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 лабораторной работе

«Многошаговая схема решения двумерных задач глобальной оптимизации. Распараллеливание по характеристикам.»

Выполнил:
студент группы 381708-1
Федотов В. И.

Проверил:
Доцент,
Кандидат технических наук
Сысоев А. В.

Нижний Новгород
2019

Содержание

Введение

Поиск глобального минимума функции — нетривиальная задача в общем случае. Сложность её, в частности, заключается в следующем:

- не всегда можно применить аналитические методы ввиду огромного количества вычислений и сложности функции;
- при увеличении размерности сложность поиска глобального минимума возрастает экспоненциально («проклятие размерности»).

Поэтому зачастую требуется прибегнуть к определенным вычислительным методам, позволяющим автоматизировать данную задачу. Но как найти оптимальный метод, который обладал бы и **хорошей производительностью**, и достаточной **точностью** измерений?

Как известно, достичь обе желаемые характеристики одновременно на практике удается крайне редко. Тем не менее, разработаны различные вычислительные методы, позволяющие стать ответом на поставленную задачу.

Рассмотрим один из таких методов в контексте лабораторной работы по параллельному программированию, а именно **многошаговую схему решения двумерных задач глобальной оптимизации (редукция размерности)**, детальная постановка и метод решения которой приводятся ниже.

Постановка задачи

Целью данной лабораторной работы является реализация решения двумерной задачи (функции двух переменных) глобальной оптимизации путем применения редукции размерности. Кроме того, требуется оценить эффективность полученного решения, сравнив параллельную реализацию с последовательной.

Для достижения данной цели необходимо выполнение ряда задач:

1. Изучение теоретической базы для понимания формальной постановки задачи и метода её решения с помощью математического аппарата.

Для этого необходимо изучить:

- соответствующие разделы монографии «Параллельные вычисления в задачах глобальной оптимизации» (авторы: Р. Г. Стронгин, В. П. Гергель, В. А. Гришагин, К. А. Баркалов);
- статью «Многомерная многоэкстремальная оптимизация на основе адаптивной многошаговой редукции размерности» (авторы: В.П. Гергель, В.А. Гришагин, А.В. Гергель)
- видеолекцию Стронгина Р. Г. на тему глобальной оптимизации, размещённую на YouTube
- другие источники.

2. Разработать реализацию решения одномерной задачи глобальной оптимизации на основе АГП (алгоритма глобального поиска), предложенного и опубликованного Стронгиным Р. Г. в упомянутой выше монографии.

3. Разработать реализацию обобщения решения одномерной задачи глобальной оптимизации на двумерный случай путём применения метода редукции размерности, то есть сведением двумерной задачи к ряду одномерных задач.

4. Разработать параллельную реализацию решения, упомянутого в предыдущем пункте.

5. Убедиться в корректности работы программы путем модульного тестирования с использованием фреймворка Google Test.

6. Оценить эффективность параллельной программы по сравнению с последовательной.

Метод решения

Для решения поставленной задачи воспользуемся алгоритмом глобального поиска для одномерной задачи глобальной оптимизации (то есть функции одной переменной), описанным в упомянутой выше монографии.

Приведём здесь описание формальной постановки задачи и вычислительного метода решения.

Рассмотрим одномерную задачу минимизации функции на отрезке:

$$\varphi(x) \to \min, \quad x \in Q = [a, b].$$

В качестве поисковой

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

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

Пусть проведено $k \geqslant 2$ испытаний и получена информация (2.18). Для выбора точки x^{k+1} нового испытания необходимо выполнить следующие действия.

1. Перенумеровать нижним индексом (начиная с нулевого значения) точки x^i, $1 \leqslant i \leqslant k$, из (2.18) в порядке возрастания, т. е.

$$a = x_0 < x_1 < \ldots < x_{k-1} = b.$$

2. Полагая $z_i = \varphi(x_i)$, $1 \leqslant i \leqslant k$, вычислить величину

$$M = \max_{1 \leqslant i \leqslant k-1} \left| \frac{z_i - z_{i-1}}{x_i - x_{i-1}} \right|$$

и положить

$$m = \begin{cases} rM, & M > 0, \\ 1, & M = 0, \end{cases} \tag{2.19}$$

где $r > 1$ является заданным параметром метода.

