

if timeit:

    #   Фиксация начала вычислений start_time
    = datetime.datetime.now()

#   Инициализируем список псевдопроекций начальной точкой self.pseudoprojections.append(x0)
#   Добавляем первую псевдопроекцию
self._append_pseudoprojection(coeffs_type)

if timeit:

    #   Инициализируем счётчик итераций фейеровского процесса
    iter_num = 1

while not self._stopping_criterion(precision_order):

    self._append_pseudoprojection(coeffs_type) if timeit:
        # Инкрементируем счётчик итераций
        iter_num += 1

if timeit:

    #   Фиксируем время окончания вычислений
    stop_time = datetime.datetime.now()
    #   Рассчитываем общее время решений
    eval_time = stop_time - start_time

if not timeit:

    #   При timeit = False, возвращаем только решение return
    self.pseudoprojections[-1]

else:
```

```
#   При timeit = True, возвращаем решение
#   а также время вычислений и количество итераций return
self.pseudoprojections[-1], eval_time, iter_num
```

Приложение 3

Листинг модуля cross. py

```
import numpy as np


class Cross:

    @staticmethod def sign(x):



        :param x:
        :return: знак числа.
        ii и и
        if x < 0:

            return -1


        elif x == 0:

            return 0

        elif x > 0:

            return 1
    @staticmethod def calc_P(n,
    K):



        :param n: размерность следящей области.
        :param K: кол-во ячеек по одному измерению.
        :return: общее кол-во ячеек следящей области.
        и и и
        return п * (K - 1) + 1

    @staticmethod
    def calc_chi(alpha. K):
        и и и


        :param alpha: порядковый номер ячейки.
        :param K: кол-во ячеек по одному измерению.
        :return: измерение chi ячейки для маркерной нумерации.
        и и и
        if alpha == 0:

            return 0 else:

            return (alpha - 1) // (K - 1)

    @staticmethod
    def calc_eta(alpha, K):
        и и и



        :param alpha: порядковый номер ячейки.
        :param K: кол-во ячеек по одному измерению.
        :return: индекс eta для маркерной нумерации.
```

```
        it и и
        if alpha == 0: return 0
        elif (alpha - 1) %(K - 1) < (K - 1) / 2:
            return (alpha - 1) %((K - 1) / 2) - ((K - 1) / 2)
        elif (alpha - 1) %(K - 1) >= (K - 1) / 2: return (alpha - 1) % ((K - 1) / 2) + 1
    @staticmethod
    def calc_alpha(chi, eta, K):
        :param chi: измерение ячейки (маркерная нумерация). :param eta: индекс ячейки
        (маркерная нумерация).
        :param K: кол-во ячеек по одному измерению.
        :return: порядковый номер ячейки (линейная нумерация).
        и и и
        if eta == 0: return 0 elif eta < 0:
            return eta + ((K - 1) / 2) + chi * (K - 1) + 1 elif eta > 0:
            return eta + ((K - 1) / 2) + chi * (K - 1)
    @staticmethod def is_odd(x):
        и и и
        :param x:
        :return: True, если число x нечётное. False - в противном случае.
        и и и
        if x % 2 != 0: return True else:
            return False @staticmethod
    def calc_zero_vertex_coords(
            ë,
            chi,
            eta,
            s
    ):
        и и и
        Вычисляет координаты нулевой вершины ячейки (chi, eta).
```

```
        :param g: координаты нулевой вершины центральной ячейки.
        :param chi: измерение ячейки.
        :param eta: индекс ячейки.
        :param s: длина ребра ячейки.
        :return:
        it и и

        zero_vertex_coords = list() for j, _ in enumerate(g):

            assert g[j].shape == (1,) if j == chi:

                zero_vertex_coord = g[chi] + eta * s else:

                zero_vertex_coord = g[j]

zero_vertex_coords.append(zero_vertex_coord) zero_vertex_coords

= np.array(zero_vertex_coords) return zero_vertex_coords def

_form_points(self):
        и и и
        Формирует структуру ячеек следящей области.
        :return:
        и и и
        points = list()

        for alpha in range(self.P):

