МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского» (ННГУ)

Институт информационных технологий, математики и механики

Направление подготовки: «Прикладная математика и информатика»
Профиль подготовки: «Вычислительные методы и суперкомпьютерные технологии»

Отчет
по лабораторной работе
на тему «Схема Кранка-Николсона для уравнения теплопроводности методом циклической редукции »

Выполнил:
студент группы 3823М1ПМвм
Бекетов Е.В.

Проверил:
д.т.н., доц., зав.каф. МОСТ
Баркалов К.А.

Нижний Новгород
2024

Оглавление

1. Введение

Моделирование физических задач из области механики, гидродинамики и прочих и сводится к решению дифференциальных уравнений. Однако большинство задач описываются сложными и не решаемыми аналитически уравнениями. В связи с этим было создано достаточное количество численных методов, большинство из которых заточено на решение задач в определенной области.

Такой задачей является моделирование распространение тепла. Эта задача классическая и достаточно простая, однако это уравнение в частных производных и в этом ее главная сложность.

Для решения уравнений с частными производными как правило используются сеточные методы, суть которых в разбиении области моделирования на сетку и просчет значений в узлах сетки. На основе этой сетки строиться разностное уравнение и уже по нему ищется решение.

Часто при подобном подходе применяются итерационные методы. Вычислительная схема в этом случае описывает, как следующее состояние сетки зависит от предыдущего. В результате моделирования получается приближенное решение уравнений с частными производными.

Однако чем больше сетка тем больше вычислительных узлов приходится просчитывать, а значит мы непременно сталкиваемся с проблемой временных затрат. Одно из решений это применение технологий распараллеливания.

В данной лабораторной работе будет рассмотрен метод циклической редукции для уравнения теплопереноса с применением схемы Кранка-Николсона.

2. Постановка задачи

Условие:
С целью решения динамической задачи теплопроводности реализовать схему Кранка-Николсона ($\sigma = \frac{1}{2}$), используя метод циклической редукции для решения трехдиагональных систем линейных уравнений.
Уравнение теплопроводности имеет вид:

$$u_t^{'}(x,t) = u_{xx}^{''}(x,t) + f(x,t),\ x \in [0,L],\ t \in [0,T]$$

Численное решение рассматривается на сетке:

$$G = \{(x_i, t_j) : x_i = ih, t_j = j\tau, i = \overline{0,n}, j = \overline{0,m}\}$$

где $h = \frac{L}{n}, \tau = \frac{T}{m}$.

Требования:
Программа на языке C++ должна реализовывать функцию со следующим заголовком:
void heat_equation_crank_nicolson(heat_task task, double * v);
class heat_task
{
public:
double T; // момент времени, в который необходимо аппроксимировать u(x, t)
double L; // длина стержня
int n; // размер сетки по x
int m; // размер сетки по t
double initial_condition(double x); // функция, задающая начальное условие
double left_condition (double t); // функция, задающая граничное условие при x = 0
double right_condition (double t); // функция, задающая граничное условие при x = L
double f(double x, double t); // функция, задающая внешнее воздействие
};

Формат входа:
Функция получает в аргументах следующие переменные:
task – экземепляр класса типа heat_task, в котором хранится описание задачи.

Формат выхода:
v – указатель на массив размера $n+1$, в котором необходимо записать численное решение на сетке G для момента времени T.
Ответ считается корректным, если выполнены условия на порядок сходимости метода $O(h^2+\tau^2)$ и при достаточно больших n, m выполнено: $\Delta_{numeric} = |u(x_i,T) - v_i|_\infty < 10^{-6}$.
Порядок сходимости метода проверяется следующим образом. Выберем две пары произвольных размерностей $(n_1, m_1), (n_2, m_2)$. Вычислим для них теоретическую величину $\Delta_{theory} = h^2 + \tau^2$, тогда

$$R_{theory} = \frac{\Delta_{theory1}}{\Delta_{theory2}} \text{ – ожидаемое число раз изменения погрешности}$$

$$R_{numeric} = \frac{\Delta_{numeric1}}{\Delta_{numeric2}} \text{ – реальное число раз изменения погрешности}$$

