

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»

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Тема:
«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	3
3.1. Сравнение методов оптимизации	6
3.2. Классы тестовых задач	6
4. Применение локального поиска для ускорения сходимости АГП	7
5. Смешанный алгоритм глобального поиска и его эффективная реализация	8
6. Многоуровневая схема редукции размерности с помощью разверток	10
7. Применение локальных оценок константы Гёльдера для ускорения сходимости АГП	12
8. Прикладная задача поиска оптимального управления	17
9. Заключение	20
Список литературы	22
10. Приложения	23
10.1. Приложение 1	23
10.2. Приложение 2	31
10.3. Приложение 3	40
10.4. Приложение 4	45

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 системы 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, решённая с помощью системы Globalizer, а также проведено исследование нескольких способов ускорения сходимости базового алгоритма глобальной оптимизации, 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 $D = \{y \in R^N : a_i \leq x_i \leq b_i, 1 \leq i \leq N\}$. Для построения оценки глобального минимума по конечному количеству вычислений значения функции требуется, чтобы $\varphi(y)$ удовлетворяла условию Липшица.

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

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

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

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

На рис. 2.1 представлены построенные численно с низкой точностью развёртки на плоскости и в трёхмерном пространстве.

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 Гельдеровой:

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

где константа Гельдера H связана с константой Липшица L соотношением

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

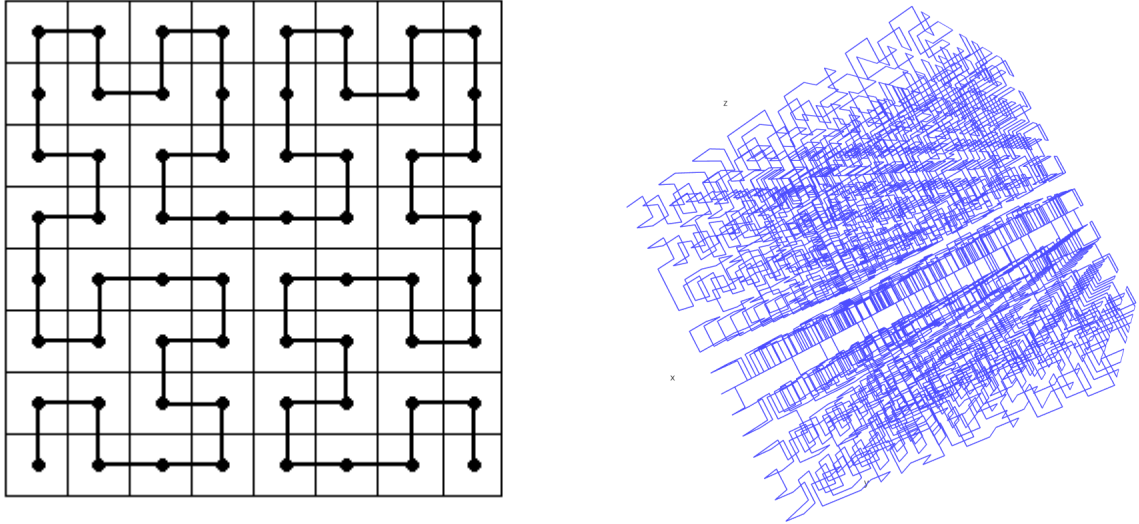


Рис. 2.1: Численно построенная развёртка в двухмерном и трёхмерном случаях

На рис. 2.2 приведена одномерная функция, полученная после применения развёртки к парабоиду вращения $\varphi(y) = y_1^2 + y_2^2$.

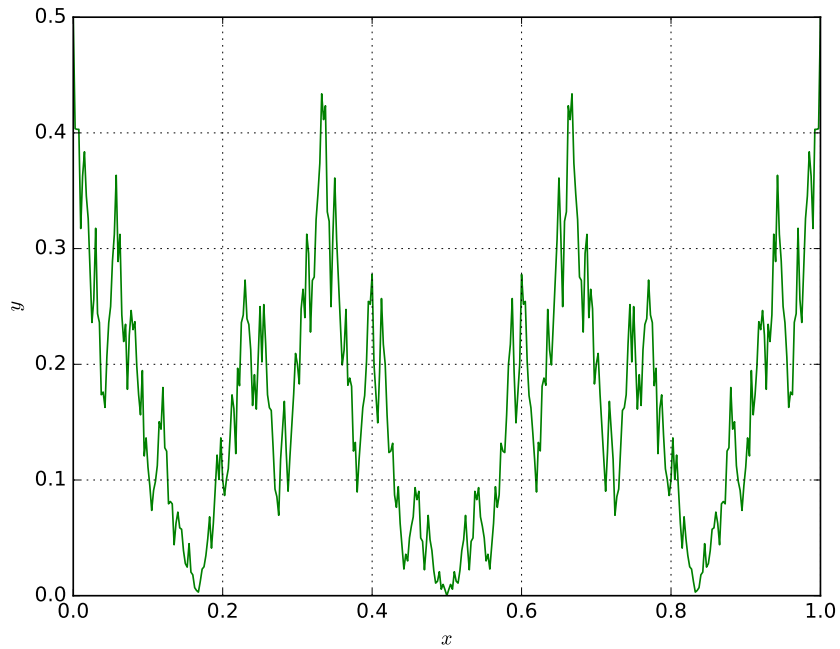


Рис. 2.2: Пример одномерной функции, порождённой развёрткой

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

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

Обозначим $g_{m+1}(y) = f(y)$. Далее будем предполагать, что все функции $g_k(y)$, $1 \leq k \leq m+1$ 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

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

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 $g_1(x), \dots, g_\nu(x)$, где значение индекса ν определяется условиями: $x \in Q_j, 0 \leq j < \nu, x \notin Q_\nu$. 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 $1 \leq \nu \leq m + 1$, называется результатом испытания в точке x .

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

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

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 $\psi(x)$ гёльдеровы 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 $\{(x_k, \nu_k, z_k)\}, 1 \leq k \leq n$ 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 $k \geq 1$ итераций метода, в процессе которых были проведены испытания в k точках $x_i, 1 \leq i \leq k$. Тогда точка x^{k+1} поисковых испытаний следующей $(k+1)$ -ой итерации определяются в соответствии с правилами:

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

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

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

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

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

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

Шаг 3. Вычислить текущие оценки для неизвестных констант Гёльдера:

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

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 $\varepsilon_R = (\varepsilon_1, \dots, \varepsilon_m)$ называется вектором резервов.

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

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

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:

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

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

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

а затем увеличить k на 1.

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

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

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 останова при минимизации функции из данного класса, а P – статистическая вероятность того, что к моменту останова 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 заранее известными свойствами. Это достигается за счёт модификации параболоида $g(x) = \|x - T\| + t$ в шаровых окрестностях некоторых случайно сгенерированных точек $M_i, i = \overline{1, m}$. В точках M_i располагаются локальные минимумы со значениями, превосходящими значение $g(T) = t$. Таким образом, координаты глобального минимума в задачах GKLS всегда заведомо известны. В данной работе используется несколько классов, сгенерированный GKLS: 2d Simple, 4d Simple и 5d Simple, параметры которых также описаны в [6]. Функции рассматриваемых классов 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 двумерных функций, предложенных В. А. Гришагиным [7]. Каждая функция существенно многоэкстремальна и задаётся формулой:

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

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 локального поиска для ускорения сходимости АГП

Методы локального поиска могут применяться в сочетании с глобальными алгоритмами для улучшения полученных решений или текущих оценок оптимума. В первом случае локальный метод стартует из точки, найденной глобальным методом, и уточняет решение практически до любой нужной точности. Это позволяет избежать чрезмерных затрат на поиск решения с высокой точностью глобальным методом.

Во втором случае локальный метод используется для ускорения обнаружения локальных оптимумов. Информационно-статистический метод Стронгина позволяет обновлять свою поисковую информацию из любых посторонних источников, в том числе из точек испытаний, полученных от локального метода. Как только глобальный метод находит новую оценку оптимума, из этой точки стартует локальный метод и все или часть испытаний, проведённых им добавляется в поисковую информацию, далее глобальный метод продолжает работу. Каких-либо теоретических исследований подобной схемы не проводилось, поэтому её эффективность проверялась экспериментально.

В качестве метода локальной оптимизации был выбран метод Хука-Дживса [8]. Он прост в реализации и для его работы не требуется знать значений производных оптимизируемой функции.

Были проведены две серии экспериментов, соответствующих следующим схемам добавления точек, полученных локальным методом в поисковую информацию:

- добавление единственной точки, к которой сошёлся локальный метод;
- добавление всех промежуточных точек.

Эксперименты проводились на классах GKLS 4d Simple и GKLS 5d Simple, параметры метода были заданы такие же, как в разделе 6. Из рис. 4.3, 4.4 можно сделать вывод, что вариант с использованием только одной лучшей точки, полученной локальным методом, оказался наилучшим. В обоих случаях он заметно ускорил сходимость глобального алгоритма.

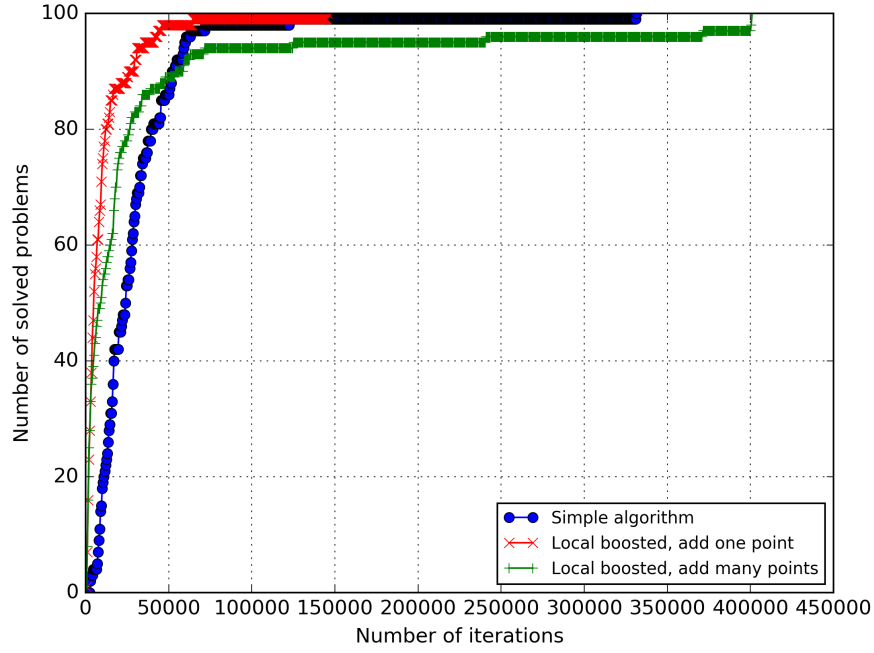


Рис. 4.3: Операционные характеристики на классе GKLS 4d Simple при различных вариантах использования локального метода

5. Смешанный алгоритм глобального поиска и его эффективная реализация

Ещё одной модификацией метода Стронгина, позволяющей при оптимизации лучше учитывать данные о локальных оптимумах, найденных в процессе поиска, является смешанный алгоритм Стронгина-Маркина [9]. Наряду с характеристикой интервала $R(i)$ (3.3) можно рассматривать $R^*(i)$, которая будет более чувствительна к наличию в интервале текущего найденного минимума функции x_k^* :

$$R^*(i) = \frac{R(i)}{\sqrt{(z_i - z^*)(z_{i-1} - z^*)/\mu + 1.5^{-\alpha}}}$$

где $f(x_k^*) = z^*$, а $\alpha \in [1; 30]$ — степень локальности. Чем она больше, тем более высокая характеристика у интервала, содержащего x_k^* , по сравнению с остальными.

