МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
**«Национальный исследовательский
Нижегородский государственный университет им. Н.И. Лобачевского»
(ННГУ)**

Институт информационных технологий, математики и механики

Кафедра: Математического обеспечения и суперкомпьютерных технологий

Направление подготовки: «Прикладная математика и информатика»
Магистерская программа: «Системное программирование»

ОТЧЁТ

по производственной практике

на тему:

«Исследование методов учёта локальных свойств целевой функции в задачах глобальной оптимизации»

Выполнил: студент группы 381503м4
_____ Соврасов В.В.
Подпись
Научный руководитель:
доцент, к.ф.м.н.
_____ Баркалов К.А.
Подпись

Нижний Новгород
2017

Содержание

1. Введение **1**

2. Постановка задачи глобальной липшицевой оптимизации **1**

3. Алгоритм глобального поиска **2**

3.1. Сравнение методов оптимизации 5

3.2. Классы тестовых задач 5

4. Применение локальных оценок константы Гёльдера для ускорения сходимости АГП **6**

5. Прикладная задача поиска оптимального управления **9**

6. Заключение **14**

Список литературы **15**

7. Приложения **16**

7.1. Приложение 1 16

7.2. Приложение 2 24

1. Введение

Задачи нелинейной глобальной оптимизации встречаются в различных прикладных областях и традиционно считаются одними из самых трудоёмких среди оптимизационных задач. Их сложность экспоненциально растёт в зависимости от размерности пространства поиска, поэтому для решения существенно многомерных задач требуются суперкомпьютерные вычисления.

В настоящее время на кафедре МОиСТ активно ведётся разработка программной системы для глобальной оптимизации функций многих вещественных переменных Globalizer. Эта система включает в себя последние теоретические разработки, сделанные на кафедре в этой сфере, в том числе и блочную многошаговую схему редукции размерности [1]. Отличительной чертой ситемы является то, что, она может работать как на CPU, так на разных типах ускорителей вычислений с высокой степенью параллельности (XeonPhi, GPU Nvidia) [2, 3].

В данной работе будут описана прикладная задача оптимизации, решаемая с помощью системы Globalizer, а также проведено исследование одного из способов ускорения сходимости базового алгортима глобальной оптимизции, используемого в системе.

2. Постановка задачи глобальной липшицевой оптимизации

Одна из постановок задачи глобальной оптимизации звучит следующим образом: найти глобальный минимум N-мерной функции $\varphi(y)$ в гиперинтервале $D = \{y \in R^N : a_i \leqslant x_i \leqslant b_i, 1 \leqslant i \leqslant N\}$. Для построения оценки глобального минимума по конечному количеству вычислений значения функции требуется, чтобы $\varphi(y)$ удовлетворяла условию Липшица.

$$\varphi(y^*) = \min\{\varphi(y) : y \in D\}$$

$$|\varphi(y_1) - \varphi(y_2)| \leqslant L\|y_1 - y_2\|, y_1, y_2 \in D, 0 < L < \infty$$

Классической схемой редукции размерности для алгоритмов глобальной оптимизации является использование разверток — кривых, заполняющих пространство [4].

$$\{y \in R^N : -2^{-1} \leqslant y_i \leqslant 2^{-1}, 1 \leqslant i \leqslant N\} = \{y(x) : 0 \leqslant x \leqslant 1\}$$

Такое отображение позволяет свести задачу в многомерном пространстве к решению одномерной ценой ухудшения её свойств. В частности, одномерная функция $\varphi(y(x))$ является не Липшицевой, а Гёльдеровой:

$$|\varphi(y(x_1)) - \varphi(y(x_2))| \leqslant H|x_1 - x_2|^{\frac{1}{N}}, x_1, x_2 \in [0, 1]$$

где константа Гельдера H связана с константой Липшица L соотношением

$$H = 4Ld\sqrt{N}, d = \max\{b_i - a_i : 1 \leqslant i \leqslant N\}.$$

Область D также может быть задана с помощью функциональных ограничений. Постановка задачи глобальной оптимизации в этом случае будет иметь следующий вид:

$$\varphi(y^*) = \min\{\varphi(y) : g_j(y) \leqslant 0, 1 \leqslant j \leqslant m\} \tag{2.1}$$

Обозначим $g_{m+1}(y) = f(y)$. Далее будем предполагать, что все функции $g_k(y), 1 \leqslant k \leqslant m+1$ удовлетворяют условию Липшица в некотором гиперинтервале, включающем D.

3. Алгоритм глобального поиска

Для дальнейшего изложения потребуется описание метода глобальной оптимизации, используемого в системе Globalizer. Многомерные задачи сводятся к одномерным с помощью различных схем редукции размерности, поэтому можно рассматривать минимизацию одномерной функции $f(x), x \in [0,1]$, удовлетворяющей условию Гёльдера, при ограничениях, также удовлетворяющих этому условию на интервале $[0,1]$.

Рассматриваемый алгоритм решения одномерной задачи (2.1) предполагает построение последовательности точек x_k, в которых вычисляются значения минимизируемой функции или ограничений $z_k = g_s(x_k)$. Для учёта последних используется индексная схема [5]. Пусть $Q_0 = [0,1]$. Ограничение, имеющее номер j, выполняется во всех точках области

$$Q_j = \left\{x \in [0,1] : g_j(x) \leq 0\right\},$$

которая называется допустимой для этого ограничения. При этом допустимая область D исходной задачи определяется равенством: $D = \cap_{j=0}^{m} Q_j$. Испытание в точке $x \in [0,1]$ состоит в последовательном вычислении значений величин $g_1(x), ..., g_\nu(x)$, где значение индекса ν определяется условиями: $x \in Q_j, 0 \leqslant j < \nu, x \notin Q_\nu$. Выявление первого нарушенного ограничения прерывает испытание в точке x. В случае, когда точка x допустима, т. е. $x \in D$ испытание включает в себя вычисление всех функций задачи. При этом значение индекса принимается равным величине $\nu = m+1$. Пара $\nu = \nu(x), z = g_\nu(x)$, где индекс ν лежит в границах $1 \leqslant \nu \leqslant m+1$, называется результатом испытания в точке x.

Такой подход к проведению испытаний позволяет свести исходную задачу с функциональными ограничениями к безусловной задаче минимизации разрывной функции:

$$\psi(x^*) = \min_{x\in[0,1]} \psi(x),$$
$$\psi(x) = \begin{cases} g_\nu(x)/H_\nu & \nu < M \\ (g_M(x) - g_M^*)/H_M & \nu = M \end{cases}$$

Здесь $M = \max\{\nu(x) : x \in [0,1]\}$, а $g_M^* = \min\{g_M(x) : x \in \cap_{i=0}^{M-1} Q_i\}$. В силу определения числа M, задача отыскания g_M^* всегда имеет решение, а если $M = m+1$, то $g_M^* = f(x^*)$. Дуги функции $\psi(x)$ гельдеровы на множествах $\cap_{i=0}^{j} Q_i, 0 \leq j \leq M-1$ с константой 1, а сама $\psi(x)$ может иметь разрывы первого рода на границах этих множеств. Несмотря на то, что

значения констант Гёльдера H_k и величина g_M^* заранее неизвестны, они могут быть оценены в процессе решения задачи.

Множество троек $\{(x_k, \nu_k, z_k)\}, 1 \leqslant k \leqslant n$ составляет поисковую информацию, накопленную методом после проведения n шагов.

На первой итерации метода испытание проводится в произвольной внутренней точке x_1 интервала $[0; 1]$. Индексы точек 0 и 1 считаются нулевыми, значения z в них не определены. Пусть выполнено $k \geqslant 1$ итераций метода, в процессе которых были проведены испытания в k точках $x_i, 1 \leqslant i \leqslant k$. Тогда точка x^{k+1} поисковых испытаний следующей $(k+1)$-ой итерации определяются в соответствии с правилами:

Шаг 1. Перенумеровать точки множества $X_k = \{x^1, ..., x^k\} \cup \{0\} \cup \{1\}$, которое включает в себя граничные точки интервала $[0, 1]$, а также точки предшествующих испытаний, нижними индексами в порядке увеличения значений координаты, т.е.

$$0 = x_0 < x_1 < ... < x_{k+1} = 1$$

и сопоставить им значения $z_i = g_\nu(x_i), \nu = \nu(x_i), i = \overline{1, k}$.

Шаг 2. Для каждого целого числа $\nu, 1 \leqslant \nu \leqslant m+1$ определить соответствующее ему множество I_ν нижних индексов точек, в которых вычислялись значения функций $g_\nu(x)$:

$$I_\nu = \{i : \nu(x_i) = \nu, 1 \leqslant i \leqslant k\}, 1 \leq \nu \leqslant m+1,$$

определить максимальное значение индекса $M = \max\{\nu(x_i), 1 \leq i \leq k\}$.

Шаг 3. Вычислить текущие оценки для неизвестных констант Гёльдера:

$$\mu_\nu = \max\{\frac{|g_\nu(x_i) - g_\nu(x_j)|}{(x_i - x_j)^{\frac{1}{N}}} : i, j \in I_\nu, i > j\}. \tag{3.2}$$

Если множество I_ν содержит менее двух элементов или если значение μ_ν оказывается равным нулю, то принять $\mu_\nu = 1$.

Шаг 4. Для всех непустых множеств $I_\nu, \nu = \overline{1, M}$ вычислить оценки

$$z_\nu^* = \begin{cases} \min\{g_\nu(x_i) : x_i \in I_\nu\} & \nu = M \\ -\varepsilon_\nu & \nu < M \end{cases},$$

где вектор с неотрицательными координатами $\varepsilon_R = (\varepsilon_1, .., \varepsilon_m)$ называется вектором резервов.

Шаг 5. Для каждого интервала $(x_{i-1}; x_i), 1 \leqslant i \leqslant k$ вычислить характеристику

$$R(i) = \begin{cases} \Delta_i + \frac{(z_i - z_{i-1})^2}{(r_\nu \mu_\nu)^2 \Delta_i} - 2\frac{z_i + z_{i-1} - 2z_\nu^*}{r_\nu \mu_\nu} & \nu = \nu(x_i) = \nu(x_{i-1}) \\ 2\Delta_i - 4\frac{z_{i-1} - z_\nu^*}{r_\nu \mu_\nu} & \nu = \nu(x_{i-1}) > \nu(x_i) \\ 2\Delta_i - 4\frac{z_i - z_\nu^*}{r_\nu \mu_\nu} & \nu = \nu(x_i) > \nu(x_{i-1}) \end{cases} \tag{3.3}$$

где $\Delta_i = (x_i - x_{i-1})^{\frac{1}{N}}$. Величины $r_\nu > 1, \nu = \overline{1,m}$ являются параметрами алгоритма. От них зависят произведения $r_\nu \mu_\nu$, используемые при вычислении характеристик в качестве оценок неизвестных констант Гёльдера.

