Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования
Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

Отчет по по практике по получению первичных профессиональных умений и навыков

«Алгоритм глобального поиска для одномерных многоэкстремальных задач оптимизации»

Выполнил:
студент группы 381803-2
Смирнов А. И.
Проверил:
доцент кафедры МОСТ,
кандидат технических наук
Сысоев А. В.

Нижний Новгород
2021

Оглавление

Введение

Многие задачи принятия оптимальных решений, возникающие в различных сферах человеческой деятельности, такие как моделирование климата, генная инженерия, проектирование интегральных схем, создание лекарственных препаратов и др., могут быть сформулированы как задачи оптимизации. Увеличение числа прикладных проблем, описываемых математическими моделями подобного типа, и бурное развитие вычислительной техники инициировали развитие глобальной оптимизации. При этом большое практическое значение имеют не только многомерные, но и одномерные задачи поиска глобально-оптимальных решений, часто встречающиеся, например, в электротехнике и электронике. Также одномерные алгоритмы поиска глобального экстремума могут быть использованы в качестве основы для конструирования численных методов решения многомерных задач оптимизации посредством применения схем редукции размерности.

Усложнение математических моделей оптимизируемых объектов затрудняет поиск оптимальной комбинации параметров, и как следствие, не представляется возможным найти такую комбинацию аналитически, поэтому возникает необходимость построения численных методов для ее поиска. Численные методы глобальной оптимизации существенно отличаются от стандартных локальных методов поиска, которые часто неспособны найти глобальное решение рассматриваемых задач, т. к. не в состоянии покинуть зоны притяжения локальных оптимумов и, соответственно, упускают глобальный оптимум.

Проблема численного решения задач оптимизации, в свою очередь, может быть сопряжена со значительными трудностями. Такие задачи могут характеризоваться многоэкстремальной, недифференцируемой или же заданной в форме черного ящика (т. е. в виде некоторой вычислительной процедуры, на вход которой подается аргумент, а на выходе наблюдается соответствующее значение) целевой функцией. Многоэкстремальные оптимизационные модели обладают высокой трудоемкостью численного анализа, поскольку для них характерен экспоненциальный рост вычислительных затрат с ростом размерности (количества параметров модели). Именно поэтому разработка эффективных параллельных методов для численного решения задач многоэкстремальной оптимизации и создание программных средств их реализации на современных многопроцессорных системах является стратегическим направлением по существенному развитию проблематики исследования сложных задач оптимального выбора.

Постановка задачи

Цель: Необходимо разработать и реализовать последовательный и параллельный варианты алгоритма глобального поиска для одномерных многоэкстремальных задач оптимизации, проверить корректность работы алгоритмов, провести вычислительные эксперименты и сравнить эффективность их работы в зависимости от различных входных данных и параметров.

Задачи:

- разработать и реализовать последовательный вариант алгоритма глобального поиска;
- разработать и реализовать параллельный вариант алгоритма глобального поиска с помощью OpenMP;
- разработать и реализовать параллельный вариант алгоритма глобального поиска с помощью std::Thread;
- протестировать алгоритм;
- сравнить эффективность в зависимости от входных данных.

Входные данные:

- минимизируемая функция: $\phi(x)$;
- отрезок: $x \in Q = [a, b]$;
- заданная точность поиска: ϵ.

Выходные данные:

- координаты минимального вычисленного значения функции: (ϕ_k^*, x_k^*).

Глава 1

Теоретическая часть

1.1 Последовательный алгоритм глобального поиска

Задачей оптимизации будем называть задачу следующего вида:

Найти точную нижнюю грань $\phi^* = inf\{\phi(x) : x \in Q\}$ *и, если множество точек глобального минимума* $Q^* \equiv Arg\ min\{\phi(x) : x \in Q\}$ *не пусто, найти хотя бы одну точку* $x^* \in Q^*$.

Рассмотрим одномерную задачу минимизации функции на отрезке:

$$\phi(x) \to min, x \in Q = [a, b]$$

Вычислительная схема АГП:

Дадим детальное описание вычислительной схемы АГП (алгоритма глобального поиска), применяемого к решению сформулированной задачи, рассматривая при этом в качестве поисковой информации множество:

$$\omega = \omega_k = \{(x_i, z_i), 1 \le i \le k\}.(*)$$

Согласно алгоритму, два первых испытания проводятся на концах отрезка $[a, b]$, т.е. $x^1 = a, x^2 = b$, вычисляются значения функции $z^1 = \phi(a), z^2 = \phi(b)$, и количество k проведенных испытаний полагается равным 2.

Пусть проведено $k \ge 2$ испытаний и получена информация $(*)$. Для выбора точки x^{k+1} нового испытания необходимо выполнить следующие действия:

1. Перенумеровать нижним индексом (начиная с нулевого значения) точки $x_i, 1 \le i \le k$, из $(*)$ в порядке возврастания, т.е. $a = x_0 < x_1 < \cdots < x_{k-1} = b$.
2. Полагая $z_i = \phi(x_i), 1 \le i \le k$, вычислить величину $M = \max\limits_{1 \le i \le k-1} |\frac{z_i - z_{i-1}}{x_i - x_{i-1}}|$ и положить

$$m = \begin{cases} rM, & M > 0 \\ 1, & M = 0 \end{cases}$$

, где $r > 1$ является заданным параметром метода.

3. Для каждого интервала $(x_{i-1}, x_i), 1 \leq i \leq k-1$ вычислить характеристику $R(i) = m(x_i - x_{i-1}) + \frac{(z_i - z_{i-1})^2}{/} m(x_i - x_{i-1}) - 2(z_i + z_{i-1})$.

4. Найти интервал (x_{t-1}, x_t), которому соответствует максимальная характеристика $R(t) = max\{R(i) : 1 \leq i \leq k-1\}$ (в случае нескольких интервалов выбирается интервал с наименьшим номером t).

5. Провести новое испытание в точке $x^{k+1} = \frac{1}{2}(x_t + x_{t-1}) - \frac{z_t - z_{t-1}}{2m}$, вычислить значение $z^{k+1} = \phi(x^{k+1})$ и увеличить номер шага поиска на единицу: $k = k + 1$.

Правило остановки задается в форме:

$$H_k(\Phi, \omega_k) = \begin{cases} 0, & x_t - x_{t-1} \leq \epsilon \text{ or } k \geq k_{max} \\ 1, & x_t - x_{t-1} > \epsilon \text{ or } k < k_{max} \end{cases}$$

, где $\epsilon > 0$ - заданная точность поиска (по координате), k_{max} – максимальное количество испытаний.

Наконец, в качестве оценки экстремума выбирается пара $e^k = (\phi_k^*, x_k^*)$, где ϕ_k^* - минимальное вычисленное значение функции, т.е. $\phi_k^* = \min\limits_{1 \leq i \leq k} \phi(x^i)$, а x_k^* - координата этого значения: $x_k^* = \arg \min\limits_{1 \leq i \leq k} \phi(x^i)$

1.2 Параллельный алгоритм глобального поиска

Распараллеливание АГП можно производить различными способами:

1. разделить отрезок, на котором производится поиск глобального минимума, на подотрезки, внутри которых запустить последовательный алгоритм глобального поиска;
2. распараллелить вычисление внутренних характеристик последовательного алгоритма глобального поиска;
3. распараллеливание и модифицирование алгоритма глобального поиска, обеспечивая одновременное выполнение нескольких испытаний.

Реализация параллельного алгоритма с помощью функций библиотеки межпроцессного взаимодействия OpenMP:

1. способ:
   - вычислить новые границы отрезков, на которых будет производиться поиск глобального минимума;
   - в каждом потоке вызвать последовательную реализацию глобального поиска на соответствующем новом отрезке;
   - собрать данные каждого потока и определить среди них глобальный минимум.
2. способ:
   - для вычисления характеристики M используем директивы $\#pragma\ omp\ for\ nowait$ - для паралельного вычисления характеристики на каждом интервале, и $\#pragma\ omp\ critical$ - для сравнения данных между потоками и определения максимальной характеристики M;
   - аналогичным способом высчитываем максимальную характеристику R среди интервалов.
3. способ:
   - определяем список максимальных характеристик R равное количеству потоков;
   - с помощью $\#pragma\ omp\ for\ nowait$ и $\#pragma\ omp\ critical$ высчитываем на интервалах новые точки и проверяем критерий выхода алгоритма;
   - добавить новые точки в конец вектора исходных точек и отсортировать его.

