|  |  |  |
| --- | --- | --- |
| Министерство науки и высшего образования  Российской Федерации | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования | | |
| «Новосибирский государственный технический университет» | | |
|  | | |
| Кафедра теоретической и прикладной информатики | | |
|  | | |
| Курсовой проект | | |
| по дисциплине «Уравнения математической физики» | | |
|  | | |
| **МКЭ для краевой задачи** | | |
|  | | |
|  | Факультет: | ПМИ |
| Группа: | ПМ-81 |
| Вариант: | Место для ввода текста. |
| Студенты: | Яницкий Борис Леонидович |
|  |  |
|  |  |
| Преподаватели: | Патрушев Илья Игоревич  Задорожный Александр Геннадьевич  Рояк Михаил Эммануилович |
|  | | |
| Новосибирск | | |
| 2021 | | |

1. **Постановка задачи**
   1. **Условие задачи**

МКЭ для двумерной краевой задачи для гиперболического уравнения в декартовой системе координат. Базисные функции линейные. Краевые условия всех типов. Коэффициент диффузии разложить по билинейным базисным функциям. Матрицу СЛАУ сгенерировать в разреженном строчном формате. Для решения СЛАУ использовать МСГ или ЛОС с неполной факторизацией. Трехслойная неявная схема по времени.

* 1. **Решаемое уравнение**
     1. **В общем виде**
     2. **В декартовой системе координат**
  2. **Краевые условия**

1. **Теоретическая часть** 
   1. **Вариационная постановка**

Потребуем, чтобы невязка дифференциального уравнения была ортогональна некоторому пространству пробных функций .

Используя формулу Грина, получим:

Так как , преобразуем интегралы по границам и , воспользовавшись краевыми условиями 2 и 3:

Исключаем так как краевыми условиями не определяется значение . Тогда потребуем, чтобы содержало только - функции, которые принимают нулевые значения на границе . Тогда в качестве выберем - пространство функций, имеющие суммируемые с квадратом производные и равных нулю на границе .

Таким образом, получим вариационное уравнение вида

* 1. **Конечноэлементная дискретизация и переход к локальным матрицам**
     1. **Конечноэлементная дискретизация**

Разобьем Ω на подобласти, получим:

Функцию будем искать в виде разложения по базисным функциям с соответствующими весами :

Итак, используя базисные функции, принимающих нулевые значения во всех узлах сетки кроме одного, СЛАУ для вектора весов может быть записана в матричном виде:

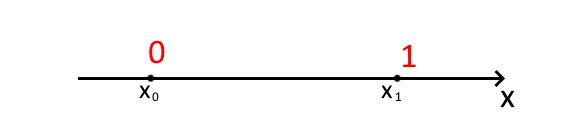
где компоненты матрицы A и вектора b определяются соотношениями:

Матрица жесткости:

Матрица массы:

* + 1. **Построение базисных функций**

Шаблонный конечный элемент для линейных базисных функций выглядит следующим образом:

****

На конечном элементе выберем две базисные функции:

где .

Сделаем замену переменной:

Тогда базисные функции примут следующий вид:

* + 1. **Локальные матрицы**

С учетом замены переменной формула для расчета интегралов для локальной матрицы жесткости примет следующий вид:

Окончательно формула для расчета матрицы жесткости будет иметь вид:

Матрица массы:

Локальный вектор правой части с учетом того что функция на конечном элементе представлена в виде интерполянта

может быть вычислен с помощью соотношения

где - вектор, составленный из значений правой части дифференциального уравнения в узлах элемента, – матрица массы, деленная на .

Краевые условия первого рода учитываются после полной сборки глобальной матрицы и правой части путем фиксации соответствующих весов , при решении СЛАУ. Таким образом, из сгенерированной СЛАУ можно исключить уравнения с теми номерами, которые являются уравнениями узлов, лежащих на границе с краевыми условиями первого рода, а весам с этими номерами присвоить значения первого краевого условия в соответствующих узлах сетки. В работе реализован способ с занулением соответствующей строки глобальной матрицы системы, установкой единицы на главной диагонали и значения точного решения в соответствующей компоненте вектора правой части.