Шаг 5. Выбрать наибольшую характеристику:

$$t = \arg\max_{1 \leqslant i \leqslant k+1} R(i) \tag{3.4}$$

Шаг 6. Провести очередное испытание в середине интервала $(x_{t-1}; x_t)$, если индексы его концевых точек не совпадают: $x^{k+1} = \frac{1}{2}(x_t + x_{t-1})$. В противном случае провести испытание в точке

$$x^{k+1} = \frac{1}{2}(x_t + x_{t-1}) - \operatorname{sgn}(z_t - z_{t-1}) \frac{|z_t - z_{t-1}|^n}{2 r_\nu \mu_\nu^n}, \nu = \nu(x_t) = \nu(x_{t-1}),$$

а затем увеличить k на 1.

Алгоритм прекращает работу, если выполняется условие $\Delta_t \leqslant \varepsilon$, где $\varepsilon > 0$ есть заданная точность. В качестве оценки глобально-оптимального решения задачи выбираются значения

$$f_k^* = \min_{1 \leqslant i \leqslant k} f(x_i), x_k^* = \arg\min_{1 \leqslant i \leqslant k} f(x_i) \tag{3.5}$$

Достаточные условия сходимости метода определяются следующей теоремой:

Теорема 1 *Пусть исходная задача оптимизации имеет решение x^* и выполняются следующие условия:*

- *каждая область $Q_j, j = \overline{1,m}$ представляет собой объединение конечного числа отрезков, имеющих положительную длину;*
- *каждая функция $g_j(x), j = \overline{1,m+1}$ удовлетворяет условию Гёльдера с соответствующей константой H_j;*
- *компоненты вектора резервов удовлетворяют неравенствам $0 \leq 2\varepsilon_\nu < L_\nu(\beta - \alpha)$, где $\beta - \alpha$ — длина отрезка $[\alpha; \beta]$, лежащего в допустимой области D и содержащего точку x^*;*
- *начиная с некоторого значения k величины μ_ν, соответствующие непустым множествам I_ν, удовлетворяют неравенствам $r_\nu \mu_\nu > 2H_\nu$.*

Тогда верно следующее:

- *точка x^* является предельной точкой последовательности $\{x^k\}$, порождаемой методом при $\varepsilon = 0$ в условии остановки;*
- *любая предельная точка x^0 последовательности $\{x^k\}$ является решением исходной задачи оптимизации;*
- *сходимость к предельной точке x^0 является двухсторонней, если $x^0 \neq a, x^0 \neq b$.*

Подробнее метод и теорема о его сходимости описаны в [5].

3.1. Сравнение методов оптимизации

Существует несколько критериев оптимальности алгоритмов поиска (минимаксный, критерий одношаговой оптимальности), но большинстве случаев представляет интерес сравнение методов по среднему результату, достижимому на конкретном подклассе липшицевых функций. Достоинством такого подхода является то, что средний показатель можно оценить по конечной случайной выборке задач, используя методы математической статистики.

В качестве оценки эффективности алгоритма будем использовать, операционную характеристику, которая определяется множеством точек на плоскости (K, P), где K – среднее число поисковых испытаний, предшествующих выполнению условия остановки при минимизации функции из данного класса, а P – статистическая вероятность того, что к моменту остановки глобальный экстремум будет найден с заданной точностью. Если при выбранном K операционная характеристика одного метода лежит выше характеристики другого, то это значит, что при фиксированных затратах на поиск первый метод найдёт решение с большей статистической вероятностью. Если же зафиксировать некоторое значение P, и характеристика одного метода лежит левее характеристики другого, то первый метод требует меньше затрат на достижение той же надёжности.

3.2. Классы тестовых задач

Для сравнения алгоритмов глобального поиска в смысле операционной характеристики требуется иметь некоторое множество тестовых задач. Генератор задач GKLS, описанный в [6] позволяет получить такое множество задач с заренее известными свойствами. В данной работе используется одни класс, сгенерированный GKLS: 2d Simple, параметры которого также описаны в [6]. Функции рассматриваемого класса являются непрерывно дифференцируемыми и имеют 10 локальных минимумов, один из которых является глобальным.

Ещё одним тестовым классом, используемым в данной работе для сравнения методов, является набор двухмерных функций, предложенных В. А. Гришагиным [7]. Каждая функция существенно многоэкстремальна и задаётся формулой:

$$\phi(y) = \sqrt{\left(\sum_{i=1}^{7}\sum_{j-1}^{7} A_{ij}g_{ij}(y) + B_{ij}h_{ij}(y)\right)^2 + \left(\sum_{i=1}^{7}\sum_{j-1}^{7} C_{ij}g_{ij}(y) - D_{ij}h_{ij}(y)\right)^2}$$

где

$$y \in [0;1]^2,$$

$$g_{ij} = \sin(i\pi y_1)\sin(j\pi y_2),$$

$$h_{ij} = \cos(i\pi y_1)\cos(j\pi y_2),$$

где коэффициенты $A_{ij}, B_{ij}, C_{ij}, D_{ij}$ генерируются случайно с равномерным в интервале $[-1;1]$ распределением.

4. Применение локальных оценок константы Гёльдера для ускорения сходимости АГП

В части работы будем рассматривать задачу глобальной оптимизации в постановке без функциональных ограничений. Как видно из описания метода, независимо от локальных свойств оптимизируемой одномерной функции, для вычисления характеристик всех интервалов (3.3) используется одно и то же значение оценки константы Гёльдера (3.2). В работе [8] было предложено использовать различные значения $M = \mu_1$ (μ_1 из (3.2) в случае отсутствия функциональных ограничений), (3.3)) для каждого интревала, а также показана эффективность такого подхода в случае одномерной оптимизации функций, удовлетворяющих условию Липшица. В работе [9] рассмотрено применение адаптивных оценок констант Липшица в схеме многомерной вложенной оптимизации.

Для каждого интревала локальная оценка константы является максимум-аддитивной свёрткой «глобальной» и «локальной» компонент (γ и λ соответственно):

$$\lambda_i = \max\{H_{i-1}, H_i, H_{i+1}\}$$
$$H_i = \frac{|z_i - z_{i-1}|}{\Delta_i}$$
$$H^k = \max\{H_i : i = 2, \dots, k\}$$
$$\gamma_i = H^k \frac{\Delta_i}{\Delta^{max}}$$
$$\Delta^{max} = \max\{\Delta_i : i = 2, \dots, k\}$$

$$M_i = r \cdot \max\{H_i, \frac{1}{2}(\lambda_i + \gamma_i), \xi\} \tag{4.6}$$

Параметр ξ предотвращает обнуление оценки M_i в случае, если оптимизируемая фнкция является тождественной константой, и выбирается достаточно малым. Данный вариант свёртки не зависит от параметра r, однако в [10] также рассматриваетсяи адаптивная свёртка:

$$M_i = r \cdot \max\{H_i, \frac{\lambda_i}{r} + \frac{r-1}{r}\gamma_i, \xi\} \tag{4.7}$$

Если априори известно, что оптимизируемая функция имеет сложный рельеф с множеством локальных минимумов, то r изначально задаётся большим, что ведёт к преобладанию в адаптивной свёртке «глобальной» сооставляющей γ.

Оба варианта свёртки (4.6), (4.7) были предложены и детально рассмотрены в [10] для случая одномерных функций, удовлетворяющих условию Липшица. В данном разделе будет рассмотрено применение этих свёрток для оптимизации двумерных функций в случае редукции размерности с помощью развёрток.

В [10] приведена теорема о сходимости метода в случае, если целевая функция липшицева, однако, как правило, подобные утверждения справедливы и в Гёльдеровой метрике, поэтому, предположительно, будет верна следующая гепотеза:

Гепотеза 1 *Пусть целевая функция $f(x)$ удовлетвворяет условию Гёльдера с конечной константой $H > 0$, и пусть x является предельной точкой последовательности $\{x_k\}$, порождаемой алгоритмом. Тогда верны следующие утверждения:*

1. *Если $x \in (0; 1)$, то сходимость к точке x является двухсторонней, т.е. существуют две подпоследовательности $\{x_k\}$, сходящиеся к x: одна слева, а другая справа;*

2. *$f(x_k) \geqslant f(x)$ для всех точек испытаний $x_k, k \geqslant 1$;*

3. *Если существует другая предельна точка $x^* = x$, то $f(x) = f(x^*)$;*

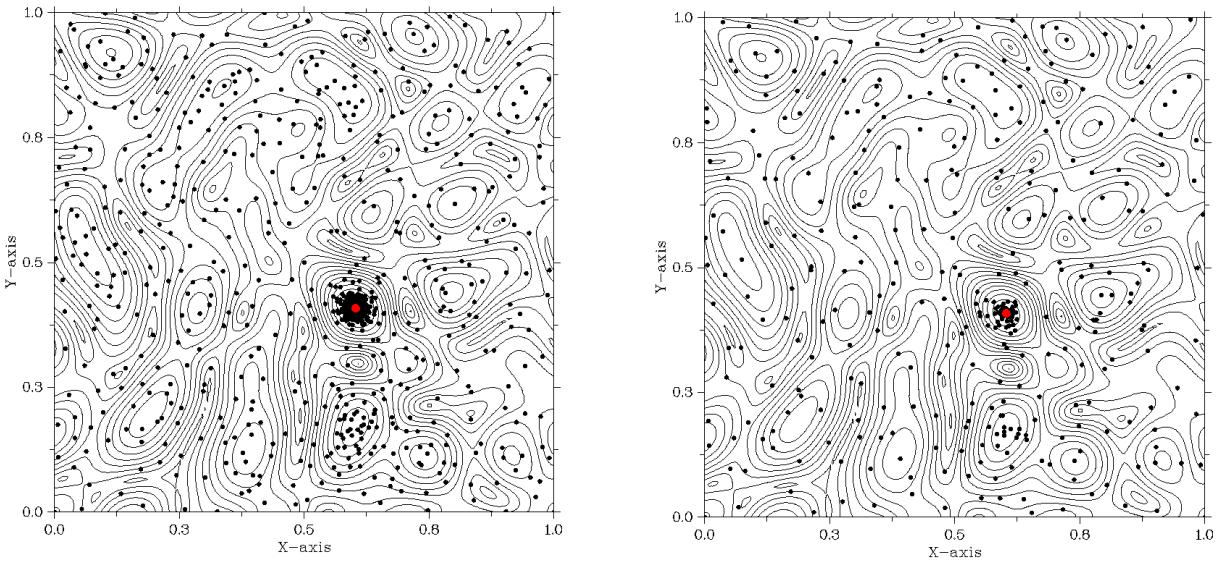
4. *Если функция $f(x)$ имеет конечное число локальных минимумов на отрезке $[0, 1]$, то точка x является локально оптимальной;*

5. *(Существенное условие сходимости к глобальномк минимуму). Пусть x^* является глобальным минимумом $f(x)$. Если существует такое число k^*, что для всех итераций с номерами $k > k^*$ неравенство $M_j(k) > H_j(k)$ выполняется, где $H_j(k)$ — это константа Гёльдера на интервале $[x_{j(k)-1}, x_{j(k)}]$, содержащем x^*, а $M_{j(k)}$ её оценка. Тогда множество предельных точек последовательности $\{x_k\}$ совпадает с множеством глобальных минимумов функции $f(x)$.*

Доказательство гепотезы требует отдельных теоретических исследований. В рамках данной работы оно не будет проведено, наличие сходимости установлено только численно.