Реализация параллельного алгоритма с помощью функций библиотеки std::Thread:

1. способ:
   - определить возможное количество потоков и создать равное им количество $std :: promise$, $std :: future$ и $std :: thread$;
   - вычислить новые границы отрезков, на которых будет производиться поиск глобального минимума;
   - в каждом потоке вызвать последовательную реализацию глобального поиска на соответствующем новом отрезке;

- собрать данные каждого потока и определить среди них глобальный минимум.

2. способ.
   - определить возможное количество потоков и создать равное им количество $std :: promise$, $std :: future$ и $std :: thread$;
   - реализовать функции для вычисления характеристик M и R, которые будут работать на отдельных потоках;
   - в каждом потоке вызвать эти функции;
   - собрать данные каждого потока и определить среди них наибольшую характеристику.

3. способ:
   - определить возможное количество потоков и создать равное им количество $std :: promise$, $std :: future$ и $std :: thread$;
   - определяем список максимальных характеристик R равное количеству потоков;
   - реализовать функцию для высчитывания новой точки и проверки условия остановки, которую будет запускать поток;
   - собрать с каждого потока и добавить новые точки в конец вектора исходных точек и отсортировать его.

Глава 2

Практическая часть

Описание структуры программы

Код программы содержится в следующих файлах:

- Sequestion - проект, содержащий в себе реализацию последовательного алгоритма глобального поиска;
  - fun.h – включает объявление функций последовательного алгоритма глобального поиска;
  - fun.cpp – включает реализацию функций последовательного иалгоритма глобального поиска;
  - hansen_functions.h - включает объявления вспомогательных математических функций, а также массивы отрезков, на которых производится поиск, и значений глобального минимума;
  - hansen_functions.cpp - включает реализацию вспомогательных математических функций;
  - main.cpp – включает функцию main, тестирующую алгоритм глобального поиска.
- OpenMP - проект, содержащий в себе реализацию параллельного алгоритма глобального поиска с помощью библиотеки OpenMP;
  - ops_omp.h – включает объявление функций параллельного алгоритма глобального поиска;
  - ops_omp.cpp – включает реализацию функций параллельного алгоритмаглобального поиска;
  - hansen_functions.h - включает объявления вспомогательных математических функций, а также массивы отрезков, на которых производится поиск, и значений глобального минимума;
  - hansen_functions.cpp - включает реализацию вспомогательных математических функций;
  - main.cpp – включает функцию main, тестирующую алгоритм глобального поиска.
- Thread - проект, содержащий в себе реализацию параллельного алгоритма глобаль-

ного поиска с помощью библиотек thread и future;

- ops_std.h - включает объявление функций параллельного алгоритма глобального поиска;
- ops_std.cpp – включает реализацию функций параллельного алгоритмаглобального поиска;
- hansen_functions.h - включает объявления вспомогательных математических функций, а также массивы отрезков, на которых производится поиск, и значений глобального минимума;
- hansen_functions.cpp - включает реализацию вспомогательных математических функций;
- main.cpp – включает функцию main, тестирующую алгоритм глобального поиска.

2.1 Описание последовательной версии

Описание функций

```
size_t Rmax(std::vector<double>* a)
```

- назначние: поиск максимальной характеристики R;
- входные данные:
  - a – указатель вектор характеристик R каждого интервала.
- выходные данные:
  - count – номер интервала с максимальной характеристикой.

```
double GSA(double inter[2], double fun(double x), double r, double e)
```

- назначние: существляет последовательный поиск глобального минимума;
- входные данные:
  - inter – массив содержащий границы отрезка;
  - fun – указатель на минимизируемую функцию, принимающую и возвращающую вещественное число;
  - r – параметр метода;
  - e – вещественное число, заданная точность поиска.
- выходные данные:
  - new_point – вещественное число, координата минимального значения функции.

2.2 Описание OpenMP-версии

Описание функций

```
size_t Rmax(std::vector<double>* a)
```

- назначние: поиск максимальной характеристики R;
- входные данные:
  - a – указатель вектор характеристик R каждого интервала.
- выходные данные:
  - count – номер интервала с максимальной характеристикой.

```
double GSA(double inter[2], double fun(double x), double r, double e)
```

- назначние: существляет последовательный поиск глобального минимума, нужен для первого метода;
- входные данные:
  - inter – массив содержащий границы отрезка;
  - fun – указатель на минимизируемую функцию, принимающую и возвращающую вещественное число;
  - r – параметр метода;
  - e – вещественное число, заданная точность поиска.
- выходные данные:
  - new_point – вещественное число, координата минимального значения функции.

```
double ParallelGSA(double inter[2], double fun(double x), double r, double
    e)
```

- назначние: осуществляет параллельный поиск глобального минимума первым способом;
- входные данные:
  - inter – массив содержащий границы отрезка;
  - fun – указатель на минимизируемую функцию, принимающую и возвращающую вещественное число;
  - r – параметр метода;
  - e – вещественное число, заданная точность поиска.
- выходные данные:
  - new_point – вещественное число, координата минимального значения функции.

```
double ParallelOperations(double inter[2], double fun(double x), double r,
    double e)
```

- назначние: осуществляет параллельный поиск глобального минимума вторым способом;
- входные данные:
  - inter – массив содержащий границы отрезка;

  - fun – указатель на минимизируемую функцию, принимающую и возвращающую вещественное число;
  - r – параметр метода;
  - e – вещественное число, заданная точность поиска.
- выходные данные:
  - new_point – вещественное число, координата минимального значения функции.

```
size_t* Rmax(std::vector<double>* a, size_t n)
```

- назначние: поиск максимальной характеристики R;
- входные данные:
  - a – указатель вектор характеристик R каждого интервала;
  - n – количество нужных максимальных характеристик R.
- выходные данные:
  - r_max – указатель на массив с номерами интервалов с максимальной характеристикой осортированные по убыванию.

```
double ParallelNewPoints(double inter[2], double fun(double x), double r,
    double e)
```

- назначние: осуществляет параллельный поиск глобального минимума третьим способом;
- входные данные:
  - inter – массив содержащий границы отрезка;
  - fun – указатель на минимизируемую функцию, принимающую и возвращающую вещественное число;
  - r – параметр метода;
  - e – вещественное число, заданная точность поиска.
- выходные данные:
  - new_point – вещественное число, координата минимального значения функции.

2.3 Описание std::Thread-версии

```
size_t Rmax(std::vector<double>* a)
```

- назначние: поиск максимальной характеристики R;
- входные данные:
  - a – указатель вектор характеристик R каждого интервала.
- выходные данные:

– count – номер интервала с максимальной характеристикой.

```
void GSA(double inter[2], double fun(double x), double r, double e,
    std::promise<int>&& pr)
```

- назначние: существляет последовательный поиск глобального минимума, нужен для первого метода;
- входные данные:
  – inter – массив содержащий границы отрезка;
  – fun – указатель на минимизируемую функцию, принимающую и возвращающую вещественное число;
  – r – параметр метода;
  – e – вещественное число, заданная точность поиска;
  – pr - ссылка на вектор хранящий значения высчитанного каждым потоком.
- нет выходных данных.

```
double ParallelGSA(double inter[2], double fun(double x), double r, double
    e)
```

- назначние: осуществляет параллельный поиск глобального минимума первым способом;
- входные данные:
  – inter – массив содержащий границы отрезка;
  – fun – указатель на минимизируемую функцию, принимающую и возвращающую вещественное число;
  – r – параметр метода;
  – e – вещественное число, заданная точность поиска.
- выходные данные:
  – new_point – вещественное число, координата минимального значения функции.

```
void findM(std::vector<std::pair<double, double>>* point, size_t rank,
    std::promise<int>&& pr)
```

- назначние: осуществляет параллельный поиск максимального значения характеристики M;
- входные данные:
  – point – указатель на вектор хранящий точки;
  – rank – номер потока;
  – pr - ссылка на вектор хранящий значения высчитанного каждым потоком.

