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

«Решение систем линейных уравнений методом сопряженных градиентов»

Выполнил:
студент группы 381706-1
Денисов В.Л.

Проверил:
Доцент кафедры МОСТ,
кандидат технических наук,
Сысоев А.В.

Нижний Новгород
2019

Содержание

Введение

Решение систем линейных алгебраических уравнений (СЛАУ) – одна из классических задач линейной алгебры.

Существуют различные способы решения СЛАУ. Одним из таких является метод сопряженных градиентов, который будет рассмотрен в данной лабораторной работе.

Метод сопряженных градиентов – итерационный метод для численного решения системы уравнений с симметричной и положительно определенной матрицей коэффициентов. Основная идея метода заключается в том, чтобы минимизировать норму так называемой ошибки.

Матрица – математический объект, записываемый в виде прямоугольной таблицы элементов (например, целых или действительных чисел), которая представляет собой совокупность строк и столбцов, на пересечении которых находятся её элементы.

Матрица A является **симметричной**, если она совпадает со своей транспонированной, т.е. $A = A^T$.

Симметричная матрица A называется **положительно определенной**, если для любого вектора x выполняется следующее неравенство: $x^T Ax > 0$.

Подробное описание метода сопряженных градиентов будет рассмотрено в теоретической части.

Постановка задачи

В рамках лабораторной работы ставится задача разработки библиотеки, реализующей алгоритм метода сопряженных градиентов для решения СЛАУ.

При работе с библиотекой будут доступны следующие операции:

- Автоматическая генерация симметричной матрицы коэффициентов и вектора правой части системы линейных уравнений,
- Последовательный алгоритм метода сопряженных градиентов,
- Параллельный алгоритм метода сопряженных градиентов,
- Печать СЛАУ, а также ответа на консоль.

Программное решение будет выглядеть следующим образом:

1. Модуль, содержащий реализацию библиотеки.
2. Набор автоматических тестов с использованием Google C++ Testing Framework.

Метод решения

Как уже упоминалось во Введении, метод сопряженных градиентов – численный метод решения систем линейных алгебраических уравнений.

Пусть дана система n линейных уравнений $Ax = b$, где A симметричная положительно определённая матрица размером $n \times n$, а x и b – вектора размера n.

Тогда процесс решения СЛАУ можно реализовать как минимизацию следующего функционала: $F(x) = (Ax, x) + 2(b, x) \rightarrow \min$.

В самом деле, функция $F(x)$ достигает своего минимального значения тогда и только тогда, когда ее градиент $\nabla F(x) = Ax - b$ обращается в ноль. Таким образом, решение системы линейных уравнений можно искать как решение задачи безусловной минимизации.

Рассмотрим сам алгоритм метода сопряженных градиентов.

Итерационный процесс начинается с указания некоторого произвольного приближения x, а также нахождения вектора невязки r и вектора направления h (в дальнейшем – градиент). В результате подготовительного шага получаем:

$$x = 1 - \text{единичный вектор}, \qquad r = h = b - Ax$$

Основные шаги выполняются последовательно и представляют собой следующий набор действий:

- В процессе очередной итерации вычисляется значение скалярного шага $alpha$:

$$alpha = \frac{(r,\ r)}{(Ah,\ h)}$$

- Находим очередное приближение:

$$x = x - alpha * h$$

- Обновляем значение вектора невязки, которое будет использоваться на следующей итерации:

$$rnext = r - alpha * Ah$$

- Вычисляем коэффициент $beta$, соответствующий выполнению условия сопряженности $(Ahnext, h) = 0$, где $hnext$ – значение градиента, которое будет получено для выполнения следующего шага:

$$beta = \frac{(rnext, rnext)}{(r, r)}$$

- Находим новое значение градиента:

$$h = rnext + beta * h$$

Из анализа расчетных формул метода видно, что на каждой итерации требуется выполнить умножение матрицы A на вектор h, несколько скалярных произведений векторов и несколько операций над векторами.