Эксперименты по оценке эффективности метода с локально-адаптивной оценкой константы Гёльдера производились на двумерных классах задач Гришагина (F_{GR}) и GKLS Simple 2d, упомянутых в разделе 3.2. Каждый из классов содержит 100 многоэкстремальных функций. Развёртка во всех экспериментах строилась с плотностью $m = 12$, параметр ε в критерии остановки был равен 10^{-3}. Параметр r выбирался минимально возможным, при котором заданный метод решает все задачи класса. Шаг поиска r равен 0.1.

Для наглядной иллюстрации преимущества локально-адаптивной схемы оценки константы H рассмотрим результаты работы метода на конкретном примере. На рис. 4.1 показаны линии уровня одной из функций класса F_{GR} и точки испытаний, проведённых методом с глобальной оценкой константы Гёльдера и с оценкой по формуле (4.6). Как видно из рисунков, метод с глобальной оценкой константы проводит большое число испытаний в окрестности точки глобального минимума прежде (всего проведено 1086 испытаний), чем выполнится условие остановки, в то врямя, как метод с локально-адаптивной оценкой гораздо быстрее сходится (всего проведено 385 испытаний). Аналогичная ситуация имеет место при оптимизации одной из функций класса GKLS Simple 2d (рис. 4.2). Метод с глобальной оценкой константы произвёл 2600 испытаний, а метод с локально-адаптивной оценкой – 1190.



(a) глобальная оценка H (b) локально-адаптивная оценка H

Рис. 4.1: Линии уровня одной из функций класса F_{GR}

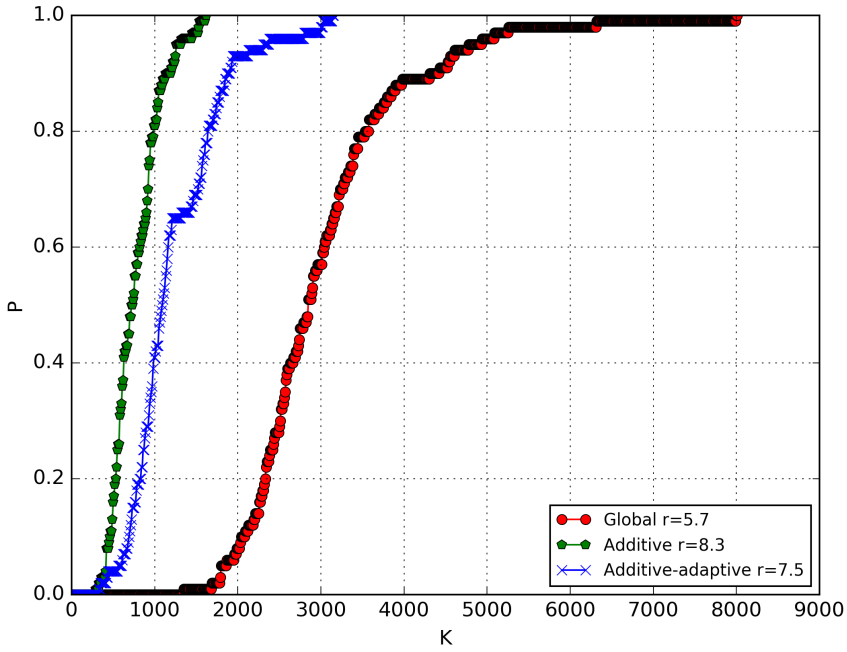
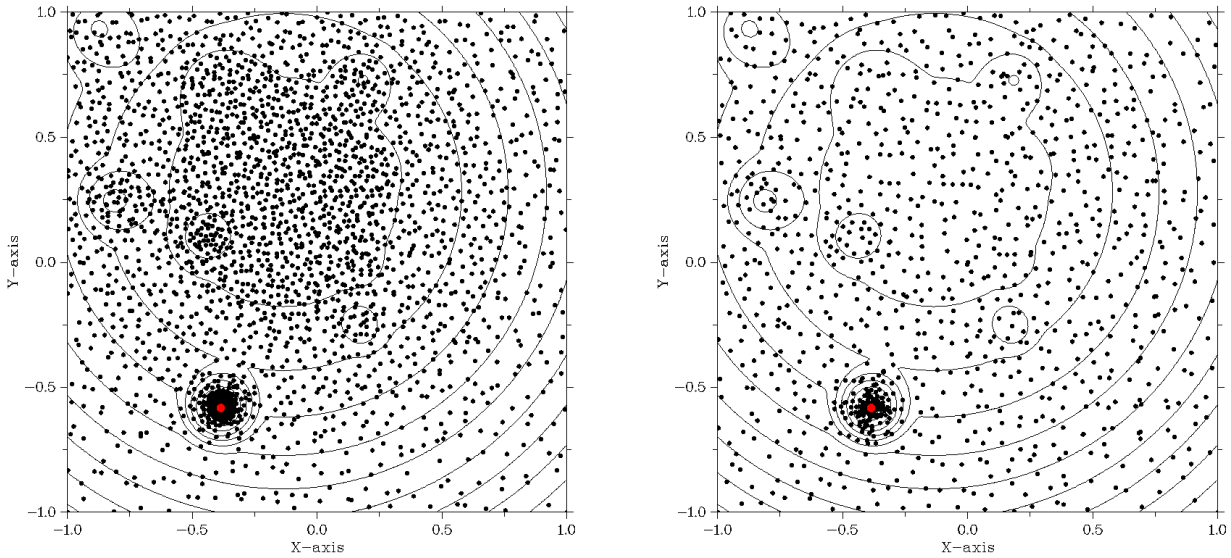


Рис. 4.4: Операционные характеристики методов на классе задач GKLS Simple 2d

Далее перейдём к сравнению различных вариантов метода на рассматриваемых классах задач. В качестве оценки эффективности алгоритма будем использовать, операционную характеристику, описанную в разделе 3.1.

Как видно из операционных характеристик на рис. 4.3 и 4.4, оба метода с локально-адаптивной оценкой константы Гёльдера показали существенное преимущество, однако они требуют задавать более высокое значение параметра r. Если сравнивать между собой методы с адаптивной и неадаптивной свёрткой (4.6), (4.7), то наглядно видно преимущество последнего, хотя он и



(a) глобальная оценка H (b) локально-адаптивная оценка H

Рис. 4.2: Линии уровня одной из функций класса GKLS Simple 2d

требует большее значение параметра надёжности r для решения задач из класса GKLS.

5. Прикладная задача поиска оптимального управления

Задача глобальной оптимизации возникает при синтезе оптимальных с точки зрения некоторых критериев управлений в линейных системах ОДУ. Если управление является линейной обратной связью по состоянию, то система с управлением имеет вид:

$$\dot{x} = (A + B_u\Theta)x + B_v v, x(0) = 0, \tag{5.8}$$

где $v(t) \in L_2$ — некоторое возмущение. Выходы системы описываются формулами $z_k = (C_k + B_u\Theta), k = \overline{1, N}$. Влияние возмущения на k-й выход системы описывается критерием

$$J_k(\Theta) = \sup_{v \in L_2} \frac{\max_{1 \le i \le n_k} \sup_{t \ge 0} |z_k^{(i)}(\Theta, t)|}{||v||_2}.$$

Нужно найти компоненты вектора Θ, минимизирующие один из критериев при заданных ограничениях на другие:

$$J_1(\Theta^*) = \min\{J_1(\Theta) : J_k(\Theta) \leqslant S_k, k = \overline{2, N}\}.$$

В [11] указан способ вычисления критериев, состоящий в следующем:

- найти матрицу Y из уравнения:

$$(A + B_u\Theta)Y + Y(A + B_u\Theta)^\mathsf{T} + B_v B_v^\mathsf{T} = 0$$

- если Y положительно определена, то используя её, вычислить функционалы $J_k(\Theta)$:

$$J_k(\Theta) = \sqrt{\max_{1 \leqslant i \leqslant n_k} \{(C_k^{(i)} + D_k^{(i)}\Theta)Y(C_k^{(i)} + D_k^{(i)}\Theta)^\mathsf{T}\}}, k = \overline{1, N},$$

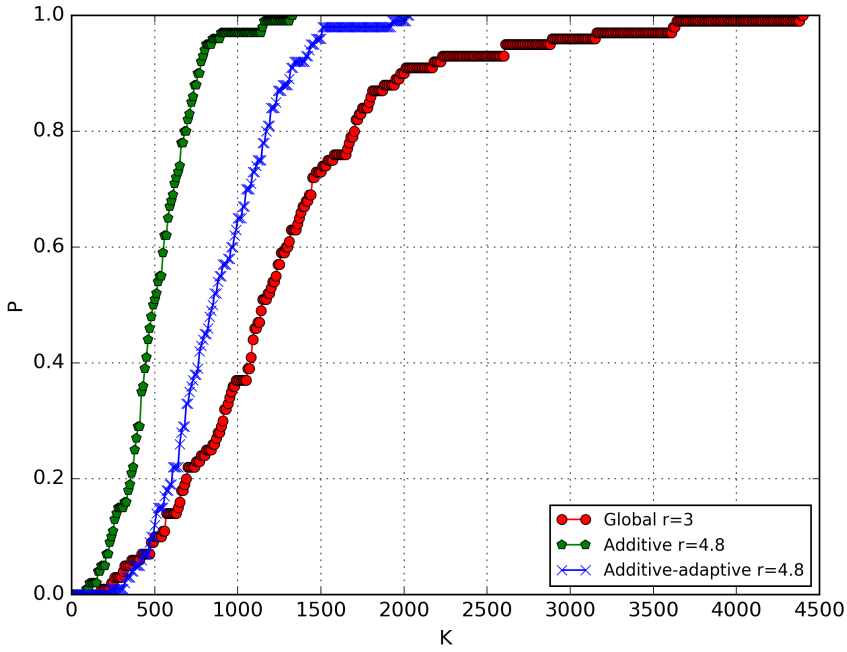


Рис. 4.3: Операционные характеристики методов на классе задач F_{GR}

где $C_k^{(i)}, D_k^{(i)}$ — i-е строки матриц C_k и D_k соответственно.