3. Для каждого интервала (x_{i-1}, x_i), $1 \leqslant i \leqslant k-1$ 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

$$R(t) = \max\{R(i) : 1 \leqslant i \leqslant k-1\} \tag{2.20}$$

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 $z^{k+1} = \varphi(x^{k+1})$ и увеличить номер шага поиска на единицу: $k = k + 1$.

Операции пунктов 1–5 описывают решающее правило АГП (1.14) из общей схемы (1.9).

Правило остановки (1.15) задается в форме

$$H_k(\Phi, \omega_k) = \begin{cases} 0, & x_t - x_{t-1} \leqslant \varepsilon, \\ 1, & x_t - x_{t-1} > \varepsilon, \end{cases} \qquad (2.21)$$

где $\varepsilon > 0$ — заданная точность поиска (по координате).

Наконец, в качестве оценки экстремума выбирается пара

$$e^k = (\varphi_k^*, x_k^*),$$

где φ_k^* — минимальное вычисленное значение функции, т. е.

$$\varphi_k^* = \min_{1 \leqslant i \leqslant k} \varphi(x^i),$$

а x_k^* — координата этого значения:

$$x_k^* = \arg \min_{1 \leqslant i \leqslant k} \varphi(x^i).$$

Другими словами, идея алгоритма заключается в следующем:

1. Накапливаем множество проведенных испытаний, представляющих собой пару:

   - аргумент функции (точка);

   - значение функции в данной точке.

2. На каждой итерации вычисляем, в какой точке наиболее «целесообразно» проводить следующее испытание. Разумеется, последовательный перебор точек «в лоб» с некоторым изменением аргумента («дельта X») крайне неэффективен.

   Поэтому происходит вычисление определенных характеристик на каждом интервале из множества уже проведенных испытаний.

   Тот интервал, на котором характеристика R принимает максимальное значение, выбирается как основа для вычисления точки следующего испытания по формуле п. 5.

3. Когда мы должны остановить процедуру поиска новой точки? Это происходит в одном из двух случаев:

   - либо точность интервала аргумента достигла заданного значения (переданного как входной параметр функции);

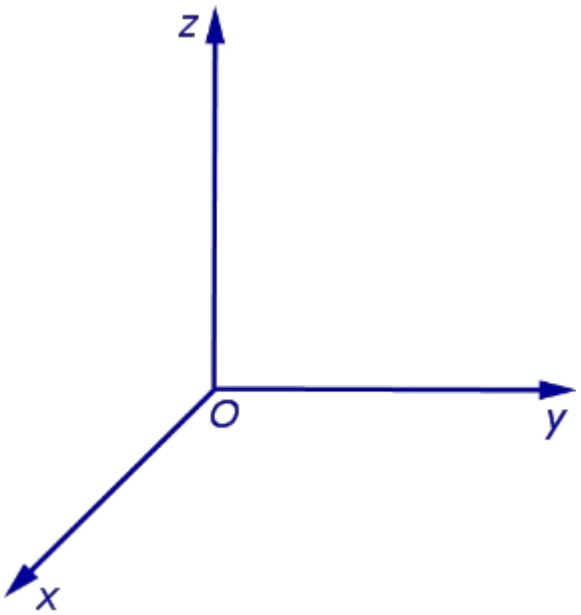
   - либо количество итераций достигло заданного максимального количества итераций (тоже переданного как входной параметр).

4. В результате у нас накоплено множество проведенных испытаний, из которого выбирается испытание с минимальным значением функции. Возвращаемое значение: аргумент и значение функции от аргумента.

Обобщение для двумерного случая

Каким образом можно обобщить описанную выше схему, чтобы найти минимум функции от двух переменных?

Воспользуемся методом редукции размерности, то есть сведем задачу к более простому случаю, а именно к оптимизации функции одной переменной. Таким образом, мы можем взять за основу описанный выше алгоритм.



Представим трехмерную систему координат.

1. Зафиксируем точку на оси oX и рассмотрим получившееся сечение.