            #   Рассчитываем измерение и индекс ячейки для маркерной нумерации, chi =
            self.calc_chi(alpha, self.K)
            eta = self.calc_eta(alpha, self.K)

            #   Удостоверяемся, что расчётное alpha
            #   соответствует ожидаемому
            #   (проверяем обратную совместимость
            #   маркерной и линейной нумераций). calculated_alpha =
            self.calc_alpha(chi, eta, self.K) assert alpha == calculated_alpha

            zero_vertex_coords = self.calc_zero_vertex_coords(
                self.central_cell_coords, chi, eta, self.s
            )

            points.append(
                {
                    'alpha': alpha,
                    'chi': chi,
                    'eta': eta,
```

```
                    'zero vertex coordinates': zero_vertex_coords
                }
            )

        return points

def ___init__ (
        self j
        central_cell_coords,
        14
        K,
        s
):


    :param central_cell_coords: координаты нулевой
    вершины центральной ячейки.
    :param п: размерность следящей области.
    :рагаш К: кол-во ячеек по одному измерению. :param s:
    длина ребра ячейки.


    # Кол-во ячеек по одному измерению должно быть нечётным assert
    self.is_odd(K)

    self.central_cell_coords = central_cell_coords self.n = n self.K = K self.s
    = s

    self.P = self.calc_P(self.n, self.K) self.points = self._form_points()
```

Приложение 4

Листинг модуля nslp. py

```
import sys

import numpy as np
import datetime

from cross import Cross from
quest import Quest from
model_problem import
ModelProblem


# Исключение для прерывания программы при слишком долгом поиске пересечений class
TargetingDoesNotConvergeError(Exception): pass



class NSLP:

    def _form_cross(self):
        ii и и


        :return: Экземпляр класса Cross
        (структуру, содержащую крестообразную следящую область).
        и и и
        return Cross(
            central_cell_coords=self.current_central_cell_coords, n=self.n_dim,
            K=self.K, s=self.s
        )

    def _solve_inequalities_system( self,
            X0,
            coeffs_type=''
    ):
        и и и


        :param x0: начальная (из которой осуществляется псевдопроектирование). :param
        coeffs_type: тип коэффициентов (расширенные или обычные матрицы А и Ь).
        :return: точку псевдопроекции, полученную с помощью фейеровских отображений.
        и и и
        return Quest(
            problem=self.problem, l=self.1
        ).find_solution_of_inequalities_system(
            X0=X0,
            precision_order=self.fejer_map_precision_order,
            coeffs_type=coeffs_type
        )

    def _catch_polyhedron_initially(self, x0):
        и и и
        Реализует фазу Quest. Находит решение системы неравенств задачи.
        :param x0: начальная точка.
```

```
        :return:
        ii и и
        self.current_central_cell_coords = self._solve_inequalities_system( X0=X0
        )
        self.cross = self._form_cross()

def ___init __ (
            self, n_dim,
            targeting_eps,
            1,
            X0,
            fejer-map-precision-Order,
            K,
            initial_s, W,
            d=None,
            non_stationary=False, period=None
):
```

```
        :param n_dim: размерность задачи.
        :param targeting_eps: точность вычислений на фазе Targeting.
        :param 1: коэффициент релаксации.
        :param x0: координаты начальной точки для фазы Quest.
        :param fejer_map_precision_order: порядок точности вычислений на фазе Quest.
        :param K: кол-во ячеек следящей области по одному измерению.
        :param initial_s: начальное значение длины ребра ячейки.
        :param w: коэффициент масштабирования длины ребра ячейки при пустом
        пересечении многогранника и следящей области.
        :param d: координаты вектора,
        на который осуществляется параллельный перенос многогранника.
        :param non_stationary: флаг стационарности/нестационарности задачи. :param
        period: период изменения исходных данных в секундах.
        и и и
        self.n_dim = n_dim
        self.targeting_eps = targeting_eps
        self.d = d
        self.1 = 1
        self.x0 = x0
        self.fejer_map_precision_order = fejer_map_precision_order
        self.K = K
        self.s = initial_s
        self.w = w


        if non_stationary:
            # Если задача нестационарна, то должен быть задан вектор сдвига, assert self.d
            is not None

        self.problem = ModelProblem( n=n_dim, d=d,
            non_stationary=non_stationary,
            period=period
```