- нет выходных данных.

```
void calculateR(std::vector <double>* R, std::vector<std::pair<double,
    double>>* point, size_t rank, double m, std::promise<int>&& pr)
```

- назначние: осуществляет параллельный поиск максимального значения характеристики R;
- входные данные:
  - point – указатель на вектор хранящий точки;
  - rank – номер потока;
  - pr - ссылка на вектор хранящий значения высчитанного каждым потоком.
- нет выходных данных.

```
double ParallelOperations(double inter[2], double fun(double x), double r,
    double e)
```

- назначние: осуществляет параллельный поиск глобального минимума вторым способом;
- входные данные:
  - inter – массив содержащий границы отрезка;
  - fun – указатель на минимизируемую функцию, принимающую и возвращающую вещественное число;
  - r – параметр метода;
  - e – вещественное число, заданная точность поиска.
- выходные данные:
  - new_point – вещественное число, координата минимального значения функции.

```
size_t* Rmax(std::vector<double>* a, size_t n)
```

- назначние: поиск максимальной характеристики R;
- входные данные:
  - a – указатель вектор характеристик R каждого интервала;
  - n – количество нужных максимальных характеристик R.
- выходные данные:
  - r_max – указатель на массив с номерами интервалов с максимальной характеристикой осортированные по убыванию.

```
void calculateNewPoint(std::vector<std::pair<double, double>>* point,
    double (*fun)(double x), double *m, double *e, int *k, size_t *r, int i,
    double *result, std::promise<int>&& pr)
```

- назначние: осуществляет параллельное высчитывание новых точек;
- входные данные:
  - point – указатель на вектор хранящий точки;
  - fun – указатель на минимизируемую функцию, принимающую и возвращающую вещественное число;
  - m – указатель характеристику M;
  - e – указатель на заданную точность метода;
  - k – указатель на сдвиг;
  - i – номер потока;
  - result – указатель на конечную точку;
  - pr - ссылка на вектор хранящий значения высчитанного каждым потоком.
- нет выходных данных.

```
double ParallelNewPoints(double inter[2], double fun(double x), double r,
    double e)
```

- назначние: осуществляет параллельный поиск глобального минимума третьим способом;
- входные данные:
  - inter – массив содержащий границы отрезка;
  - fun – указатель на минимизируемую функцию, принимающую и возвращающую вещественное число;
  - r – параметр метода;
  - e – вещественное число, заданная точность поиска.
- выходные данные:
  - new_point – вещественное число, координата минимального значения функции.

Результаты экспериментов

Вычислительные эксперименты для оценки эффективности последовательного и параллельного алгоритмов глобального поиска проводились на оборудовании со следующей аппаратной конфигурацией:

- Процессор: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz, 6 ядра;
- Оперативная память: 16 ГБ;
- ОС: Microsoft Windows 10 Home 64-bit.

Вычисления производились с заданной точностью $\epsilon = 10^{-3}$ и параметром метода $k = 2$ на 12 потоках.

Запуск проводился 101 раз и бралась медиана по данным, для того чтобы отсеить какие-либо погрешности.

В таблице ниже представлено время работы в миллисекундах каждого реализованного алгоритма поиска глобального минимума. Первый столбец содержит порядковый номер тестируемой функции.

№	Sequential	OMP 1st	OMP 2nd	OMP 3rd	Thread 1st	Thread 2nd	Thread 3rd
1	140	17	8	9	41	2291	197
2	39	12	3	12	24	780	128
3	272	62	25	16	48	4481	249
4	153	3	6	7	14	1115	144
5	11	4	1	4	15	534	99
6	99	36	5	7	71	963	157
7	28	12	2	6	22	1009	121
8	246	58	18	12	51	3651	221
9	126	45	8	12	45	2098	231
10	107	17	7	11	25	1952	199
11	196	10	22	23	22	3237	262
12	239	11	13	18	18	2506	241
13	119	2	3	5	11	905	125
14	31	12	1	3	16	739	75
15	630	19	59	55	24	3362	475
16	681	10	39	36	17	5601	375
17	942	17	60	57	24	6908	434
18	250	6	7	10	13	1881	197
19	64	13	3	7	17	1316	138
20	53	33	5	7	41	4507	152

Таблица 1. Время работы алгоитмов глобального поиска. Время в миллисекундах

Из таблицы 1 видно, что наиболее эффективно работает 2-ой и 3-ий способы распараллеливания по характеристикам и точкам с помощью OpenMP. А расспараелеливание std::Thread 2-ой и 3-ий методы работают хуже, так как на каждой итерации мы вынуждены обнулять и заново создавать переменные для синхронизации потоков, что значительно ухуджает производительность.

Заключение

Был изучен алгоритм глобального поиска для одномерных многоэкстремальных задач оптимизации. Также были разработаны последовательная и параллельная реализации данного алгоритма и протестированы на различных минимизируемых функциях с разными входными параметрами.

Основной задачей данной работы была реализация эффективной параллельной версии. Эта цель была успешно достигнута, что подтверждается результатами экспериментов, проведенных в ходе работы. Из результатов тестирования можно сделать вывод, что реализованные алгоритмы работают корректно.

В ходе дальнейшей работы планируется реализовать последовательный алгоритм глобального поиска для многомерных многоэкстремальных задач оптимизации.

Также в ходе работы был изучены, настроены и внедрены Cmake проекта, необходимый для автоматизированной сборки программного обеспечения из исходного кода, и Continuous Integration на GitHub для проверки корректности работы.

Литература

1. Стронгин Р.Г., Гергель В.П., Гришагин В.А., Баркалов К.А. Параллельные вычисления в задачах глобальной оптимизации: Монография / Предисл.: В.А. Садовничий. – М.: Издательство Московского университета, 2013. – 280 с., илл.

2. Сергеев Я.Д., Квасов Д.Е. Краткое введение в теорию липшицевой глобальной оптимизации: Учебно-методическое пособие. – Нижний Новгород: Изд-во ННГУ, 2016. – 48с.

3. Документация по TBB [Электронный ресурс] // URL: `https://software.intel.com/content/www/ru/ru/develop/articles/tbb_async_io`

Приложение

Sequestion проект:

main.cpp

```
// Copyright 2021 Smirnov Aleksandr
#include <gtest/gtest.h>

#include <vector>
#include <ctime>

#include "./func.h"
#include "./hansen_functions.h"

::testing::AssertionResult resultInExpected(std::vector<double> expected,
    double result) {
    for (size_t i = 0; i < expected.size(); i++) {
        if (round(expected[i] * 10) / 10 == round(result * 10) / 10) {
            return ::testing::AssertionSuccess();
        }
    }
    std::cout << "Actual:\n    " << result << "\nExpected:\n    {";
    for (double i : expected) std::cout << i << " ";
    std::cout << "}" << std::endl;
    return ::testing::AssertionFailure();

}

typedef testing::TestWithParam<std::tuple< double(*)(double), double*,
    std::vector<double>, double, double>> Sequential;
TEST_P(Sequential, Test) {
    // Arrange
    double (*fun)(double) = std::get<0>(GetParam());
    double* param = std::get<1>(GetParam());
    std::vector<double> expected = std::get<2>(GetParam());
    double r = std::get<3>(GetParam());
    double e = std::get<4>(GetParam());
    double result;
    double start = clock();

    // Actz
    result = GSA(param, fun, r, e);

    printf(">>> TIME = %.4lf sec\n", (clock() - start) / CLOCKS_PER_SEC);

    //Assert
    ASSERT_TRUE(resultInExpected(expected, result));
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}

INSTANTIATE_TEST_SUITE_P(/**/, Sequential, testing::Values(
    std::make_tuple(pfn[0], intervals[0], res[0], 2., 1e-3),
    std::make_tuple(pfn[1], intervals[1], res[1], 2., 1e-3),
    std::make_tuple(pfn[2], intervals[2], res[2], 2., 1e-3),
```

```
    std::make_tuple(pfn[3], intervals[3], res[3], 2., 1e-3),
    std::make_tuple(pfn[4], intervals[4], res[4], 2., 1e-3),
    std::make_tuple(pfn[5], intervals[5], res[5], 2., 1e-3),
    std::make_tuple(pfn[6], intervals[6], res[6], 2., 1e-3),
    std::make_tuple(pfn[7], intervals[7], res[7], 2., 1e-3),
    std::make_tuple(pfn[8], intervals[8], res[8], 2., 1e-3),
    std::make_tuple(pfn[9], intervals[9], res[9], 2., 1e-3),
    std::make_tuple(pfn[10], intervals[10], res[10], 2., 1e-3),
    std::make_tuple(pfn[11], intervals[11], res[11], 2., 1e-3),
    std::make_tuple(pfn[12], intervals[12], res[12], 2., 1e-3),
    std::make_tuple(pfn[13], intervals[13], res[13], 2., 1e-3),
    std::make_tuple(pfn[14], intervals[14], res[14], 2., 1e-3),
    std::make_tuple(pfn[15], intervals[15], res[15], 2., 1e-3),
    std::make_tuple(pfn[16], intervals[16], res[16], 2., 1e-3),
    std::make_tuple(pfn[17], intervals[17], res[17], 2., 1e-3),
    std::make_tuple(pfn[18], intervals[18], res[18], 2., 1e-3),
    std::make_tuple(pfn[19], intervals[19], res[19], 2., 1e-3)
));
```