2. Будем решать задачу нахождения минимума функции от одной переменной на плоскости zOy при фиксированном значении переменной X (то есть передаем его в функцию двух переменных как константу).

3. Снова и снова выбирая новую точку X, которую наиболее целесообразно использовать для фиксирования (используется тот же самый алгоритм, описанный выше для одномерного случая), мы получим множество решений одномерной задачи на плоскости zOy при различных фиксированных значениях X, которые представляют из себя элементы следующего вида:

    - координата X;

    - координата Y;

    - значение функции двух переменных Z.

4. Выберем испытание с минимальным значением функции, подобно тому, как это было сделано для одномерной задачи.

Схема распараллеливания

Предположим, что реализация последовательного алгоритма нахождения глобального минимума функции двух переменных написана.

Если количество процессов меньше двух, то будем вызывать функцию, представляющую собой последовательное решение.

В противном случае поступим следующим образом:

1. Разделим область поиска по оси oX на равные части в зависимости от количества процессов, указанных при запуске программы.
2. Каждому процессу сообщим исходную фиксированную точку отсчета по оси oX.
3. Будем ожидать в нулевом процессе приема сообщений от остальных процессов, содержащих вычисленные локальные минимумы при фиксированной переменной X.
4. В цикле (аналогично последовательному решению) вычисляем характеристику R для каждого интервала, но, в отличие от последовательного решения, выбираем не максимальную характеристику, а для каждой найденной характеристики вычисляем точку следующего испытания и отсылаем её другому процессу.
5. Снова ожидаем результатов испытаний от других процессов.
6. Завершаем работу остальных процессов.
7. Выбираем минимум из множества проведенных испытаний на всех процессах (множество хранится в нулевом процессе).
8. В чем заключается работа остальных процессов?

   - принимать сообщения от нулевого процесса. Если это признак завершения, то прекратить работу, иначе:

   - вызывать функцию последовательного решения для одномерной задачи при фиксированном значении переменной X, которое было получено как сообщение от нулевого процесса.

Описание программной реализации

Введем классы точек Point2D и Point3D, назначение которых не требует объяснений.

Для решения одномерной задачи используется функция **getGlobalMinimumOnSegment**, возвращающая объект Point2D. Содержимое данной функции представляет собой реализацию алгоритма глобального поиска для функции одной переменной, описанного в пункте «Метод решения».

Замечания, которые следует сделать по реализации данной функции (getGlobalMinimumOnSegment):

- фиксированная переменная по оси oX принимается в качестве параметра и в дальнейшем используется при вызове непосредственно математической функции. Функция принимает, кроме того, границы области поиска, указатель на математическую функцию и вспомогательные параметры, такие как параметр надежности, максимальное количество итераций и точность измерений.
- Множество произведенных испытаний реализуется при помощи такой структуры данных стандартной библиотеки шаблонов C++, как set:

```
std::set<Point2D, Point2DComparator> trials;
```

- Интервалы не хранятся специальным образом в виде структуры данных, а представляют собой два итератора, отличающихся на одну позицию:

```
iteration = trials.begin();
iteration++;
previousIteration = trials.begin();
```

Для решения двумерной задачи глобальной оптимизации используется функция **getGlobalMinimumOnPlane,** возвращающая объект Point3D.

Поскольку с математической точки зрения используется тот же самый алгоритм (мы используем схему редукции размерности), то и реализация будет отличаться лишь незначительно:

- используем объекты Point3D вместо Point2D;

- для каждого испытания вызываем функцию, представляющую собой последовательное решение и принимающую в качестве аргумента новую точку, вычисленную для следующего испытания.

Параллельное решение реализуется в функции **getGlobalMinimumOnPlaneParallelly,** реализованную аналогично getGlobalMinimumOnPlane, но с изменениями, которые позволяют распараллелить вычисления и которые были описаны в предыдущем разделе.

Подтверждение корректности

Для подтверждения корректности в программе предполагается запуск модульных тестов. Каким образом мы можем оценить корректность выполнения реализованных алгоритмов?

Для этого нам необходимо располагать информацией об априори известных глобальных минимумах в заданных областях тех или иных функций.