```
        )

    def _is_pseudoprojection_in_polyhedron(self, pseudoprojection):
        """
        Определяет, принадлежит ли точка многограннику.
        Принадлежность точки определяется путём подстановки в неравенство. :param
        pseudoprojection: точка псевдопроекции на пересечение ячейки с многогранником.
        :return: True/False
        """
        left = np.round(np.dot(
            self.problem.A, pseudoprojection),
            self.fejer_map_precision_order - 2
        )
        #   Вид правой части зависит от того, стационарна ли задача.
        #   Стационарность задачи определяется путём определения,
        #   есть ли у неё атрибут 'counter'.

        #   Если задача стационарна ноебходимо
        #   подставить точку в неравенство вида Ax <= b if not hasattr(self.problem,
        'counter'):

            right = self.problem.b

        #   Если задача стационарна ноебходимо
        #   подставить точку в неравенство вида Ax <= b + A*t*d else:

            right = self.problem.b + np.dot(
                self.problem.A * self.problem.counter.t,
                self.problem.d
            )

        right = np.round( right,
            self.fejer_map_precision_order - 2
        )

        return all(left <= right)

    def _get_pseudoprojection_on_single_intersection_data(self, point):
        """
        Вычисляет псевдопроекцию на пересечение ячейки с многогранником. :param
        point: ячейка следящей облачти.
        :return: словарь, содержащий координаты псевдопроекции,
        флаг принадлежности псевдопроекции многограннику и данные ячейки.
        """
        zero_vertex_coordinates = point['zero vertex coordinates']

        #   Формируем расширенные матрицы A' и Ь' self.problem.extend_coeffic
        ient_matrices(
            zero_vertex_coordinates, self.cross.s
        )

        #   Получаем решение расширенной системы неравенств pseudoprojection =
        self._solve_inequalities_system(
            x0=zero_vertex_coordinates,
```

```
            coeffs_type='extended_'
        )
        # Определеяем, принадлежит ли псевдопроекция многограннику
        in_polyhedron = \
            self._is_pseudoprojection_in_polyhedron(pseudoprojection)

        pseudoprojection_data = {
            'coordinates': pseudoprojection,
            'in polyhedron': in_polyhedron,
            'cross point': point
        }

        return pseudoprojection_data def

_get_pseudoprojections_on_intersections(self):


        :return: псевдопроекции на пересечения ячеек с многогранником.
        it и и
        pseudoprojections_data = list(map(
            self._get_pseudoprojection_on_single_intersection_data, self.cross.points
        ))
        #   Отсеиваем не принадлежащие многограннику псевдопроекции pseudoprojections_data = [x for x
        in pseudoprojections_data if
                                            x['in polyhedron']]

        return pseudoprojections_data

def _handle_calculation_of_pseudoprojections_on_intersections(self):
        и и и
        Обрабатывает вычисление псевдопроекций на пересечения ячеек с
        многогранником.
        :return: структуру pseudoprojections_data.
        и и и
        #   Инициализируем счётчик попыток получения непустого множества
        #   пересечений следящей области с многогранником
        n_attempts = 0

        while True:

            n_attempts += 1

            #   На время вычислений псевдопроекций
            #   на пересечения прерываем счётчик.
            #   Если этого не делать, многогранник может сдвинуться
            #   в процессе вычислений и по некоторым измерениям будут
            #   отсутствовать псевдопроекции.
            if hasattr(self.problem, 'counter'):

                self.problem.counter.stop()

            pseudoprojections_data = \

                self._get_pseudoprojections_on_intersections()

            if hasattr(self.problem, 'counter'):

                self.problem.counter.start()
```

```
            if pseudoprojections_data:

                return pseudoprojections_data else:

                if n_attempts == 10:
                                        if hasattr(self.problem, 'counter'):
                                            self.problem.counter.stop()

                                    raise TargetingDoesNotConvergeError

                #   Масштибирование ячеек в случае отсутствия пересечений print('Hn
                одна из ячеек не пересекается с многогранником.')