fun.h

```
// Copyright 2021 Smirnov Aleksandr
#ifndef MODULES_SEQUENCE_FUNC_H_
#define MODULES_SEQUENCE_FUNC_H_

#include <vector>

double GSA(double inter[2], double fun(double x), double r, double e);

#endif  // MODULES_SEQUENCE_FUNC_H_
```

fun.cpp

```
// Copyright 2021 Smirnov Aleksandr
#include "stdio.h"
#include "math.h"
#include "time.h"

#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>

#include "../../modules/Sequence/func.h"

size_t Rmax(std::vector<double> *a){
    double *R = a->data();
    size_t count = 0;
    for (size_t i = 1; i < a->size(); i++) {
        if (*(a->data() + i) > *R) {
            R = a->data() + i;
            count = i;
        }
    }
    return count;
};
```

```
double GSA(double inter[2], double fun(double x), double r, double e){
    double left = inter[0];
    double right = inter[1];
    double M = 0;
    double new_point;
    double m;
    size_t k = 2;
    size_t r_max = 0;
    std::vector <std::pair<double, double>> point{ std::pair<double,
        double>(left, fun(left)),
                                                   std::pair<double,
                                                       double>(right,
                                                       fun(right)) };
    std::vector <double> R;
    while ((point[r_max + 1].first - point[r_max].first) > e) {
        M = 0;
        for (auto it1 = point.begin(), it2 = ++point.begin(); it2 !=
            point.end(); it1++, it2++)
            M = fmax(M, abs((it2->second - it1->second) / (it2->first -
                it1->first)));
        m = (M == 0) ? 1 : r * M;
        for (auto it1 = point.begin(), it2 = ++point.begin(); it2 !=
            point.end(); it1++, it2++)
            R.push_back(m * (it2->first - it1->first) + ((it2->second -
                it1->second) * (it2->second - it1->second)) /
                        (m * (it2->first - it1->first)) - 2 * (it2->second +
                            it1->second));
        r_max = Rmax(&R);
        R.clear();
        new_point = (point[r_max + 1].first + point[r_max].first) / 2 -
            (point[r_max + 1].second - point[r_max].second) / (2 * m);
        point.push_back(std::pair<double, double>(new_point, fun(new_point)));
        std::sort(point.begin(), point.end());
        k++;
    }
    return point[r_max].first;
}
```

OpenMP проект:

main.cpp

```
// Copyright 2021 Smirnov Aleksandr
#include <gtest/gtest.h>

#include <vector>

#include "./ops_omp.h"
#include "./hansen_functions.h"
#include <omp.h>

::testing::AssertionResult resultInExpected(std::vector<double> expected,
    double result) {
    for (size_t i = 0; i < expected.size(); i++) {
        if (round(expected[i] * 10) / 10 == round(result * 10) / 10) {
            return ::testing::AssertionSuccess();
        }
    }
```

```
    std::cout << "Actual:\n    " << result << "\nExpected:\n    {";
    for (double i : expected) std::cout << i << " ";
    std::cout << "}" << std::endl;
    return ::testing::AssertionFailure();

}

typedef testing::TestWithParam<std::tuple< double(*)(double), double*,
    std::vector<double>, double, double>> OpenMP;
TEST_P(OpenMP, Division_into_segments) {
    // Arrange
    double (*fun)(double) = std::get<0>(GetParam());
    double* param = std::get<1>(GetParam());
    std::vector<double> expected = std::get<2>(GetParam());
    double r = std::get<3>(GetParam());
    double e = std::get<4>(GetParam());
    double result;
    double start = clock();

    // Actz
    result = ParallelGSA(param, fun, r, e);

    printf(">>> TIME = %.4lf sec\n", (clock() - start) / CLOCKS_PER_SEC);

    //Assert
    ASSERT_TRUE(resultInExpected(expected, result));
}

typedef testing::TestWithParam<std::tuple< double(*)(double), double*,
    std::vector<double>, double, double>> OpenMP;
TEST_P(OpenMP, Internals_parallelization) {
    // Arrange
    double (*fun)(double) = std::get<0>(GetParam());
    double* param = std::get<1>(GetParam());
    std::vector<double> expected = std::get<2>(GetParam());
    double r = std::get<3>(GetParam());
    double e = std::get<4>(GetParam());
    double result;
    double start = clock();

    // Actz
    result = ParallelOperations(param, fun, r, e);

    printf(">>> TIME = %.4lf sec\n", (clock() - start) / CLOCKS_PER_SEC);

    //Assert
    ASSERT_TRUE(resultInExpected(expected, result));
}

typedef testing::TestWithParam<std::tuple< double(*)(double), double*,
    std::vector<double>, double, double>> OpenMP;
TEST_P(OpenMP, Parallel_search_for_new_points) {
    // Arrange
    double (*fun)(double) = std::get<0>(GetParam());
    double* param = std::get<1>(GetParam());
    std::vector<double> expected = std::get<2>(GetParam());
    double r = std::get<3>(GetParam());
    double e = std::get<4>(GetParam());
    double result;
    double start = clock();
```

```
    // Actz
    result = ParallelNewPoints(param, fun, r, e);

    printf(">>> TIME = %.4lf sec\n", (clock() - start) / CLOCKS_PER_SEC);

    //Assert
    ASSERT_TRUE(resultInExpected(expected, result));
}

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}

INSTANTIATE_TEST_SUITE_P(/**/, OpenMP, testing::Values(
    std::make_tuple(pfn[0], intervals[0], res[0], 2., 1e-3),
    std::make_tuple(pfn[1], intervals[1], res[1], 2., 1e-3),
    std::make_tuple(pfn[2], intervals[2], res[2], 2., 1e-3),
    std::make_tuple(pfn[3], intervals[3], res[3], 2., 1e-3),
    std::make_tuple(pfn[4], intervals[4], res[4], 2., 1e-3),
    std::make_tuple(pfn[5], intervals[5], res[5], 2., 1e-3),
    std::make_tuple(pfn[6], intervals[6], res[6], 2., 1e-3),
    std::make_tuple(pfn[7], intervals[7], res[7], 2., 1e-3),
    std::make_tuple(pfn[8], intervals[8], res[8], 2., 1e-3),
    std::make_tuple(pfn[9], intervals[9], res[9], 2., 1e-3),
    std::make_tuple(pfn[10], intervals[10], res[10], 2., 1e-3),
    std::make_tuple(pfn[11], intervals[11], res[11], 2., 1e-3),
    std::make_tuple(pfn[12], intervals[12], res[12], 2., 1e-3),
    std::make_tuple(pfn[13], intervals[13], res[13], 2., 1e-3),
    std::make_tuple(pfn[14], intervals[14], res[14], 2., 1e-3),
    std::make_tuple(pfn[15], intervals[15], res[15], 2., 1e-3),
    std::make_tuple(pfn[16], intervals[16], res[16], 2., 1e-3),
    std::make_tuple(pfn[17], intervals[17], res[17], 2., 1e-3),
    std::make_tuple(pfn[18], intervals[18], res[18], 2., 1e-3),
    std::make_tuple(pfn[19], intervals[19], res[19], 2., 1e-3)
));
```