Для предотвращения случаев, когда Y незнакоопределена, в качестве дополнительного ограничения использовался критеий устойчивости линейной системы ОДУ: все действительные части собственных чисел матрицы $A + B_u\Theta$ должны быть отрицательны:

$$g_0(\Theta) = \min_j \mathrm{Re}(\lambda_j(\Theta)) < 0$$

С целью проверки корректности реализации вычисления критериев задачи индексным алгоритмом глобального поиска были решены две задачи рассматриваемого типа (их решение также приведено и в [11]).

В задаче виброзащиты параметры системы (5.8) определются следующим образом:

$$A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, B_v = B_u = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, C_1 = \begin{bmatrix} 0 & 1 \end{bmatrix}, D_1 = 0, C_2 = \begin{bmatrix} 0 & 1 \end{bmatrix}, D_2 = 1.$$

Управление имеет вид $u = [\theta_1, \theta_2]x$, где $\theta_1 \leqslant 0, \theta_2 \leqslant 0$ — оптимизируемые параметры. Сама задача ставится следующим образом:

$$J_2(\Theta^*) = \min\{J_2(\Theta) : J_1(\Theta) \leqslant 1, g_0(\Theta) \leqslant -0.02\}. \tag{5.9}$$

При решении этой задачи АГП с параметрами $r = 2.3$, $\varepsilon = 10^{-2}$ произвёл 65 испытаний, причём целевая функция была вычислена 27 раз. Найдена оптимальная точка с координатами $\tilde{\theta}_1 = -0.503223, \tilde{\theta}_2 = -0.997168$, $J_2(\widetilde{\Theta}) = 0.866551$, а ограничение на J_1 активно. На рис.

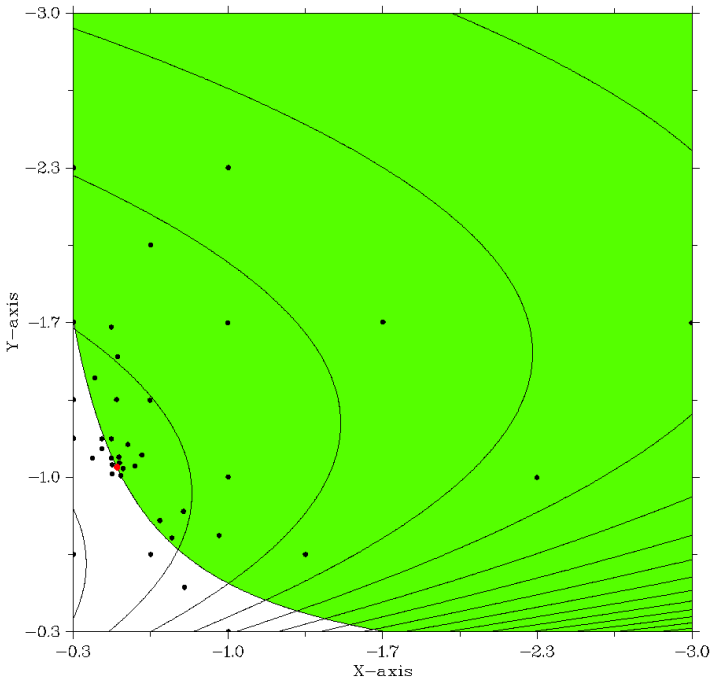


Рис. 5.5: Линии уровня для задачи виброзащиты с отмеченными точками испытаний АГП

5.5 представлены линии уровня целевой функции задачи виброзащиты. В данном случае допустимая область (закрашена на рисунке) имеет довольно простую границу, а целевая функция унимодальна.

В задаче гашения колебаний параметры системы (5.8) определются следующим образом:

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -2 & 1 & -2\beta & \beta \\ 1 & -1 & \beta & -\beta \end{bmatrix}, \beta = 0.1, B_v = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}, B_u = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix},$$

$$C_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \end{bmatrix}, D_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, C_2 = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}, D_2 = 1.$$

Управление имеет вид $u = [\theta_1, \theta_2, \theta_3, \theta_4]x$, где $\theta_1, \theta_2, \theta_3, \theta_4$ — оптимизируемые параметры. Сама задача ставится так же, как и предыдущая:

$$J_2(\Theta^*) = \min\{J_2(\Theta) : J_1(\Theta) \leqslant 1, g_0(\Theta) \leqslant -0.02\}.$$

В процессе решения этой задачи АГП с указанными ранее параметрами сделал 336575 испытания, причём целевая функция была вычислена 5173 раза. Найдена оптимальная точка с координатами $\tilde{\theta}_1 = 0.322954, \tilde{\theta}_2 = -0.583130, \tilde{\theta}_3 = -0.453491, \tilde{\theta}_4 = -0.970581$, $J_2(\widetilde{\Theta}) = 1.056579$, а ограничение на J_1 активно.

Рассмариваемые задачи поиска оптимального управления по состоянию интересны, пержде всего, в многокритериальной постановке. В данной работе рассматриваются задачи с двумя

критерими: один отвечает за максимальное смещение колебательного объекта, а другой — за максимальное управляющее воздействие. Для отыскания Парето-границы на плоскости критериев достаточно воспользоваться постановкой (5.9). Варьируя максимальное значение критерия $J_1(\theta)$ и каждый раз убеждаясь, что ограничение на $J_1(\theta)$ активно, мы получим пары (J_1, J_2), соответствующие Парето-границе. Этот метод неприменим в случае, когда одному значению J_1 соответствует несколько значений J_2, лежащих на Парето-границе (то есть в границу входит вертикально ориентированный отрезок). При применении этого метода в задаче, которая будет описана далее, не было выявлено каких-либо непердвиденных разрывов и резких скачков кривой-границы, поэтому другие способы решения многокритериальных задач не рассматривались.

Задача, в которой требовалось найти Парето-границу является расширением упомянутой ранее задачи виброзащиты, однако объект защиты представлен многомассовой механической системой. Приведём уравнения, описывающую двухмасовую систему:

$$\begin{cases} \dot{x}_1 = x_3 \\ \dot{x}_2 = x_4 \\ \dot{x}_3 = -x_1 + x_2 - \beta x_3 + \beta x_4 + u + v \\ \dot{x}_4 = x_1 - x_2 + \beta x_3 - \beta x_4 + u \end{cases}$$
$$x_1(0) = x_2(0) = x_3(0) = x_4(0) = 0$$

В случае n-массовой системы матирцы из (5.8) определяются следующим образом:

$$A = \begin{bmatrix} 0_{n\times n} & I_n \\ -K & -\beta K \end{bmatrix}, \beta = 0.1, B_v = \begin{bmatrix} 0_{n\times 1} \\ p \end{bmatrix}, B_u = \begin{bmatrix} 0_{n\times 1} \\ q \end{bmatrix}, p = \begin{bmatrix} 1 \\ 1 \\ ... \\ 1 \end{bmatrix}, q = \begin{bmatrix} 1 \\ 0 \\ ... \\ 0 \end{bmatrix},$$

$$K = \begin{bmatrix} 1 & -1 & 0 & ... & ... & 0 \\ -1 & 2 & -1 & ... & ... & 0 \\ ... & ... & ... & ... & ... & ... \\ ... & ... & ... & -1 & 2 & -1 \\ ... & ... & ... & 0 & -1 & 1 \end{bmatrix},$$

$$C_1 = \begin{bmatrix} 1 & 0 & ... & 0 & 0 & 0 \end{bmatrix}, D_1 = \begin{bmatrix} 0 \end{bmatrix}, C_1 = \begin{bmatrix} -1 & 1 & 0 & ... & 0 & 0 \\ 0 & -1 & 1 & ... & 0 & 0 \\ ... & ... & ... & ... & ... & ... \\ 0 & ... & ... & ... & 0 & 0 \\ 0 & ... & -1 & 1 & ... & 0 \end{bmatrix}, D_2 = \begin{bmatrix} 0 \\ 0 \\ ... \\ 0 \\ 0 \end{bmatrix},$$

где $K \in \mathbf{R}^{n \times n}$; $p, q \in \mathbf{R}^n$, $C_1 \in \mathbf{R}^{1 \times 2n}$, $C_2 \in \mathbf{R}^{n-1 \times 2n}$, $D_2 \in \mathbf{R}^{1 \times 2n}$.

Будем рассмаривать описанную задачу при $n = 10$. В отличие от ранее приведённых задач в управление включены не все фазовые переменные, а только три из них: $u = \theta_1 x_1 + \theta_2 x_3 + \theta_3 (x_2 - x_3), \theta_1 \leqslant 0, \theta_2 \leqslant 0$. Таким образом, решаемая задача оптимизации является трёхмерной. Если считать систему полностью наблюдаемой, то количество переменных увеличится до 20.

В [11] показано, что при полностью наблюдаемом состоянии системы Парето-граница в рассматриваемой задаче может быть получена с использованием линейных матричных неравенств. Кривая γ_{inf}, полученная коллегами с кафедры Кафедра Дифференциальных уравнений, математического и численного анализа, представлена на рис. 5.6. При её построении на абсолютные значения компонент вектора Θ не накладывалось никаких ограничений, поэтому её можно считать некоторой предельной, идеальной кривой, которая нереализуема в реальной системе.