Зная математически обоснованный результат испытания, сравним его с программно вычисленным:

- зададим область поиска и требуемые параметры;
- зададим априори известный глобальный минимум в виде:

  ```
  Point3D trueGlobalMin(5, 3, 0);
  ```

- вычислим программно глобальный минимум:

  ```
  Point3D countedGlobalMin = getGlobalMinimumOnPlane(xLeftBorder, xRightBorder, yBottomBorder, yTopBorder, func, maxIterationsCount, r, accuracy);
  ```

- Если модуль разности известного и вычисленного значений различаются не более, чем на заданную в виде параметра требуемую точность измерения, то считаем программу работающей корректно:

  ```
  EXPECT_EQ(1, std::abs(countedGlobalMin.x - trueGlobalMin.x) <= accuracy);

  EXPECT_EQ(1, std::abs(countedGlobalMin.y - trueGlobalMin.y)  <= accuracy);
  ```

Результаты экспериментов

Эксперименты проводились на ПК со следующими параметрами:

1. ОС: Linux Ubuntu 18.04 LTS
2. Процессор: 4 ядра, 8 потоков (8 x 2100.00 MHz)

Для проведения экспериментов осуществлялся поиск глобального минимума функции `0.5*(x-5)*(x-5) + y — 3` на области `[-10; 10], [-10; 10]`.

Количество процессов	Последовательный алгоритм (время, с)	Параллельный алгоритм (время, с)	Ускорение
1	0.000463	-	-
2	0.000529	0.000918	0.576252723
3	0.000505	0.000700	0.721428571
4	0.000544	0.000508	1.070866142
5	0.000736	0.000077	9.558441558
6	0.000725	0.000675	1.074074074
7	0.000718	0.000367	1.95640327
8	0.000764	0.000545	1.401834862

По данным экспериментов видно, что параллельный алгоритм в некоторых случаях оказывается эффективнее последовательного.

Заключение

Таким образом, в результате лабораторной работы реализован один из вычислительных методов глобального поиска: многошаговая схема решения двумерных задач глобальной оптимизации (редукция размерности). Были проведены эксперименты, в результате которых выяснилось, что параллельная реализация оказалась эффективнее.

Литература

1. Монография «**Параллельные вычисления в задачах глобальной оптимизации**» (авторы: Р. Г. Стронгин, В. П. Гергель, В. А. Гришагин, К. А. Баркалов);

2. Статья «**Многомерная многоэкстремальная оптимизация на основе адаптивной многошаговой редукции размерности**» (авторы: В.П. Гергель, В.А. Гришагин, А.В. Гергель)