Известно, что умножение матрицы на вектор имеет квадратичную сложность $O(n^2)$, а сложность умножения векторов или выполнения операций с ними – $O(n)$. Следовательно, сложность выполнения одной итерации можем оценить сверху, как $O(n^2)$.

Кроме того, известно, что для нахождения точного решения системы линейных уравнений с положительно определенной симметричной матрицей необходимо выполнить не более n итераций, тем самым, сложность всего метода составляет $O(n^3)$.

Схема распараллеливания

Выполнение итераций метода осуществляется последовательно, следовательно, наиболее целесообразный подход состоит в распараллеливании вычислений, реализуемых в ходе выполнения отдельных итераций.

Самой ресурсоёмкой операцией в алгоритме метода сопряженных градиентов является умножение матрицы на вектор. Поэтому было принято решение воспользоваться ленточным разбиением матрицы для обработки каждым процессом своей части.

Каждый из K процессов, участвующих в решении системы линейных уравнений размерности N, получает по $delta = \lfloor N/K \rfloor$ строк, за исключением процесса с рангом 0, который получает $delta + remainder$ строк, где $remainder = N \% K$– оставшиеся строки.

Дополнительные вычисления, имеющие меньший порядок сложности, представляют собой различные операции обработки векторов (скалярное произведение, сложение и вычитание, умножение на скаляр). Организация этих вычислений заключается в разделении необходимых векторов на блоки, согласованных по размеру и положению с частью матрицы текущего процесса.

При необходимости процессы обмениваются информацией, посредством выполнения операций MPI_Allreduce, MPI_Bcast, MPI_Send или MPI_Recv.

Описание программной реализации

Программа включает в себя следующие функции для организации решения системы линейных уравнений с помощью метода сопряженных градиентов:

- createRandomMatrix – создает симметричную матрицу, заполненную случайными элементами. Принимает на вход размерность матрицы.
- createRandomVector – создает вектор из случайных элементов. Принимает на вход размерность вектора.
- vectorMult – возвращает результат скалярного произведения двух векторов, принимаемых в качестве аргументов.
- matrixVectorMult – возвращает вектор, как результат произведения матрицы на вектор, принимаемых в качестве аргументов.
- getSolveSeq – возвращает вектор, как результат последовательного выполнения алгоритма метода сопряженных градиентов. Принимает на вход матрицу коэффициентов системы, вектор правой части, а также размерность системы.
- getSolvePar – возвращает вектор, как результат параллельного выполнения алгоритма метода сопряженных градиентов. Входные данные аналогичны предыдущему пункту.
- printMatrix, printVector, printSystem – производит на консоль печать матрицы коэффициентов, вектора или системы линейных уравнений (матрица коэффициентов и вектор правой части) соответственно. Входные данные: матрица/вектор/матрица и вектор, а также размерность.

Рассмотрим подробнее, как устроено выполнение алгоритма в параллельном случае. Здесь будут опущены формулы, которые используются в самом методе, так как они описаны в соответствующем разделе. Внимание уделим только общей схеме распараллеливания.

- Каждый процесс узнает число всех процессов, параллельно выполняемых вместе с ним, а также свой ранг посредством вызова *MPI_Comm_size* и *MPI_Comm_rank*.
- Определяется число строк, обрабатываемых каждым процессом, а также вычисляется остаток, передаваемый процессу с рангом 0.
- Выполняется разделение входной матрицы коэффициентов на ленты, передаваемые каждому процессу, а также их рассылка процессам посредством *MPI_Send* и *MPI_Recv*.

- Создается вектор направления h размерностью системы, т.к. на каждом процессе он требуется целиком. Вектор невязки r создается размером по числу строк, обрабатываемых отдельным процессом.
- При выполнении подготовительного шага алгоритма, на каждом процессе вычисляется своя часть вектора направлений h. Затем полученные результаты передаются процессу с рангом 0 посредством *MPI_Send/MPI_Recv*.