Смешанный алгоритм состоит в следующем: в процессе работы метода каждые S итераций интервал для последующего разбиения выбирается по характеристикам $R^*(i)$. S — параметр смешивания. Такой подход позволяет существенно ускорить сходимость метода. На рис. 5.5 приведены операционные характеристики чисто глобального и смешанного алгоритма на классе GKLS 4d Simple. Параметр смешивания S равен 5, $\alpha = 15$, остальные параметры метода были заданы такие же, как в разделе 6.

Из-за того, что интервал имеет сразу две характеристики, появляется проблема эффективной реализации смешанного алгоритма. Если интервал имеет одну характеристику, то для выбора максимальной достаточно организовать приоритетную очередь характеристик [10]. Причём перезаполнение такой очереди необходимо не на каждой итерации: в большинстве случаев

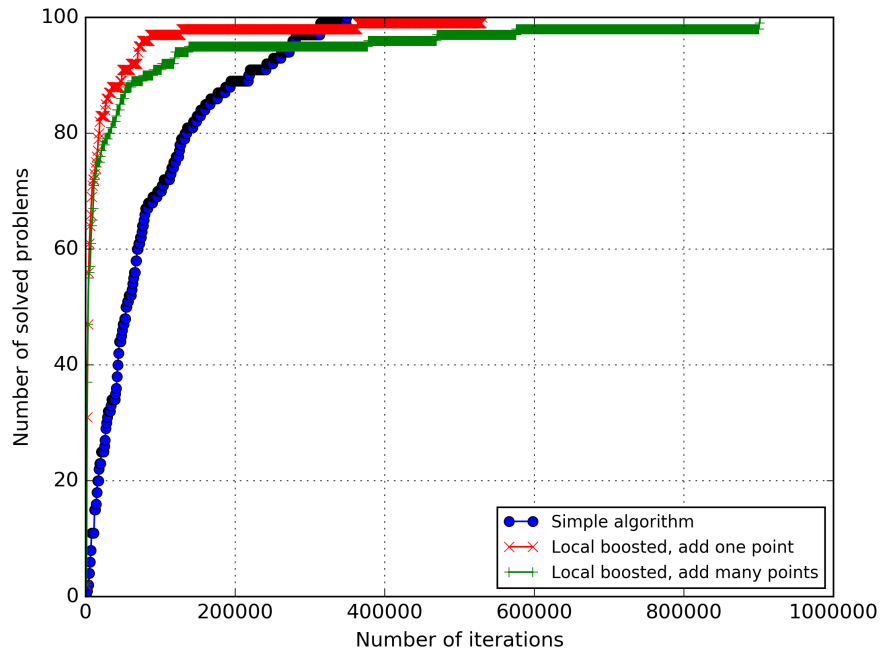


Рис. 4.4: Операционные характеристики на классе GKLS 5d Simple при различных вариантах использования локального метода

характеристики интервалов не меняются, достаточно удалить разбиваемый интервал и вставить в очередь два новых. Такая организация работы метода позволяет существенно сократить объём вычислений. При наличии у интервала двух характеристик можно организовать две связанные очереди. В этом случае необходимо предусмотреть процедуру синхронизации двух очередей.

Перечислим операции, при которых необходима синхронизация:

- вставка интервала сразу в обе очереди;
- удаление интервала из какой-либо очереди.

Синхронизация достигается путём введения перекрёстных ссылок между элементами очередей. На рис. 5.6 приведена схема связанных очередей. Элемент очереди представляет собой совокупность ключа (**LocalR** или **R**), указателя на интервал (**pInterval**) и указателя на элемент связанной очереди, соответствующий тому же интервалу (**pLinkedElement**). Опишем подробнее алгоритмы вставки и удаления элементов.

Вставка элемента в пару связанных очередей:

1. Попытаться вставить элемент в очередь глобальных характеристик (он может быть не вставлен, если имеет слишком низкий приоритет).
2. Попытаться вставить элемент в очередь локальных характеристик (он может быть не вставлен, если имеет слишком низкий приоритет).
3. Если элемент интервал вставлен в обе очереди, то выставить перекрёстные ссылки.

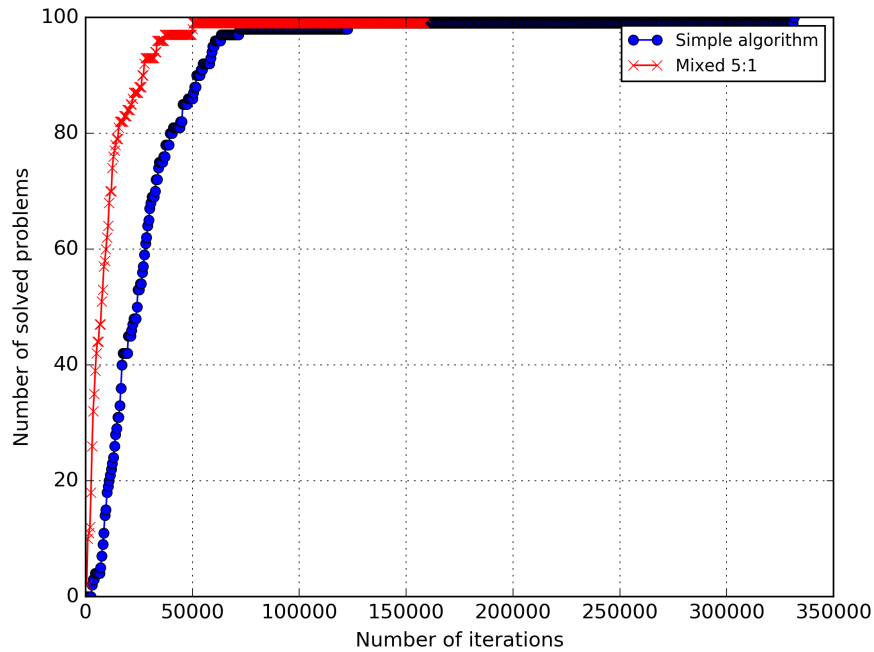


Рис. 5.5: Операционные характеристики обычного и смешанного АГП на классе GKLS 4d Simple

Удаление элемента с минимальным ключом:

1. Удалить элемент с минимальным ключом из очереди, запомнить указатель **pLinkedElement**.
2. Если **pLinkedElement** ненулевой, то вызвать процедуру удаления элемента, на который указывает **pLinkedElement** в структуре данных, хранящей связанную очередь.

Последний момент, который надо учесть при реализации: вставка или удаление элемента очереди приводит к тому, что необходимо восстановить её внутреннюю структуру. Если очередь хранится в куче, то восстанавливается свойство кучеобразности. Во время этого процесса требуется производить попарные перестановки элементов, а значит, необходимо обновлять ссылки на эти элементы в связанной очереди.

Стоит заметить, что внесённые модификации (в основном эта работа со ссылками) не увеличивают асимптотическую сложность выполнения операций вставки и удаления по сравнению с единственной очередью.

6. Многоуровневая схема редукции размерности с помощью разверток

Теоретически с помощью развёрток можно решить задачу любой размерности, однако на ЭВМ развёртка строится с помощью конечноразрядной арифметики, из-за чего, начиная с некоторого N^* , построение разветки невозможно (значение N^* зависит от максимального количества значащих разрядов в арифметике с плавающей точкой). Понять почему это происходит нетрудно, обратившись, например к [4].

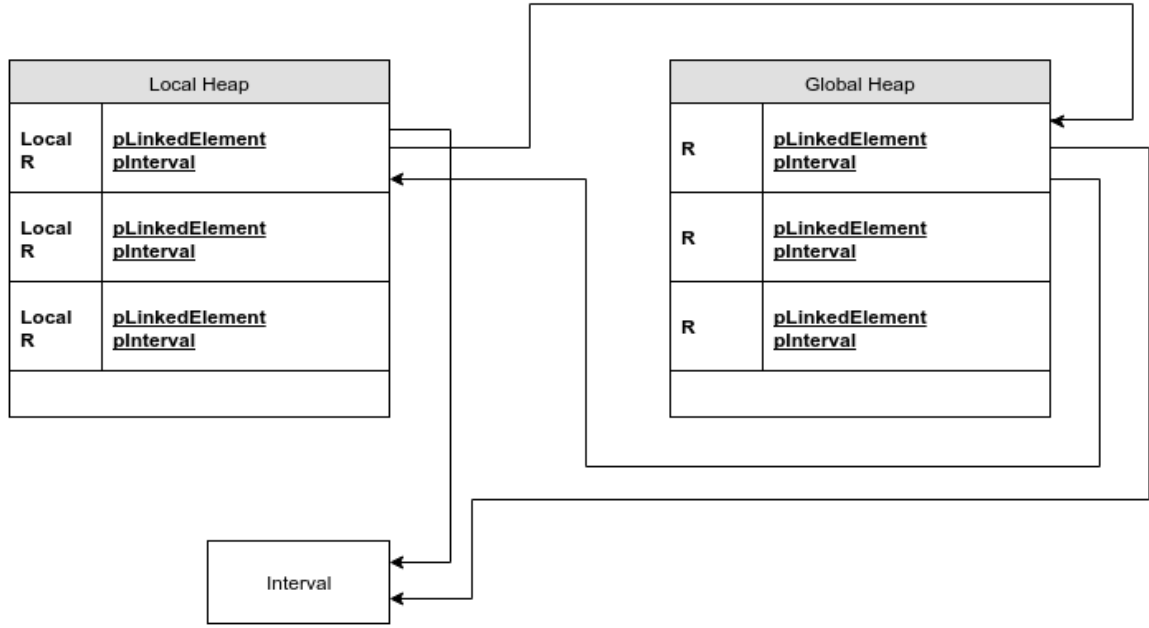


Рис. 5.6: Схема устройства связанных очередей

Чтобы преодолеть эту проблему профессором В. П. Гергелем была предложена следующая идея: использовать композицию развёрток меньшей размерности для построения отображения $z(x) : [0; 1] \rightarrow D \in R^N$. Поясним эту схему на примере редукции размерности в четырёхмерной задаче. Пусть $y_2(x)$ — двухмерная развёртка (отображает отрезок в прямоугольник), тогда рассмотрим функцию $\psi(x_1, x_2) = \varphi(y_2(x_1), y_2(x_2))$. К $\psi(x_1, x_2)$ можно также применить редукцию размерности с помощью развёртки. Таким образом, задав точку $x^* \in [0; 1]$, вычислив $y_2(x^*) = (x_1, x_2)$ и пару векторов $(y_2(x_1), y_2(x_2))$, получим четырёхмерную точку. Из инъективности $y_2(x)$ следует инъективность $z(x)$.

Проблемой этого подхода является выяснение свойств функции $\varphi(z(x))$ и возможности использования одномерного метода Стронгина с гёльдеровой метрикой для оптимизации $\varphi(z(x))$. Чтобы не тратить время на теоретическое исследование, были проведены численные эксперименты с целью оценить возможность применения многоуровневой развёртки в четырёхмерном случае.

Прежде всего, рассмотрим линии уровня функции $\psi(x_1, x_2) = \varphi(y_2(x_1), y_2(x_2))$ при $\varphi(t) = \sum_{i=1}^4 (t_i - 0.5)^2$. Как видно из рис. 6.7, линии уровня имеют довольно сложную структуру, что говорит о возможных сложностях применения одномерного метода с развёрткой.