3. Другие источники

Приложение

```
#include <mpi.h>
#include <cstdio>
#include <cstdlib>
#include <set>
#include <cmath>
#include <iostream>
#include "../../../modules/task_3/fedotov_v_global_optimization/global_optimization.h"

// global min of ONE argument function.
// fixed variable is need for dimensional reduction method,
// its destination will be clear in getGlobalMinimumOnPlane() function
Point2D getGlobalMinimumOnSegment(double fixedVariable, double leftBorder,
double rightBorder, double(*func)(double x, double y), int maxIterationsCount,
double r, double accuracy) {
std::set<Point2D, Point2DComparator> trials; // calculated points

double M, maxM, m, R, maxR;
int countOfTrials = 0;

// handle borders
trials.insert(Point2D(leftBorder, func(fixedVariable, leftBorder)));
countOfTrials++;
trials.insert(Point2D(rightBorder, func(fixedVariable, rightBorder)));
countOfTrials++;

while (countOfTrials < maxIterationsCount) {
maxM = -999;
auto iteration = trials.begin();
iteration++;
auto previousIteration = trials.begin();

// calculate max M, which actually is max derivative on intervals
while (iteration != trials.end()) {
M = std::abs(static_cast<double>(
(iteration->y - previousIteration->y) /
(iteration->x - previousIteration->x)));
if (M > maxM)
maxM = M;
iteration++;
```

```
previousIteration++;
}

// calculate m depending on M
if (maxM > 0)
m = r * maxM;
else
m = 1;

// restore iterators to beginning
iteration = trials.begin();
iteration++;
previousIteration = trials.begin();

maxR = -999;

auto iterationOnMaxR = trials.begin();
auto previousIterationOnMaxR = trials.begin();

// calculate R
while (iteration != trials.end()) {
R = m*(iteration->x - previousIteration->x) + (std::pow(
(iteration->y - previousIteration->y), 2) /
(m * (iteration->x - previousIteration->x))) - 2 *
(iteration->y + previousIteration->y);
if (R > maxR) {
maxR = R;
iterationOnMaxR = iteration;
previousIterationOnMaxR = previousIteration;
}
iteration++;
previousIteration++;
}

countOfTrials++;

// calculating X of new point for trial
double newPointForTrial = (0.5)*(iterationOnMaxR->x +
previousIterationOnMaxR->x) - ((iterationOnMaxR->y -
previousIterationOnMaxR->y) / (2 * m));
```

```
// save new counted value of function in new point
trials.insert(Point2D(newPointForTrial, func(fixedVariable,
newPointForTrial)));

// finish our work when interval is less than required accuracy
if (iterationOnMaxR->x - previousIterationOnMaxR->x <= accuracy)
break;
}

// find and return calculated min value from set of trials

auto iterationOnGlobalMin = trials.begin();

for (auto it=trials.begin(); it != trials.end(); ++it) {
// std::cout << fixedVariable << ' ' << (it->x) << ' ' <<
// (it->y) << std::endl;
if (it->y < iterationOnGlobalMin->y) {
iterationOnGlobalMin = it;
}
}

Point2D globalMin(iterationOnGlobalMin->x, iterationOnGlobalMin->y);
// std::cout << fixedVariable << ' ' << (iterationOnGlobalMin->x) <<
// ' ' << (iterationOnGlobalMin->y) << std::endl;

return globalMin;
}

// global min of TWO argument function
// suppose oX is left-right and oY is bottom-top. oZ is the target function
// we are going to fix oX coordinate, then find minimum on oY segment,
// then choose min of mins
Point3D getGlobalMinimumOnPlane(double xLeftBorder, double xRightBorder,
double yBottomBorder, double yTopBorder, double(*func)(double x, double y),
int maxIterationsCount, double r, double accuracy) {
std::set<Point3D, Point3DComparator> trials; // calculated points

double M, maxM, m, R, maxR;
int countOfTrials = 0;
```

```
// handle borders
// the second param after xLeftBorder will return point2D,
// and overloaded constructor for Point3D will be used
trials.insert(Point3D(xLeftBorder, getGlobalMinimumOnSegment(xLeftBorder,
yBottomBorder, yTopBorder, func, maxIterationsCount, r, accuracy)));
countOfTrials++;
trials.insert(Point3D(xRightBorder, getGlobalMinimumOnSegment(xRightBorder,