  Итоговый целый вектор направлений рассылается всем процессам от процесса с рангом 0 с помощью *MPI_Bcast*.
- Умножение каждой части матрицы коэффициентов A на соответствующую часть вектора направления h выполняется процессами параллельно.
- При вычислении скалярных произведений векторов для нахождения коэффициентов $alpha$ и $beta$ используется *MPI_Allreduce* с параметром *MPI_SUM* для получения правильных значений на каждом процессе. Необходимость использования этой операции объясняется тем, что каждый процесс выполняет скалярное произведение только своей части обрабатываемого вектора.
- При вычислении нового значения вектора направления h снова используем пересылку данных процессу с рангом 0 посредством *MPI_Send/MPI_Recv*, который затем выполнит передачу обновленного целого вектора h всем остальным процессом с помощью *MPI_Bcast*.

Подтверждение корректности

Для подтверждения корректности в программе представлен набор тестов, разработанных с помощью использования Google C++ Testing Framework.

Набор представляет из себя тесты на проверку вспомогательных методов, а именно, невозможность создания матрицы коэффициентов или вектора правой части отрицательного размера; тесты на корректность входных данных при умножении вектора на вектор, а также матрицы на вектор – они должны быть одной размерности; проверку правильности умножения вектора на вектор и матрицы на вектор.

К проверке основных методов относится тест на правильность работы последовательного алгоритма метода сопряженных градиентов – результат, полученный по заранее известным входным данным, сравнивается с эталонным. Наконец, проверяется параллельный алгоритм путем генерации случайных входных данных и сравнения результатов его работы с ответом, полученным при последовательном варианте решения.

Успешное прохождение всех тестов доказывает корректность работы программного комплекса.

Результаты экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта метода сопряженных градиентов для решения систем линейных уравнений с симметричной положительно определенной матрицей проводились на оборудовании со следующей аппаратной конфигурацией:

- Процессор: Intel Core i5-7200U, 2700 MHz,
- Оперативная память: 6012 МБ (DDR4 SDRAM), 2400 MHz,
- ОС: Microsoft Windows 10 Home, версия 10.0.18363 сборка 18383.

Результаты экспериментов представлены в Таблице 1.

Таблица 1 Результаты вычислительных экспериментов.

Размер матрицы	Последовательный алгоритм	Параллельный алгоритм			
		2 процесса		4 процесса	
		время, с	ускорение	время, с	ускорение
500	0,19	0,11	1,72	0,09	2,11
1000	1,53	1,17	1,31	0,68	2,25
1500	4,72	3,16	1,49	2,07	2,28
2000	11,36	7,04	1,61	5,08	2,25
2500	21,87	13,01	1,68	9,99	2,18
3000	37,64	21,33	1,76	16,75	2,24

По данным, полученным в результате экспериментов, можно сделать вывод о том, что параллельный случай работает действительно быстрее, чем последовательный. Причем при увеличении числа процессов заметно, что ускорение неплохо возрастает.

Это объясняется тем, что основная вычислительная нагрузка происходит при умножении матрицы на вектор, а эта операция достаточно хорошо распараллеливается. Обмен данными, который происходит в ходе выполнения итераций, вносит не слишком большой вклад по сравнению с умножением матрицы на вектор, поэтому не сильно замедляет программу при больших объемах данных.

Однако такая картина будет наблюдаться только при достаточном количестве вычислительных ресурсов. В противном случае, наоборот произойдет замедление в связи с тем, что некоторые процессы будут ожидать своей очереди на выполнение операций или обмен данными со всеми остальными при коллективных операциях.

Заключение

В результате лабораторной работы была разработана библиотека, реализующая метод сопряженных градиентов для решения системы линейных уравнений.

Основной задачей данной лабораторной работы была реализация параллельной версии алгоритма. Эта задача была успешно достигнута, о чем говорят результаты экспериментов, проведенных в ходе работы. Они показывают, что параллельный случай работает действительно быстрее, чем последовательный.

Кроме того, были разработаны и доведены до успешного выполнения тесты, созданные для данного программного проекта с использованием Google C++ Testing Framework и необходимые для подтверждения корректности работы программы.