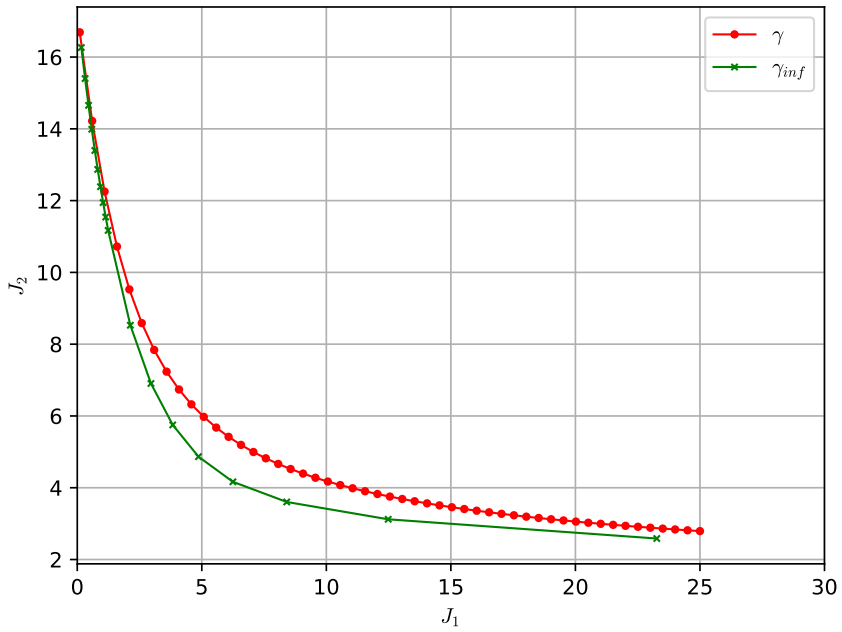


Рис. 5.6: Парето-границы в задаче виброизоляции при $n = 10$, построенные для частично и полностью наблюдаемого состояния

Построенная с помощью системы Globalizer описанным ранее способом Парето-граница показана на рис. 5.6 (кривая γ). При её построении были ограничены абсолютные значения коэффициентов обратной связи: $|\theta_i| < 10^4$. Сравнивая кривые γ_{inf} и γ, можно сказать, что потеря

качества управления при неполной наблюдаемости и ограниченных коэффициентов обратной связи некритическая.

6. Заключение

В ходе работы были получены следующие практические результаты:

- рассмотрено применение ранее предложенного для одномерных методов способа учёта локального поведения оптимизируемой функции в методе многомерной многоэкстремальной оптимизации. Учёт локальных свойств выражается в использовании различных оценок константы Гёльдера в различных областях поиска. Эффективность рассматриваемого подхода была подтверждена решением существенно многоэкстремальных задач из двух тестовых классов. Реализация модифицированного алгоритма глобального поиска на языке C++ представлена в приложении 7.1.;
- реализована схема вычисления целевой функции в задаче поиска оптимального управления, описанной в разделе 5. При реализации использовадась библиотека Eigen [12] для решения задач линейной алгебры. Исходный код можно найти в приложении 7.2. Два варианта задачи поиска оптимального управления, приведённые в [11], решены с помощью системы Globalizer;
- решена прикладная задача построения Парето-границы в одной задаче поиска оптимального управления.

Список литературы

[1] А.В. Сысоев, К.А. Баркалов, В.П. Гергель. «Блочная многошаговая схема параллельного решения задач многомерной глобальной оптимизации». В: *Материалы XIV Международной конференции "Высокопроизводительные параллельные вычисления на кластерных системах 10-12 ноября, ПНИПУ, Пермь*. 2014, 425–432.

[2] Сысоев А.В. Баркалов К.А. Гергель В.П. Лебедев И.Г. «MPI-реализация блочной многошаговой схемы параллельного решения задач глобальной оптимизации». В: *Суперкомпьютерные дни в России: Труды международной конференции (28-29 сентября 2015 г., г. Москва)*. М.: Изд-во МГУ, 2015, с. 411—419.

[3] К.А. Баркалов, И.Г. Лебедев, В.В. Соврасов, А.В. Сысоев. «Реализация параллельного алгоритма поиска глобального экстремума функции на Intel Xeon Phi». В: *Вычислительные методы и программирование* 17 (2016), с. 101—110.

[4] Стронгин Р. Г. *«Численные методы в многоэкстремальных задачах (информационно-статистические алгоритмы)»*. «Наука», М., 1978, 240 стр.

[5] Strongin R.G., Sergeyev Ya.D. *Global optimization with non-convex constraints. Sequential and parallel algorithms*. Dordrecht: Kluwer Academic Publishers, 2000.

[6] Квасов Д. Е., Сергеев Я. Д. *Исследование методов глобальной оптимизации при помощи генератора классов тестовых функций*. Н.Новгород: Изд-во ННГУ, 2011.

[7] В.А. Гришагин. «Операционные характеристики некоторых алгоритмов глобального поиска». В: *Проблемы случайного поиска* 7 (1978), с. 198—206.

[8] Sergeev, Ya.D. «An information global optimization algorithm with local tuning». В: *SIAM Journal on Optimization* 5.4 (1995), с. 390—407.

[9] Gergel, V., Grishagin, V., Israfilov, R. «Local tuning in nested scheme of global optimization». В: *Procedia Computer Science* 51 (2015), с. 865—874.

[10] Sergeyev, Y.D., Mukhametzhanov, M.S., Kvasov, D.E., Lera, D. «Derivative-Free Local Tuning and Local Improvement Techniques Embedded in the Univariate Global Optimization». В: *J Optim Theory Appl* (2016), с. 1—23.

[11] Д.В. Баландин М.М. Коган. «Оптимальное по Парето обобщенное H_2-управление и задачи виброзащиты». В: *Автоматика и телемеханика* (2017).

[12] *API documentation for Eigen3*. `https://eigen.tuxfamily.org/dox/`. [Online; accessed 26-December-2016].

7. Приложения

7.1. Приложение 1

```
#ifndef OPTIMIZER_ALGORITHM_UNCONSTRAINED_HPP
#define OPTIMIZER_ALGORITHM_UNCONSTRAINED_HPP

#include ”OptimizerCoreGlobal.hpp”
#include ”OptimizerTask.hpp”
#include ”OptimizerSolution.hpp”
#include ”OptimizerFunction.hpp”
#include ”OptimizerDataStructures.hpp”
#include ”OptimizerSolution.hpp”
#include ”OptimizerResult.hpp”
#include ”OptimizerSearchSequence.hpp”

#include <set>

namespace optimizercore
{

  class EXPORT_API OptimizerAlgorithmUnconstrained final
  {

  private:

    bool mLocalMixType;
    bool mIsAlgorithmMemoryAllocated;
    bool mIsParamsInitialized;
    bool mIsTaskInitialized;
    bool mNeedLocalVerification;

    int mNumberOfThreads;
    int mLocalStartIterationNumber;
    int mMaxNumberOfIterations;
    int mMapTightness;
    int mMethodDimension;
    int mAlpha;
    int mLocalMixParameter;
    int mMapType;

    OptimizerSpaceTransformation mSpaceTransform;
    OptimizerFunction *mTargetFunction;
    OptimizerFunctionPtr mTargetFunctionSmartPtr;

    OptimizerInterval *mIntervalsForTrials;
    std::set<OptimizerTrialPoint> mSearchInformationStorage;
    OptimizerTrialPoint mOptimumEvaluation, *mNextTrialsPoints;

    LocalTuningMode mLocalTuningMode;

    double mGlobalM, mZ, eps, r, mMaxIntervalNorm;
    double **mNextPoints;

    void AllocMem();
    void InitializeInformationStorage();
    void UpdateGlobalM(std::set<OptimizerTrialPoint>::iterator&);
    int UpdateRanks(bool isLocal);
    bool InsertNewTrials(int trailsNumber);
    OptimizerSolution DoLocalVerification(OptimizerSolution startPoint);