                #   Увеличиваем длину ребра ячейки в w раз self.s *= self.w
                #   Обновляем следящую область self.cross = self._form_cross()

@staticmethod
                    def _get_pseudoprojection_data_by_dim(pseudoprojections_data, dim):
```

```
        :param pseudoprojections_data: данные по псевдопроекциям. :param
        dim: измерение.
        :return: выборка псевдопроекций по измерению dim.


        return [x for x in pseudoprojections_data if x['cross point']['chi'] == dim]

def _subs_point_into_target_func(self, point_coords):
        ii и и
        Подставляет точку в целевую функцию.
        :param point_coords: координаты точки.
        :return: результат подстановки точки в целевую функцию.



        return np.dot(self.problem.c, point_coords)

@staticmethod
def _get_best_point_data_for_dim(dim_pseudoprojections_data):
        и и и
        Выбирает точку, в которой достигается максимальное значение
        целевой функции.
        :param dim_pseudoprojections_data: псевдопроекции по измерению. :return:
        и и и
        return max(
            dim_pseudoprojections_data, key=lambda x:
            x['target function value']
        )

def _calc_target_func_vals_for_pseudoprojections( self,
```

```
        pseudoprojections_data
):
    ii и и
    Вычисляет значения целевой функции в точках псевдопроекции.
    :param pseudoprojections_data:
    :return:
    и и и
    for idXj pseudoprojection_data in enumerate(pseudoprojections_data):

        pseudoprojection_coords = pseudoprojection_data['coordinates']
        pseudoprojections_data[idx]['target function value'] = \
            self._subs_point_into_target_func(pseudoprojection_coords)

    return pseudoprojections_data

def _get_best_points_data(self, pseudoprojections_data):



    :param pseudoprojections_data:
    :return: псевдопроекции с максимальными значениями целевой функции по
    каждому измерению.


    best_points_data = list()

    for dim in range(self.problem.n):

        # Выделяем псевдопроекции по заданному измерению curr_dim_pseudoprojections_data = \
            self._get_pseudoprojection_data_by_dim( pseudoprojections_data, dim
            )

        best_points_data.append(
            self._get_best_point_data_for_dim( curr_dim_pseudoprojections_data
            )
        )

    return best_points_data @staticmethod
def _calc_mass_center(best_points_data):









    :param best_points_data: список псевдопроекций
    с максимальными значениями целевой функции по каждому из измерений. :return:
    центр масс точек.


    #   Центр масс рассчитывается как среднее арифметическое
    #   соответствующих координат
    return np.hstack(
        [point_data['coordinates'] for point_data in best_points_data] ).mean(axis=0).reshape( -1, 1)

def _get_temp_solution(self):
```

```
        """

        :return: Временное решение задачи.
        """
        #   Получаем псевдопроекции
        pseudoprojections_data = \
            self._handle_calculation_of_pseudoprojections_on_intersections()

        #   Вычисляем для каждой из псевдопроекций значение целевой функции pseudoprojections_data
        = \
            self._calc_target_func_vals_for_pseudoprojections( pseudoprojections_data
            )
        #   Выбираем для каждого измерения псевдопроекцию с максимальным
        #   значением целевой функции
        best_points_data = self._get_best_points_data(pseudoprojections_data)

        #   Количество "лучших" точек должно совпадать с размерностью задачи assert
        len(best_points_data) == self.problem.n