* 1. **Построение схемы по времени**

В трехслойной неявной схеме по времени искомая функция может быть представлена в следующем виде:

где функции являются квадратичными полиномами Лагранжа и имеют следующий вид:

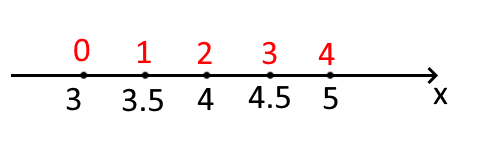
Вычислим производные функций по t в точке :

Вычислим вторые производные функций по t в точке :

Тогда задача сводится к решению следующей системы:

1. **Описание разработанных программ** 
   1. **Структуры данных, используемые для задания расчетной области и конечноэлементной сетки**

Разберем задание расчетной области и конечноэлементной сетки на следующем примере (красным выделены номера узлов):



Расчетная область и конечноэлементная сетка задаются в файле **“grid.txt”** в следующем формате:

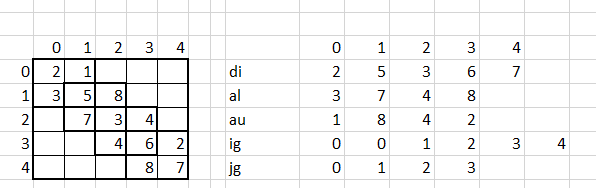
3 5 4

Первое число задает координату левой границы расчетной области, второе – координату павой границы расчетной области. Третье число задает количество разбиений расчетной области. После ввода этих трех чисел генерируется сетка. После генерации сетка хранится в массиве **grid.**

Сетка по времени считывается из файла **“time\_grid.txt”** по схожему принципу. Хранится сетка по времени в массиве **time\_grid.**

* 1. **Структура основных модулей программы, в том числе генерация портрета СЛАУ, вычисление локальных матриц, генерация глобальных матриц, решение СЛАУ.**
     1. **Структура хранения глобальной матрицы**

Разберем формат хранения матрицы на следующем примере:

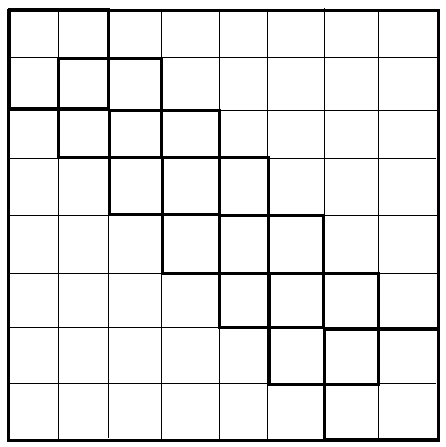


Для хранения диагонали используется массив di.

Для хранения элементов в нижнем и верхнем треугольниках матрицы используются массивы al и au соответственно. В массиве ig на i-той позиции стоит индекс в массиве al (au), с которого в этом массиве начинаются элементы профиля. Длинна профиля в i-той строке равно значению ig[i + 1] – ig[i]. В массиве jg находятся индексы столбцов элементов из массивов al и au.

* + 1. **Генерация глобальной матрицы**

Из-за того, что программа работает с линейными конечными элементами, довольно просто строить глобальную матрицу:



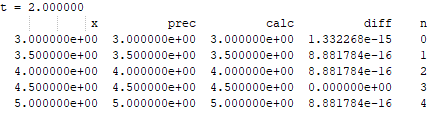
Каждый конечный элемент вносит вклад только в соответствующую область 4х4, выделенную жирным. Так как матрица имеет профильный формат, достаточно идти по конечным элементам, для конечного элемента **k** заносить вклад в диагональ и в вектор правой части с индексами **k** и **k+1**, в треугольники матрицы с индексами **k** (так как треугольники матрицы системы, несмотря на то, что матрица хранится в разреженном формате, будут представлять собой диагонали трехдиагональной матрицы).

* + 1. **Решение СЛАУ**

Решение СЛАУ осуществляется с помощью МСГ.