Ответ считается корректным, если $|R_{theory} - R_{numeric}| < 0.1$

3. Теоретическая часть

3.1. Схема Кранка-Николсона

В основе решения уравнения теплопроводности лежит аппроксимация частных производных:

$$(\frac{\partial u}{\partial t})_{ij} \approx \frac{u_i^{j+1} - u_i^j}{\tau} \text{ – аппроксимация первого порядка по } \tau$$

$$(\frac{\partial^2 u}{\partial x^2})_{ij} \approx \frac{u_{i+1}^j - 2u_i^j + u_{i-1}^j}{h^2} \text{ – аппроксимация второго порядка по } h$$

тогда можно записать разностную схему вида:

$$\frac{v_i^{j+1} - v_i^j}{\tau} = \sigma \frac{v_{i+1}^{j+1} - 2v_i^{j+1} + v_{i-1}^{j+1}}{h^2} + (1-\sigma)\frac{v_{i+1}^j - 2v_i^j + v_{i-1}^j}{h^2} + \phi_i^j$$

$$v_i^0 = u_0(x_i),\ v_0^j = u_1(t_j),\ v_n^j = u_2(t_j)$$

где v_i^j – сеточная функция, являющаяся точным решением разностной схемы и аппроксимирующая точное решение дифференциального уравнения в узлах сетки; $0 \leq \sigma \leq 1$ – параметр, называемый весом, а ϕ_i^j – некоторая правая часть, например $\phi_i^j = f_i^j$

В общем случае данная схема определена на шеститочечном шаблоне, изображенном на Рис. 1.

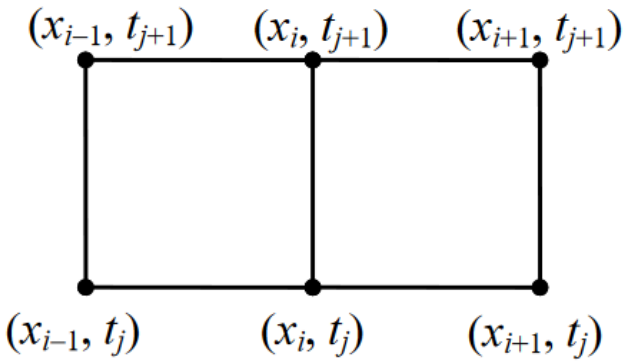

Рис. 1: Шаблон схемы с весом σ.

В случае когда $\sigma = \frac{1}{2}$ получаем следующее разностное уравнение:

$$\frac{v_i^{j+1} - v_i^j}{\tau} = \frac{1}{2}[\frac{v_{i+1}^{j+1} - 2v_i^{j+1} + v_{i-1}^{j+1}}{h^2} + \frac{v_{i+1}^j - 2v_i^j + v_{i-1}^j}{h^2}] + \phi_i^{j+1/2}$$

$$v_i^0 = u_0(x_i),\ v_0^j = u_1(t_j),\ v_n^j = u_2(t_j)$$

данная схема называется схемой Кранка-Николсона. Известно, что симметричная разностная схема с весом $1/2$ будет абсолютно устойчива, а ее погрешность аппроксимации составит $O(\tau^2 + h^2)$. Данные свойства делают схему с весом $1/2$ предпочтительной для проведения расчетов, т.к. она обеспечивает хорошую точность при не слишком малых шагах сетки.

В итоге получается трехдиагональная матрица, которую уже удобно решать с помощью циклической редукции.

3.2. Циклическая редукция

Этот метод достаточно хорош и эффективен, однако есть ограничение, он работает только в случаях, когда матрица имеет размерность равную степени двух, но это не слишком большая проблема.

Смысл метода состоит в последовательном исключении переменных с нечетными индексами (прямой ход редукции) и обратном восстановлении значений нечетных переменных на основании известных значений переменных с четными номерами (обратный ход).

На каждой итерации прямого хода редукции рассматриваются тройки уравнений, неперекрывающихся по уравнениям с четными индексами. Из каждой такой тройки исключаются переменные с нечетными индексами, после чего переменные перенумеровываются и на следующей итера- ции снова исключаются переменные с нечетными индексами.Схема исключения переменных для $N = 8$ случая приведена на Рис. 2.