        #   Решение вычисляется как центр масс "лучших" точек
        temp_solution = self._calc_mass_center(best_points_data)

        return temp_solution

    @staticmethod
    def _calc_dist(first_point, second_point):
        """
        Вычисляет евклидово расстояние между точками.
        :param first_point:
        :param second_point:
        :return:
        """




        return np.linalg.norm(first_point - second_point) def

    _adjust_s(self, solution_coords):
        """
        Корректирует длину ребра ячейки следящей области. :param
        solution_coords: координаты решения.
        :return:


        dist = self._calc_dist(
            solution_coords,
            self.current_central_cell_coords
        )

        if dist < 0.25 * self.s: self.s /= 2

        elif dist > 0.75 * self.s: self.s

            * = 1 . 5

    def _handle_cross_transition(self, temp_solution_coords):
```

```
        Обрабатывает параллельный перенос и масштабирование следящей области.
        :param temp_solution_coords: координаты временного решения.
        :return:
        it и и
        #   Корректируем длину ребра ячейки self._adjust_s(temp_solution_coords)
        #   Переносим центральную точку во временное решение (т.е., в центр масс)
        self.current_central_cell_coords = temp_solution_coords
        #   Строим новую следящую область self.cross = self._form_cross()

    def _stopping_criterion(self, solution_coords_lst):




        :param solution_coords_lst: список координат временных решений :return: True/False -
        верность критерия останова итерационного процесса фазы Targeting.
        и и и
        dist = self._calc_dist(
                    solution_coords_lst[-1],
                    solution_coords_lst[-2]
        )
        #   Вычисления прекращаются, когда расстояние между двумя
        #   последними точками становится меныие/равно targeting_eps return dist <=
        self.targeting_eps

    def solve(
                self,
                timeit=False,
    ):
        и и и
        Публичный метод. Получает решение задачи.
        :param timeit: фиксация времени и числа итераций при установке в True. Используется при
        проведении вычислительных экспериментов.
        :return:
        и и и



        if timeit:

            # Фиксируем время начала вычисления
            start_time = datetime.datetime.now()

        #   FlaxoflHM решение системы неравенств задачи,
        #   т.е. находим точку на многограннике (выполняем фазу Quest),
        self._catch_polyhedron_initially(self.x0)

        #   Инициализируем список временных решений точкой,
        #   полученной в результате выполнения фазы Quest. solution_coords_lst =
        [self.current_central_cell_coords]

        #   Получаем первое временное решение temp_solution_coords =
        self._get_temp_solution() solution_coords_lst.append(temp_solution_coords)
        #   Осуществляем масштабирование и перенос следящей области
        self._handle_cross_transition(temp_solution_coords)

        if timeit:
```

```
    #   Инициализируем счётчик итераций фазы Targeting iter_num
    = 1

while not self._stopping_criterion(solution_coords_lst):

    #   Выполняем те же шаги в цикле
    #   (до удовлетворения критерия останова).
    temp_solution_coords = self._get_temp_solution()
    solution_coords_lst.append(temp_solution_coords)
    self._handle_cross_transition(temp_solution_coords)

    if timeit:

        # Инкрементируем счётчик итераций
        iter_num += 1

if timeit:

    #   Фиксируем время окончания вычисления
    stop_time = datetime.datetime.now()
    #   Рассчитываем общее время решения
    eval_time = stop_time - start_time

# Останавливаем счётчик, если задача нестационарна if
hasattr(self.problem, 'counter'): self.problem.counter.stop()

if not timeit:

    #   При timeit = False, возвращаем только решение return
    solution_coords_lst[-1]

else:

    #   При timeit = True, возвращаем решение
    #   а также время вычислений и количество итераций return
    solution_coords_lst[-1], eval_time, iter_num
```

Приложение 5

Листинг модуля plotting. py

```
import pickle

import matplotlib.pyplot as pit

#   Конфигурация графиков
pit.rcParams['figure.figsize'] = (12, 5.5)
pit.rcParams['lines.linewidth'] = 2 . 5
pit.rcParams['lines.markeredgecolor'] = 'k'
pit.rcParams['lines.markersize'] = 8
pit.rcParams['legend.handlelength'] = 5 alpha = 0.85

n_dim_vals = [10, 20, 30]