Литература

1. Баркалов К.А. Методы параллельных вычислений. Н. Новгород: Изд-во Нижегородского госуниверситета им. Н.И. Лобачевского, 2011.
2. Википедия: свободная электронная энциклопедия: на русском языке [Электронный ресурс] // URL: https://ru.wikipedia.org/wiki/Симметричная_матрица (дата обращения: 07.12.2019)
3. Википедия: свободная электронная энциклопедия: на русском языке [Электронный ресурс] // URL: https://ru.wikipedia.org/wiki/Метод_сопряжённых_градиентов_(для_решения_СЛАУ) (дата обращения: 07.12.2019)

Приложение

denisov_v_gradient_method.h

```
// Copyright 2019 Denisov Vladislav
#ifndef MODULES_TASK_3_DENISOV_V_GRADIENT_METHOD_GRADIENT_METHOD_H_
#define MODULES_TASK_3_DENISOV_V_GRADIENT_METHOD_GRADIENT_METHOD_H_


#include <vector>

std::vector<double> createRandomMatrix(int sizeSide);
std::vector<double> createRandomVector(int size);

double vectorMult(const std::vector<double>& vectorA, const std::vector<double>& vectorB);
std::vector<double> matrixVectorMult(const std::vector<double>& matrix,
                                     const std::vector<double>& vector);

std::vector<double> getSolveSeq(const std::vector<double>& matrix,
                                const std::vector<double>& vector, int size);
std::vector<double> getSolvePar(const std::vector<double>& matrix,
                                 const std::vector<double>& vector, int sizeSide);

void printMatrix(std::vector<double> matrix, int size);
void printVector(std::vector<double> vector, int size);
void printSystem(std::vector<double> matrix, std::vector<double> vector, int size);

#endif  // MODULES_TASK_3_DENISOV_V_GRADIENT_METHOD_GRADIENT_METHOD_H_
```