Далее был проведён более масштабный вычислительный эксперимент: с помощью многоуровневой развёртки решались 100 задач из класса GKLS 4d Simple. На рис. 6.8 приведены операционные характеристики метода с простой и многоуровневой развёртками. При этом были зафиксированы следующие параметры алгоритма: надёжность $r = 4.5$, плотность построения всех развёрток 12, критерий остановки попадание точки, поставленной методом в квадрат со стороной $\varepsilon = 10^{-2}$, центром которого является решение задачи. Как видно из графика, операционная характеристика метода с многоуровневой развёрткой лежит выше, чем анало-

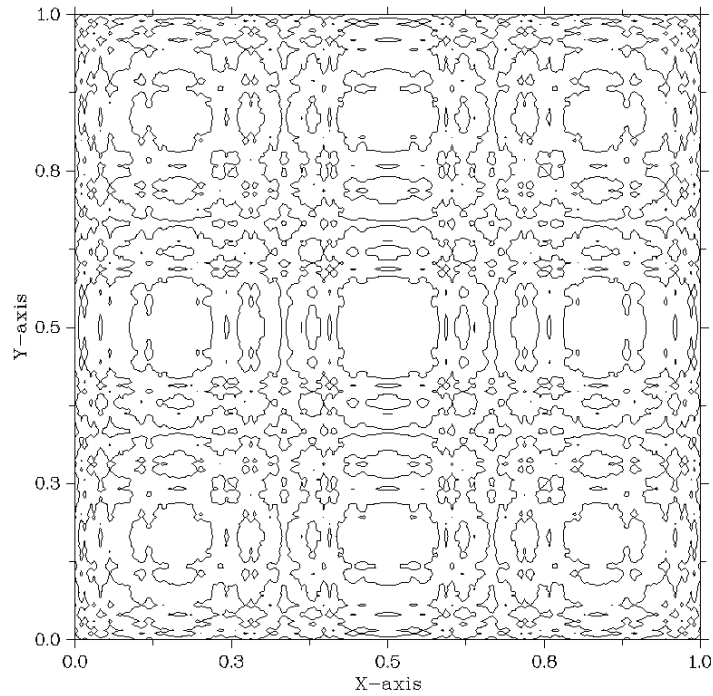


Рис. 6.7: Линии уровня функции $\psi(x_1, x_2) = \varphi(y_2(x_1), y_2(x_2))$

гичная кривая метода с простой развёрткой. Кроме того, метод с многоуровневой развёрткой заметно раньше вышел на стопроцентную надёжность на решаемой выборке задач. Исходя из первых экспериментов можно сказать, что при данных параметрах алгоритма многоуровневая развёртка лучше, но при уменьшении точности алгоритма ε до значения 10^{-3} выполнения критерия остановки в случае использования многоуровневой развёртки с выбранной плотностью построения внутренних развёрток не происходит. Если плотность увеличить до 16, то остановка также не будет происходить. Исходя из этого можно делать вывод, что использование на практике многоуровневых развёрток с большой долей вероятности невозможно из-за описанного дефекта сходимости.

7. 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 [11] было предложено использовать различные значения $M = \mu_1$ (μ_1 из (3.2) в случае отсутствия функциональных ограничений), (3.3)) для каждого интервала, а также показана эффективность такого подхода в случае одномерной оптимизации функций, удовлетворяющих условию Липшица. В работе [12] рассмотрено применение адаптивных оценок констант Липшица в схеме многомерной вложенной оптимизации.

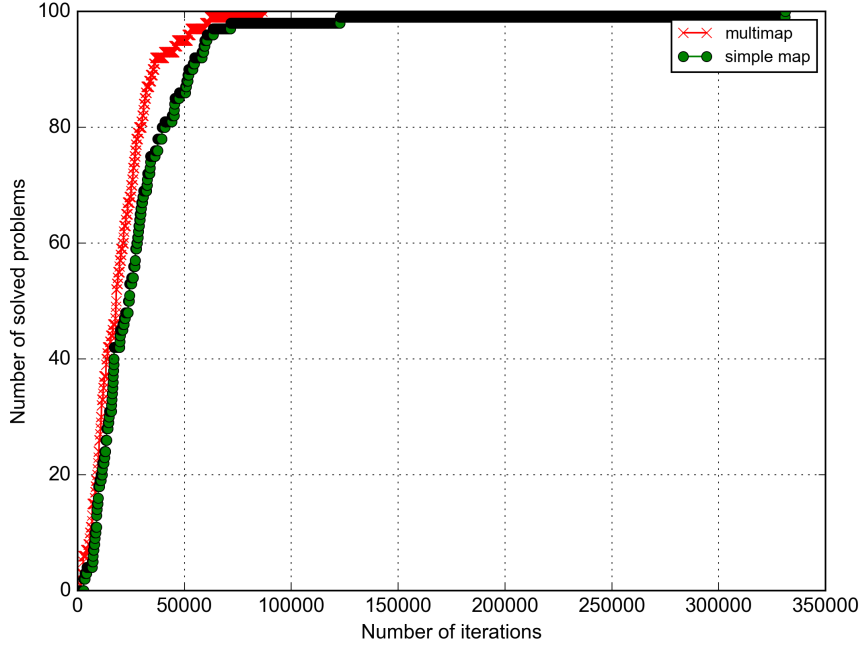


Рис. 6.8: Операционная характеристика метода с различными развёртками на классе GKLS 4d Simple

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

$$\lambda_i = \max\{H_{i-1}, H_i, H_{i+1}\}$$

$$H_i = \frac{|z_i - z_{i-1}|}{\Delta_i}$$

$$H^k = \max\{H_i : i = 2, \dots, k\}$$

$$\gamma_i = H^k \frac{\Delta_i}{\Delta^{max}}$$

$$\Delta^{max} = \max\{\Delta_i : i = 2, \dots, k\}$$

$$M_i = r \cdot \max\{H_i, \frac{1}{2}(\lambda_i + \gamma_i), \xi\} \quad (7.6)$$

Параметр ξ предотвращает обнуление оценки M_i в случае, если оптимизируемая функция является тождественной константой, и выбирается достаточно малым. Данный вариант свёртки не зависит от параметра r , однако в [13] также рассматривается адаптивная свёртка:

$$M_i = r \cdot \max\{H_i, \frac{\lambda_i}{r} + \frac{r-1}{r}\gamma_i, \xi\} \quad (7.7)$$

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» составляющей γ .

Оба варианта свёртки (7.6), (7.7) были предложены и детально рассмотрены в [13] для случая одномерных функций, удовлетворяющих условию Липшица. В данном разделе будет рас-

смотрено применение этих свёрток для оптимизации двумерных функций в случае редукции размерности с помощью развёрток.

В [13] 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 гипотеза:

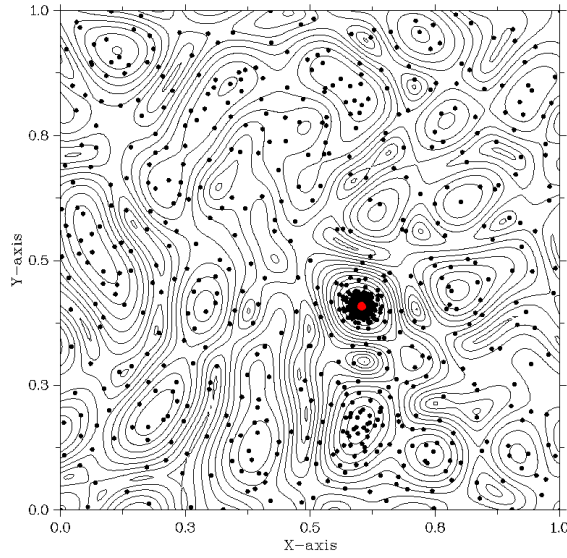
Гипотеза 1 Пусть целевая функция $f(x)$ удовлетворяет условию Гёльдера с конечной константой $H > 0$, и пусть x является предельной точкой последовательности $\{x_k\}$, порождаемой алгоритмом. Тогда верны следующие утверждения:

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. $f(x_k) \geq f(x)$ для всех точек испытаний $x_k, k \geq 1$;
3. Если существует другая предельная 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 глобальному 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 $f(x)$.

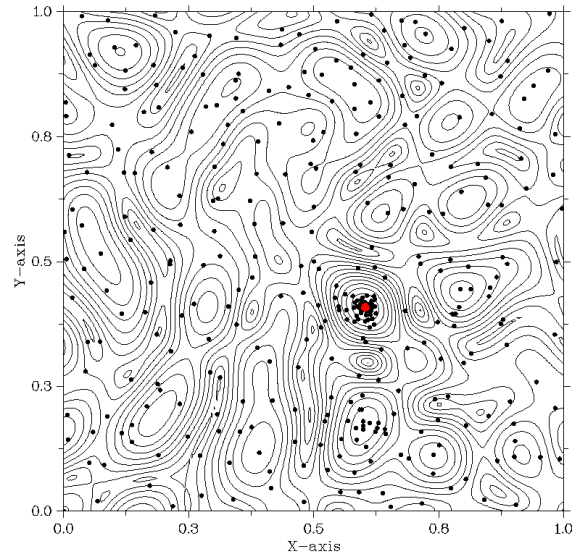
Доказательство гипотезы требует отдельных теоретических исследований. В рамках данной работы оно не будет проведено, наличие сходимости установлено только численно.

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 останова 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 преимуществ локально-адаптивной схемы оценки константы H рассмотрим результаты работы метода на конкретном примере. На рис. 7.9 показаны линии уровня одной из функций класса F_{GR} и точки испытаний, проведённых методом с глобальной оценкой константы Гёльдера и с оценкой по формуле (7.6). 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 останова,



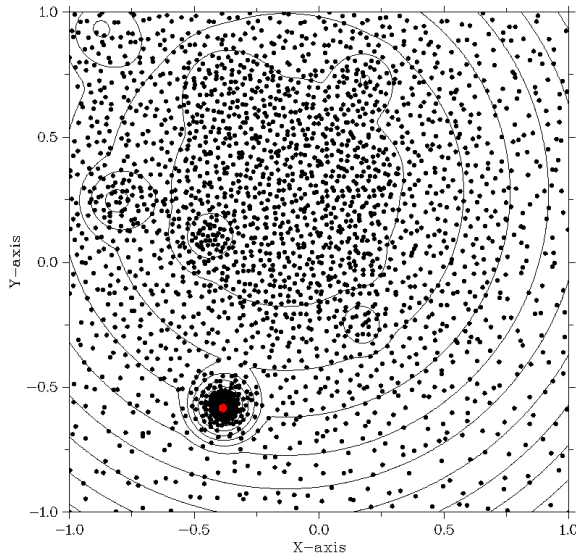
(a) глобальная оценка H



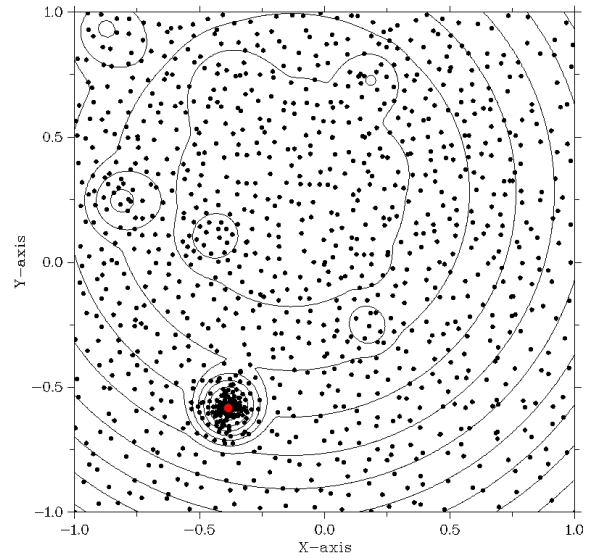
(b) локально-адаптивная оценка H

Рис. 7.9: Линии уровня одной из функций класса F_{GR}

в то время, 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. 7.10). 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. 7.10: Линии уровня одной из функций класса GKLS Simple 2d