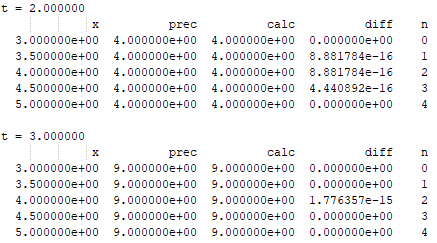
1. **Описание тестирования программы**
   1. **Тестирование на работоспособность с сеткой по пространству**

|  |  |
| --- | --- |
|  | **grid.txt** |
| 3 5 4 |



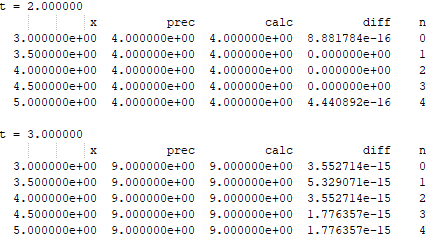
* 1. **Тестирование на работоспособность параболической задачи**

|  |  |
| --- | --- |
|  | **grid.txt** |
| 3 5 4 |



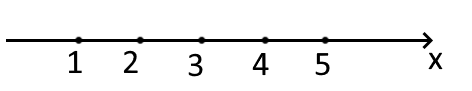
* 1. **Тестирование на работоспособность гиперболической задачи**

|  |  |
| --- | --- |
|  | **grid.txt** |
| 3 5 4 |

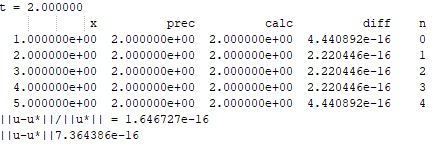


1. **Исследование порядка аппроксимации**

**Сетка по пространству для всех тестов:**

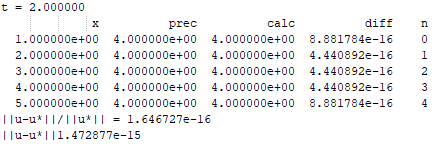
****

* 1. **Исследование порядка аппроксимации по времени**
     1. **Тест 1**



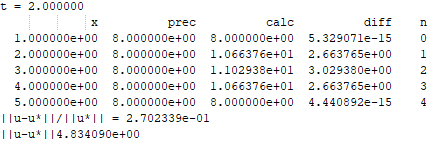
Погрешности нет.

* + 1. **Тест 2**



Погрешности нет.

* + 1. **Тест 3**

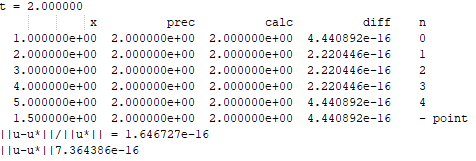


При увеличении степени t в искомой функции, начиная с , происходит увеличение погрешности. Следовательно, порядок аппроксимации схемы по времени = 2.

1. **Исследование порядка аппроксимации по пространству**

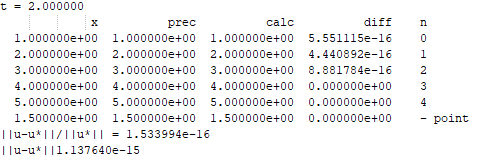
Помимо решения в узлах также выведем решение в произвольной точке расчетной области.

* + 1. **Тест 1**



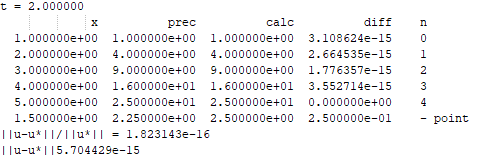
Погрешности нет.

* + 1. **Тест 2**



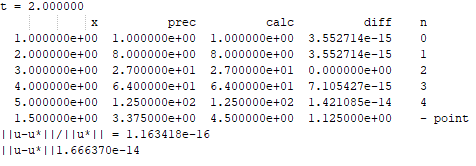
Погрешности нет.