denisov_v_gradient_method.cpp

```
// Copyright 2019 Denisov Vladislav
#include <mpi.h>
#include <random>
#include <ctime>
#include <vector>
#include <iostream>
#include "../../../modules/task_3/denisov_v_gradient_method/gradient_method.h"

std::vector<double> createRandomMatrix(int sizeSide) {
    if (sizeSide <= 0)
        throw "Error size of matrix";

    std::default_random_engine generator;
    generator.seed(static_cast<unsigned int>(time(0)));

    int sizeVector = sizeSide * sizeSide;
    std::vector <double> matrix(sizeVector);
    for (int i = 0; i < sizeSide; ++i) {
        for (int j = i; j < sizeSide; ++j) {
            matrix[i * sizeSide + j] = matrix[j * sizeSide + i] = generator() % 10;
        }
    }

    return matrix;
}

std::vector<double> createRandomVector(int size) {
    if (size <= 0)
        throw "Error size of vector";

    std::default_random_engine generator;
    generator.seed(static_cast<unsigned int>(time(0)));
```

```
    std::vector <double> vector(size);
    for (int i = 0; i < size; ++i) {
        vector[i] = generator() % 10 + 1;
    }

    return vector;
}
void printMatrix(std::vector<double> matrix, int size) {
    std::cout << "\nPrint Matrix:" << std::endl;

    if (size > 10) {
        std::cout << "The matrix is too large to display on the console." << std::endl <<
std::endl;
        return;
    }

    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            std::cout << matrix[i * size + j] << " ";
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;
}

void printVector(std::vector<double> vector, int size) {
    std::cout << "\nPrint Vector:" << std::endl;

    if (size > 10) {
        std::cout << "The vector is too large to display on the console." << std::endl <<
std::endl;
        return;
    }

    for (int i = 0; i < size; ++i) {
        std::cout << vector[i] << std::endl;
    }
}

void printSystem(std::vector<double> matrix, std::vector<double> vector, int size) {
    std::cout << "\nPrint System:" << std::endl;

    if (size > 10) {
        std::cout << "The system is too large to display on the console." << std::endl <<
std::endl;
        return;
    }

    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            std::cout << matrix[i * size + j] << " ";
        }
        std::cout << "| "<< vector[i] << std::endl;
    }
    std::cout << std::endl;
}

double vectorMult(const std::vector<double>& vectorA, const std::vector<double>& vectorB) {
    if (vectorA.size() != vectorB.size())
        throw "The dimensions of vectors are not consistent";

    double res = 0.0;
    for (size_t i = 0; i < vectorA.size(); ++i)
        res += vectorA[i] * vectorB[i];

    return res;
```

```
}

std::vector<double> matrixVectorMult(const std::vector<double>& matrix, const
std::vector<double>& vector) {
    if (0 != (matrix.size() % vector.size()))
        throw "The dimensions of the matrix and vector are not consistent";

    std::vector<double> res(matrix.size() / vector.size());
    for (size_t i = 0; i < (matrix.size() / vector.size()); ++i) {
        res[i] = 0.0;
        for (size_t j = 0; j < vector.size(); ++j) {
            res[i] += matrix[i * vector.size() + j] * vector[j];
        }
    }

    return res;
}

std::vector<double> getSolveSeq(const std::vector<double>& matrix, const
std::vector<double>& vector, int size) {
    if (size <= 0)
        throw "Error size";

    int iters = 0;
    double eps = 0.1, beta = 0.0, alpha = 0.0, check = 0.0;

    std::vector<double> result(size);
    for (int i = 0; i < size; i++) {
        result[i] = 1;
    }

    std::vector<double> Ah = matrixVectorMult(matrix, result);
    std::vector<double> rprev(size), rnext(size);
    for (int i = 0; i < size; i++)
        rprev[i] = vector[i] - Ah[i];

    std::vector<double> h(rprev);

    do {
        iters++;
        Ah = matrixVectorMult(matrix, h);
        alpha = vectorMult(rprev, rprev) / vectorMult(h, Ah);
        for (int i = 0; i < size; i++) {
            result[i] += alpha * h[i];
            rnext[i] = rprev[i] - alpha * Ah[i];
        }
        beta = vectorMult(rnext, rnext) / vectorMult(rprev, rprev);

        check = sqrt(vectorMult(rnext, rnext));

        for (int j = 0; j < size; j++)
            h[j] = rnext[j] + beta * h[j];

        rprev = rnext;
    } while ((check > eps) && (iters <= size));

    return result;
}
std::vector<double> getSolvePar(const std::vector<double>& matrixInput, const
std::vector<double>& vectorInput,
    int sizeSide) {
    if (sizeSide <= 0)
        throw "Error size";

    std::vector<double> matrix = matrixInput;
    std::vector<double> vector = vectorInput;
```

```
    MPI_Bcast(matrix.data(), sizeSide * sizeSide, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(vector.data(), sizeSide, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    int comm_size, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int delta = sizeSide / comm_size;