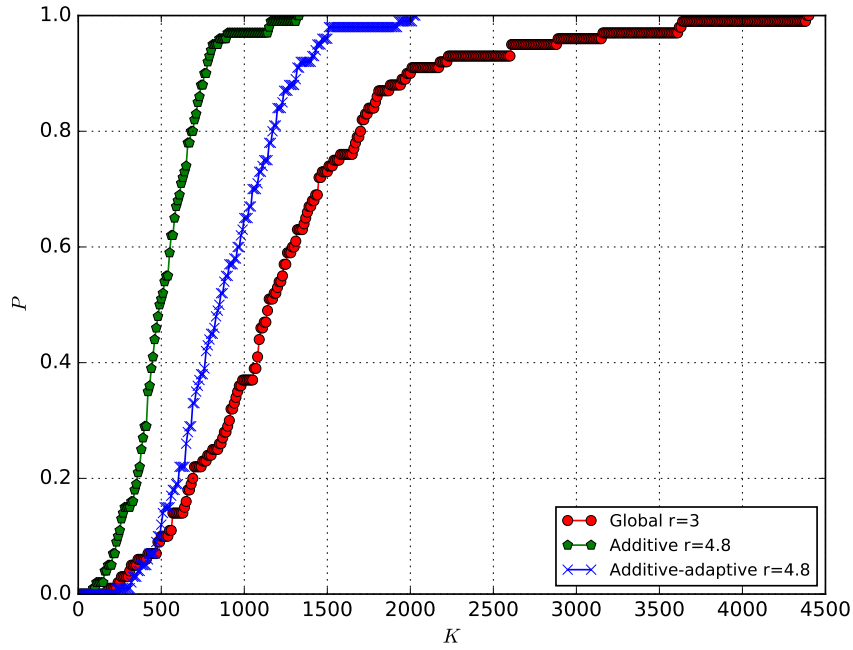


Рис. 7.11: Операционные характеристики методов на классе задач F_{GR}

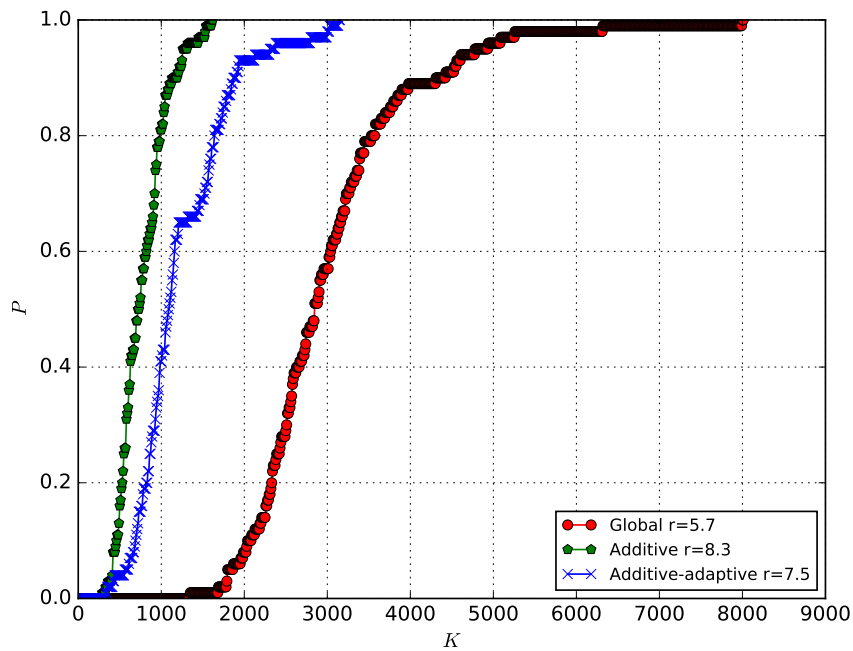


Рис. 7.12: Операционные характеристики методов на классе задач GKLS Simple 2d

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. 7.11 и 7.12, 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 (7.6), (7.7), то наглядно видно преимущество последнего, хотя он и требует большее значение параметра надёжности r для решения задач из класса GKLS.

8. 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:

$$\dot{x} = (A + B_u \Theta)x + B_v v, x(0) = 0, \quad (8.8)$$

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

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

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

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

В [14] указан способ вычисления критериев, состоящий в следующем:

- найти матрицу Y из уравнения:

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

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

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

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 критерий устойчивости линейной системы ОДУ: все действительные части собственных чисел матрицы $A + B_u \Theta$ должны быть отрицательны:

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

С целью проверки корректности реализации вычисления критериев задачи индексным алгоритмом глобального поиска были решены две задачи рассматриваемого типа (их решение также приведено и в [14]).

В задаче виброзащиты параметры системы (8.8) определяются 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 $\theta_1 \leq 0, \theta_2 \leq 0$ — оптимизируемые параметры. Сама задача ставится следующим образом:

$$J_2(\Theta^*) = \min\{J_2(\Theta) : J_1(\Theta) \leq 1, g_0(\Theta) \leq -0.02\}. \quad (8.9)$$

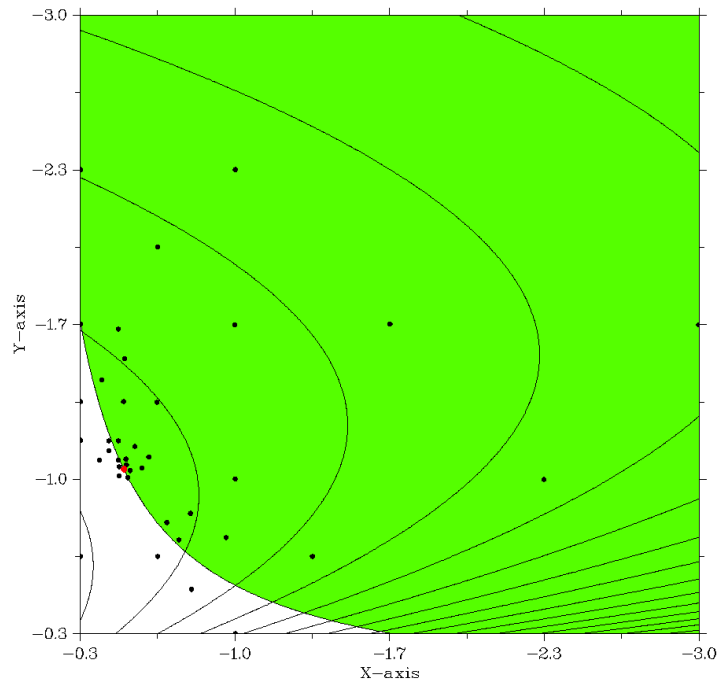


Рис. 8.13: Линии уровня для задачи виброзащиты с отмеченными точками испытаний АГП

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 $\tilde{\theta}_1 = -0.503223, \tilde{\theta}_2 = -0.997168, J_2(\tilde{\Theta}) = 0.866551$, а ограничение на J_1 активно. На рис. 8.13 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 (8.8) определяются 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:

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

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

Рассматриваемые задачи поиска оптимального управления по состоянию интересны, прежде всего, в многокритериальной постановке. В данной работе рассматриваются задачи с двумя критериями: один отвечает за максимальное смещение колебательного объекта, а другой — за максимальное управляющее воздействие. Для отыскания Парето-границы на плоскости критериев достаточно воспользоваться постановкой (8.9). 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 неподвижных 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 двухмассовую 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 матрицы из (8.8) определяются следующим образом:

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

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

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

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

В [14] 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. 8.14. 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.

Построенная с помощью системы Globalizer описанным ранее способом Парето-граница показана на рис. 8.14 (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 коэффициентах обратной связи не критическая.

9. Заключение

В ходе работы были получены следующие практические результаты:

- реализован метод локальной оптимизации Хука-Дживса (код в приложении 10.3.);
- в системе Globalizer реализованы различные стратегии использования локального поиска, исследована их эффективность;

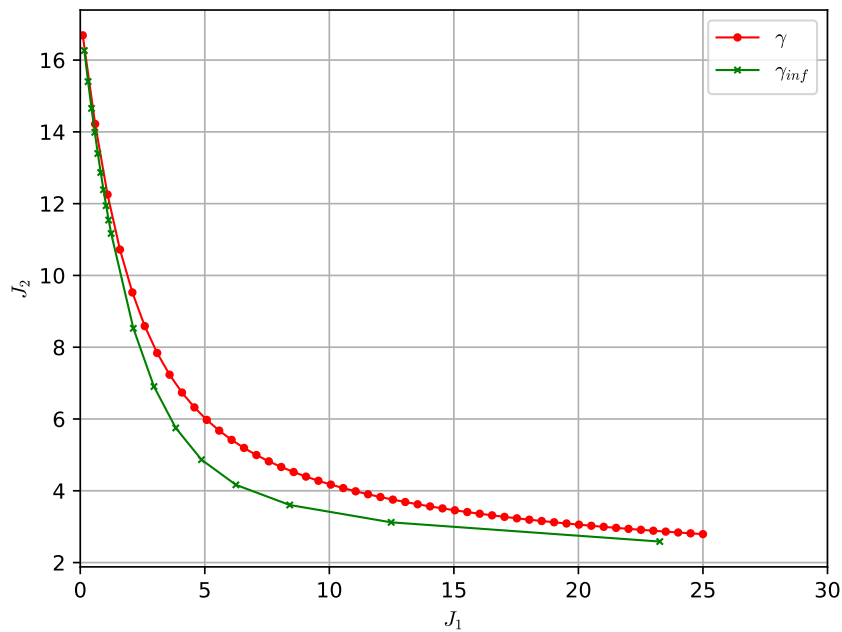


Рис. 8.14: Парето-границы в задаче виброизоляции при $n = 10$, построенные для частично и полностью наблюдаемого состояния

- в рамках Globalizer реализована поддержка смешанного алгоритма глобального поиска, а также эффективные структуры данных, необходимые для этого алгоритма (фрагменты кода в приложении 10.4.)
- был проведён отдельный эксперимент для выяснения возможности практического использования многоуровневых развёрток.
- 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 10.1.;
- реализована схема вычисления целевой функции в задаче поиска оптимального управления, описанной в разделе 8. При реализации использовалась библиотека Eigen [15] для решения задач линейной алгебры. Исходный код можно найти в приложении 10.2. Два варианта задачи поиска оптимального управления, приведённые в [14], 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] Химмельблау Д. М. *Прикладное нелинейное программирование*. Издательство МИР, Москва, 1975.
- [9] Д. Л. Маркин, Р. Г. Стронгин. «Метод решения многоэкстремальных задач с невыпуклыми ограничениями, использующий априорную информацию об оценках оптимума». В: *Ж. вычисл. матем. и матем. физ.* 27:1 (1987), 52–62.
- [10] M. D. Atkinson, J.R. Sack, N. Santoro, T. Strothotte. «Min-Max Heaps and Generalized Priority Queues». В: *Communications of the ACM* 29 (1986), с. 996—1000.
- [11] Sergeyev, Ya.D. «An information global optimization algorithm with local tuning». В: *SIAM Journal on Optimization* 5.4 (1995), с. 390—407.
- [12] Gergel, V., Grishagin, V., Israfilov, R. «Local tuning in nested scheme of global optimization». В: *Procedia Computer Science* 51 (2015), с. 865—874.
- [13] Sergeyev, Y.D., Mukhametzhonov, 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.
- [14] Д.В. Баландин М.М. Коган. «Оптимальное по Парето обобщенное H_2 -управление и задачи виброзащиты». В: *Автоматика и телемеханика* (2017).
- [15] *API documentation for Eigen3*. <https://eigen.tuxfamily.org/dox/>. [Online; accessed 26-December-2016].