yBottomBorder, yTopBorder, func, maxIterationsCount, r, accuracy)));
countOfTrials++;

while (countOfTrials < maxIterationsCount) {
maxM = -999;
auto iteration = trials.begin();
iteration++;
auto previousIteration = trials.begin();

// calculate max M, which actually is max derivative on intervals
while (iteration != trials.end()) {
M = std::abs(static_cast<double>(
(iteration->z - previousIteration->z) /
(iteration->x - previousIteration->x)));
if (M > maxM)
maxM = M;
iteration++;
previousIteration++;
}

// calculate m depending on M
if (maxM > 0)
m = r * maxM;
else
m = 1;

// restore iterators to beginning
iteration = trials.begin();
iteration++;
previousIteration = trials.begin();

maxR = -999;
```

```
auto iterationOnMaxR = trials.begin();
auto previousIterationOnMaxR = trials.begin();

// calculate R
while (iteration != trials.end()) {
R = m*(iteration->x - previousIteration->x) +
(std::pow((iteration->z - previousIteration->z), 2) /
(m * (iteration->x - previousIteration->x))) - 2 *
(iteration->z + previousIteration->z);
if (R > maxR) {
maxR = R;
iterationOnMaxR = iteration;
previousIterationOnMaxR = previousIteration;
}
iteration++;
previousIteration++;
}

countOfTrials++;

// calculating X of new point for trial
double newPointForTrial = (0.5)*(iterationOnMaxR->x +
previousIterationOnMaxR->x) - ((iterationOnMaxR->z -
previousIterationOnMaxR->z) / (2 * m));

// save new counted value of function in new point
trials.insert(Point3D(newPointForTrial,
getGlobalMinimumOnSegment(newPointForTrial, yBottomBorder,
yTopBorder, func, maxIterationsCount, r, accuracy)));

// finish our work when interval is less than required accuracy
if (iterationOnMaxR->x - previousIterationOnMaxR->x <= accuracy &&
iterationOnMaxR->y - previousIterationOnMaxR->y <= accuracy)
break;
}

// find and return calculated min value from set of trials

auto iterationOnGlobalMin = trials.begin();
```

```
for (auto it=trials.begin(); it != trials.end(); ++it) {
// std::cout << (it->x) << ' ' << (it->y)
// << ' ' << (it->z) << std::endl;
if (it->z < iterationOnGlobalMin->z) {
iterationOnGlobalMin = it;
}
}

Point3D globalMin(iterationOnGlobalMin->x, iterationOnGlobalMin->y,
iterationOnGlobalMin->z);
// std::cout << (iterationOnGlobalMin->x) << ' ' <<
// (iterationOnGlobalMin->y) << ' ' << (iterationOnGlobalMin->z)
// << std::endl;

return globalMin;
}

Point3D getGlobalMinimumOnPlaneParallelly(double xLeftBorder,
double xRightBorder, double yBottomBorder, double yTopBorder,
double(*func)(double x, double y), int maxIterationsCount, double r,
double accuracy) {
int size, rank;
MPI_Comm_size(MPI_COMM_WORLD, &size);

if (!(size > 1))
return getGlobalMinimumOnPlane(xLeftBorder, xRightBorder,
yBottomBorder, yTopBorder, func, maxIterationsCount, r, accuracy);

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Status status;

Point3D localMin(-9999, -9999, -9999);
Point3D globalMin(-9999, -9999, -9999);
// printf("Before %d\n", rank);

if (rank == 0) {
// printf("Hello from 0\n");
std::set<Point3D, Point3DComparator> trials;
double M, maxM, m, R;
```

```
int countOfTrials = 0;
// printf("rank 0 after var init\n");

double partPerProcess = (xRightBorder - xLeftBorder) / (size - 1);
for (int proc = 1; proc < size; ++proc) {
double fixedVariable = xLeftBorder + proc * partPerProcess;
MPI_Send(&fixedVariable, 1, MPI_DOUBLE, proc, 1, MPI_COMM_WORLD);
}

// printf("rank 0 after send\n");

trials.insert(Point3D(xLeftBorder, getGlobalMinimumOnSegment(
xLeftBorder,
yBottomBorder, yTopBorder, func, maxIterationsCount, r, accuracy)));
countOfTrials++;
trials.insert(Point3D(xRightBorder, getGlobalMinimumOnSegment(
xRightBorder, yBottomBorder, yTopBorder, func,
maxIterationsCount, r, accuracy)));
countOfTrials++;

// printf("rank 0 before recv\n");
double tmpLocalMin[3];
for (int proc = 1; proc < size; ++proc) {
MPI_Recv(tmpLocalMin, 3, MPI_DOUBLE, proc, 1,