ops_omp.h

```
// Copyright 2021 Smirnov Aleksandr
#ifndef MODULES_OPENMP_OPS_OMP_H_
#define MODULES_OPENMP_OPS_OMP_H_

#include <vector>

double ParallelGSA(double inter[2], double (*fun)(double x), double r, double
    e);
double ParallelOperations(double inter[2], double (*fun)(double x), double r,
    double e);
double ParallelNewPoints(double inter[2], double (*fun)(double x), double r,
    double e);

#endif  // MODULES_OPENMP_OPS_OMP_H_
```

ops_omp.cpp

```cpp
// Copyright 2021 Smirnov Aleksandr
#include <omp.h>
#include "stdio.h"
#include "math.h"
#include "time.h"

#include <iostream>
#include <algorithm>
#include <limits>
#include <vector>
#include <iterator>

#include "../../modules/OpenMP/ops_omp.h"

#define NUM_THREADS 12

size_t Rmax(std::vector<double>* a) {
    double* R = a->data();
    size_t count = 0;
    for (size_t i = 1; i < a->size(); i++) {
        if (*(a->data() + i) > *R) {
            R = a->data() + i;
            count = i;
        }
    }
    return count;
};

// true algorithm
double GSA(double inter[2], double (*fun)(double x), double r, double e) {
    double left = inter[0];
    double right = inter[1];
    double M = 0;
    double new_point;
    double m;
    size_t k = 2;
    size_t r_max = 0;
    std::vector <std::pair<double, double>> point{ std::pair<double,
       double>(left, fun(left)), std::pair<double, double>(right, fun(right)) };
    std::vector <double> R;
    while (abs(point[r_max + 1].first - point[r_max].first) > e) {
        M = 0;
        for (auto it1 = point.begin(), it2 = ++point.begin(); it2 !=
            point.end(); it1++, it2++)
            M = fmax(M, abs((it2->second - it1->second) / (it2->first -
                it1->first)));
        m = (M == 0) ? 1 : r * M;
        for (auto it1 = point.begin(), it2 = ++point.begin(); it2 !=
            point.end(); it1++, it2++)
            R.push_back(m * (it2->first - it1->first) + ((it2->second -
                it1->second) * (it2->second - it1->second)) / (m * (it2->first -
                it1->first)) - 2 * (it2->second + it1->second));
        r_max = Rmax(&R);
        R.clear();
        new_point = (point[r_max + 1].first + point[r_max].first) / 2 -
            (point[r_max + 1].second - point[r_max].second) / (2 * m);
        point.push_back(std::pair<double, double>(new_point, fun(new_point)));
        std::sort(point.begin(), point.end());
        k++;
```

```
    }
    return point[r_max].first;
};

// parallelization into segments
double ParallelGSA(double inter[2], double (*fun)(double x), double r, double
    e) {
    omp_set_num_threads(NUM_THREADS);
    double left = inter[0];
    double right = inter[1];
    double min = left;

    double h = (right - left) / (double)NUM_THREADS;

#pragma omp parallel
    {
        double min_local;
#pragma omp for nowait
        for (int i = 0; i < NUM_THREADS; i++) {
            int thread_num = omp_get_thread_num();
            double lr[2] = { left + h * thread_num, left + h * (thread_num + 1)
                };
            min_local = GSA(lr, fun, r, e);
        }
#pragma omp critical
        {
            if (fun(min_local) <= fun(min)) {
                min = min_local;
            }
        }
    }

    return min;
};

// parallelization internals
double ParallelOperations(double inter[2], double (*fun)(double x), double r,
    double e) {
    double left = inter[0];
    double right = inter[1];

    double M;
    double new_point;
    double m;
    int k = 2;
    int r_max = 0;
    std::vector <std::pair<double, double>> point{ std::pair<double,
        double>(left, fun(left)), std::pair<double, double>(right, fun(right)) };
    std::vector <double> R;
    while ((point[r_max + 1].first - point[r_max].first) > e) {
        M = 0;
#pragma omp parallel
        {
#pragma omp for nowait
            for (int i = 0; i < k - 1; i++) {
                double tmpM = abs((point[i + 1].second - point[i].second) /
                    (point[i + 1].first - point[i].first));
#pragma omp critical
                if (M < tmpM) M = tmpM;
            }
```

```
        }
        m = (M == 0) ? 1 : r * M;

        R.push_back(0);

#pragma omp for  //nowait
        for (int i = 0; i < k - 1; i++)
            R[i] = m * (point[i + 1].first - point[i].first) + ((point[i +
                1].second - point[i].second) * (point[i + 1].second -
                point[i].second)) / (m * (point[i + 1].first - point[i].first))
                - 2 * (point[i + 1].second + point[i].second);

        r_max = Rmax(&R);
        new_point = (point[r_max + 1].first + point[r_max].first) / 2 -
            (point[r_max + 1].second - point[r_max].second) / (2 * m);
        point.push_back(std::pair<double, double>(new_point, fun(new_point)));
        std::sort(point.begin(), point.end());
        k++;
    }
    return point[r_max].first;
};

size_t* RmaxN(std::vector <std::pair<double, int>> R, size_t n) {
    size_t* r_max = new size_t[n];
    std::sort(R.begin(), R.end());
    for (size_t i = 0; i < n; i++)
        r_max[i] = R[R.size() - 1 - i].second;
    return r_max;
};

// parallel search for newpoints
double ParallelNewPoints(double inter[2], double (*fun)(double x), double r,
    double e) {
    double left = inter[0];
    double right = inter[1];
    double M;
    double new_point = 0;
    double m;
    int k = 1;
    size_t* r_max;
    std::vector <std::pair<double, double>> point{ std::pair<double,
        double>(left, fun(left)), std::pair<double, double>(right, fun(right)) };
    std::vector <std::pair<double, int>> R = { std::pair<double, int>(0, 0) };
    bool flag = false;
    while (!flag) {
        M = 0;
        for (auto it1 = point.begin(), it2 = ++point.begin(); it2 !=
            point.end(); it1++, it2++)
            M = fmax(M, abs((it2->second - it1->second) / (it2->first -
                it1->first)));
        m = (M == 0) ? 1 : r * M;

        for (int i = 0; i < k; i++)
            R[i].first = m * (point[i + 1].first - point[i].first) + ((point[i
                + 1].second - point[i].second) * (point[i + 1].second -
                point[i].second)) / (m * (point[i + 1].first - point[i].first))
                - 2 * (point[i + 1].second + point[i].second);

        int n = R.size() < NUM_THREADS ? R.size() : NUM_THREADS;
        omp_set_num_threads(n);
```

```
        for (int i = 0; i < n; i++)
            point.push_back(std::pair<double, double>(0, 0));
        r_max = RmaxN(R, n);
#pragma omp parallel
        {
            double new_point_local;
#pragma omp for
            for (int i = 0; i < n; i++) {
                size_t r = r_max[i];
                new_point_local = (point[r + 1].first + point[r].first) / 2 -
                    (point[r + 1].second - point[r].second) / (2 * m);
                point[k + 1 + i] = std::pair<double, double>(new_point_local,
                    fun(new_point_local));
#pragma omp critical
                if (point[r + 1].first - point[r].first < e) {
                    new_point = new_point_local;
                    flag = true;
                }
            }
        }
        std::sort(point.begin(), point.end());

        for (int i = 0; i < n; i++)
            R.push_back(std::pair<double, int>(0, R.size()));

        k += n;
        delete[]r_max;
    }
    return new_point;
};
```

std::Thread проект:

