Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу «Численные методы»

Тема: «Решение систем линейных алгебраических уравнений с симметричными разреженными матрицами большой размерности. Метод сопряженных градиентов.»

Студент:	Д. А. Тарпанов
Преподаватель:	Д. Л. Ревизников
Группа:	М8О-307Б-19
Дата:	
Оценка:	
Подпись:	

Москва, 2022

Постановка задачи

Задача: Дана система линейных уравнений в матричном виде. Найти решение системы с помощью метода сопряженных градиентов.

Сравнить производительность метода сопряженных градиентов с методом Зейделя.

1 Описание

Метод сопряженных градиентов

Рассмотрим следующую задачу оптимизации: $F(x) = 1/2*(Ax, x) - (b, x) -> inf, x \in R^n$. Здесь A – положительно определенная симметричная разреженная матрица размера $n \cdot n$. Заметим, что $F'(x) = Ax - b$ и условие $F'(x) = 0$ эквивалентно $Ax - b = 0$. Функция F достигает своей нижней грани в единственной точке x^*, определяемой уравнением $Ax^* = b$.
Будем говорить, что ненулевые векторы $p^{(0)}, p^{(1)}, \dots, p^{(m-1)}$ называются *взаимно сопряженными* относительно матрицы A, если $(Ap^{(n)}, p^{(l)}) = 0$ для всех $n \neq l$.
Под *методом сопряженных направлений* для минимизации квадратичной функции будем понимать метод

$$x^{(n+1)} = x^{(n)} + \alpha_n p^{(n)} (n = 0, 1, 2, \dots, m-1),$$

в котором направления $p^{(0)}, p^{(1)}, \dots, p^{(m-1)}$ взаимно сопряжены, а шаги

$$\alpha_n = \frac{(r^{(n)}, p^{(n)})}{(Ap^{(n)}, p^{(n)})},$$

где $r^{(n)} = r^{(n-1)} - \alpha_n Ap^{(n)}$ – антиградиент, получаются как решение задач одномерной минимизации:

$$\phi_n(\alpha_n) = min_{\alpha \geq 0} \phi_n(\alpha), \phi_n(\alpha) = F(x^{(n)}) + \alpha p^{(n)}$$

В методе сопряженных градиентов направления $p^{(n)}$ строят по правилу:

$$p^{(0)} = r^{(0)}, p^{(n)} = r^{(n)} + \beta_{n-1} p^{(n-1)}, n \geq 1,$$

где

$$\beta_{n-1} = \frac{(r^{(n)}, r^{(n)})}{(r^{(n-1)}, r^{(n-1)})},$$

$$r^{(n)} = r^{(n-1)} - \alpha_{n-1} Ap^{(n-1)}.$$

Анализ метода

Для метода сопряжённых градиентов справедлива следующая теорема: Теорема Пусть $F(x) = \frac{1}{2}(Ax, x) - (b, x)$, Aгде - симметричная положительно определённая матрица размера n. Тогда метод сопряжённых градиентов сходится не более чем за n шагов. В компьютерном представлении, однако, существуют проблемы с представлением вещественных чисел, в связи с чем, количество итераций может превышать n.

Сходимость

Более тонкий анализ показывает, что число итераций не превышает m, где m - число различных собственных значений матрицы A. Для оценки скорости сходимости верна следующая (довольно грубая) оценка:

$$||x_k - x_*||_A \leq (\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1})||x_0 - x_*||_A,$$

где $\kappa(A) = ||A||\,||A^{-1}|| = \lambda_1/\lambda_n$. Она позволяет оценить скорость сходимости, если известны оценки для максимального λ_1 и минимального λ_n собственных значений матрицы A. На практике чаще всего используют следующий критерий останова:

$$||r_k|| < \varepsilon.$$

Вычислительная сложность

На каждой итерации метода выполняется $O(n^2)$ операций. Такое количество операций требуется для вычисления произведения $Ap^{(n)}$ - это самая трудоёмкая процедура на каждой итерации. Отальные вычисления требуют $O(n)$ операций. Суммарная вычислительная сложность метода не превышает $O(n^3)$ - так как число итераций не больше n.