MPI_COMM_WORLD, &status);
trials.insert(Point3D(tmpLocalMin[0], tmpLocalMin[1],
tmpLocalMin[2]));
}

// printf("rank 0 after recv\n");

std::set<characteristicR> characteristics;

bool accuracyAchieved = false;
while (!accuracyAchieved && countOfTrials < maxIterationsCount) {
// printf("rank 0 in while\n");
characteristics.clear();
maxM = -999;
auto iteration = trials.begin();
iteration++;
```

```
auto previousIteration = trials.begin();

// calculate max M, which actually is max derivative on intervals
while (iteration != trials.end()) {
// printf("rank 0 in inner while\n");
M = std::abs(static_cast<double>(
(iteration->z - previousIteration->z) /
(iteration->x - previousIteration->x)));
if (M > maxM)
maxM = M;
iteration++;
previousIteration++;
}

// printf("rank 0 after inner while\n");

// calculate m depending on M
if (maxM > 0)
m = r * maxM;
else
m = 1;

// restore iterators to beginning
iteration = trials.begin();
iteration++;
previousIteration = trials.begin();

// calculate R
while (iteration != trials.end()) {
// printf("rank 0 in R inner while\n");
R = m*(iteration->x - previousIteration->x) +
(std::pow((iteration->z - previousIteration->z), 2) /
(m * (iteration->x - previousIteration->x))) - 2 *
(iteration->z + previousIteration->z);

characteristics.insert(
characteristicR(R, iteration->x, iteration->z,
previousIteration->x, previousIteration->z));

iteration++;
```

```
previousIteration++;
}

// printf("rank 0 after R inner while\n");

auto iteratorR = characteristics.begin();

for (int proc = 1; proc < size; ++proc) {
countOfTrials++;
double newPointForTrial = (0.5)*
(iteratorR->x + iteratorR->xPrevious) -
((iteratorR->z - iteratorR->zPrevious) / (2 * m));
MPI_Send(&newPointForTrial, 1, MPI_DOUBLE, proc, 1,
MPI_COMM_WORLD);
if (iteratorR->x - iteratorR->xPrevious <= accuracy) {
accuracyAchieved = true;
}
iteratorR++;
}
// printf("rank 0 after send newPointForTrial\n");

if (accuracyAchieved == true) {
break;
}

double tmpLocalMin_2[3];
for (int proc = 1; proc < size; ++proc) {
MPI_Recv(tmpLocalMin_2, 3, MPI_DOUBLE, proc,
1, MPI_COMM_WORLD, &status);
trials.insert(Point3D(tmpLocalMin_2[0],
tmpLocalMin_2[1], tmpLocalMin_2[2]));
}
// printf("rank 0 after recv localMin\n");
}

// printf("rank 0 after outer while\n");

for (int proc = 1; proc < size; ++proc) {
double terminate = accuracy * 0.001;
MPI_Send(&terminate, 1, MPI_DOUBLE, proc, 1, MPI_COMM_WORLD);
```

```
}

// MPI_Barrier(MPI_COMM_WORLD);

// find and return calculated min value from set of trials

auto iterationOnGlobalMin = trials.begin();

for (auto it=trials.begin(); it != trials.end(); ++it) {
// std::cout << (it->x) << ' '
// << (it->y) << ' ' << (it->z) << std::endl;
if (it->z < iterationOnGlobalMin->z) {
iterationOnGlobalMin = it;
}
}

globalMin = Point3D(iterationOnGlobalMin->x,
iterationOnGlobalMin->y, iterationOnGlobalMin->z);

} else {
// printf("Hello from %d", rank);
bool terminate = false;
while (!terminate) {
double message;
MPI_Recv(&message, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, &status);
if (message == accuracy * 0.001) {
terminate = true;
} else {
localMin = Point3D(message,
getGlobalMinimumOnSegment(message, yBottomBorder, yTopBorder,
func, maxIterationsCount, r, accuracy));
double tmpLocalMin_3[3] = {localMin.x, localMin.y, localMin.z };
MPI_Send(&tmpLocalMin_3[0], 3, MPI_DOUBLE, 0, 1,
MPI_COMM_WORLD);
}
}
}
// printf("after: %d", rank);
return globalMin;
}
```

```
double function_1(double x, double y) {
return x*x + y*y;
}

double function_2(double x, double y) {
return 0.5*(x-5)*(x-5) + y - 3;
}

double function_3(double x, double y) {
return (x-5)*(x-5) + (y-3)*(y-3);
}

bool operator<(const Point2D& firstPoint,
const Point2D& secondPoint) {
return firstPoint.x < secondPoint.x;
}

bool operator<(const Point3D& firstPoint,
const Point3D& secondPoint) {
return firstPoint.x < secondPoint.x;
}
```