#   Каждой размерности соответствует свой стиль
style_by_dim = {
    10: {
        'linestyle':
        'markerstyle': 'D',
        'color': '#EC7063'
    b
    20: {
        'linestyle':
        'markerstyle': ' o ',
        'color': '#58D68D'
    b
    30: {
        'linestyle': ' : ',
        'markerstyle': ' л ',
        'color': '#F4D03F'
    }
}



def plot_results(
    path_to_results,
    x_vals, title, xlabel,
    marker_needed,
    is_times
):
    ii и и
    Выводит на экран график.
    :param path_to_results: путь к файлу с результатами экспериментов. :param x_vals:
    варьируемые в эксперименте значения.
    :param title: заголовок графика.
    :param xlabel: подпись оси абсцисс.
    :param marker_needed: флаг необходимости маркеров на графике. :param is_times:
    флаг, исследуется ли время вычисления (или кол-во итераций).
    :return:
    и и и
    with open(path_to_results, ' r b ')  as f:  results_dct = pickle.load(f)
```

```
for n_dim in n_dim_vals:

    style = style_by_dim[n_dim]

    Is = style['linestyle']

    if not marker_needed:

        marker = None

    else:

        marker = style['markerstyle'] c =

    style['color'] if is_times:
        y_vals = [elem.seconds + elem.microseconds / (10**6) for elem in
    results_dct[n_dim]] ylabel = 'Время вычисления, с'

    else:

        y_vals = results_dct[n_dim] ylabel = 'Количество итераций'

    pit.plot( x_vals, y_vals, ls=ls,
        marker=marker,
        c=c,
        alpha=alpha,
        label=f'Размерность $n = {n_dim}$'
    )
pit.title(title)
pit.xlabel(xlabel)
pit.ylabel(ylabel) pit.legend()
plt.gridQ
```

Приложение 6

Листинг кода по исследованию стационарного варианта фазы Quest

```
import pickle

import numpy as np
from tqdm.notebook import tqdm
import matplotlib.pyplot as pit

from quest import Quest
from model_problem import ModelProblem

pit.rcParams['figure.figsize'] = (12, 7)

n_dim_vals = [10, 20, 30] l_vals = np.arange(0.1, 10.1, 0.1)

times_dct = dict() iter_dct = dict()

for n_dim in tqdm(n_dim_vals, desc='n_dim loop'):

    current_n_dim_times_lst = list()
    current_n_dim_iter_lst = list()

    for 1 in tqdm(l_vals, desc='l loop'):

        d = np.ones((n_dim, 1))
        non_stationary = False period = None
        x0 = np.zeros((n_dim, 1))
        fejer_mapp_precision_order = 3

        problem = ModelProblem( n=n_dim,
            d=d,
            non_stationary=non_stationary,
            period=period
        )
        quest = Quest(
            problem=problem,
            1=1
        )
        sol, eval_time, iter_num = quest.find_solution_of_inequalities_system( X0=X0,
            precision_order=fejer_map_precision_order,
            timeit=True
        )
        if hasattr(problem, 'counter'): problem.counter.stop()

        current_n_dim_times_lst.append(eval_time) current_n_dim_iter_lst.append(iter_num)

    times_dct[n_dim] = current_n_dim_times_lst
    iter_dct[n_dim] = current_n_dim_iter_lst
```

```
    with open('результаты экспериментов/время от коэффициента релаксации                quest
стационарный случай', 'wb') as f:
pickle.dump(times_dct, f)

    with open('результаты экспериментов/итерации от коэффициента релаксации             quest
стационарный случай', 'wb') as f:
pickle.dump(iter_dct, f)
```

Приложение 7

Листинг кода по исследованию нестационарного варианта фазы Quest

```
from IPython.core.debugger import set_trace import pickle

import datetime
import numpy as np
from tqdm.notebook import tqdm
import matplotlib.pyplot as pit

from quest import Quest

from model_problem import ModelProblem

pit.rcParams['figure.figsize'] = (12, 7)

n_dim_vals = [10, 20, 30]
d_coef_vals = [1] + list(np.arange(100, 1001, 100))

times_dct = dict() iter_dct = dict()

for n_dim in tqdm(n_dim_vals, desc='n_dim loop'):

    current_n_dim_times_lst = list() current_n_dim_iter_lst = list()