    int remainder = sizeSide % comm_size;

    std::vector<double> matrixLocal(delta * sizeSide);
    if (rank == 0) {
        if (remainder != 0) {
            matrixLocal.resize(sizeSide * delta + remainder * sizeSide);
        }
        if (delta != 0) {
            for (int proc = 1; proc < comm_size; proc++) {
                MPI_Send(&matrix[0] + proc * delta * sizeSide + remainder * sizeSide,
                    delta * sizeSide, MPI_DOUBLE, proc, 1, MPI_COMM_WORLD);
            }
        }
    }

    if (rank == 0) {
        if (remainder != 0) {
            for (int i = 0; i < sizeSide * delta + sizeSide * remainder; i++) {
                matrixLocal[i] = matrix[i];
            }
        } else {
            for (int i = 0; i < sizeSide * delta; i++) {
                matrixLocal[i] = matrix[i];
            }
        }
    } else {
        MPI_Status status;
        if (delta != 0) {
            MPI_Recv(&matrixLocal[0], delta * sizeSide, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD,
&status);
        }
    }

    int iters = 0;
    double eps = 0.1, beta = 0.0, alpha = 0.0, check = 0.0;

    std::vector<double> result(sizeSide);
    for (int i = 0; i < sizeSide; i++) {
        result[i] = 1;
    }
    std::vector<double> Ah = matrixVectorMult(matrixLocal, result);


    std::vector<double> rprev(delta), rnext(delta);
    if (rank == 0) {
        if (remainder != 0) {
            rprev.resize(delta + remainder);
            rnext.resize(delta + remainder);
        }
        for (int i = 0; i < delta + remainder; i++)
            rprev[i] = vector[i] - Ah[i];
    } else {
        for (int i = 0; i < delta; i++)
            rprev[i] = vector[rank * delta + remainder + i] - Ah[i];
    }

    std::vector<double> h(sizeSide);
```

```
if (rank == 0) {
    if (remainder != 0) {
        for (int i = 0; i < delta + remainder; i++) {
            h[i] = rprev[i];
        }
    } else {
        for (int i = 0; i < delta; i++) {
            h[i] = rprev[i];
        }
    }
    if (delta != 0) {
        MPI_Status status;
        for (int proc = 1; proc < comm_size; proc++) {
            MPI_Recv(&h[0] + proc * delta + remainder,
                delta, MPI_DOUBLE, proc, 2, MPI_COMM_WORLD, &status);
        }
    }
} else {
    if (delta != 0) {
        MPI_Send(&rprev[0], delta, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD);
    }
}
MPI_Bcast(h.data(), sizeSide, MPI_DOUBLE, 0, MPI_COMM_WORLD);

std::vector<double> hblock(delta);
std::vector<double> hlocal(delta);
if (rank == 0) {
    if (remainder != 0) {
        hblock.resize(delta + remainder);
    }
}

do {
    iters++;
    Ah = matrixVectorMult(matrixLocal, h);

    if (rank == 0) {
        for (int i = 0; i < delta + remainder; i++) {
            hblock[i] = h[i];
        }
    } else {
        for (int i = 0; i < delta; i++) {
            hblock[i] = h[rank * delta + remainder + i];
        }
    }

    double tmp1 = vectorMult(rprev, rprev), rprevScal;
    double tmp2 = vectorMult(hblock, Ah), znam;
    MPI_Allreduce(&tmp1, &rprevScal, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    MPI_Allreduce(&tmp2, &znam, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

    alpha = rprevScal / znam;
    for (int i = 0; i < sizeSide; i++) {
        result[i] += alpha * h[i];
    }
    if (rank == 0) {
        for (int i = 0; i < delta + remainder; i++) {
            rnext[i] = rprev[i] - alpha * Ah[i];
        }
    } else {
        for (int i = 0; i < delta; i++) {
            rnext[i] = rprev[i] - alpha * Ah[i];
        }
    }

    double rnextScal;
```

```
        tmp1 = vectorMult(rnext, rnext);
        MPI_Allreduce(&tmp1, &rnextScal, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
        beta = rnextScal / rprevScal;

        // Norma neviazki
        check = sqrt(rnextScal);

        if (rank == 0) {
            for (int i = 0; i < delta + remainder; i++) {
                h[i] = rnext[i] + beta * h[i];
            }

            if (delta != 0) {
                MPI_Status status;
                for (int proc = 1; proc < comm_size; proc++) {
                    MPI_Recv(&h[0] + proc * delta + remainder,
                        delta, MPI_DOUBLE, proc, 3, MPI_COMM_WORLD, &status);
                }
            }
        } else {
            for (int i = 0; i < delta; i++) {
                hlocal[i] = rnext[i] + beta * h[rank * delta + remainder + i];
            }

            if (delta != 0) {
                MPI_Send(&hlocal[0], delta, MPI_DOUBLE, 0, 3, MPI_COMM_WORLD);
            }
        }
        MPI_Bcast(h.data(), sizeSide, MPI_DOUBLE, 0, MPI_COMM_WORLD);

        rprev = rnext;
    } while ((check > eps) && (iters <= sizeSide));

    return result;
}
```

