Московский государственный университет имени М. В. Ломоносова
механико-математический факультет
кафедра Теории Вероятностей

Отчёт по практикуму на ЭВМ
студента 409 группы
Сидоренко Артура Павловича

Оптимальный многошаговый метод решения СЛУ

Москва, 2019

1 Постановка задачи. Описание метода

Пусть дана СЛУ $Ax = b$, где A - кадратная невырожденная матрица. В таком случае решение единственно, и оно равно $x = A^{-1}b$. Требуется решить СЛУ на ЭВМ.

Предлагается использовать итеративные методы. Их идея заключается в следующем: берётся начальное приближение искомого решения x^0, далее считается последовательность приближений по формуле $x^{k+1} = Gx^k + c$ для некоторой марицы G и вектора c. Для нашей СЛУ предлагается взять формулу

$$\frac{x^{k+1} - xk}{\tau_{k+1}} + Ax^k = b, \tag{1}$$

т.е.

$$x^{k+1} = (I - \tau_{k+1}A)x^k + \tau b. \tag{2}$$

Тогда вектор ошибки $z^k = x^k - x$ считается по формуле

$$z^{k+1} = (I - \tau_{k+1}A)z^k. \tag{3}$$

Отдельный вопрос - подбор параметров τ. Здесь мы ограничимся случаем, когда $A = A^T > 0$. Предлагается взять $\tau_1, \ldots, \tau_N$ таким образом, чтобы минимизировать норму матрицы $(I - \tau_1 A) \ldots (I - \tau_N A)$. Тогда получим и оценку на норму z^k. Эту задачу удаётся решить, если взять евклидову норму. Оказывается, что тогда надо брать

$$\tau_k = \frac{1}{0.5 * (m + M) + 0.5 * (M - m) * w_k}, \tag{4}$$

где m и M - минимальное и максимальное собственные числа A, w_k - k-ый корен многочлена Чебышёва N-ой степени,

$$w_k = cos(\pi * (2k - 1)/(2N)), k = 1, \ldots, N. \tag{5}$$

Теперь самое интересное. В теории, все матрицы $(I - \tau_k A)$ коммутируют, т.е. порядок не имеет значения. На практике же от расстановки матриц зависит как скорость сходимости метода, так и сам факт того, что метод сойдётся.

Рассмотрим три способа выбрать порядок сомножителей. **Способ А**. Берём τ_k в порядке возрастания.

Способ Б. Группировка $\tau_1, \tau_N, \tau_2, \tau_{N-1}, ...$

Способ В. Оптимальная ггруппировка для $N = 2^k$. Для $k = 1$ группировка $1, 2$. Для $k = 2$ считаю так: после единицы вписываю $4 + 1 - 1$, после двух - $4 + 1 - 2$. Получу $1, 4, 2, 3$. Для $k = 3$ получаем, по аналогии, $1, 8, 4, 5, 2, 7, 3, 6$. И так далее. Этот способ в некотором плане продолжает идею способа Б.

Как мы увидим, способ А обеспечивает расходимость, а способы Б и В - сходимость, причём способ В - лучше.

2 Структура программы

Код был выполнен на C++11. Основа программы - класс матрицы с операциями сложения и умножения. Метод ООП позволяет для матричных операций использовать обычные + и *, что весьма удобно. Кроме, код становися безопасным и легко интегрируемым в сложные проекты. Помимо этого, класс матрицы обспечивает работу не только с квадратными матрицами, но и векторами. Хранение сведений происходит в двумерном массиве, причём первое измерение отвечает за столбцы. Это позволяет хранить векторы-столбцы в памяти более оптимальным образом.

Итеративный метод реализован в отдельной функции n_steps_method, принимающей на вход матрицу A и векторы правой части и начального приближения, а так же массив со значениями τ. Данный метод производит определённое число итераций, зависящее от нормы вектора невязки: остановка происходит либо если невязка мала, либо велика, либо прошло много итераций. Имеется также функция $test\_n\_steps\_method$, которая требует на вход точное решение и сравнивает его с получаемыми приближениями в смысле нормы L_∞.