```

```
  public:
    OptimizerAlgorithmUnconstrained();
    ~OptimizerAlgorithmUnconstrained();

    void SetTask(OptimizerFunctionPtr function,
      OptimizerSpaceTransformation spaceTransform);
    void SetThreadsNum(int num);
    void SetParameters(OptimizerParameters params);

    OptimizerResult StartOptimization(const double* xOpt,
      StopCriterionType stopType);

    double GetLipschitzConst() const;
    OptimizerSearchSequence GetSearchSequence() const;

  };
}
#endif
```

```
  #include "OptimizerAlgorithmUnconstrained.hpp"
  #include "HookeJeevesLocalMethod.hpp"

  #include <cassert>
  #include <algorithm>

  using namespace optimizercore;
  using namespace optimizercore::utils;

  OptimizerAlgorithmUnconstrained::OptimizerAlgorithmUnconstrained()
  {
    mIsAlgorithmMemoryAllocated = false;

    mLocalStartIterationNumber = 1;
    mNumberOfThreads = 1;
    mMaxNumberOfIterations = 5000;
    mNextPoints = nullptr;
    mNextTrialsPoints = nullptr;
    mIntervalsForTrials = nullptr;
    r = 2;
    mLocalTuningMode = LocalTuningMode::None;

    mIsTaskInitialized = false;
    mIsParamsInitialized = false;
  }

  void OptimizerAlgorithmUnconstrained::SetTask(OptimizerFunctionPtr function,
    OptimizerSpaceTransformation spaceTransform)
  {
    assert(function);

    mTargetFunctionSmartPtr = function;
    mTargetFunction = function.get();
    mSpaceTransform = spaceTransform;

    mIsTaskInitialized = true;
  }

  OptimizerSearchSequence OptimizerAlgorithmUnconstrained::GetSearchSequence()
    const
```

```
{
  return OptimizerSearchSequence(mSearchInformationStorage, mMethodDimension,
    static_cast<MapType> (mMapType), mMapTightness, mSpaceTransform);
}

double OptimizerAlgorithmUnconstrained::GetLipschitzConst() const
{
  return mGlobalM;
}

void OptimizerAlgorithmUnconstrained::SetParameters(OptimizerParameters params
 )
{
  assert(params.algDimention);
  assert(params.eps > 0);
  assert(params.localAlgStartIterationNumber > 0);
  assert(params.mapTightness > 5 && params.mapTightness <= 20);
  assert(params.maxIterationsNumber > 0);
  assert(params.localMixParameter >= 0 && params.localMixParameter <= 20);
  assert(params.r != nullptr);
  assert(params.numberOfThreads > 0);
  assert(params.reserves != nullptr);
  assert(params.numberOfMaps > 0);

  mLocalStartIterationNumber = params.localAlgStartIterationNumber;
  eps = params.eps;
  if (params.localMixParameter <= 10) {
    mLocalMixParameter = params.localMixParameter;
    mLocalMixType = true;
  }
  else  {
    mLocalMixParameter = 20 - params.localMixParameter;
    mLocalMixType = false;
  }
  mNeedLocalVerification = params.localVerification;
  mAlpha = params.localExponent;
  mMethodDimension = params.algDimention;
  mMapTightness = params.mapTightness;
  mMapType = static_cast<int>(params.mapType);
  mMaxNumberOfIterations = params.maxIterationsNumber;
  mLocalTuningMode = params.localTuningMode;
  r = *params.r;
  if (mNextPoints)
    utils::DeleteMatrix(mNextPoints, mNumberOfThreads);
  mNextPoints = utils::AllocateMatrix<double>(mNumberOfThreads,
 mMethodDimension);
  this->SetThreadsNum(params.numberOfThreads);

  mIsParamsInitialized = true;
}

void OptimizerAlgorithmUnconstrained::InitializeInformationStorage()
{
  if (!mIsAlgorithmMemoryAllocated){
    AllocMem();
    mIsAlgorithmMemoryAllocated = true;
  }

  mZ = HUGE_VAL;
  mGlobalM = 1;
  mMaxIntervalNorm = 0;
```

```
99
100     mSearchInformationStorage.clear();
101
102     mapd(0.0, mMapTightness, mNextPoints[0], mMethodDimension, mMapType);
103     mSpaceTransform.Transform(mNextPoints[0], mNextPoints[0]);
104     mSearchInformationStorage.emplace(0.0, mTargetFunction->Calculate(
        mNextPoints[0]), 0);
105
106     mapd(1.0, mMapTightness, mNextPoints[0], mMethodDimension, mMapType);
107     mSpaceTransform.Transform(mNextPoints[0], mNextPoints[0]);
108     mSearchInformationStorage.emplace(1.0, mTargetFunction->Calculate(
        mNextPoints[0]), 0);
109   }
110
111   bool OptimizerAlgorithmUnconstrained::InsertNewTrials(int trailsNumber)
112   {
113     bool storageInsertionError;
114     if (mMapType == 3)
115     {
116       int preimagesNumber = 0;
117       double preimages[32];
118       for (int i = 0; i < trailsNumber; i++)
119       {
120         invmad(mMapTightness, preimages, 32,
121           &preimagesNumber, mNextPoints[i], mMethodDimension, 4);
122         for (int k = 0; k < preimagesNumber; k++)
123         {
124           mNextTrialsPoints[i].x = preimages[k];
125           auto insertionResult =
126             mSearchInformationStorage.insert(mNextTrialsPoints[i]);
127
128           if (!(storageInsertionError = insertionResult.second))
129             break;
130
131           UpdateGlobalM(insertionResult.first);
132         }
133       }
134     }
135     else
136       for (int i = 0; i < trailsNumber; i++)
137       {
138         auto insertionResult =
139           mSearchInformationStorage.insert(mNextTrialsPoints[i]);
140
141         if (!(storageInsertionError = insertionResult.second))
142           break;
143
144         UpdateGlobalM(insertionResult.first);
145       }
146     return storageInsertionError;
147   }
148
149   OptimizerResult OptimizerAlgorithmUnconstrained::StartOptimization(
150     const double* a, StopCriterionType stopType)
151   {
152     assert(mIsParamsInitialized && mIsTaskInitialized);
153     assert(mSpaceTransform.GetDomainDimension() == mMethodDimension);
154
155     InitializeInformationStorage();
156
157     double *y;
```

```
    bool stop = false;
    int iterationsCount = 0,
      currentThrNum = 1, ranksUpdateErrCode;

    mNextTrialsPoints[0].x = 0.5;
    mapd(mNextTrialsPoints[0].x, mMapTightness, mNextPoints[0],
      mMethodDimension, mMapType);
    mSpaceTransform.Transform(mNextPoints[0], mNextPoints[0]);

    while (iterationsCount < mMaxNumberOfIterations && !stop) {
      iterationsCount++;

#pragma omp parallel for num_threads(currentThrNum)
      for (int i = 0; i < currentThrNum; i++) {
        mNextTrialsPoints[i].val = mTargetFunction->Calculate(mNextPoints[i]);
        if (mMapType == 3)
          mSpaceTransform.InvertTransform(mNextPoints[i], mNextPoints[i]);
#pragma omp critical
        if (mNextTrialsPoints[i].val < mZ)
          mZ = mNextTrialsPoints[i].val;
      }

      if (!InsertNewTrials(currentThrNum))
        break;

      if (iterationsCount >= mLocalStartIterationNumber)  {
        if (iterationsCount % (12 - mLocalMixParameter) == 0
          && mLocalMixParameter > 0)
          ranksUpdateErrCode = UpdateRanks(mLocalMixType);
        else
          ranksUpdateErrCode = UpdateRanks(!mLocalMixType);
      }
      else
        ranksUpdateErrCode = UpdateRanks(false);

      if (iterationsCount >= mNumberOfThreads + 10)
        currentThrNum = mNumberOfThreads;

      for (int i = 0; i < currentThrNum && !stop; i++)  {
        OptimizerTrialPoint left = mIntervalsForTrials[i].left;
        OptimizerTrialPoint right = mIntervalsForTrials[i].right;

        mNextTrialsPoints[i].x = (left.x + right.x) / 2
          - sgn(right.val - left.val)*pow(fabs(right.val - left.val)
          / mIntervalsForTrials[i].localM, mMethodDimension) / (2 * r);

        mapd(mNextTrialsPoints[i].x, mMapTightness, mNextPoints[i],
          mMethodDimension, mMapType);
        mSpaceTransform.Transform(mNextPoints[i], mNextPoints[i]);

        y = mNextPoints[i];

        if (stopType == StopCriterionType::OptimalPoint)  {
          if (NormNDimMax(y, a, mMethodDimension) < eps)  {
            stop = true;
            mOptimumEvaluation = mNextTrialsPoints[i];
          }
        }
        else  {