* + 1. **Тест 3**



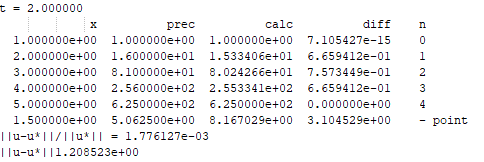
Появилась погрешность между узами сетки.

* + 1. **Тест 4**



Наблюдается погрешность только между узлами.

* + 1. **Тест 5**

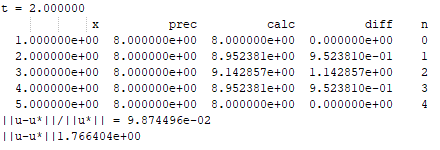


При увеличении степени x в искомой функции, начиная с , происходит увеличение погрешности в неузловой точке, а при и в узловой точке. Следовательно, порядок аппроксимации схемы по пространству = 1.

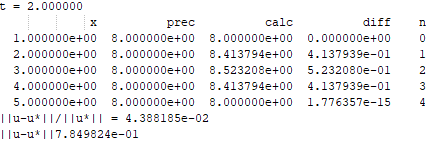
1. **Исследование порядка сходимости**
   1. **Исследование порядка сходимости по времени для параболической задачи**

Для удобства будем отображать только временные слои, которые были на изначальной временной сетке.

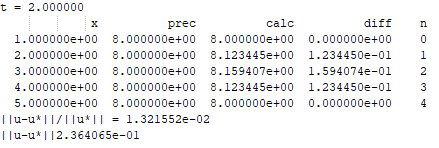
* + 1. **Тест 1**



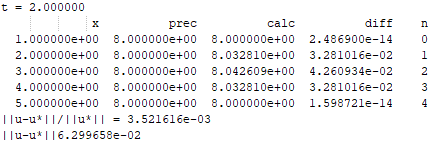
* + 1. **Тест 2**



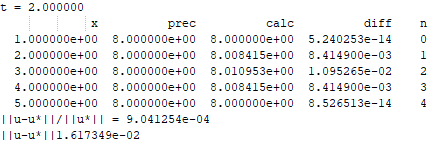
* + 1. **Тест 3**



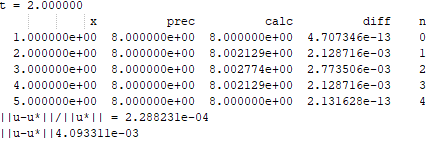
* + 1. **Тест 4**



* + 1. **Тест 5**



* + 1. **Тест 6**



По результатам тестов можно составить таблицу для сравнения погрешностей во внутренней точке области:

|  |  |  |
| --- | --- | --- |
| n | Погрешность | Отношение к предыдущей погрешности |
| 2 | 1,14E+00 |  |
| 4 | 5,23E-01 | 2,18432631 |
| 8 | 1,59E-01 | 3,282206472 |
| 16 | 4,26E-02 | 3,741137506 |
| 32 | 1,10E-02 | 3,890322433 |
| 64 | 2,77E-03 | 3,949026972 |

И по норме вектора погрешности:

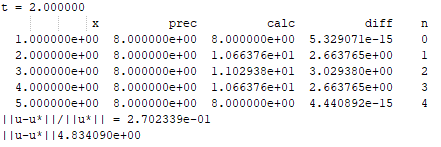
|  |  |  |
| --- | --- | --- |
| n | Погрешность | Отношение к предыдущей погрешности |
| 2 | 1,77E+00 |  |
| 4 | 7,85E-01 | 2,25024663 |
| 8 | 2,36E-01 | 3,320477229 |
| 16 | 6,30E-02 | 3,752687844 |
| 32 | 1,62E-02 | 3,895051717 |
| 64 | 4,09E-03 | 3,951199897 |

По таблице видно, что порядок сходимости схемы для параболической задачи = 2.

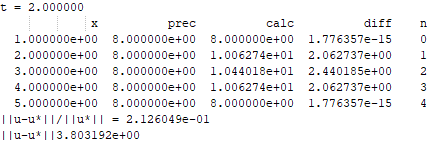
* 1. **Исследование порядка сходимости по времени для гиперболической задачи**

Для удобства будем отображать только временные слои, которые были на изначальной временной сетке.