10. Приложения

10.1. Приложение 1

```
1 #ifndef OPTIMIZER_ALGORITHM_UNCONSTRAINED_HPP
2 #define OPTIMIZER_ALGORITHM_UNCONSTRAINED_HPP
3
4 #include "OptimizerCoreGlobal.hpp"
5 #include "OptimizerTask.hpp"
6 #include "OptimizerSolution.hpp"
7 #include "OptimizerFunction.hpp"
8 #include "OptimizerDataStructures.hpp"
9 #include "OptimizerSolution.hpp"
10 #include "OptimizerResult.hpp"
11 #include "OptimizerSearchSequence.hpp"
12
13 #include <set>
14
15 namespace optimizercore
16 {
17
18     class EXPORT_API OptimizerAlgorithmUnconstrained final
19     {
20
21     private:
22
23         bool mLocalMixType;
24         bool mIsAlgorithmMemoryAllocated;
25         bool mIsParamsInitialized;
26         bool mIsTaskInitialized;
27         bool mNeedLocalVerification;
28
29         int mNumberOfThreads;
30         int mLocalStartIterationNumber;
31         int mMaxNumberOfIterations;
32         int mMapTightness;
33         int mMethodDimension;
34         int mAlpha;
35         int mLocalMixParameter;
36         int mMapType;
37
38         OptimizerSpaceTransformation mSpaceTransform;
39         OptimizerFunction *mTargetFunction;
40         OptimizerFunctionPtr mTargetFunctionSmartPtr;
41
42         OptimizerInterval *mIntervalsForTrials;
43         std::set<OptimizerTrialPoint> mSearchInformationStorage;
44         OptimizerTrialPoint mOptimumEvaluation, *mNextTrialsPoints;
45
46         LocalTuningMode mLocalTuningMode;
47
48         double mGlobalM, mZ, eps, r, mMaxIntervalNorm;
49         double **mNextPoints;
50
51         void AllocMem();
52         void InitializeInformationStorage();
53         void UpdateGlobalM(std::set<OptimizerTrialPoint>::iterator&);
54         int UpdateRanks(bool isLocal);
55         bool InsertNewTrials(int trialsNumber);
56         OptimizerSolution DoLocalVerification(OptimizerSolution startPoint);
57     }
```

```

58 public:
59     OptimizerAlgorithmUnconstrained();
60     ~OptimizerAlgorithmUnconstrained();
61
62     void SetTask(OptimizerFunctionPtr function,
63                 OptimizerSpaceTransformation spaceTransform);
64     void SetThreadsNum(int num);
65     void SetParameters(OptimizerParameters params);
66
67     OptimizerResult StartOptimization(const double* xOpt,
68                                     StopCriterionType stopType);
69
70     double GetLipschitzConst() const;
71     OptimizerSearchSequence GetSearchSequence() const;
72
73 };
74 }
75 #endif

```

```

1  #include "OptimizerAlgorithmUnconstrained.hpp"
2  #include "HookeJeevesLocalMethod.hpp"
3
4  #include <cassert>
5  #include <algorithm>
6
7  using namespace optimizercore;
8  using namespace optimizercore::utils;
9
10 OptimizerAlgorithmUnconstrained::OptimizerAlgorithmUnconstrained()
11 {
12     mIsAlgorithmMemoryAllocated = false;
13
14     mLocalStartIterationNumber = 1;
15     mNumberOfThreads = 1;
16     mMaxNumberOfIterations = 5000;
17     mNextPoints = nullptr;
18     mNextTrialsPoints = nullptr;
19     mIntervalsForTrials = nullptr;
20     r = 2;
21     mLocalTuningMode = LocalTuningMode::None;
22
23     mIsTaskInitialized = false;
24     mIsParamsInitialized = false;
25 }
26
27 void OptimizerAlgorithmUnconstrained::SetTask(OptimizerFunctionPtr function,
28 OptimizerSpaceTransformation spaceTransform)
29 {
30     assert(function);
31
32     mTargetFunctionSmartPtr = function;
33     mTargetFunction = function.get();
34     mSpaceTransform = spaceTransform;
35
36     mIsTaskInitialized = true;
37 }
38
39 OptimizerSearchSequence OptimizerAlgorithmUnconstrained::GetSearchSequence()
40 const

```

```

40 {
41     return OptimizerSearchSequence(mSearchInformationStorage, mMethodDimension,
42         static_cast<MapType> (mMapType), mMapTightness, mSpaceTransform);
43 }
44
45 double OptimizerAlgorithmUnconstrained::GetLipschitzConst() const
46 {
47     return mGlobalM;
48 }
49
50 void OptimizerAlgorithmUnconstrained::SetParameters(OptimizerParameters params
51 )
52 {
53     assert(params.algDimention);
54     assert(params.eps > 0);
55     assert(params.localAlgStartIterationNumber > 0);
56     assert(params.mapTightness > 5 && params.mapTightness <= 20);
57     assert(params.maxIterationsNumber > 0);
58     assert(params.localMixParameter >= 0 && params.localMixParameter <= 20);
59     assert(params.r != nullptr);
60     assert(params.numberOfThreads > 0);
61     assert(params.reserves != nullptr);
62     assert(params.numberOfMaps > 0);
63
64     mLocalStartIterationNumber = params.localAlgStartIterationNumber;
65     eps = params.eps;
66     if (params.localMixParameter <= 10) {
67         mLocalMixParameter = params.localMixParameter;
68         mLocalMixType = true;
69     }
70     else {
71         mLocalMixParameter = 20 - params.localMixParameter;
72         mLocalMixType = false;
73     }
74     mNeedLocalVerification = params.localVerification;
75     mAlpha = params.localExponent;
76     mMethodDimension = params.algDimention;
77     mMapTightness = params.mapTightness;
78     mMapType = static_cast<int>(params.mapType);
79     mMaxNumberOfIterations = params.maxIterationsNumber;
80     mLocalTuningMode = params.localTuningMode;
81     r = *params.r;
82     if (mNextPoints)
83         utils::DeleteMatrix(mNextPoints, mNumberOfThreads);
84     mNextPoints = utils::AllocateMatrix<double>(mNumberOfThreads,
85         mMethodDimension);
86     this->SetThreadsNum(params.numberOfThreads);
87
88     mIsParamsInitialized = true;
89 }
90
91 void OptimizerAlgorithmUnconstrained::InitializeInformationStorage()
92 {
93     if (!mIsAlgorithmMemoryAllocated){
94         AllocMem();
95         mIsAlgorithmMemoryAllocated = true;
96     }
97
98     mZ = HUGE_VAL;
99     mGlobalM = 1;
100     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(
105         mNextPoints[0]), 0);
106
107     mapd(1.0, mMapTightness, mNextPoints[0], mMethodDimension, mMapType);
108     mSpaceTransform.Transform(mNextPoints[0], mNextPoints[0]);
109     mSearchInformationStorage.emplace(1.0, mTargetFunction->Calculate(
110         mNextPoints[0]), 0);
111 }
112
113 bool OptimizerAlgorithmUnconstrained::InsertNewTrials(int trailsNumber)
114 {
115     bool storageInsertionError;
116     if (mMapType == 3)
117     {
118         int preimagesNumber = 0;
119         double preimages[32];
120         for (int i = 0; i < trailsNumber; i++)
121         {
122             invmad(mMapTightness, preimages, 32,
123                 &preimagesNumber, mNextPoints[i], mMethodDimension, 4);
124             for (int k = 0; k < preimagesNumber; k++)
125             {
126                 mNextTrialsPoints[i].x = preimages[k];
127                 auto insertionResult =
128                     mSearchInformationStorage.insert(mNextTrialsPoints[i]);
129
130                 if (!(storageInsertionError = insertionResult.second))
131                     break;
132
133                 UpdateGlobalM(insertionResult.first);
134             }
135         }
136     }
137     else
138     {
139         for (int i = 0; i < trailsNumber; i++)
140         {
141             auto insertionResult =
142                 mSearchInformationStorage.insert(mNextTrialsPoints[i]);
143
144             if (!(storageInsertionError = insertionResult.second))
145                 break;
146
147             UpdateGlobalM(insertionResult.first);
148         }
149     }
150     return storageInsertionError;
151 }
152
153 OptimizerResult OptimizerAlgorithmUnconstrained::StartOptimization(
154     const double* a, StopCriterionType stopType)
155 {
156     assert(mIsParamsInitialized && mIsTaskInitialized);
157     assert(mSpaceTransform.GetDomainDimension() == mMethodDimension);
158
159     InitializeInformationStorage();
160
161     double *y;

```

```

158     bool stop = false;
159     int iterationsCount = 0,
160         currentThrNum = 1, ranksUpdateErrCode;
161
162     mNextTrialsPoints[0].x = 0.5;
163     mapd(mNextTrialsPoints[0].x, mMapTightness, mNextPoints[0],
164         mMethodDimension, mMapType);
165     mSpaceTransform.Transform(mNextPoints[0], mNextPoints[0]);
166
167     while (iterationsCount < mMaxNumberOfIterations && !stop) {
168         iterationsCount++;
169
170     #pragma omp parallel for num_threads(currentThrNum)
171         for (int i = 0; i < currentThrNum; i++) {
172             mNextTrialsPoints[i].val = mTargetFunction->Calculate(mNextPoints[i]);
173             if (mMapType == 3)
174                 mSpaceTransform.InvertTransform(mNextPoints[i], mNextPoints[i]);
175     #pragma omp critical
176         if (mNextTrialsPoints[i].val < mZ)
177             mZ = mNextTrialsPoints[i].val;
178         }
179
180         if (!InsertNewTrials(currentThrNum))
181             break;
182
183         if (iterationsCount >= mLocalStartIterationNumber) {
184             if (iterationsCount % (12 - mLocalMixParameter) == 0
185                 && mLocalMixParameter > 0)
186                 ranksUpdateErrCode = UpdateRanks(mLocalMixType);
187             else
188                 ranksUpdateErrCode = UpdateRanks(!mLocalMixType);
189         }
190         else
191             ranksUpdateErrCode = UpdateRanks(false);
192
193         if (iterationsCount >= mNumberOfThreads + 10)
194             currentThrNum = mNumberOfThreads;
195
196         for (int i = 0; i < currentThrNum && !stop; i++) {
197             OptimizerTrialPoint left = mIntervalsForTrials[i].left;
198             OptimizerTrialPoint right = mIntervalsForTrials[i].right;
199
200             mNextTrialsPoints[i].x = (left.x + right.x) / 2
201                 - sgn(right.val - left.val)*pow(fabs(right.val - left.val)
202                 / mIntervalsForTrials[i].localM, mMethodDimension) / (2 * r);
203
204             mapd(mNextTrialsPoints[i].x, mMapTightness, mNextPoints[i],
205                 mMethodDimension, mMapType);
206             mSpaceTransform.Transform(mNextPoints[i], mNextPoints[i]);
207
208             y = mNextPoints[i];
209
210             if (stopType == StopCriterionType::OptimalPoint) {
211                 if (NormNDimMax(y, a, mMethodDimension) < eps) {
212                     stop = true;
213                     mOptimumEvaluation = mNextTrialsPoints[i];
214                 }
215             }
216             else {
217                 if (pow(right.x - left.x, 1.0 / mMethodDimension) < eps) {
218                     stop = true;

```

```

219         mOptimumEvaluation = mNextTrialsPoints[i];
220     }
221 }
222 }
223 }
224
225 mOptimumEvaluation.val = mTargetFunction->Calculate(y);
226 mSearchInformationStorage.insert(mOptimumEvaluation);
227
228 if (stopType == StopCriterionType::Precision)
229     mOptimumEvaluation = *std::min_element(mSearchInformationStorage.begin(),
230     mSearchInformationStorage.cend(),
231     [](OptimizerTrialPoint p1, OptimizerTrialPoint p2)
232     {
233         return p1.val < p2.val;
234     });
235
236 mapd(mOptimumEvaluation.x, mMapTightness, y, mMethodDimension, mMapType);
237 mSpaceTransform.Transform(y, y);
238
239 SharedVector optPoint(new double[mMethodDimension], array_deleter<double>())
240 ;
241 std::memcpy(optPoint.get(), y, mMethodDimension*sizeof(double));
242
243 OptimizerSolution solution(iterationsCount, mOptimumEvaluation.val,
244     mOptimumEvaluation.x, mMethodDimension, optPoint);
245
246 if (mNeedLocalVerification)
247     return OptimizerResult(DoLocalVerification(solution));
248 else
249     return OptimizerResult(solution);
250 }
251 OptimizerSolution OptimizerAlgorithmUnconstrained::DoLocalVerification(
252     OptimizerSolution startSolution)
253 {
254     OptimizerFunctionPtr *functions = new OptimizerFunctionPtr[1];
255     functions[0] = mTargetFunctionSmartPtr;
256
257     OptimizerTask localTask(std::shared_ptr<OptimizerFunctionPtr>(functions,
258     utils::array_deleter<OptimizerFunctionPtr>()),
259     0, mMethodDimension, mSpaceTransform.GetLeftDomainBound(),
260     mSpaceTransform.GetRightDomainBound());
261
262     localoptimizer::HookeJeevesLocalMethod localMethod;
263     localMethod.SetEps(eps / 100);
264     localMethod.SetInitialStep(2 * eps);
265     localMethod.SetProblem(localTask);
266     localMethod.SetStepMultiplier(2);
267     localMethod.SetStartPoint(startSolution.GetOptimumPoint().get(),
268     localTask.GetTaskDimension());
269
270     SharedVector localOptimum(new double[mMethodDimension], array_deleter<double>());
271     localMethod.StartOptimization(localOptimum.get());
272     double bestLocalValue = mTargetFunction->Calculate(localOptimum.get());
273
274     if (startSolution.GetOptimumValue() > bestLocalValue)
275         return OptimizerSolution(startSolution.GetIterationsCount(),
276             bestLocalValue, 0.5, mMethodDimension, localOptimum);
277
278     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);