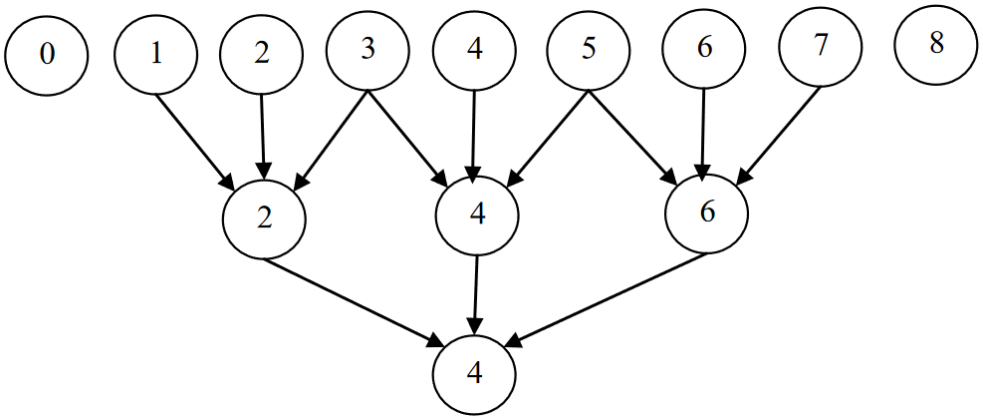

Рис. 2: Схема исключения переменных с нечетными номерами.

На последней $q-1$ итерации исключения переменных останется одно значимое уравнение с двумя фиктивными переменными:

$$x_0 = f_0 = 0$$

$$a^{q-2}x_0 + b^{q-2}x_{N/2} + c^{q-2}x_N = f_{N/2}^{q-2}$$

$$x_N = f_N = 0$$

Данное уравнение можно разрешить относительно переменной $x_{N/2}$:

$$x_{N/2} = \frac{f_{N/2}^{q-2} - a^{q-2}x_0 - c^{q-2}x_N}{b^{q-2}}$$

Таким образом, развертывается обратный ход редукции. На произвольной l-ой итерации можно восстановить переменную с нечетным индексом i , выразив ее из соответствующего уравнения

$$a^l x_{i-2^l} + b^l x_i + c^l x_{i+2^l} = f_i^l$$

$$x_{Ni} = \frac{f_i^l - a^l x_{i-2^l} - c^l x_{x+26l}}{b^l}$$

через известные переменные, полученные на предшествующем шаге обратного хода.

На Рис. 3 приведена схема восстановления решения системы при $N = 8$. Согласно данной схеме переменные пересчитываются последовательно снизу вверх (пересчитываемые на каждом шаге переменные выделены, стрелками от них указаны переменные,

значения которых используются при восстановлении согласно формуле выше). Такая схема восстановления позволяет при пересчете правых частей уравнений в прямом ходе редукции затирать значения, полученные на предыдущем шаге.

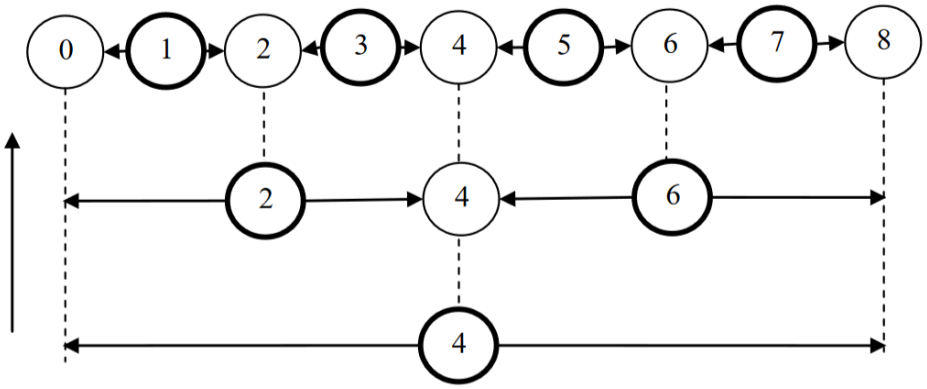

Рис. 3: Схема восстановления переменных на каждой итерации обратного хода редукции .

4. Практическая часть

Для проведения дальнейших численных экспериментов были написаны 2 версии алгоритма: последовательный и параллельный. Код каждой приведен в конце отчета в приложениях.

Эксперименты проводились с использованием следующей конфигурации:

Процессор	8 ядерный AMD Ryzen 7 5800HS (2.8 GHz)
Память, кэш	32 GB, L1 – 32 Kb, L2 – 512 Kb, L3 – 16 Mb
Операционная система	Windows 10 x64
Среда разработки	Visual Studio 2022
Библиотеки	OpenMP
Компилятор	Intel oneAPI DPC++/C++ Compiler (2024.0.2)

4.1. Эксперименты

Проведем серию экспериментов для выявления оптимального числа потоков для сетки размера 8192, 16384, 32768, 65536.