          if (pow(right.x - left.x, 1.0 / mMethodDimension) < eps)  {
            stop = true;
```

```
            mOptimumEvaluation = mNextTrialsPoints[i];
          }
        }
      }
    }

    mOptimumEvaluation.val = mTargetFunction->Calculate(y);
    mSearchInformationStorage.insert(mOptimumEvaluation);

    if (stopType == StopCriterionType::Precision)
      mOptimumEvaluation = *std::min_element(mSearchInformationStorage.begin(),
        mSearchInformationStorage.cend(),
        [](OptimizerTrialPoint p1, OptimizerTrialPoint p2)
      {
        return p1.val < p2.val;
      });

    mapd(mOptimumEvaluation.x, mMapTightness, y, mMethodDimension, mMapType);
    mSpaceTransform.Transform(y, y);

    SharedVector optPoint(new double[mMethodDimension], array_deleter<double>())
    ;
    std::memcpy(optPoint.get(), y, mMethodDimension*sizeof(double));

    OptimizerSolution solution(iterationsCount, mOptimumEvaluation.val,
      mOptimumEvaluation.x, mMethodDimension, optPoint);

    if (mNeedLocalVerification)
      return OptimizerResult(DoLocalVerification(solution));
    else
      return OptimizerResult(solution);
  }
  OptimizerSolution OptimizerAlgorithmUnconstrained::DoLocalVerification(
    OptimizerSolution startSolution)
  {
    OptimizerFunctionPtr *functions = new OptimizerFunctionPtr[1];
    functions[0] = mTargetFunctionSmartPtr;

    OptimizerTask localTask(std::shared_ptr<OptimizerFunctionPtr>(functions,
      utils::array_deleter<OptimizerFunctionPtr>()),
      0, mMethodDimension, mSpaceTransform.GetLeftDomainBound(),
      mSpaceTransform.GetRightDomainBound());

    localoptimizer::HookeJeevesLocalMethod localMethod;
    localMethod.SetEps(eps / 100);
    localMethod.SetInitialStep(2 * eps);
    localMethod.SetProblem(localTask);
    localMethod.SetStepMultiplier(2);
    localMethod.SetStartPoint(startSolution.GetOptimumPoint().get(),
      localTask.GetTaskDimension());

    SharedVector localOptimum(new double[mMethodDimension], array_deleter<double
    >());
    localMethod.StartOptimization(localOptimum.get());
    double bestLocalValue = mTargetFunction->Calculate(localOptimum.get());

    if (startSolution.GetOptimumValue() > bestLocalValue)
      return OptimizerSolution(startSolution.GetIterationsCount(),
      bestLocalValue, 0.5, mMethodDimension, localOptimum);

    return startSolution;
```

```
277   }
278   void OptimizerAlgorithmUnconstrained::SetThreadsNum(int num)
279   {
280     if (num > 0 && num < 100)
281     {
282       if (mNextPoints != nullptr)
283         utils::DeleteMatrix(mNextPoints, mNumberOfThreads);
284       mNumberOfThreads = num;
285       if (mNextTrialsPoints)
286         delete[] mNextTrialsPoints;
287       if (mIntervalsForTrials)
288         delete[] mIntervalsForTrials;
289       mIntervalsForTrials = new OptimizerInterval[num];
290       mNextTrialsPoints = new OptimizerTrialPoint[num];
291       mNextPoints = utils::AllocateMatrix<double>(
292         mNumberOfThreads, mMethodDimension);
293     }
294   }
295   OptimizerAlgorithmUnconstrained::~OptimizerAlgorithmUnconstrained()
296   {
297     if (mIntervalsForTrials)
298       delete[] mIntervalsForTrials;
299     if (mNextPoints)
300       utils::DeleteMatrix(mNextPoints, mNumberOfThreads);
301     if (mNextTrialsPoints)
302       delete[] mNextTrialsPoints;
303     if (mIsAlgorithmMemoryAllocated)
304     {
305     }
306   }
307   void OptimizerAlgorithmUnconstrained::UpdateGlobalM(
308     std::set<OptimizerTrialPoint>::iterator& newPointIt)
309   {
310     double max = mGlobalM;
311     if (max == 1) max = 0;
312
313
314     auto leftPointIt = newPointIt;
315     auto rightPointIt = newPointIt;
316     --leftPointIt;
317     ++rightPointIt;
318
319     double leftIntervalNorm = pow(newPointIt->x - leftPointIt->x, 1.0 /
      mMethodDimension);
320     double rightIntervalNorm = pow(rightPointIt->x - newPointIt->x, 1.0 /
      mMethodDimension);
321
322
323     max = fmax(fmax(fabs(newPointIt->val - leftPointIt->val) / leftIntervalNorm,
324       fabs(rightPointIt->val - newPointIt->val) / rightIntervalNorm), max);
325
326
327     mMaxIntervalNorm = 0;
328     auto currentPointIt = mSearchInformationStorage.begin();
329     auto nextPointIt = currentPointIt;
330     ++nextPointIt;
331
332     while (nextPointIt != mSearchInformationStorage.cend())
333     {
334       if (mLocalTuningMode != LocalTuningMode::None)
335         mMaxIntervalNorm = fmax(
```

```
336             pow(nextPointIt->x - currentPointIt->x, 1.0 / mMethodDimension),
337             mMaxIntervalNorm);
338
339       ++currentPointIt;
340       ++nextPointIt;
341     }
342     if (max != 0)
343       mGlobalM = max;
344     else
345       mGlobalM = 1;
346   }
347   int OptimizerAlgorithmUnconstrained::UpdateRanks(bool isLocal)
348   {
349     double dx, curr_rank, mu1 = -HUGE_VAL, localM = mGlobalM;
350     double localMConsts[3];
351
352     for (int i = 0; i < mNumberOfThreads; i++)
353       mIntervalsForTrials[i].R = -HUGE_VAL;
354
355     auto leftIt = mSearchInformationStorage.begin();
356     auto rightIt = mSearchInformationStorage.begin();
357     ++rightIt;
358
359     int storageSize = mSearchInformationStorage.size();
360
361     for (int j = 0; j < storageSize - 1; j++)
362     {
363       dx = pow(rightIt->x - leftIt->x, 1.0 / mMethodDimension);
364
365       if (dx == 0)
366         return 1;
367
368       if (mLocalTuningMode != LocalTuningMode::None)  {
369         std::set<OptimizerTrialPoint>::iterator rightRightIt = rightIt;
370
371         if (j > 0 && j < storageSize - 2) {
372           ++rightRightIt;
373
374           std::swap(localMConsts[0], localMConsts[1]);
375           std::swap(localMConsts[1], localMConsts[2]);
376
377           localMConsts[2] = fabs(rightRightIt->val - rightIt->val)
378             / pow(rightRightIt->x - rightIt->x, 1.0 / mMethodDimension);
379
380           mu1 = fmax(fmax(localMConsts[0], localMConsts[1]), localMConsts[2]);
381         }
382         else if (j == 0)  {
383           ++rightRightIt;
384
385           localMConsts[1] = fabs(rightIt->val - leftIt->val) / dx;
386           localMConsts[2] = fabs(rightRightIt->val - rightIt->val) /
387             pow(rightRightIt->x - rightIt->x, 1.0 / mMethodDimension);
388           mu1 = fmax(localMConsts[1], localMConsts[2]);
389         }
390         else
391           mu1 = fmax(localMConsts[1], localMConsts[2]);
392
393         double mu2 = mGlobalM*dx / mMaxIntervalNorm;
394
395         if (mLocalTuningMode == LocalTuningMode::Maximum) {
396           localM = fmax(fmax(mu1, mu2), 0.01);
```

```
        }
        else// LocalTuningMode::Adaptive
          localM = fmax(mu1 / r + (1 - 1 / r)*mGlobalM, 0.01);
          //localM = fmax(mu1*(1 - dx / mMaxIntervalNorm) + mu2, 0.01);
          //localM = fmax(mu1*mMConvolution + (1 - mMConvolution)*mu2, 0.01);
      }

      curr_rank = dx + Pow2((rightIt->val - leftIt->val) / (r * localM)) / dx
        - 2 * (rightIt->val + leftIt->val - 2 * mZ) / (r * localM);
      if (isLocal)
        curr_rank /= sqrt((rightIt->val - mZ)*
        (leftIt->val - mZ)) / localM + pow(1.5, -mAlpha);

      if (curr_rank > mIntervalsForTrials[mNumberOfThreads - 1].R)
      {
        OptimizerInterval newInterval(
          OptimizerTrialPoint(*leftIt),
          OptimizerTrialPoint(*rightIt), curr_rank, localM);
        for (int i = 0; i < mNumberOfThreads; i++)
          if (mIntervalsForTrials[i].R < newInterval.R)
            std::swap(mIntervalsForTrials[i], newInterval);
      }
      ++leftIt;
      ++rightIt;
    }
    return 0;
  }
  void OptimizerAlgorithmUnconstrained::AllocMem()
  {
  }
```