```



```

397     }
398     else// LocalTuningMode::Adaptive
399         localM = fmax(mu1 / r + (1 - 1 / r)*mGlobalM, 0.01);
400         //localM = fmax(mu1*(1 - dx / mMaxIntervalNorm) + mu2, 0.01);
401         //localM = fmax(mu1*mMConvolution + (1 - mMConvolution)*mu2, 0.01);
402     }
403
404     curr_rank = dx + Pow2((rightIt->val - leftIt->val) / (r * localM)) / dx
405         - 2 * (rightIt->val + leftIt->val - 2 * mZ) / (r * localM);
406     if (isLocal)
407         curr_rank /= sqrt((rightIt->val - mZ)*
408             (leftIt->val - mZ)) / localM + pow(1.5, -mAlpha);
409
410     if (curr_rank > mIntervalsForTrials[mNumberOfThreads - 1].R)
411     {
412         OptimizerInterval newInterval(
413             OptimizerTrialPoint(*leftIt),
414             OptimizerTrialPoint(*rightIt), curr_rank, localM);
415         for (int i = 0; i < mNumberOfThreads; i++)
416             if (mIntervalsForTrials[i].R < newInterval.R)
417                 std::swap(mIntervalsForTrials[i], newInterval);
418     }
419     ++leftIt;
420     ++rightIt;
421 }
422 return 0;
423 }
424 void OptimizerAlgorithmUnconstrained::AllocMem()
425 {
426 }

```

10.2. Приложение 2

```

1  #ifndef __OPTMAL_CONTROL_PROBLEM_H__
2  #define __OPTMAL_CONTROL_PROBLEM_H__
3
4  #include "problem_interface.h"
5  #include "optimal_problem_base.h"
6
7  #include <Eigen/Dense>
8  #include <string>
9
10 class TOptimalControlProblem : public IProblem
11 {
12 protected:
13
14     OptimalControlProblemBase* mPPProblemImpl;
15     int mDimension;
16     bool mIsInitialized;
17     std::string mConfigPath;
18
19 public:
20     TOptimalControlProblem();
21     virtual int SetConfigPath(const std::string& configPath);
22     virtual int SetDimension(int dimension);
23     virtual int GetDimension() const;
24     virtual int Initialize();
25 }

```

```

26 virtual void GetBounds(double* lower, double *upper);
27 virtual int GetOptimumValue(double& value) const;
28 virtual int GetOptimumPoint(double* x) const;
29
30 virtual int GetNumberOfFunctions() const;
31 virtual int GetNumberOfConstraints() const;
32 virtual int GetNumberOfCriteria() const;
33
34 virtual double CalculateFunctionals(const double* x, int fNumber);
35
36 ~TOptimalControlProblem();
37 };
38
39 extern "C" LIB_EXPORT_API IProblem* create();
40 extern "C" LIB_EXPORT_API void destroy(IProblem* ptr);
41 #endif

```

```

1 #include "optimalControl.h"
2 #include "test_problems.h"
3 #include "problemA.h"
4 #include "problemB.h"
5 #include "pugixml.hpp"
6
7 #include <string>
8 #include <stdexcept>
9
10 TOptimalControlProblem::TOptimalControlProblem()
11 {
12     mIsInitialized = false;
13 }
14
15 int TOptimalControlProblem::SetConfigPath(const std::string& configPath)
16 {
17     mConfigPath = std::string(configPath);
18     return IProblem::OK;
19 }
20
21 int TOptimalControlProblem::SetDimension(int dimension)
22 {
23     return IProblem::OK;
24 }
25
26 int TOptimalControlProblem::GetDimension() const
27 {
28     return mDimension;
29 }
30
31 int TOptimalControlProblem::Initialize()
32 {
33     if (mIsInitialized == false)
34     {
35         mIsInitialized = true;
36
37         pugi::xml_document doc;
38         pugi::xml_parse_result result = doc.load_file(mConfigPath.c_str());
39         if (result.status != pugi::status_ok)
40             return IProblem::ERROR;
41
42         pugi::xml_node config = doc.child("config");

```

```

43     std::string problemName = config.child("problem_name").child_value();
44     double secondCriterionLevel = 0.;
45     double lambda = 0.;
46     int dimension = 0;
47     try {
48         secondCriterionLevel = std::stod(config.child("S").child_value());
49         lambda = std::stod(config.child("lambda").child_value());
50         dimension = std::stoi(config.child("N").child_value());
51     }
52     catch (std::invalid_argument& exp) {
53         return IProblem::ERROR;
54     }
55
56     if (problemName == std::string("v"))
57         mPPProblemImpl = new VibroisolationProblem();
58     else if (problemName == std::string("od"))
59         mPPProblemImpl = new OscillationDampingProblem();
60     else if (problemName == std::string("a2d"))
61         mPPProblemImpl = new ProblemA2d(secondCriterionLevel);
62     else if (problemName == std::string("a3d"))
63         mPPProblemImpl = new ProblemA3d(secondCriterionLevel);
64     else if (problemName == std::string("bNd") && dimension == 2)
65         mPPProblemImpl = new ProblemB2d(secondCriterionLevel);
66     else if (problemName == std::string("bNd"))
67         mPPProblemImpl = new ProblemB(secondCriterionLevel, dimension);
68     else if (problemName == std::string("bNdC"))
69         mPPProblemImpl = new ProblemBConvolved(lambda, dimension);
70     else
71         return IProblem::ERROR;
72
73     mDimension = mPPProblemImpl->GetDimension();
74
75     return IProblem::OK;
76 }
77 else
78     return IProblem::ERROR;
79 }
80
81 void TOptimalControlProblem::GetBounds(double* lower, double *upper)
82 {
83     if (mIsInitialized)
84         mPPProblemImpl->GetBounds(lower, upper);
85 }
86
87 int TOptimalControlProblem::GetOptimumValue(double& value) const
88 {
89     return IProblem::UNDEFINED;
90 }
91
92 int TOptimalControlProblem::GetOptimumPoint(double* point) const
93 {
94     return IProblem::UNDEFINED;
95 }
96
97 int TOptimalControlProblem::GetNumberOfFunctions() const
98 {
99     return mPPProblemImpl->GetNumberOfFunctions();
100 }
101
102 int TOptimalControlProblem::GetNumberOfConstraints() const
103 {

```

```

104     return mPPProblemImpl->GetNumberOfConstraints();
105 }
106
107 int TOptimalControlProblem::GetNumberOfCriteriaions() const
108 {
109     return 1;
110 }
111
112 //
113
114 double TOptimalControlProblem::CalculateFunctionals(const double* x, int fNumber
115 )
116 {
117     return mPPProblemImpl->CalculateFunctionals(x, fNumber);
118 }
119
120 TOptimalControlProblem::~TOptimalControlProblem()
121 {
122     if(mIsInitialized)
123     {
124         mPPProblemImpl->PrintFinalMessage();
125         delete mPPProblemImpl;
126     }
127 }
128
129 LIB_EXPORT_API IProblem* create()
130 {
131     return new TOptimalControlProblem();
132 }
133
134 LIB_EXPORT_API void destroy(IProblem* ptr)
135 {
136     delete ptr;
137 }

```

```

1  #ifndef OPTIMAL_CONTROL_BASE_H
2  #define OPTIMAL_CONTROL_BASE_H
3
4  #ifndef EIGEN_DONT_PARALLELIZE
5  #define EIGEN_DONT_PARALLELIZE
6  #endif
7
8  #include <Eigen/Dense>
9  #include <vector>
10
11 class OptimalControlProblemBase
12 {
13 protected:
14     int mDimension;
15     int n_x;
16     int n_v;
17     int n_u;
18     std::vector<int> n_k;
19
20     Eigen::MatrixXd A;
21     Eigen::MatrixXd B_u;
22     Eigen::MatrixXd B_v;
23     Eigen::MatrixXd* YVecs;

```

```

24     std::vector<Eigen::MatrixXd> CMatrices;
25     std::vector<Eigen::MatrixXd> DMatrices;
26
27     double mZeroConstraintOffset;
28     double mS;
29
30     virtual Eigen::RowVectorXd getTheta(const double* x);
31     virtual double CalculateCriterionValue(const Eigen::RowVectorXd& theta, int
        fNumber);
32
33 public:
34     OptimalControlProblemBase();
35     virtual ~OptimalControlProblemBase();
36     virtual void GetBounds(double* lower, double *upper) const = 0;
37     virtual double CalculateFunctionals(const double* x, int fNumber);
38     int GetDimension() const { return mDimension; }
39     virtual int GetNumberOfConstraints() const { return 2; }
40     virtual int GetNumberOfFunctions() const { return 3; }
41     virtual void PrintFinalMessage() {}
42 };
43
44 #endif

```

```

1  #include "optimal_problem_base.h"
2
3  #define _USE_MATH_DEFINES
4  #include <math.h>
5  #include <algorithm>
6  #include <omp.h>
7
8  using namespace Eigen;
9
10 OptimalControlProblemBase::OptimalControlProblemBase()
11 {
12     YVecs = new MatrixXd[omp_get_num_procs()];
13 }
14
15 OptimalControlProblemBase::~~OptimalControlProblemBase()
16 {
17     delete[] YVecs;
18 }
19
20 RowVectorXd OptimalControlProblemBase::getTheta(const double* x)
21 {
22     return Map<const RowVectorXd>(x, mDimension);
23 }
24
25 MatrixXd buildVectorizationMatrix(const MatrixXd& A)
26 {
27     size_t n = A.cols();
28
29     MatrixXd E = MatrixXd::Identity(n, n);
30     MatrixXd S(n*n, n*n);
31
32     for(size_t i = 0; i < n; i++)
33         for(size_t j = 0; j < n; j++)
34         {
35             if(i != j)
36                 S.block(i*n, j*n, n, n) = A.coeff(i, j)*E;