4.2. Параллельный алгоритм

Сделаем замеры алгоритма при разном числе потоков Рис. 4.

Параллельный алгоритм от числа потоков				
Число потоков	Размер матрицы			
	8192	16384	32768	65536
1	0,501	1,623	5,874	25,474
2	0,293	1,081	3,378	13,257
3	0,248	0,942	2,704	9,395
4	0,19	0,709	2,452	8,185

Рис. 4: Время работы параллельного алгоритма в секундах.

Видно, что увеличение числа потоков уменьшает время работы, для любого из рассматриваемых размеров сетки.

Посмотрим для наглядности на таблицу ускорения на Рис. 5.

Ускорение параллельного алгоритма от числа потоков				
Число потоков	Размер матрицы			
	8192	16384	32768	65536
1	1	1	1	1
2	1,709898	1,501388	1,738899	1,921551
3	2,020161	1,72293	2,172337	2,711442
4	2,636842	2,28914	2,395595	3,112279

Рис. 5: Ускорение параллельного алгоритма.

На Рис. 4 и Рис. 5 видно, что параллельный алгоритм работает исправно, не наблюдается никаких аномалий, за исключением небольшого спада производительности для первых трех сеток при использовании 3 и 4 потоков (ожидалась близость к ускорению в 3 и 4 раза соответственно), однако куда большую эффективность по ускорению демонстрирует самая большая сетка (65356). Сверхлинейное расширение не наблюдается. Результаты вполне близки к ожидаемым.

5. Заключение

В ходе данной лабораторной работы был изучен метод циклической редукции для решения уравнения теплопереноса с использованием схемы Кранка-Николсона.

Из проведенных экспериментов было видно, что алгоритм работает исправно, не наблюдается аномалий, однако присутствует не критичный спад производительности.

6. Литература

1. Баркалов К.А. Образовательный комплекс «Параллельные численные методы». - Н.Новгород, Изд-во ННГУ 2011.

2. Самарский А.А., Гулин А.В. Численные методы. – М.: Наука, 1989.

3. Е.А. Рындин, И.В. Куликова, И.Е. Лысенко. Основы численных методов: теория и практика, 2015.

4. Бахвалов Н.С., Жидков Н.П., Кобельков Г.М. Численные методы. – М.: Наука, 1987.

5. Тихонов А.Н., Самарский А.А. Уравнения математической физики. – М.: Наука, 1977.

7. Приложение А – Код последовательной функции

```
void heat_equation_crank_nicolson(heat_task task, double* v) {
    std::vector<double> u(task.n + 1);
    double dx = task.L / task.n;
    double dt = task.T / task.m;
    int log_n = std::log2(task.n);

    for (int i = 0; i <= task.n; ++i) {
        u[i] = task.initial_condition(i * dx);
    }

    double alpha = dt / (2 * dx * dx);
    double beta = -1 - dt / (dx * dx);
    double gamma1 = dt / (dx * dx) - 1;
    double gamma2 = dt / (2 * dx * dx);

    std::vector<double> a(log_n + 1);
    std::vector<double> b(log_n + 1);
    std::vector<double> c(log_n + 1);
    std::vector<double> rhs(task.n + 1);

    for (int j = 1; j <= task.m; ++j) {
        for (int i = 1; i < task.n; ++i) {
            rhs[i] = gamma1 * u[i] - gamma2 * (u[i - 1] + u[i + 1]) -
            dt * task.f(i * dx, (j - 0.5) * dt);
        }
        rhs[1] -= task.left_condition(j * dt) * (dt / (dx * dx * 2));
        rhs[task.n - 1] -= task.right_condition(j * dt) *
        (dt / (dx * dx * 2));

        a[0] = alpha;
        b[0] = beta;
        c[0] = alpha;

        rhs[0] = 0;
        rhs[task.n] = 0;
        u[0] = 0;
        u[task.n] = 0;

        int start = 2;
        int size_n = task.n;
        int step = 1;
        for (int j = 0; j < log_n; ++j) {
            double a_k = -a[j] / b[j];
            double c_k = -c[j] / b[j];
            a[j + 1] = a_k * a[j];
            b[j + 1] = b[j] + a_k * a[j] +
            c_k * c[j];
            c[j + 1] = c_k * c[j];
```

```
            size_n = (size_n - 1) / 2;
            for (int i = 0; i < size_n; ++i) {
                int idx = start * (i + 1);
                rhs[idx] = a_k * rhs[idx - step] +
                rhs[idx] + c_k * rhs[idx + step];
            }
            start = 2 * start;
            step = 2 * step;
        }

        start = task.n / 2;
        size_n = 1;
        for (int k = log_n - 1; k >= 0; --k) {
            double a_k = -a[k] / b[k];
            double c_k = -c[k] / b[k];
            for (int i = 0; i < size_n; ++i) {
                int idx = start * (2 * i + 1);
                u[idx] = rhs[idx] / b[k] + a_k * u[idx - start] +
                c_k * u[idx + start];
            }
            start = start / 2;
            size_n = size_n * 2;
        }

        u[0] = task.left_condition(j * dt);
        u[task.n] = task.right_condition(j * dt);
    }
    for (int i = 0; i <= task.n; ++i) {
        v[i] = u[i];
    }
}
```