main.cpp

```
// Copyright 2021 Smirnov Aleksandr
#include <gtest/gtest.h>

#include <vector>

#include "../../3rdparty/unapproved/unapproved.h"
#include "./ops_std.h"
#include "./hansen_functions.h"

::testing::AssertionResult resultInExpected(std::vector<double> expected,
    double result) {
    for (size_t i = 0; i < expected.size(); i++) {
        if (round(expected[i] * 10) / 10 == round(result * 10) / 10) {
            return ::testing::AssertionSuccess();
        }
    }
    std::cout << "Actual:\n␣␣␣␣" << result << "\nExpected:\n␣␣␣␣{␣";
    for (double i : expected) std::cout << i << "␣";
    std::cout << "}" << std::endl;
    return ::testing::AssertionFailure();

}
```

```
typedef testing::TestWithParam<std::tuple< double(*)(double), double*,
    std::vector<double>, double, double>> Threads;
TEST_P(Threads, Division_into_segments) {
    // Arrange
    double (*fun)(double) = std::get<0>(GetParam());
    double* param = std::get<1>(GetParam());
    std::vector<double> expected = std::get<2>(GetParam());
    double r = std::get<3>(GetParam());
    double e = std::get<4>(GetParam());
    double result;
    double start = clock();

    // Actz
    result = ParallelGSA(param, fun, r, e);

    printf(">>> TIME = %.4lf sec\n", (clock() - start) / CLOCKS_PER_SEC);

    //Assert
    ASSERT_TRUE(resultInExpected(expected, result));
}

typedef testing::TestWithParam<std::tuple< double(*)(double), double*,
    std::vector<double>, double, double>> Threads;
TEST_P(Threads, Internals_parallelization) {
    // Arrange
    double (*fun)(double) = std::get<0>(GetParam());
    double* param = std::get<1>(GetParam());
    std::vector<double> expected = std::get<2>(GetParam());
    double r = std::get<3>(GetParam());
    double e = std::get<4>(GetParam());
    double result;
    double start = clock();

    // Actz
    result = ParallelOperations(param, fun, r, e);

    printf(">>> TIME = %.4lf sec\n", (clock() - start) / CLOCKS_PER_SEC);

    //Assert
    ASSERT_TRUE(resultInExpected(expected, result));
}

typedef testing::TestWithParam<std::tuple< double(*)(double), double*,
    std::vector<double>, double, double>> Threads;
TEST_P(Threads, Parallel_search_for_new_points) {
    // Arrange
    double (*fun)(double) = std::get<0>(GetParam());
    double* param = std::get<1>(GetParam());
    std::vector<double> expected = std::get<2>(GetParam());
    double r = std::get<3>(GetParam());
    double e = std::get<4>(GetParam());
    double result;
    double start = clock();

    // Actz
    result = ParallelNewPoints(param, fun, r, e);

    printf(">>> TIME = %.4lf sec\n", (clock() - start) / CLOCKS_PER_SEC);

    //Assert
```

```
    ASSERT_TRUE(resultInExpected(expected, result));
}

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}

INSTANTIATE_TEST_SUITE_P(/**/, Threads, testing::Values(
    std::make_tuple(pfn[0], intervals[0], res[0], 2., 1e-3),
    std::make_tuple(pfn[1], intervals[1], res[1], 2., 1e-3),
    std::make_tuple(pfn[2], intervals[2], res[2], 2., 1e-3),
    std::make_tuple(pfn[3], intervals[3], res[3], 2., 1e-3),
    std::make_tuple(pfn[4], intervals[4], res[4], 2., 1e-3),
    std::make_tuple(pfn[5], intervals[5], res[5], 2., 1e-3),
    std::make_tuple(pfn[6], intervals[6], res[6], 2., 1e-3),
    std::make_tuple(pfn[7], intervals[7], res[7], 2., 1e-3),
    std::make_tuple(pfn[8], intervals[8], res[8], 2., 1e-3),
    std::make_tuple(pfn[9], intervals[9], res[9], 2., 1e-3),
    std::make_tuple(pfn[10], intervals[10], res[10], 2., 1e-3),
    std::make_tuple(pfn[11], intervals[11], res[11], 2., 1e-3),
    std::make_tuple(pfn[12], intervals[12], res[12], 2., 1e-3),
    std::make_tuple(pfn[13], intervals[13], res[13], 2., 1e-3),
    std::make_tuple(pfn[14], intervals[14], res[14], 2., 1e-3),
    std::make_tuple(pfn[15], intervals[15], res[15], 2., 1e-3),
    std::make_tuple(pfn[16], intervals[16], res[16], 2., 1e-3),
    std::make_tuple(pfn[17], intervals[17], res[17], 2., 1e-3),
    std::make_tuple(pfn[18], intervals[18], res[18], 2., 1e-3),
    std::make_tuple(pfn[19], intervals[19], res[19], 2., 1e-3)
));
```

ops_std.h

```
// Copyright 2021 Smirnov Aleksandr
#ifndef MODULES_THREAD_OPS_STD_H_
#define MODULES_THREAD_OPS_STD_H_

#include <vector>

double ParallelGSA(double inter[2], double (*fun)(double x), double r, double
    e);
double ParallelOperations(double inter[2], double (*fun)(double x), double r,
    double e);
double ParallelNewPoints(double inter[2], double (*fun)(double x), double r,
    double e);

#endif  // MODULES_THREAD_OPS_STD_H_
```