    for d_coef in tqdm(d_coef_vals, desc='d_coef loop'):

        tmp_times_lst = list()
        tmp_iter_lst = list()

        for i in range(1000):

            d = d_coef * np.ones((n_dim, 1))
            1=10
            non_stationary = True period = 1.3
            x0 = np.zeros((n_dim, 1))
            fejer_map_precision_order = 3

            problem = ModelProblem( n=n_dim, d=d,
                non_stationary=non_stationary, period=period
            )
            quest = Quest(
                problem=problem,
                1=1
            )
            sol, eval_time, iter_num = quest.find_solution_of_inequalities_system( X0=X0,
                precision_order=fejer_map_precision_order,
                timeit=True
            )
```

```
                                        if hasattr(problem, 'counter'):
                                            problem.counter.stop()

                    tmp_times_lst.append(eval_time)
                tmp_iter_lst.append(iter_num)
                current_n_dim_times_lst.append(np.mean(tmp_times_lst))
                current_n_dim_iter_lst.append(np.mean(tmp_iter_lst))

            times_dct[n_dim] = current_n_dim_times_lst iter_dct[n_dim] =
            current_n_dim_iter_lst


    with open('результаты экспериментов/время от модуля вектора сдвига quest
нестационарный случай', 'wb') as f:
pickle.dump(times_dct, f)

    with open('результаты экспериментов/итерации от модуля вектора сдвига quest
нестационарный случай', 'wb') as f:
        pickle.dump(iter_dct, f)
```

Приложение 8

Листинг кода по исследованию стационарного варианта алгоритма NSLP

```
from IPython.core.debugger import set_trace import pickle

import datetime
import numpy as np
from tqdm.notebook import tqdm
import matplotlib.pyplot as pit

from nslp import NSLP

pit.rcParams['figure.figsize'] = (12, 7)

n_dim_vals = [10, 20, 30]
K_vals = np.arange(ll, 1012, 100)

times_dct = dict() iter_dct = dict()

for n_dim in tqdm(n_dim_vals, desc='n_dim loop'):

    current_n_dim_times_lst = list()
    current_n_dim_iter_lst = list()

    for K in tqdm(K_vals, desc='K loop'):

        targeting_eps = 2 d = np.ones((n_dim, 1)) non_stationary = False
        period = None 1=10
        x0 = np.zeros((n_dim, 1)) fejer_mapprecision_order = 3 initial_s =
        0.01 w = 2

        nslp = NSLP(
            n_dim=n_dim,
            targeting_eps=targeting_eps, d=d,
            non_stationary=non_stationary,
            period=period,
            1=1,
            X0=X0,
            fejer_mapprecision_order=fejer_map_precision_order,
            K=K,
            initial_s=initial_s,
            W=W
        )

        sol, eval_time, iter_num = nslp.solve(timeit=True)

        current_n_dim_times_lst.append(eval_time) current_n_dim_iter_lst.append(iter_num)

    times_dct[n_dim] = current_n_dim_times_lst
    iter_dct[n_dim] = current_n_dim_iter_lst
```

```
with open('результаты экспериментов/время от кол-ва ячеек по одному измерению nslp стационарный случай', 'wb') as f: pickle.dump(times_dct, f)
```

Приложение 9

Листинг кода по исследованию зависимости сходимости алгоритма NSLP от количества ячеек следящей области по одному измерению при решении нестационарных задач

```
from IPython.core.debugger import set_trace import pickle

import datetime
import numpy as np
from tqdm.notebook import tqdm
import matplotlib.pyplot as pit

from nslp import NSLP

pit.rcParams['figure.figsize'] = (12, 7)

n_dim_vals = [10, 20, 30]
K_vals = np.arange(ll, 1012, 100)

times_dct = dict() iter_dct = dict()

for n_dim in tqdm(n_dim_vals, desc='n_dim loop'):

    current_n_dim_times_lst = list()
    current_n_dim_iter_lst = list()

    for K in tqdm(K_vals, desc='K loop'):

        targeting_eps = 2 d = np.ones((n_dim,
        1)) non_stationary = True period = 1.5
        1=10
        x0 = np.zeros((n_dim, 1))
        fejer_map_precision_order = 3 initial_s
        = 0.01 w = 10

        nslp = NSLP(
            n_dim=n_dim,
            targeting_eps=targeting_eps, d=d,
            non_stationary=non_stationary,
            period=period,
            1=1,
            x0=x0,
            fejer_map_precision_order=fejer_map_precision_order,
            K=K,
            initial_s=initial_s,
            w=w
        )

        sol, eval_time, iter_num = nslp.solve(timeit=True) current_n_dim_times_lst.append(eval_time)
```