Ниже приведён код заголовочного файла.

```
#pragma once

#include<iostream>
#include<fstream>
#include<cstdlib>
#include<string>
#include<algorithm>
#include<utility>
#include<exception>
#include<iomanip>
#include<cmath>
#include<vector>


const double PI = 4 * atan(1);
typedef std::vector<double> method_tau;
typedef std::vector<double> sol_logs; //logs of system solving
typedef std::vector<size_t> permut;

/*
Custom matrix class
Unlike common approaches, this matrix is stored column-wise, i.e.
the first index is in charge of columns, not rows.
It allows to represent vector-columns in the same class.
*/
```

```
class matrix {
	size_t rows_, cols_; //sizes
	double **data_; //contents
public:
	//constructors

	matrix(size_t rows, size_t cols); //zero matrix constructor
	matrix(const matrix &other); //copy construnctor
	matrix(matrix &&other); //move constructor
	~matrix();


	//operators =
	matrix operator=(const matrix &other);
	matrix operator=(matrix &&other);

	//pluses and multiplies
	//introduce convenient syntax
	matrix operator+(const matrix &M) const;
	matrix operator-(const matrix &M) const;
	matrix operator*(const matrix &M) const;
	matrix operator*(double x) const; //one can write either A*x,...
	friend matrix operator*(double x, const matrix &A);  //...or x*A


	//norms
	double l1_norm() const;
	double linf_norm() const;

	//access functions
	double get(size_t i, size_t j) const;
	void set(size_t i, size_t j, double x);
	void print(std::ofstream &s) const; //prints matrix to file
	size_t rows() const { return rows_; };
	size_t cols() const { return cols_; };
};

//Identity matrix
matrix id(size_t n);
//Lagrange matrix constructor
matrix lagrange(size_t N);

//Chebyshev grid
double cheb_grid(unsigned ord, unsigned i, double a, double b);
double inv_cheb_grid(unsigned ord, unsigned k, double a, double b);

//some n-steps method
//it performs sonsecutive multiplications like (I-t*A)*X
//method_tau - array of parameters t
//X is either a vector-column or a set columns in the one matrix
//A must be a square matrix and the number of rows of X must coincide with
```

```
    the size of A
matrix n_steps_method(const matrix &A, const matrix &X, const matrix &B,
   const method_tau &t);

//copy of the previous function to test convergence
matrix test_n_steps_method(const matrix & A, const matrix & X, const
   matrix &B, const method_tau & t, const matrix &true_ans);

bool break_loop(const sol_logs &g);

//very clever permutation of length = 2^t
permut clever_met(size_t t);
```

3 Численный эксперимент

Берём в качестве A матрицу Лагранжа размера $N = 500$x500: на главной диагонали стоит $\frac{2}{(N+1)^2}$, под и над ней - $\frac{-1}{(N+1)^2}$, в остальных ячейках - нули. В качестве решения x берём $x = (1, 1.5, 2, 2.5, ...)^T$. Вектор $b = Ax$. Начальное приближение $x_0 = 0$. Сходимсоть проверяется в смысле нормы L_∞.

Выберу оптимальный метод с 64 шагами: у меня 64 чисел τ. Расставив их способом А, я получу расходимость: норма разности $||x - x_{calc}||_\infty$ после 2000 итераций N-шагового метода даже выросла. При расстановке способами Б и В я получу сходимость, причём при выборе способа В норма убывает слегка быстрее. При этом ответ определяется с точностью примерно $5 * 10^{-11}$ в смысле той же нормы. Данная точность достигается за 800-900 итераций в обоих случаях, после чего не меняется. Так же отметим, что сходимотсь способом В гарантируется определёнными теоретическими результатами.

Таким образом, мы установили налаичие зависимости сзодимости метода от порядка сомножителей. Осталось объяснить то, откуда она появляется. Для этого рассмотрим L_2-норму матрицы $I - \tau A$. $||I - \tau A||_2 = \max|1 - \tau\lambda| : \lambda \in sp(A)$. Можем считать, что $sp(A) \in [m, M]$ Тогда оценка нормы матрицы $I - \tau A$ будет просто максимум функции $y = |1 - \tau x|$ среди граничных точек отрезка. Заметим, что $y = 0$ при $x = \frac{1}{\tau}$. Если $frac1\tau$ - корень многочлена Чебышёва большой степени на отрезке $[m, M]$, то это τ может быть близко к m. Тогда максимум для $y = \frac{M}{m} - 1 = cond_2(A) - 1$. Если матрица A сильно обусловлена, то тогда мы можем получить сильно большую норму для $I - \tau A$ для чисел τ, близких к m. Если такие "плохие"матрицы идут друг за другом в произведении, то норма произведения может галопировать. Это приведёт к тому, что малая численная ошибка, зародившшаяся в уже в самом начале, может резко вырасти и подорвать все расчёты. Возможный вариант преодолеть это препятствие - это чередование чисел $\frac{1}{\tau}$, близких к m, с их "родственниками"около M. Тогда "родственник"сможет "погасить"накопление ошибок. На практике мы видим, что это сделало метод работающим.

4 Выводы

Был изучен метод оптимальной n-шаговой итерации для решения СЛУ. Показано, что его работоспособность и эффективность зависит от расстановки скобок в слагаемых, коммутирующих в теории.