ops_std.cpp

```
// Copyright 2021 Smirnov Aleksandr
#include <vector>
#include <string>
#include <utility>
#include <random>
#include <iostream>
#include <algorithm>
```

```
#include "../../modules/Thread/ops_std.h"
#include "../../3rdparty/unapproved/unapproved.h"

std::mutex my_mutex;

size_t Rmax(std::vector<double>* a) {
    double* R = a->data();
    size_t count = 0;
    for (size_t i = 1; i < a->size(); i++) {
        if (*(a->data() + i) > *R) {
            R = a->data() + i;
            count = i;
        }
    }
    return count;
};

// true algorithm
void GSA(double inter[2], double (*fun)(double x), double r, double e,
    std::promise<int>&& pr) {
    double left = inter[0];
    double right = inter[1];
    double M = 0;
    double new_point;
    double m;
    size_t k = 2;
    size_t r_max = 0;
    std::vector <std::pair<double, double>> point{ std::pair<double,
        double>(left, fun(left)), std::pair<double, double>(right, fun(right)) };
    std::vector <double> R;
    while ((point[r_max + 1].first - point[r_max].first) > e) {
        M = 0;
        for (auto it1 = point.begin(), it2 = ++point.begin(); it2 !=
            point.end(); it1++, it2++)
            M = fmax(M, std::abs((it2->second - it1->second) / (it2->first -
                it1->first)));
        m = (M == 0) ? 1 : r * M;
        for (auto it1 = point.begin(), it2 = ++point.begin(); it2 !=
            point.end(); it1++, it2++)
            R.push_back(m * (it2->first - it1->first) + ((it2->second -
                it1->second) * (it2->second - it1->second)) / (m * (it2->first -
                it1->first)) - 2 * (it2->second + it1->second));
        r_max = Rmax(&R);
        R.clear();
        new_point = (point[r_max + 1].first + point[r_max].first) / 2 -
            (point[r_max + 1].second - point[r_max].second) / (2 * m);
        point.push_back(std::pair<double, double>(new_point, fun(new_point)));
        std::sort(point.begin(), point.end());
        k++;
    }
    pr.set_value(round(point[r_max].first * 10));
};

// parallelization into segments
double ParallelGSA(double inter[2], double (*fun)(double x), double r, double
    e) {
    const int nthreads = std::thread::hardware_concurrency();
    const double delta = (inter[1] - inter[0]) / nthreads;
    std::promise<int>* promises = new std::promise<int>[nthreads];
    std::future<int>* futures = new std::future<int>[nthreads];
```

```
    std::thread* threads = new std::thread[nthreads];

    for (int i = 0; i < nthreads; i++) {
        futures[i] = promises[i].get_future();
        double lr[2] = { inter[0] + (double)i * delta,inter[0] + (double)(i +
            1) * delta };
        threads[i] = std::thread(GSA, lr, fun, r, e, std::move(promises[i]));
        threads[i].join();
    }
    double min = inter[0];
    for (int i = 0; i < nthreads; i++) {
        double tmp = (double)futures[i].get() / 10.0;
        double fmin = fun(min);
        double ftmp = fun(tmp);
        if (ftmp <= fmin) {
            min = tmp;
        }
    }
    delete[] promises;
    delete[] futures;
    delete[] threads;
    return min;
};

void findM(std::vector<std::pair<double, double>>* point, size_t rank,
    std::promise<int>&& pr) {
    const int nthreads = std::thread::hardware_concurrency();
    std::pair<double, double>* point1 = point->data() + rank;
    std::pair<double, double>* point2 = point->data() + rank + 1;
    double M = 0;
    for (size_t i = rank + 1; i < point->size(); i += nthreads, point1 +=
        nthreads, point2 += nthreads) {
        double tmpM = std::abs((point2->second - point1->second) /
            (point2->first - point1->first));
        if (M < tmpM) M = tmpM;
    }
    pr.set_value(round(M * 10000));
};

void calculateR(std::vector <double>* R, std::vector<std::pair<double,
    double>>* point, size_t rank, double m, std::promise<int>&& pr) {
    const int nthreads = std::thread::hardware_concurrency();
    std::pair<double, double>* point1 = point->data() + rank;
    std::pair<double, double>* point2 = point->data() + rank + 1;
    double* Ri = R->data() + rank;
    for (size_t i = rank; i < R->size(); i += nthreads, Ri += nthreads, point1
        += nthreads, point2 += nthreads) {
        double tmp = (m * (point2->first - point1->first) + ((point2->second -
            point1->second) * (point2->second - point1->second)) / (m *
            (point2->first - point1->first)) - 2 * (point2->second +
            point1->second));
        *Ri = tmp;
    }
    pr.set_value(1);
};

// parallelization internals
double ParallelOperations(double inter[2], double (*fun)(double x), double r,
    double e) {
    const int nthreads = std::thread::hardware_concurrency();
```

```
std::promise<int>* promisesM;
std::future<int>* futuresM = new std::future<int>[nthreads];
std::thread* threadsM = new std::thread[nthreads];

std::promise<int>* promisesR;
std::future<int>* futuresR = new std::future<int>[nthreads];
std::thread* threadsR = new std::thread[nthreads];

double left = inter[0];
double right = inter[1];
double M;
double new_point;
double m;
int k = 2;
int r_max = 0;
std::vector <std::pair<double, double>> point{ std::pair<double,
    double>(left, fun(left)), std::pair<double, double>(right, fun(right)) };
std::vector <double> R;
while ((point[r_max + 1].first - point[r_max].first) > e) {
    promisesM = new std::promise<int>[nthreads];
    promisesR = new std::promise<int>[nthreads];
    M = 0;
    if (k >= nthreads + 1) {
        for (int i = 0; i < nthreads; i++) {
            futuresM[i] = promisesM[i].get_future();
            threadsM[i] = std::thread(findM, &point, i,
                std::move(promisesM[i]));
            threadsM[i].join();
        }
        for (int i = 0; i < nthreads && i < k - 1; i++) {
            int tmp = (double)futuresM[i].get() / 10000.0;
            if (M < tmp) M = tmp;
            futuresM[i].valid();
        }
    }
    else {
        for (int i = 0; i < k - 1; i++) {
            double tmpM = std::abs((point[i + 1].second - point[i].second)
                / (point[i + 1].first - point[i].first));
            if (M < tmpM) M = tmpM;
        }
    }

    m = (M == 0) ? 1 : r * M;

    R.push_back(0);

    if (k >= nthreads + 1) {
        int count = 0;
        for (int i = 0; i < nthreads; i++) {
            futuresR[i] = promisesR[i].get_future();
            threadsR[i] = std::thread(calculateR, &R, &point, i, m,
                std::move(promisesR[i]));
            threadsR[i].join();
            count += futuresR[i].get();
        }
    }
    else {
        for (int i = 0; i < k - 1; i++) {
            R[i] = m * (point[i + 1].first - point[i].first) + ((point[i +
```

```
                    1].second - point[i].second) * (point[i + 1].second -
                    point[i].second)) / (m * (point[i + 1].first -
                    point[i].first)) - 2 * (point[i + 1].second +
                    point[i].second);
            }
        }

        r_max = Rmax(&R);
        new_point = (point[r_max + 1].first + point[r_max].first) / 2 -
            (point[r_max + 1].second - point[r_max].second) / (2 * m);
        size_t left = 0, right = point.size();
        while (left < right) {
            size_t mid = (left + right) / 2;
            if (new_point < point[mid].first)
                right = mid;
            else
                left = mid + 1;
        }
        point.insert(point.begin() + left, std::pair<double, double>(new_point,
            fun(new_point)));
        k++;
        delete[] promisesM;
        delete[] promisesR;
    }
    delete[] threadsM;
    delete[] futuresM;
    delete[] threadsR;
    delete[] futuresR;
    return point[r_max].first;
};

size_t* RmaxN(std::vector<std::pair<double, int>> R, size_t n) {
    size_t* r_max = new size_t[n];
    std::sort(R.begin(), R.end());
    for (size_t i = 0; i < n; i++)
        r_max[i] = R[R.size() - 1 - i].second;
    return r_max;
};

void calculateNewPoint(std::vector<std::pair<double, double>>* point, double
    (*fun)(double x), double *m, double *e, int *k, size_t *r, int i /*rank*/,
    double *result, std::promise<int>&& pr) {
    double new_point = ((*(point->data() + *r + 1)).first + (*(point->data() +
        *r)).first) / 2 - ((*(point->data() + *r + 1)).second - (*(point->data()
        + *r)).second) / (2 * (*m));
    *(point->data() + *k + 1 + i) = std::pair<double, double>(new_point,
        fun(new_point));
    if ((*(point->data() + *r + 1)).first - (*(point->data() + *r)).first < *e)
        {
        pr.set_value(1);
        *result = new_point;
    }
    else
        pr.set_value(0);
};

// parallel search for newpoints
double ParallelNewPoints(double inter[2], double (*fun)(double x), double r,
    double e) {
    const int nthreads = std::thread::hardware_concurrency();
```

```
    std::promise<int>* promises;
    std::future<int>* futures = new std::future<int>[nthreads];
    std::thread* threads = new std::thread[nthreads];
    double left = inter[0];
    double right = inter[1];
    double M;
    double new_point = 0.;
    double m;
    int k = 1;
    size_t* r_max;
    std::vector <std::pair<double, double>> point{ std::pair<double,
       double>(left, fun(left)), std::pair<double, double>(right, fun(right)) };
    std::vector <std::pair<double, int>> R = { std::pair<double, int>(0, 0) };
    bool flag = false;
    while (!flag) {
        promises = new std::promise<int>[nthreads];
        M = 0;
        for (auto it1 = point.begin(), it2 = ++point.begin(); it2 !=
            point.end(); it1++, it2++)
            M = fmax(M, std::abs((it2->second - it1->second) / (it2->first -
                it1->first)));
        m = (M == 0) ? 1 : r * M;

        for (int i = 0; i < k; i++)
            R[i].first = m * (point[i + 1].first - point[i].first) + ((point[i
                + 1].second - point[i].second) * (point[i + 1].second -
                point[i].second)) / (m * (point[i + 1].first - point[i].first))
                - 2 * (point[i + 1].second + point[i].second);

        int n = (int)R.size() < nthreads ? (int)R.size() : nthreads;
        for (int i = 0; i < n; i++)
            point.push_back(std::pair<double, double>(0, 0));
        r_max = RmaxN(R, n);
        for (int i = 0; i < n; i++) {
            futures[i] = promises[i].get_future();
            threads[i] = std::thread(calculateNewPoint, &point, fun, &m, &e,
                &k, &r_max[i], i, &new_point, std::move(promises[i]));
            threads[i].join();
        }
        for (int i = 0; i < n; i++) {
            int tmp = futures[i].get();
            if (tmp == 1) {
                flag = true;
            }
            futures[i].valid();
        }
        std::sort(point.begin(), point.end());
        for (int i = 0; i < n; i++)
            R.push_back(std::pair<double, int>(0, R.size()));
        k += (int)n;
        delete[]r_max;
        delete[]promises;
    }
    delete[]threads;
    delete[]futures;
    return new_point;
}
```