```
                current_n_dim_iter_lst.append(iter_num)

            times_dct[n_dim] = current_n_dim_times_lst
            iter_dct[n_dim] = current_n_dim_iter_lst

    with open('результаты экспериментов/время от кол-ва ячеек по одному измерению nslp нестационарный случай', 'wb') as f: pickle.dump(times_dct, f)
```

Приложение 10

Листинг кода по поиску значения масштабирующего коэффициента следящей области

```
from IPython.core.debugger import set_trace import numpy as np

from nslp import NSLP, TargetingDoesNotConvergeError
n_dim = 30 targeting_eps = 2 d_coef = 200
d = d_coef * np.ones((n_dim, 1)) non_stationary = True period
= 1 . 5  1=10
x0 = np.zeros((n_dim, 1)) fejer_map_precision_order = 3 K = 11
initial_s = 0.01 w = 2

while True:

    nslp = NSLP(
        n_dim=n_dim,
        targeting_eps=targeting_eps, d=d,
        non_stationary=non_stationary, period=periodj 1=1,
        X0=X0J
        fejer_map_precision_order=fejer_map_precision_orderJ
        K=K,
        initial_s=initial_s,
        w=w
    )

    try:

        sol, eval_timej iter_num = nslp.solve(timeit=True) break

    except TargetingDoesNotConvergeError: w *= 2
```

Приложение 11

Листинг кода по исследованию зависимости сходимости алгоритма NSLP от координат вектора параллельного переноса многогранника при решении нестационарных задач

```
from IPython.core.debugger import set_trace import pickle

import datetime
import numpy as np
from tqdm.notebook import tqdm

import matplotlib.pyplot as pit

from nslp import NSLP

pit.rcParams['figure.figsize'] = (12, 7) n_dim_vals = [10, 20, 30]
d_coef_vals = [1] + list(np.arange(20, 191, 20))

times_dct = dict() iter_dct = dict()

for n_dim in tqdm(n_dim_vals, desc='n_dim loop'):

    current_n_dim_times_lst = list() current_n_dim_iter_lst = list()

    for d_coef in tqdm(d_coef_vals, desc='d_coef loop'):

        tmp_times_lst = list()
        tmp_iter_lst = list()

        for i in range(1000):

            targeting_eps = 2 d = d_coef *
            np.ones((n_dim, 1)) non_stationary =
            True period = 1 . 5  1=10
            x0 = np.zeros((n_dim, 1))
            fejer_map_precision_order = 3 K = 11
            initial_s = 0.01 w = 2

            nslp = NSLP(
                n_dim=n_dim,
                targeting_eps=targeting_eps,
                d=d,
                non_stationary=non_stationary, period=period,
                1=1,
                X0=X0,
                fejer_map_precision_order=fejer_map_precision_order,
                K=K,
                initial_s=initial_s,
```

```
                w=w
            )

            sol, eval_time, iter_num = nslp.solve(timeit=True)

            tmp_times_lst.append(eval_time)
        tmp_iter_lst.append(iter_num)
        current_n_dim_times_lst.append(np.mean(tmp_times_lst))
        current_n_dim_iter_lst.append(np.mean(tmp_iter_lst))

    times_dct[n_dim] = current_n_dim_times_lst
    iter_dct[n_dim] = current_n_dim_iter_lst

with open('результаты экспериментов/время от модуля вектора сдвига nslp нестационарный случай', 'wb') as
f:
    pickle.dump(times_dct, f)
```