```

```

37     else
38         S.block(i*n, j*n, n, n) = A + A.coeff(i, j)*E;
39     }
40
41     return S;
42 }
43
44 double OptimalControlProblemBase::CalculateCriterionValue(const Eigen::
    RowVectorXd& theta, int fNumber)
45 {
46     Map<MatrixXd> Y(YVecs[omp_get_thread_num()].data(), A.cols(), A.rows());
47
48     double value = -HUGE_VAL;
49     size_t cRows = CMatrices[fNumber].rows();
50     for (size_t i = 0; i < cRows; i++)
51     {
52         RowVectorXd currentVector = CMatrices[fNumber].row(i) + DMatrices[fNumber].
            row(i)*theta;
53         double dotProd = (currentVector*Y).dot(currentVector.transpose());
54         value = std::max(value, dotProd);
55     }
56     return value;
57 }
58
59 double OptimalControlProblemBase::CalculateFunctionals(const double* x, int
    fNumber)
60 {
61     RowVectorXd theta = getTheta(x);
62
63     if (fNumber == 0)
64     {
65         MatrixXd Atheta = A + B_u*theta;
66
67         EigenSolver<MatrixXd> eigenSolver(Atheta, false);
68         MatrixXcd eigenvalues = eigenSolver.eigenvalues();
69         double maxReal = -HUGE_VAL;
70         for (int i = 0; i < Atheta.cols(); i++)
71             maxReal = std::max(maxReal, eigenvalues.coeff(i).real());
72
73         return maxReal + mZeroConstraintOffset;
74     }
75     else if (fNumber == 1)
76     {
77         MatrixXd Atheta = A + B_u*theta;
78         MatrixXd S = buildVectorizationMatrix(Atheta);
79         MatrixXd rhs = -B_v*B_v.transpose();
80         Map<VectorXd> rhsMap(rhs.data(), rhs.size());
81         YVecs[omp_get_thread_num()] = S.partialPivLu().solve(rhsMap);
82     }
83
84     fNumber--;
85     double offset = fNumber == 0 ? -mS : 0.;
86     double value = CalculateCriterionValue(theta, fNumber);
87
88     return sqrt(value) + offset;
89 }

```

```

1 #pragma once
2 #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

```

```

64     DMatrices[0].resize(1, 1); DMatrices[0] << 0;
65
66     DMatrices[1].resize(n - 1, 1);
67     DMatrices[1].setZero();
68
69     mS = S;
70     mZeroConstraintOffset = 0.;
71 }
72
73 int GetNumberOfConstraints() const { return 2; }
74 };
75
76 class ProblemB : public ProblemB_Common
77 {
78 public:
79     ProblemB(double S, int dimension) : ProblemB_Common(S)
80     {
81         mDimension = dimension;//shoud be > 2
82     }
83
84     void GetBounds(double* lower, double *upper) const
85     {
86         for (int i = 0; i < mDimension; i++)
87         {
88             lower[i] = -10000.;
89             upper[i] = -0.01;
90         }
91
92         for (int i = 2; i < mDimension; i++)
93         {
94             upper[i] = 10000.;
95         }
96     }
97
98     RowVectorXd getTheta(const double* x)
99     {
100         RowVectorXd theta(n_x);
101         theta.setZero();
102
103         theta(0) = x[0] - x[2];
104         for (int i = 1; i < mDimension - 2; i++)
105             theta(i) = x[i + 1] - x[i + 2];
106         theta(mDimension - 2) = x[mDimension - 1];
107         theta(n_x / 2) = x[1];
108
109         return theta;
110     }
111 };
112
113 class ProblemB2d : public ProblemB_Common
114 {
115 public:
116     ProblemB2d(double S) : ProblemB_Common(S)
117     {
118         mDimension = 2;
119     }
120
121     void GetBounds(double* lower, double *upper) const
122     {
123         for (int i = 0; i < mDimension; i++)
124         {

```



```

125     lower[i] = -70.;
126     upper[i] = -0.01;
127 }
128 lower[0] = -20.;
129 }
130
131 RowVectorXd getTheta(const double* x)
132 {
133     RowVectorXd theta(n_x);
134     theta.setZero();
135     theta(0) = x[0];
136     theta(n_x / 2) = x[1];
137
138     return theta;
139 }
140 };

```

```

1  #ifndef __OPTMAL_CONTROL_TEST_IMPL_H__
2  #define __OPTMAL_CONTROL_TEST_IMPL_H__
3
4  #include "optimal_problem_base.h"
5
6  class VibroisolationProblem : public OptimalControlProblemBase
7  {
8  public:
9      VibroisolationProblem()
10     {
11         n_x = 2;
12         n_v = 1;
13         n_u = 1;
14         n_k.resize(2);
15         n_k[0] = n_k[1] = 1;
16
17         CMatrices.resize(2);
18         DMatrices.resize(2);
19
20         A.resize(2, 2); A << 0, 1, 0, 0;
21         B_u.resize(2, 1); B_u << 0, 1;
22         B_v = B_u;
23
24         CMatrices[0].resize(1, 2); CMatrices[0] << 1, 0;
25         CMatrices[1].resize(1, 2); CMatrices[1] << 0, 0;
26
27         DMatrices[0].resize(1, 1); DMatrices[0] << 0;
28         DMatrices[1].resize(1, 1); DMatrices[1] << 1;
29         mDimension = n_x;
30         mZeroConstraintOffset = 0.;
31         mS = 1.;
32     }
33
34     int GetNumberOfConstraints() const { return 2; }
35
36     void GetBounds(double* lower, double *upper) const
37     {
38         for (int i = 0; i < mDimension; i++)
39         {
40             upper[i] = -0.2;
41             lower[i] = -2.0;
42         }

```

```

43     }
44 };
45
46 class OscillationDampingProblem : public OptimalControlProblemBase
47 {
48 public:
49     OscillationDampingProblem()
50     {
51         double beta = 0.1;
52         n_x = 4;
53         n_v = 1;
54         n_u = 1;
55         n_k.resize(2);
56         n_k[0] = n_k[1] = 1;
57
58         CMatrices.resize(2);
59         DMatrices.resize(2);
60
61         A.resize(4, 4);
62         A << 0, 0, 1, 0,
63            0, 0, 0, 1,
64            -2, 1, -2*beta, beta,
65            1, -1, beta, -beta;
66
67         B_u.resize(4, 1); B_u << 0, 0, 0, 1;
68         B_v.resize(4, 1); B_v << 0, 0, 1, 1;
69
70         CMatrices[0].resize(2, 4);
71         CMatrices[0] << 1, 0, 0, 0,
72            -1, 1, 0, 0;
73         CMatrices[1].resize(1, 4); CMatrices[1] << 0, 0, 0, 0;
74
75         DMatrices[0].resize(2, 1); DMatrices[0] << 0, 0;
76         DMatrices[1].resize(1, 1); DMatrices[1] << 1;
77         mDimension = n_x;
78         mZeroConstraintOffset = 0.02;
79         mS = 1.;
80     }
81
82     int GetNumberOfConstraints() const
83     {
84         return 2;
85     }
86
87     void GetBounds(double* lower, double *upper) const
88     {
89         for (int i = 0; i < mDimension; i++)
90         {
91             upper[i] = 1;
92             lower[i] = -2.0;
93         }
94     }
95 };
96 #endif

```

10.3. Приложение 3

```
1 #ifndef __LOCALMETHOD_H__
```

```

2 #define __LOCALMETHOD_H__
3
4 #include "parameters.h"
5 #include "task.h"
6 #include "data.h"
7 #include "common.h"
8 #include <vector>
9
10 #define MAX_LOCAL_TRIALS_NUMBER 10000
11
12 class TLocalMethod
13 {
14 protected:
15
16     int mDimension;
17     int mConstraintsNumber;
18     int mTrialsCounter;
19     int mMaxTrial;
20
21     TTrial mBestPoint;
22     std::vector<TTrial> mSearchSequence;
23     TTask* mPTask;
24
25     bool mIsLogPoints;
26
27     double mEps;
28     double mStep;
29     double mStepMultiplier;
30
31     OBJECTIV_TYPE *mFunctionsArgument;
32     OBJECTIV_TYPE *mStartPoint;
33     OBJECTIV_TYPE* mCurrentPoint;
34     OBJECTIV_TYPE* mCurrentResearchDirection;
35     OBJECTIV_TYPE* mPreviousResearchDirection;
36
37     double MakeResearch(OBJECTIV_TYPE*);
38     void DoStep();
39     double EvaluateObjectiveFunction(const OBJECTIV_TYPE*);
40
41 public:
42
43     TLocalMethod();
44     TLocalMethod(TParameters _params, TTask* _pTask, TTrial _startPoint, bool
        logPoints = false);
45     ~TLocalMethod();
46
47     void SetEps(double);
48     void SetInitialStep(double);
49     void SetStepMultiplier(double);
50     void SetMaxTrials(int);
51
52     int GetTrialsCounter() const;
53     std::vector<TTrial> GetSearchSequence() const;
54
55     TTrial StartOptimization();
56 };
57
58 #endif //__LOCALMETHOD_H__

```

```

1 #include "local_method.h"
2 #include <cmath>
3 #include <cstring>
4 #include <algorithm>
5
6 TLocalMethod::TLocalMethod(TParameters _params, TTask* _pTask, TTrial
   _startPoint, bool logPoints)
7 {
8     mEps = _params.Epsilon / 100;
9     if (mEps > 0.0001)
10         mEps = 0.0001;
11     mBestPoint = _startPoint;
12     mStep = _params.Epsilon * 2;
13     mStepMultiplier = 2;
14     mTrialsCounter = 0;
15     mIsLogPoints = logPoints;
16
17     mPTask = _pTask;
18
19     mDimension = mPTask->GetN() - mPTask->GetFixedN();
20     mConstraintsNumber = mPTask->GetNumOfFunc() - 1;
21
22     mStartPoint = new OBJECTIV_TYPE[mDimension];
23     std::memcpy(mStartPoint,
24         _startPoint.y + mPTask->GetFixedN(),
25         mDimension * sizeof(OBJECTIV_TYPE));
26
27     mFunctionsArgument = new OBJECTIV_TYPE[mPTask->GetN()];
28     std::memcpy(mFunctionsArgument,
29         _startPoint.y,
30         mPTask->GetN() * sizeof(OBJECTIV_TYPE));
31     mMaxTrial = MAX_LOCAL_TRIALS_NUMBER;
32 }
33
34 TLocalMethod::TLocalMethod() : mPTask(NULL), mStartPoint(NULL),
35 mFunctionsArgument(NULL), mMaxTrial(MAX_LOCAL_TRIALS_NUMBER)
36 {
37 }
38
39 TLocalMethod::~~TLocalMethod()
40 {
41     if (mStartPoint)
42         delete[] mStartPoint;
43     if (mFunctionsArgument)
44         delete[] mFunctionsArgument;
45 }
46
47 void TLocalMethod::DoStep()
48 {
49     for (int i = 0; i < mDimension; i++)
50         mCurrentPoint[i] = (1 + mStepMultiplier)*mCurrentResearchDirection[i] -
51         mStepMultiplier*mPreviousResearchDirection[i];
52 }
53
54 TTrial TLocalMethod::StartOptimization()
55 {
56     int k = 0, i = 0;
57     bool needRestart = true;
58     double currentFValue, nextFValue;
59     mTrialsCounter = 0;
60

```

```

61 mCurrentPoint = new OBJECTIV_TYPE[mDimension];
62 mCurrentResearchDirection = new OBJECTIV_TYPE[mDimension];
63 mPreviousResearchDirection = new OBJECTIV_TYPE[mDimension];
64
65 while (mTrialsCounter < mMaxTrial) {
66     i++;
67     if (needRestart) {
68         k = 0;
69         std::memcpy(mCurrentPoint, mStartPoint, sizeof(OBJECTIV_TYPE)*mDimension);
70         std::memcpy(mCurrentResearchDirection, mStartPoint, sizeof(OBJECTIV_TYPE)*
mDimension);
71         currentFValue = EvaluateObjectiveFunciun(mCurrentPoint);
72         needRestart = false;
73     }
74
75     std::swap(mPreviousResearchDirection, mCurrentResearchDirection);
76     std::memcpy(mCurrentResearchDirection, mCurrentPoint, sizeof(OBJECTIV_TYPE)*
mDimension);
77     nextFValue = MakeResearch(mCurrentResearchDirection);
78
79     if (currentFValue > nextFValue) {
80         DoStep();
81
82         if (mIsLogPoints)
83         {
84             TTrial currentTrial;
85             currentTrial.index = mBestPoint.index;
86             currentTrial.FuncValues[currentTrial.index] = nextFValue;
87             std::memcpy(currentTrial.y, mFunctionsArgument, sizeof(OBJECTIV_TYPE)*
mPTask->GetFixedN());
88             std::memcpy(currentTrial.y + mPTask->GetFixedN(), mCurrentPoint, sizeof(
OBJECTIV_TYPE)*mDimension);
89             mSearchSequence.push_back(currentTrial);
90         }
91         k++;
92         currentFValue = nextFValue;
93     }
94     else if (mStep > mEps) {
95         if (k != 0)
96             std::memcpy(mStartPoint, mPreviousResearchDirection, sizeof(
OBJECTIV_TYPE)*mDimension);
97         else
98             mStep /= mStepMultiplier;
99         needRestart = true;
100     }
101     else
102         break;
103 }
104
105 if (currentFValue < mBestPoint.FuncValues[mConstraintsNumber])
106 {
107     std::memcpy(mBestPoint.y + mPTask->GetFixedN(),
mPreviousResearchDirection, sizeof(OBJECTIV_TYPE)*mDimension);
108     mBestPoint.FuncValues[mConstraintsNumber] = currentFValue;
109     mSearchSequence.push_back(mBestPoint);
110 }
111
112 delete[] mCurrentPoint;
113 delete[] mPreviousResearchDirection;
114 delete[] mCurrentResearchDirection;
115
116