Общие файлы

hansen_functions.h

```
// Copyright 2021 Smirnov Aleksandr
#ifndef MODULES_SEQUENCE_HANSEN_FUNCTION_H_
#define MODULES_SEQUENCE_HANSEN_FUNCTION_H_

#include <cmath>
#include <vector>

extern double intervals[20][2];

double hfunc1(double x);
double hfunc2(double x);
double hfunc3(double x);
double hfunc4(double x);
double hfunc5(double x);
double hfunc6(double x);
double hfunc7(double x);
double hfunc8(double x);
double hfunc9(double x);
double hfunc10(double x);
double hfunc11(double x);
double hfunc12(double x);
double hfunc13(double x);
double hfunc14(double x);
double hfunc15(double x);
double hfunc16(double x);
double hfunc17(double x);
double hfunc18(double x);
double hfunc19(double x);
double hfunc20(double x);
double hpfunc1(double x);
double hpfunc2(double x);
double hpfunc3(double x);
double hpfunc4(double x);
double hpfunc5(double x);
double hpfunc6(double x);
double hpfunc7(double x);
double hpfunc8(double x);
double hpfunc9(double x);
double hpfunc10(double x);
double hpfunc11(double x);
double hpfunc12(double x);
double hpfunc13(double x);
double hpfunc14(double x);
double hpfunc15(double x);
double hpfunc16(double x);
double hpfunc17(double x);
double hpfunc18(double x);
double hpfunc19(double x);
double hpfunc20(double x);

extern double(*pfn[])(double x);

extern std::vector<std::vector<double>> res;


#endif  // MODULES_SEQUENCE_HANSEN_FUNCTION_H_
```

hansen_functions.cpp

```cpp
// Copyright 2021 Smirnov Aleksandr
#include "../../modules/Sequence/hansen_functions.h"

double intervals[20][2] = { {-1.5, 11}, {2.7, 7.5}, {-10.0, 10.0}, {1.9, 3.9},
    {0.0, 1.2},
{-10.0, 10.0}, {2.7, 7.5}, {-10.0, 10.0}, {3.1, 20.4}, {0.0, 10.0},
{-1.57, 6.28}, {0.0, 6.28}, {0.001, 0.99}, {0.0, 4.0}, {-5.0, 5.0},
{-3.0, 3.0}, {-4.0, 4.0}, {0.0, 6.0}, {0.0, 6.5}, {-10.0, 10.0} };

double hfunc1(double x) {
    return pow(x, 6) / 6.0 - 52.0 / 25.0 * pow(x, 5) + 39.0 / 80.0 * pow(x, 4) +
        71.0 / 10.0 * pow(x, 3) - 79.0 / 20.0 * pow(x, 2) - x + 0.1;
}

double hfunc2(double x) {
    return sin(x) + sin(10 * x / 3);
}

double hfunc3(double x) {
    double res = 0;
    for (int i = 1; i < 6; i++)
        res += i * sin((i + 1) * x + i);
    return -res;
}

double hfunc4(double x) {
    return (-16 * x * x + 24 * x - 5) * exp(-x);
}

double hfunc5(double x) {
    return -(-3 * x + 1.4) * sin(18 * x);
}

double hfunc6(double x) {
    return -(x + sin(x)) * exp(-x * x);
}

double hfunc7(double x) {
    return sin(x) + sin(10 * x / 3) + log(x) - 0.84 * x + 3;
}

double hfunc8(double x) {
    double res = 0;
    for (int i = 1; i < 6; i++)
        res += i * cos((i + 1) * x + i);
    return -res;
}

double hfunc9(double x) {
    return sin(x) + sin(2.0 / 3.0 * x);
}

double hfunc10(double x) {
    return -x * sin(x);
}

double hfunc11(double x) {
    return 2 * cos(x) + cos(2 * x);
```

```
}

double hfunc12(double x) {
    return pow(sin(x), 3) + pow(cos(x), 3);
}

double hfunc13(double x) {
    double sgn = 0.0;
    if (x * x - 1 < 0)
        sgn = -1.0;
    else
        sgn = 1.0;
    return -pow(x * x, 1.0 / 3.0) + sgn * pow(sgn * (x * x - 1.0), 1.0 / 3.0);
}

double hfunc14(double x) {
    return -exp(-x) * sin(2 * acos(-1.0) * x);
}

double hfunc15(double x) {
    return (x * x - 5 * x + 6) / (x * x + 1);
}

double hfunc16(double x) {
    return 2 * (x - 3) * (x - 3) + exp(x * x / 2);
}

double hfunc17(double x) {
    return pow(x, 6) - 15 * pow(x, 4) + 27 * x * x + 250;
}

double hfunc18(double x) {
    if (x <= 3)
        return (x - 2) * (x - 2);
    else
        return 2 * log(x - 2) + 1;
}

double hfunc19(double x) {
    return -x + sin(3 * x) - 1;
}

double hfunc20(double x) {
    return -(x - sin(x)) * exp(-x * x);
}

double hpfunc1(double x) {
    return pow(x, 5) - 10.4 * pow(x, 4) + 1.95 * pow(x, 3) + 21.3 * x * x -
        7.9 * x - 1.0;
}

double hpfunc2(double x) {
    return cos(x) + 10.0 * cos(10.0 * x / 3.0) / 3.0;
}

double hpfunc3(double x) {
    double res = 0.0;
    for (int i = 1; i < 6; i++)
        res += i * (i + 1) * cos((i + 1) * x + i);
    return -res;
```

```
}

double hpfunc4(double x) {
    return (16.0 * x * x - 56.0 * x + 29.0) * exp(-x);
}

double hpfunc5(double x) {
    return 3.0 * sin(18.0 * x) - 18.0 * (-3.0 * x + 1.4) * cos(18.0 * x);
}

double hpfunc6(double x) {
    return (2.0 * x * (x + sin(x)) - cos(x) - 1) * exp(-x * x);
}

double hpfunc7(double x) {
    return cos(x) + 10.0 * cos(10.0 * x / 3.0) / 3.0 + 1 / x - 0.84;
}

double hpfunc8(double x) {
    double res = 0.0;
    for (int i = 1; i < 6; i++)
        res += i * (i + 1) * sin((i + 1) * x + i);
    return res;
}

double hpfunc9(double x) {
    return cos(x) + 2.0 * cos(2.0 * x / 3.0) / 3.0;
}

double hpfunc10(double x) {
    return -sin(x) - x * cos(x);
}

double hpfunc11(double x) {
    return -2.0 * (sin(x) + sin(2.0 * x));
}

double hpfunc12(double x) {
    return 3.0 * cos(x) * sin(x) * (sin(x) - cos(x));
}

double hpfunc13(double x) {
    double st = (1.0 / 3.0);
    if (x == 0.0)
        return 0.0;
    return (2.0 * x / pow((x * x - 1) * (x * x - 1), st) - 2.0 * pow(x, -st)) /
        3.0;
}

double hpfunc14(double x) {
    double pi = acos(-1.0);
    return exp(-x) * (sin(2.0 * pi * x) - 2.0 * pi * cos(2 * pi * x));
}

double hpfunc15(double x) {
    return (2.0 * x * (-x * x + 5.0 * x - 6.0) - (x * x + 1.0) * (5.0 - 2.0 *
        x))
        / ((x * x + 1.0) * (x * x + 1.0));
}
```

```
double hpfunc16(double x) {
    return 4.0 * (x - 3.0) + x * exp(x * x / 2.0);
}

double hpfunc17(double x) {
    return 6.0 * pow(x, 5) - 60.0 * x * x * x * x + 54.0 * x;
}

double hpfunc18(double x) {
    if (x <= 3)
        return 2.0 * x - 4.0;
    else
        return 2.0 / (x - 2.0);
}

double hpfunc19(double x) {
    return 3.0 * cos(3.0 * x) - 1.0;
}

double hpfunc20(double x) {
    return exp(-x * x) * (2.0 * x * (x - sin(x)) - 1.0 + cos(x));
}

double(*pfn[])(double x) = { hfunc1, hfunc2, hfunc3, hfunc4, hfunc5,
hfunc6, hfunc7, hfunc8, hfunc9, hfunc10,
hfunc11, hfunc12, hfunc13, hfunc14, hfunc15,
hfunc16, hfunc17, hfunc18, hfunc19, hfunc20 };

std::vector<std::vector<double>> res = { {10},{5.14575},{-0.49139, -6.77458,
    5.79179},{2.868},{0.966086},
{0.679578},{5.199776},{-0.80032, -7.08351, 5.48286},{17.0392},{7.97867},
{2.09444, 4.18879},{4.71239, 3.14159},{0.70711},{0.22488},{2.41421},
{1.590721},{-3, 3},{2},{5.8728656},{1.195137} };
```