2 Исходный код

Ниже приведена реализация класса разреженной матрицы:

```
template<typename T>
class SparseMatrix {

private:
    std::vector<int> rows;
    std::vector<int> columns;
    std::vector<T> data;
    int m, n;

    void insert(int ind, int row, int col, T val) {
        if (data.empty()) {
            data = std::vector<T> (1, val);
            columns = std::vector<int> (1, col);
        } else {
            data.insert(data.begin() + ind, val);
            columns.insert(columns.begin() + ind, col);
        }
        for (int i = row + 1; i <= m; ++i) {
            rows[i]++;
        }
    }

    void remove(int index, int row) {
        data.erase(data.begin() + index);
        columns.erase(columns.begin() + index);

        for (int i = row + 1; i <= m; ++i) {
            rows[i]--;
        }
    }


public:
    SparseMatrix(int row, int col) {
        m = row;
        n = col;
        rows = std::vector<int> (row + 1, 0);
    }

    SparseMatrix(int _n) {
        m = _n;
        n = _n;
        rows = std::vector<int> (_n + 1, 0);
    }

    int size() const {
```

```
        if (n == m) {
            return n;
        } else {
            return 0;
        }
    }

    SparseMatrix(const SparseMatrix<T> &matrix) {
        n = matrix.n;
        m = matrix.m;
        rows = matrix.rows;
        columns = matrix.columns;
        data = matrix.data;
    }

    SparseMatrix(const std::vector<std::vector<T>> &matrix) {
        int count = 0;
        rows.clear();
        for (int i = 0; i < matrix.size(); ++i) {
            rows.push_back(count);
            for (int j = 0; j < matrix[i].size(); ++j) {
                if (matrix[i][j] != 0)
                    ++count;
            }
        }
        rows.push_back(count);
        columns.resize(count);
        data.resize(count);
        n = matrix[0].size();
        m = matrix.size();
        int k = 0;
        for (int i = 0; i < matrix.size(); ++i) {
            for (int j = 0; j < matrix[i].size(); ++j) {
                if (matrix[i][j] != 0) {
                 columns[k] = j;
                 data[k] = matrix[i][j];
                 ++k;
                }
            }
        }
    }

    T get(int row, int column) const {
        if (column > n || row > m) {
            throw "Trying to get non-existent element";
        }
        int currcol;
        for (int pos = rows[row]; pos < rows[row+1]; ++pos) {
            currcol = columns[pos];
```

```
            if (currcol == column) {
                return data[pos];
            } else if (currcol > column) {
                break;
            }
        }
        return T();
    }

    void swapRows(int n, int m) {

    }

    void set(T val, int row, int column) {
        if (column > n || row > m) {
            throw "Trying to set non-existent element";
        }
        int pos = rows[row];
        int currCol = 0;

        for (; pos < rows[row + 1]; ++pos) {
            currCol = columns[pos];
            if (currCol >= column) {
                break;
            }
        }

        if (currCol != column || data.empty() || pos >= data.size()) {
            insert(pos, row, column, val);
        } else if (val == T()) {
            remove(pos, row);
        } else {
            data[pos] = val;
        }
    }

    std::vector<T> operator *(const std::vector<T> &x) {
        if (n != x.size()) {
            throw "Invalid dimensions on multiply";
        }
        std::vector<T> result(m, T());
        if (!data.empty()) {
            for (int i = 0; i < m; ++i) {
                T sum = T();
                for (int j = rows[i]; j < rows[i + 1]; ++j) {
                    sum = sum + data[j] * x[columns[j]];
                }
                result[i] = sum;
            }
```

```
        }
        return result;
    }

    friend SparseMatrix<T> operator *(const SparseMatrix<T> &lhs, SparseMatrix<T> &mat)
     {
        if (lhs.n != mat.m) {
            throw "Invalid dimensions on multiply";
        }
        SparseMatrix<T> result(lhs.m, mat.n);
        T a;
        for (int i = 0; i < lhs.m; ++i) {
            for (int j = 0; j < mat.n; ++j) {
                a = T();
                for (int k = 0; k < lhs.n; ++k) {
                    a = a + lhs.get(i, k) * mat.get(k, j);
                }
                result.set(a, i, j);
            }
        }
        return result;
    }

    friend SparseMatrix<T> operator +(const SparseMatrix<T> &lhs, SparseMatrix<T> &rhs)
     {
        if (lhs.m != rhs.m || lhs.n != rhs.m) {
            throw "Invalid dimensions on addition";
        }
        SparseMatrix<T> result(lhs.m, lhs.n);
        for (int i = 0; i < lhs.m; ++i) {
            for (int j = 0; j < lhs.n; ++j) {
                result.set(lhs.get(i,j) + rhs.get(i,j), i, j);
            }
        }
        return result;
    }

    friend SparseMatrix<T> operator -(const SparseMatrix<T> &lhs, SparseMatrix<T> &rhs)
     {
        if (lhs.m != rhs.m || lhs.n != rhs.m) {
            throw "Invalid dimensions on substraction";
        }
        SparseMatrix<T> result(lhs.m, lhs.n);
        for (int i = 0; i < lhs.m; ++i) {
            for (int j = 0; j < lhs.n; ++j) {
                result.set(lhs.get(i,j) - rhs.get(i,j), i, j);
            }
        }
        return result;
```

```
    }

    friend std::ostream & operator << (std::ostream &os, const SparseMatrix<T> &matrix)
         {
        for (int i = 0; i < matrix.m; ++i) {
            for (int j = 0; j < matrix.n; ++j) {
                if (j != 0) {
                    os << " ";
                }
                os << matrix.get(i, j);
            }
            if (i < matrix.m - 1) {
                os << std::endl;
            }
        }
        return os;
    }
};
```