```

```

117     return mBestPoint;
118 }
119
120 double TLocalMethod::EvaluateObjectiveFunction(const OBJECTIV_TYPE* x)
121 {
122     if (mTrialsCounter >= mMaxTrial)
123         return HUGE_VAL;
124
125     for (int i = 0; i < mDimension; i++)
126         if (x[i] < mPTask->GetA()[mPTask->GetFixedN() + i] ||
127             x[i] > mPTask->GetB()[mPTask->GetFixedN() + i])
128             return HUGE_VAL;
129
130     std::memcpy(mFunctionsArgument + mPTask->GetFixedN(), x, mDimension * sizeof(
131         OBJECTIV_TYPE));
132     for (int i = 0; i <= mConstraintsNumber; i++)
133     {
134         double value = mPTask->GetFuncs()[i](mFunctionsArgument);
135         if (i < mConstraintsNumber && value > 0)
136         {
137             mTrialsCounter++;
138             return HUGE_VAL;
139         }
140         else if (i == mConstraintsNumber)
141         {
142             mTrialsCounter++;
143             return value;
144         }
145     }
146     return HUGE_VAL;
147 }
148
149 void TLocalMethod::SetEps(double eps)
150 {
151     mEps = eps;
152 }
153
154 void TLocalMethod::SetInitialStep(double value)
155 {
156     mStep = value;
157 }
158
159 void TLocalMethod::SetStepMultiplier(double value)
160 {
161     mStepMultiplier = value;
162 }
163
164 void TLocalMethod::SetMaxTrials(int count)
165 {
166     mMaxTrial = std::min(count, MAX_LOCAL_TRIALS_NUMBER);
167 }
168
169 int TLocalMethod::GetTrialsCounter() const
170 {
171     return mTrialsCounter;
172 }
173
174 std::vector<TTrial> TLocalMethod::GetSearchSequence() const
175 {
176     return mSearchSequence;

```

```

177 }
178
179 double TLocalMethod::MakeResearch(OBJECTIV_TYPE* startPoint)
180 {
181     double bestValue = EvaluateObjectiveFunction(startPoint);
182
183     for (int i = 0; i < mDimension; i++)
184     {
185         startPoint[i] += mStep;
186         double rightFValue = EvaluateObjectiveFunction(startPoint);
187
188         if (rightFValue > bestValue)
189         {
190             startPoint[i] -= 2 * mStep;
191             double leftFValue = EvaluateObjectiveFunction(startPoint);
192             if (leftFValue > bestValue)
193                 startPoint[i] += mStep;
194             else
195                 bestValue = leftFValue;
196         }
197         else
198             bestValue = rightFValue;
199     }
200
201     return bestValue;
202 }

```

10.4. Приложение 4

```

1 #ifndef __DUAL_QUEUE_H__
2 #define __DUAL_QUEUE_H__
3
4 #include "minmaxheap.h"
5 #include "common.h"
6 #include "queue_common.h"
7
8 class TPriorityDualQueue : public TPriorityQueueCommon
9 {
10 protected:
11     int MaxSize;
12     int CurLocalSize;
13     int CurGlobalSize;
14
15     MinMaxHeap< TQueueElement, _less >* pGlobalHeap;
16     MinMaxHeap< TQueueElement, _less >* pLocalHeap;
17
18     void DeleteMinLocalElem();
19     void DeleteMinGlobalElem();
20     void ClearLocal();
21     void ClearGlobal();
22 public:
23
24     TPriorityDualQueue(int _MaxSize = DefaultQueueSize); // _MaxSize must be equal
25     ~TPriorityDualQueue();
26
27     int GetLocalSize() const;
28     int GetSize() const;

```

```

29  int GetMaxSize() const;
30  bool IsLocalEmpty() const;
31  bool IsLocalFull() const;
32  bool IsEmpty() const;
33  bool IsFull() const;
34
35  void Push(double globalKey, double localKey, void *value);
36  void PushWithPriority(double globalKey, double localKey, void *value);
37  void Pop(double *key, void **value);
38
39  void PopFromLocal(double *key, void **value);
40
41  void Clear();
42  void Resize(int size);
43 };
44 #endif

```

```

1  #include "dual_queue.h"
2
3  TPriorityDualQueue::TPriorityDualQueue(int _MaxSize)
4  {
5      MaxSize = _MaxSize;
6      CurGlobalSize = CurLocalSize = 0;
7      pLocalHeap = new MinMaxHeap< TQueueElement, _less >(MaxSize);
8      pGlobalHeap = new MinMaxHeap< TQueueElement, _less >(MaxSize);
9  }
10
11  TPriorityDualQueue::~TPriorityDualQueue()
12  {
13      delete pLocalHeap;
14      delete pGlobalHeap;
15  }
16
17  void TPriorityDualQueue::DeleteMinLocalElem()
18  {
19      TQueueElement tmp = pLocalHeap->popMin();
20      CurLocalSize--;
21
22      //update linked element in the global queue
23      if (tmp.pLinkedElement != NULL)
24          tmp.pLinkedElement->pLinkedElement = NULL;
25  }
26
27  void TPriorityDualQueue::DeleteMinGlobalElem()
28  {
29      TQueueElement tmp = pGlobalHeap->popMin();
30      CurGlobalSize--;
31
32      //update linked element in the local queue
33      if (tmp.pLinkedElement != NULL)
34          tmp.pLinkedElement->pLinkedElement = NULL;
35  }
36
37  int TPriorityDualQueue::GetLocalSize() const
38  {
39      return CurLocalSize;
40  }
41
42  int TPriorityDualQueue::GetSize() const

```



```

43 {
44     return CurGlobalSize;
45 }
46
47 int TPriorityDualQueue::GetMaxSize() const
48 {
49     return MaxSize;
50 }
51
52 bool TPriorityDualQueue::IsLocalEmpty() const
53 {
54     return CurLocalSize == 0;
55 }
56
57 bool TPriorityDualQueue::IsLocalFull() const
58 {
59     return CurLocalSize == MaxSize;
60 }
61
62 bool TPriorityDualQueue::IsEmpty() const
63 {
64     return CurGlobalSize == 0;
65 }
66
67 bool TPriorityDualQueue::IsFull() const
68 {
69     return CurGlobalSize == MaxSize;
70 }
71
72 void TPriorityDualQueue::Push(double globalKey, double localKey, void * value)
73 {
74     TQueueElement* pGlobalElem = NULL, *pLocalElem = NULL;
75     //push to a global queue
76     if (!IsFull()) {
77         CurGlobalSize++;
78         pGlobalElem = pGlobalHeap->push(TQueueElement(globalKey, value));
79     }
80     else {
81         if (globalKey > pGlobalHeap->findMin().Key) {
82             DeleteMinGlobalElem();
83             CurGlobalSize++;
84             pGlobalElem = pGlobalHeap->push(TQueueElement(globalKey, value));
85         }
86     }
87     //push to a local queue
88     if (!IsLocalFull()) {
89         CurLocalSize++;
90         pLocalElem = pLocalHeap->push(TQueueElement(localKey, value));
91     }
92     else {
93         if (localKey > pLocalHeap->findMin().Key) {
94             DeleteMinLocalElem();
95             CurLocalSize++;
96             pLocalElem = pLocalHeap->push(TQueueElement(localKey, value));
97         }
98     }
99     //link elements
100     if (pGlobalElem != NULL && pLocalElem != NULL) {
101         pGlobalElem->pLinkedElement = pLocalElem;
102         pLocalElem->pLinkedElement = pGlobalElem;
103     }

```

```

104 }
105
106 void TPriorityDualQueue::PushWithPriority(double globalKey, double localKey,
107 void * value)
108 {
109     TQueueElement* pGlobalElem = NULL, *pLocalElem = NULL;
110     //push to a global queue
111     if (!IsEmpty()) {
112         if (globalKey > pGlobalHeap->findMin().Key) {
113             if (IsFull())
114                 DeleteMinGlobalElem();
115             CurGlobalSize++;
116             pGlobalElem = pGlobalHeap->push(TQueueElement(globalKey, value));
117         }
118     }
119     else {
120         CurGlobalSize++;
121         pGlobalElem = pGlobalHeap->push(TQueueElement(globalKey, value));
122     }
123     //push to a local queue
124     if (!IsLocalEmpty()) {
125         if (localKey > pLocalHeap->findMin().Key) {
126             if (IsLocalFull())
127                 DeleteMinLocalElem();
128             CurLocalSize++;
129             pLocalElem = pLocalHeap->push(TQueueElement(localKey, value));
130         }
131     }
132     else {
133         CurLocalSize++;
134         pLocalElem = pLocalHeap->push(TQueueElement(localKey, value));
135     }
136     //link elements
137     if (pGlobalElem != NULL && pLocalElem != NULL) {
138         pGlobalElem->pLinkedElement = pLocalElem;
139         pLocalElem->pLinkedElement = pGlobalElem;
140     }
141 }
142
143 void TPriorityDualQueue::PopFromLocal(double * key, void ** value)
144 {
145     TQueueElement tmp = pLocalHeap->popMax();
146     *key = tmp.Key;
147     *value = tmp.pValue;
148     CurLocalSize--;
149
150     //delete linked element from the global queue
151     if (tmp.pLinkedElement != NULL) {
152         pGlobalHeap->deleteElement(tmp.pLinkedElement);
153         CurGlobalSize--;
154     }
155 }
156
157 void TPriorityDualQueue::Pop(double * key, void ** value)
158 {
159     TQueueElement tmp = pGlobalHeap->popMax();
160     *key = tmp.Key;
161     *value = tmp.pValue;
162     CurGlobalSize--;
163
164     //delete linked element from the local queue

```

```

164     if (tmp.pLinkedElement != NULL) {
165         pLocalHeap->deleteElement(tmp.pLinkedElement);
166         CurLocalSize--;
167     }
168 }
169
170 void TPriorityDualQueue::Clear()
171 {
172     ClearLocal();
173     ClearGlobal();
174 }
175
176 void TPriorityDualQueue::Resize(int size)
177 {
178     MaxSize = size;
179     CurGlobalSize = CurLocalSize = 0;
180     delete pLocalHeap;
181     delete pGlobalHeap;
182     pLocalHeap = new MinMaxHeap< TQueueElement, _less >(MaxSize);
183     pGlobalHeap = new MinMaxHeap< TQueueElement, _less >(MaxSize);
184 }
185
186 void TPriorityDualQueue::ClearLocal()
187 {
188     pLocalHeap->clear();
189     CurLocalSize = 0;
190 }
191
192 void TPriorityDualQueue::ClearGlobal()
193 {
194     pGlobalHeap->clear();
195     CurGlobalSize = 0;
196 }

```