7.2. Приложение 2

```
#ifndef __OPTMAL_CONTROL_PROBLEM_H__
#define __OPTMAL_CONTROL_PROBLEM_H__

#include "problem_interface.h"
#include "optimal_problem_base.h"

#include <Eigen/Dense>
#include <string>

class TOptimalControlProblem : public IProblem
{
protected:

  OptimalControlProblemBase* mPProblemImpl;
  int mDimension;
  bool mIsInitialized;
  std::string mConfigPath;

public:
  TOptimalControlProblem();
  virtual int SetConfigPath(const std::string& configPath);
  virtual int SetDimension(int dimension);
  virtual int GetDimension() const;
  virtual int Initialize();

```

```
  virtual void GetBounds(double* lower, double *upper);
  virtual int GetOptimumValue(double& value) const;
  virtual int GetOptimumPoint(double* x) const;

  virtual int GetNumberOfFunctions() const;
  virtual int GetNumberOfConstraints() const;
  virtual int GetNumberOfCriterions() const;

  virtual double CalculateFunctionals(const double* x, int fNumber);

  ~TOptimalControlProblem();
};

extern "C" LIB_EXPORT_API IProblem* create();
extern "C" LIB_EXPORT_API void destroy(IProblem* ptr);
#endif
```

```
#include "optimalControl.h"
#include "test_problems.h"
#include "problemA.h"
#include "problemB.h"
#include "pugixml.hpp"

#include <string>
#include <stdexcept>

TOptimalControlProblem::TOptimalControlProblem()
{
  mIsInitialized = false;
}

int TOptimalControlProblem::SetConfigPath(const std::string& configPath)
{
  mConfigPath = std::string(configPath);
  return IProblem::OK;
}

int TOptimalControlProblem::SetDimension(int dimension)
{
    return IProblem::OK;
}

int TOptimalControlProblem::GetDimension() const
{
  return mDimension;
}

int TOptimalControlProblem::Initialize()
{
  if (mIsInitialized == false)
  {
    mIsInitialized = true;

    pugi::xml_document doc;
    pugi::xml_parse_result result = doc.load_file(mConfigPath.c_str());
    if (result.status != pugi::status_ok)
      return IProblem::ERROR;

    pugi::xml_node config = doc.child("config");
```

```
    std::string problemName = config.child("problem_name").child_value();
    double secondCriterionLevel = 0.;
    double lambda = 0.;
    int dimension = 0;
    try  {
      secondCriterionLevel = std::stod(config.child("S").child_value());
      lambda = std::stod(config.child("lambda").child_value());
      dimension = std::stoi(config.child("N").child_value());
    }
    catch (std::invalid_argument& exp)  {
      return IProblem::ERROR;
    }

    if (problemName == std::string("v"))
      mPProblemImpl = new VibroisolationProblem();
    else if (problemName == std::string("od"))
      mPProblemImpl = new OscillationDampingProblem();
    else if (problemName == std::string("a2d"))
      mPProblemImpl = new ProblemA2d(secondCriterionLevel);
    else if (problemName == std::string("a3d"))
      mPProblemImpl = new ProblemA3d(secondCriterionLevel);
    else if (problemName == std::string("bNd") && dimension == 2)
      mPProblemImpl = new ProblemB2d(secondCriterionLevel);
    else if (problemName == std::string("bNd"))
      mPProblemImpl = new ProblemB(secondCriterionLevel, dimension);
    else if (problemName == std::string("bNdC"))
      mPProblemImpl = new ProblemBConvolved(lambda, dimension);
    else
      return IProblem::ERROR;

    mDimension = mPProblemImpl->GetDimension();

    return IProblem::OK;
  }
  else
    return IProblem::ERROR;
}

void TOptimalControlProblem::GetBounds(double* lower, double *upper)
{
  if (mIsInitialized)
    mPProblemImpl->GetBounds(lower, upper);
}

int TOptimalControlProblem::GetOptimumValue(double& value) const
{
  return IProblem::UNDEFINED;
}

int TOptimalControlProblem::GetOptimumPoint(double* point) const
{
  return IProblem::UNDEFINED;
}

int TOptimalControlProblem::GetNumberOfFunctions() const
{
  return mPProblemImpl->GetNumberOfFunctions();
}

int TOptimalControlProblem::GetNumberOfConstraints() const
{
```

```
  return mPProblemImpl->GetNumberOfConstraints();
}

int TOptimalControlProblem::GetNumberOfCriterions() const
{
  return 1;
}

//
    ------------------------------------------------------------------------------

double TOptimalControlProblem::CalculateFunctionals(const double* x, int fNumber
    )
{
  return mPProblemImpl->CalculateFunctionals(x, fNumber);
}

TOptimalControlProblem::~TOptimalControlProblem()
{
  if(mIsInitialized)
  {
    mPProblemImpl->PrintFinalMessage();
    delete mPProblemImpl;
  }
}

LIB_EXPORT_API IProblem* create()
{
  return new TOptimalControlProblem();
}

LIB_EXPORT_API void destroy(IProblem* ptr)
{
  delete ptr;
}
```

```
#ifndef OPTIMAL_CONTROL_BASE_H
#define OPTIMAL_CONTROL_BASE_H

#ifndef EIGEN_DONT_PARALLELIZE
#define EIGEN_DONT_PARALLELIZE
#endif

#include <Eigen/Dense>
#include <vector>

class OptimalControlProblemBase
{
protected:
  int mDimension;
  int n_x;
  int n_v;
  int n_u;
  std::vector<int> n_k;

  Eigen::MatrixXd A;
  Eigen::MatrixXd B_u;
  Eigen::MatrixXd B_v;
  Eigen::MatrixXd* YVecs;
```

```
  std::vector<Eigen::MatrixXd> CMatrices;
  std::vector<Eigen::MatrixXd> DMatrices;

  double mZeroConstraintOffset;
  double mS;

  virtual Eigen::RowVectorXd getTheta(const double* x);
  virtual double CalculateCriterionValue(const Eigen::RowVectorXd& theta, int
    fNumber);

public:
  OptimalControlProblemBase();
  virtual ~OptimalControlProblemBase();
  virtual void GetBounds(double* lower, double *upper) const = 0;
  virtual double CalculateFunctionals(const double* x, int fNumber);
  int GetDimension() const { return mDimension; }
  virtual int GetNumberOfConstraints() const { return 2; }
  virtual int GetNumberOfFunctions() const { return 3; }
  virtual void PrintFinalMessage() {}
};

#endif
```

```
#include "optimal_problem_base.h"

#define _USE_MATH_DEFINES
#include <math.h>
#include <algorithm>
#include <omp.h>

using namespace Eigen;

OptimalControlProblemBase::OptimalControlProblemBase()
{
  YVecs = new MatrixXd[omp_get_num_procs()];
}

OptimalControlProblemBase::~OptimalControlProblemBase()
{
  delete[] YVecs;
}

RowVectorXd OptimalControlProblemBase::getTheta(const double* x)
{
  return Map<const RowVectorXd>(x, mDimension);
}

MatrixXd buildVectorizationMatrix(const MatrixXd& A)
{
  size_t n = A.cols();

  MatrixXd E = MatrixXd::Identity(n, n);
  MatrixXd S(n*n, n*n);

  for(size_t i = 0; i < n; i++)
    for(size_t j = 0; j < n; j++)
    {
      if(i != j)
        S.block(i*n, j*n, n, n) = A.coeff(i, j)*E;
```

```
      else
        S.block(i*n, j*n, n, n) = A + A.coeff(i, j)*E;
    }

  return S;
}

double OptimalControlProblemBase::CalculateCriterionValue(const Eigen::
    RowVectorXd& theta, int fNumber)
{
  Map<MatrixXd> Y(YVecs[omp_get_thread_num()].data(), A.cols(), A.rows());

  double value = -HUGE_VAL;
  size_t cRows = CMatrices[fNumber].rows();
  for (size_t i = 0; i < cRows; i++)
  {
    RowVectorXd currentVector = CMatrices[fNumber].row(i) + DMatrices[fNumber].
    row(i)*theta;
    double dotProd = (currentVector*Y).dot(currentVector.transpose());
    value = std::max(value, dotProd);
  }
  return value;
}

double OptimalControlProblemBase::CalculateFunctionals(const double* x, int
    fNumber)
{
  RowVectorXd theta = getTheta(x);

  if (fNumber == 0)
  {
    MatrixXd Atheta = A + B_u*theta;

    EigenSolver<MatrixXd> eigenSolver(Atheta, false);
    MatrixXcd eigenvalues = eigenSolver.eigenvalues();
    double maxReal = -HUGE_VAL;
    for (int i = 0; i < Atheta.cols(); i++)
      maxReal = std::max(maxReal, eigenvalues.coeff(i).real());

    return maxReal + mZeroConstraintOffset;
  }
  else if (fNumber == 1)
  {
    MatrixXd Atheta = A + B_u*theta;
    MatrixXd S = buildVectorizationMatrix(Atheta);
    MatrixXd rhs = -B_v*B_v.transpose();
    Map<VectorXd> rhsMap(rhs.data(), rhs.size());
    YVecs[omp_get_thread_num()] = S.partialPivLu().solve(rhsMap);
  }

  fNumber--;
  double offset = fNumber == 0 ? -mS : 0.;
  double value = CalculateCriterionValue(theta, fNumber);

  return sqrt(value) + offset;
}
```

```
#pragma once
#include "optimal_problem_base.h"
```

```
3
4  #define _USE_MATH_DEFINES
5  #include <math.h>
6  #include <algorithm>
7  #include <iostream>
8
9  using namespace Eigen;
10 using Eigen::internal::BandMatrix;
11
12 class ProblemB_Common : public OptimalControlProblemBase
13 {
14 public:
15   ProblemB_Common(double S)
16   {
17     double beta = 0.1;
18
19     int n = 10;
20
21     n_x = 2*n;
22     n_v = 1;
23     n_u = 1;
24     n_k.resize(2);
25     n_k[0] = n_k[1] = 1;
26
27     CMatrices.resize(2);
28     DMatrices.resize(2);
29
30     A.resize(n_x, n_x);
31     A.topLeftCorner(n, n).setZero();
32     A.topRightCorner(n, n).setIdentity();
33
34     BandMatrix<double> K(n, n, 1, 1);
35     K.diagonal().setConstant(2.);
36     K.diagonal(-1).setConstant(-1.);
37     K.diagonal(1).setConstant(-1.);
38     K.diagonal()(0) = 1.;
39     K.diagonal()(n - 1) = 1.;
40
41     A.bottomLeftCorner(n, n) = -K.toDenseMatrix();
42     A.bottomRightCorner(n, n) = -beta*K.toDenseMatrix();
43
44     B_v.resize(n_x, 1);
45     B_v.col(0).head(n).setZero();
46     B_v.col(0).tail(n).setOnes();
47
48     B_u = VectorXd::Zero(n_x);
49     B_u(n, 0) = 1.;
50
51     CMatrices[0].resize(1, n_x);
52     CMatrices[0].setZero();
53     CMatrices[0](0, 0) = 1.;
54
55     CMatrices[1].resize(n - 1, n_x);
56
57     for(size_t i = 0; i < n - 1; i++)
58     {
59       CMatrices[1].row(i).setZero();
60       CMatrices[1].row(i)(i) = -1.;
61       CMatrices[1].row(i)(i + 1) = 1.;
62     }
63
```

```
    DMatrices[0].resize(1, 1); DMatrices[0] << 0;

    DMatrices[1].resize(n - 1, 1);
    DMatrices[1].setZero();

    mS = S;
    mZeroConstraintOffset = 0.;
  }

  int GetNumberOfConstraints() const { return 2; }
};

class ProblemB : public ProblemB_Common
{
public:
  ProblemB(double S, int dimension) : ProblemB_Common(S)
  {
    mDimension = dimension;//shoud be > 2
  }

  void GetBounds(double* lower, double *upper) const
  {
    for (int i = 0; i < mDimension; i++)
    {
      lower[i] = -10000.;
      upper[i] = -0.01;
    }

    for (int i = 2; i < mDimension; i++)
    {
      upper[i] = 10000.;
    }
  }

  RowVectorXd getTheta(const double* x)
  {
    RowVectorXd theta(n_x);
    theta.setZero();

    theta(0) = x[0] - x[2];
    for (int i = 1; i < mDimension - 2; i++)
        theta(i) = x[i + 1] - x[i + 2];
    theta(mDimension - 2) = x[mDimension - 1];
    theta(n_x / 2) = x[1];

    return theta;
  }
};

class ProblemB2d : public ProblemB_Common
{
public:
  ProblemB2d(double S) : ProblemB_Common(S)
  {
    mDimension = 2;
  }

  void GetBounds(double* lower, double *upper) const
  {
    for (int i = 0; i < mDimension; i++)
    {
```

```
      lower[i] = -70.;
      upper[i] = -0.01;
    }
    lower[0] = -20.;
  }

  RowVectorXd getTheta(const double* x)
  {
    RowVectorXd theta(n_x);
    theta.setZero();
    theta(0) = x[0];
    theta(n_x / 2) = x[1];

    return theta;
  }
};
```

```
#ifndef __OPTMAL_CONTROL_TEST_IMPL_H__
#define __OPTMAL_CONTROL_TEST_IMPL_H__

#include "optimal_problem_base.h"

class VibroisolationProblem : public OptimalControlProblemBase
{
public:
  VibroisolationProblem()
  {
    n_x = 2;
    n_v = 1;
    n_u = 1;
    n_k.resize(2);
    n_k[0] = n_k[1] = 1;

    CMatrices.resize(2);
    DMatrices.resize(2);

    A.resize(2, 2); A << 0, 1, 0, 0;
    B_u.resize(2, 1); B_u << 0, 1;
    B_v = B_u;

    CMatrices[0].resize(1, 2); CMatrices[0] << 1, 0;
    CMatrices[1].resize(1, 2); CMatrices[1] << 0, 0;

    DMatrices[0].resize(1, 1); DMatrices[0] << 0;
    DMatrices[1].resize(1, 1); DMatrices[1] << 1;
    mDimension = n_x;
    mZeroConstraintOffset = 0.;
    mS = 1.;
  }

  int GetNumberOfConstraints() const { return 2; }

  void GetBounds(double* lower, double *upper) const
  {
    for (int i = 0; i < mDimension; i++)
    {
      upper[i] = -0.2;
      lower[i] = -2.0;
    }
```

```
  }
};

class OscillationDampingProblem : public OptimalControlProblemBase
{
public:
  OscillationDampingProblem()
  {
    double beta = 0.1;
    n_x = 4;
    n_v = 1;
    n_u = 1;
    n_k.resize(2);
    n_k[0] = n_k[1] = 1;

    CMatrices.resize(2);
    DMatrices.resize(2);

    A.resize(4, 4);
    A << 0, 0, 1, 0,
      0, 0, 0, 1,
      -2, 1, -2*beta, beta,
      1, -1, beta, -beta;

    B_u.resize(4, 1); B_u << 0, 0, 0, 1;
    B_v.resize(4, 1); B_v << 0, 0, 1, 1;

    CMatrices[0].resize(2, 4);
    CMatrices[0] << 1, 0, 0, 0,
      -1, 1, 0, 0;
    CMatrices[1].resize(1, 4); CMatrices[1] << 0, 0, 0, 0;

    DMatrices[0].resize(2, 1); DMatrices[0] << 0, 0;
    DMatrices[1].resize(1, 1); DMatrices[1] << 1;
    mDimension = n_x;
    mZeroConstraintOffset = 0.02;
    mS = 1.;
  }

  int GetNumberOfConstraints() const
  {
    return 2;
  }

  void GetBounds(double* lower, double *upper) const
  {
    for (int i = 0; i < mDimension; i++)
    {
      upper[i] = 1;
      lower[i] = -2.0;
    }
  }
};
#endif
```