Ниже приведён листинг метода сопряженных градиентов:

```
template <typename T>
class ConjGrad {
public:
    SparseMatrix<T> A;
    std::vector<T> b;
    double eps;

    ConjGrad(SparseMatrix<T> &_A, std::vector<T> &_b, double _eps) : A(_A), b(_b), eps(
        _eps) {};

    std::vector<T> solve(bool flag) {
        if (flag) {
            std::cout << " : \n" << A;
            std::cout << "  : \n" << b << '\n';
        }
        std::vector<T> x1(b.size(), 0);
        std::vector<T> r1 = b - (A * x1);
        std::vector<T> p1 = r1;
        int count = 0;
        do {
            std::vector<T> temp = A * p1;
            T alpha = (r1 * r1)/(temp * p1);
            std::vector<T> xk = x1 + (p1 * alpha);
            std::vector<T> rk = r1 - (temp) * alpha;
            T t = rk * rk;
            T t2 = r1 * r1;
            T beta = (rk * rk)/(r1 * r1);
            std::vector<T> pk = rk + (p1 * beta);
            x1 = xk;
```

```
            r1 = rk;
            p1 = pk;
            count++;
        } while (norm(r1) > eps);
        std::cout << "     : " << count << "\n";
        return x1;
    }
};
```

3 Консоль

```
$Матрица системы:
1.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 1.452 0.000 0.000 0.000 0.000 0.000 0.000 -0.672 0.000
0.000 0.000 1.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 1.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 1.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 1.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 1.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 1.000 0.000 0.000
0.000 -0.672 0.000 0.000 0.000 0.000 0.000 0.000 1.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 1.000
Столбец свободных членов:
9.350 9.355 3.916 4.467 8.542 0.887 5.213 9.040 9.958 3.585

Количество итераций метода сопряженных градиентов: 3
Решение методом сопряженных градиентов:
9.350 16.050 3.916 4.467 8.542 0.887 5.213 9.040 20.749 3.585
Решение методом Зейделя:
Количество итераций метода Зейделя: 10
9.350 16.050 3.916 4.467 8.542 0.887 5.213 9.040 20.749 3.585
Бенчмарк:
Метод сопряженных градиентов: 70 microseconds
Метод Зейделя: 112 microseconds$
```

4 Тест производительности

В тесте сравнивается время решения СЛАУ с помощью метода сопряженных градиентов и метода Зейделя:

Размерность матрицы	МСГ, мс	Зейделя, мс	Количество итераций МСГ	Количество итераций Зейделя
$10 \cdot 10$	0.023	0.1	3	10
$50 \cdot 50$	0.307	28.3	24	75
$100 \cdot 100$	2.25	2132	62	815

Видно, что метод Зейделя сильно проигрывает по времени работы методу сопряженных градиентов.

5 Выводы

В ходе выполнения курсового проекта я изучил метод сопряженных градиентов для решения СЛАУ. Основной сложностью была корректная реализация класса разреженных матриц для хранения данных. Арифметические операции долго отлаживались, в связи с специфичным хранением данных (формат CSR).

Также, у меня не получился алгоритм для генерации случайной положительно определенной, разреженной и симметричной матрицы. Из-за этого, тесты я генерировал на языке python с помощью библиотеки sklearn.

В работе сравниваются методы сопряженных градиентов и Зейделя. Не уверен, что мой метод Зейделя реализован оптимально, но получилось так, что он работает на несколько порядков медленнее, чем метод сопряженных градиентов.

Список литературы

[1] *Метод сопряженных градиентов*
URL: https://en.wikipedia.org/wiki/Conjugate$_{g}radient_{m}ethod$(: 05.06.2022).

[2] *Метод сопряженных градиентов*
URL: https://scask.ru/i$_{b}ook_{c}lm.php?id = 76$(: 05.06.2022).