8. Приложение Б – Код параллельной функции

```
void heat_equation_crank_nicolson(heat_task task, double* v,
int num_threads) {
    std::vector<double> u(task.n + 1);
    double dx = task.L / task.n;
    double dt = task.T / task.m;
    int log_n = std::log2(task.n);

#pragma omp parallel for num_threads(num_threads)
    for (int i = 0; i <= task.n; ++i) {
        u[i] = task.initial_condition(i * dx);
    }

    double alpha = dt / (2 * dx * dx);
    double beta = -1 - dt / (dx * dx);
    double gamma1 = dt / (dx * dx) - 1;
    double gamma2 = dt / (2 * dx * dx);

    std::vector<double> a(log_n + 1);
    std::vector<double> b(log_n + 1);
    std::vector<double> c(log_n + 1);
    std::vector<double> rhs(task.n + 1);

    for (int j = 1; j <= task.m; ++j) {
#pragma omp parallel for num_threads(num_threads)
        for (int i = 1; i < task.n; ++i) {
            rhs[i] = gamma1 * u[i] - gamma2 * (u[i - 1] + u[i + 1]) -
            dt * task.f(i * dx, (j - 0.5) * dt);
        }
        rhs[1] -= task.left_condition(j * dt) * (dt / (dx * dx * 2));
        rhs[task.n - 1] -= task.right_condition(j * dt) *
        (dt / (dx * dx * 2));

        a[0] = alpha;
        b[0] = beta;
        c[0] = alpha;

        rhs[0] = 0;
        rhs[task.n] = 0;
        u[0] = 0;
        u[task.n] = 0;

        int start = 2;
        int size_n = task.n;
        int step = 1;
        for (int j = 0; j < log_n; ++j) {
            double a_k = -a[j] / b[j];
            double c_k = -c[j] / b[j];
            a[j + 1] = a_k * a[j];
```

```
            b[j + 1] = b[j] + a_k * a[j] +
            c_k * c[j];
            c[j + 1] = c_k * c[j];

            size_n = (size_n - 1) / 2;
#pragma omp parallel for num_threads(num_threads)
            for (int i = 0; i < size_n; ++i) {
                int idx = start * (i + 1);
                rhs[idx] = a_k * rhs[idx - step] +
                rhs[idx] + c_k * rhs[idx + step];
            }
            start = 2 * start;
            step = 2 * step;
        }

        start = task.n / 2;
        size_n = 1;
        for (int k = log_n - 1; k >= 0; --k) {
            double a_k = -a[k] / b[k];
            double c_k = -c[k] / b[k];
#pragma omp parallel for num_threads(num_threads)
            for (int i = 0; i < size_n; ++i) {
                int idx = start * (2 * i + 1);
                u[idx] = rhs[idx] / b[k] + a_k * u[idx - start] +
                c_k * u[idx + start];
            }
            start = start / 2;
            size_n = size_n * 2;
        }

        u[0] = task.left_condition(j * dt);
        u[task.n] = task.right_condition(j * dt);
    }
#pragma omp parallel for num_threads(num_threads)
    for (int i = 0; i <= task.n; ++i) {
        v[i] = u[i];
    }
}
```