main.cpp

```
// Copyright 2019 Denisov Vladislav
#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include <vector>
#include "./gradient_method.h"

TEST(gradient_method, throw_when_create_random_matrix_with_negative_size) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        ASSERT_ANY_THROW(createRandomMatrix(-1));
    }
}

TEST(gradient_method, throw_when_create_random_vector_with_negative_size) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        ASSERT_ANY_THROW(createRandomVector(-1));
    }
}

TEST(gradient_method, throw_when_vector_mult_with_not_consistent_size) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
    std::vector<double> vector1(2), vector2(3);

    if (rank == 0) {
        ASSERT_ANY_THROW(vectorMult(vector1, vector2));
    }
}

TEST(gradient_method, throw_when_matrix_vector_mult_with_not_consistent_size) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int size = 2;
    std::vector<double> matrix(size * size);
    std::vector<double> vector(size + 1);

    if (rank == 0) {
        ASSERT_ANY_THROW(matrixVectorMult(matrix, vector));
    }
}

TEST(gradient_method, check_correct_vector_mult) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int size = 2;
    std::vector<double> vector1(size);
    std::vector<double> vector2(size);
    double correctAnswer;

    vector1[0] = 1;
    vector1[1] = 2;

    vector2[0] = 5;
    vector2[1] = 6;

    correctAnswer = 17;

    if (rank == 0) {
        double result = vectorMult(vector1, vector2);
        ASSERT_NEAR(correctAnswer, result, 0.5);
    }
}

TEST(gradient_method, check_correct_matrix_vector_mult) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int size = 2;
    std::vector<double> vector(size);
    std::vector<double> matrix(size * size);
    std::vector<double> correctAnswer(size);

    vector[0] = 1;
    vector[1] = 2;

    matrix[0] = 4;
    matrix[1] = 5;
    matrix[2] = 6;
    matrix[3] = 7;

    correctAnswer[0] = 14;
    correctAnswer[1] = 20;

    if (rank == 0) {
        std::vector<double> result = matrixVectorMult(matrix, vector);
```

```
        for (size_t i = 0; i < result.size(); i++)
            ASSERT_NEAR(correctAnswer[i], result[i], 0.5);
    }
}

TEST(gradient_method, check_correct_seq_gradient_method) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int size = 3;
    std::vector<double> vector(size);
    std::vector<double> matrix(size * size);
    std::vector<double> correctAnswer(size);

    vector[0] = 1;
    vector[1] = 5;
    vector[2] = 9;

    matrix[0] = 3;
    matrix[1] = 4;
    matrix[2] = 0;
    matrix[3] = 4;
    matrix[4] = -3;
    matrix[5] = 0;
    matrix[6] = 0;
    matrix[7] = 0;
    matrix[8] = 5;

    correctAnswer[0] = 0.92;
    correctAnswer[1] = -0.44;
    correctAnswer[2] = 1.80;

    if (rank == 0) {
        std::vector<double> result = getSolveSeq(matrix, vector, size);
        for (size_t i = 0; i < result.size(); i++)
            ASSERT_NEAR(correctAnswer[i], result[i], 0.5);
    }
}

TEST(gradient_method, check_correct_par_gradient_method) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int size = 12;
    std::vector<double> vector = createRandomVector(size);
    std::vector<double> matrix = createRandomMatrix(size);

    std::vector<double> resultPar = getSolvePar(matrix, vector, size);

    if (rank == 0) {
        std::vector<double> resultSeq = getSolveSeq(matrix, vector, size);
        for (size_t i = 0; i < resultSeq.size(); i++)
            ASSERT_NEAR(resultSeq[i], resultPar[i], 0.5);
    }
}


int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    MPI_Init(&argc, &argv);

    ::testing::AddGlobalTestEnvironment(new GTestMPIListener::MPIEnvironment);
    ::testing::TestEventListeners& listeners =
        ::testing::UnitTest::GetInstance()->listeners();

    listeners.Release(listeners.default_result_printer());
```

```
    listeners.Release(listeners.default_xml_generator());

    listeners.Append(new GTestMPIListener::MPIMinimalistPrinter);
    return RUN_ALL_TESTS();
}
```