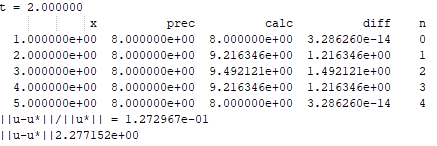
* + 1. **Тест 1**



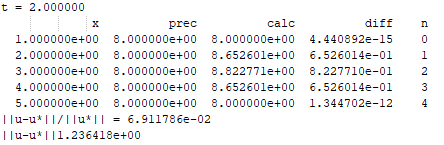
* + 1. **Тест 2**



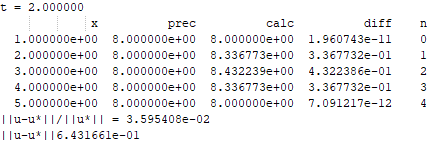
* + 1. **Тест 3**



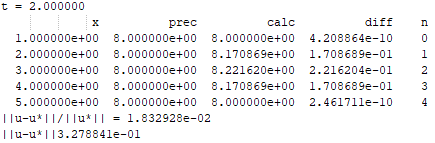
* + 1. **Тест 4**



* + 1. **Тест 5**



* + 1. **Тест 6**



По результатам тестов можно составить таблицу для сравнения погрешностей во внутренней точке области:

|  |  |  |
| --- | --- | --- |
| n | Погрешность | Отношение к предыдущей погрешности |
| 2 | 3,03E+00 |  |
| 4 | 2,44E+00 | 1,241455054 |
| 8 | 1,49E+00 | 1,635380107 |
| 16 | 8,23E-01 | 1,813531347 |
| 32 | 4,32E-01 | 1,903511163 |
| 64 | 2,22E-01 | 1,950355653 |

И по норме вектора погрешности:

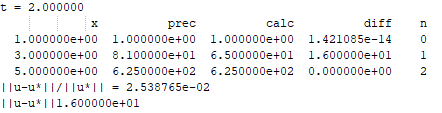
|  |  |  |
| --- | --- | --- |
| n | Погрешность | Отношение к предыдущей погрешности |
| 2 | 4,83E+00 |  |
| 4 | 3,80E+00 | 1,271061256 |
| 8 | 2,28E+00 | 1,670152893 |
| 16 | 1,24E+00 | 1,841733136 |
| 32 | 6,43E-01 | 1,922392987 |
| 64 | 3,28E-01 | 1,961565382 |

По таблице видно, что порядок сходимости схемы для гиперболической задачи = 1.

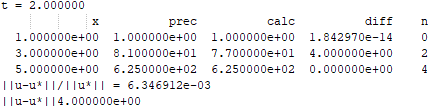
* 1. **Исследование порядка сходимости по пространству**

Для удобства будем отображать только узлы, которые были на изначальной пространственной сетке. При подсчете нормы вектора погрешности также участвуют только узлы, которые присутствовали на изначальной сетке.

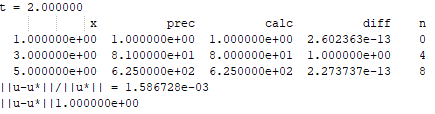
* + 1. **Тест 1**



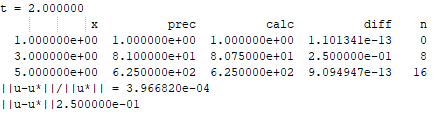
* + 1. **Тест 2**



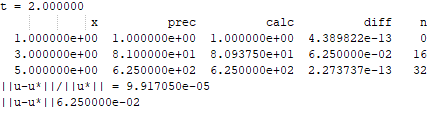
* + 1. **Тест 3**



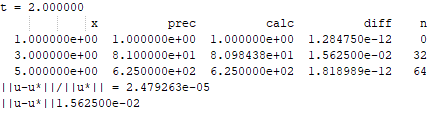
* + 1. **Тест 4**



* + 1. **Тест 5**



* + 1. **Тест 6**



По результатам тестов можно составить таблицу для сравнения погрешностей во внутренней точке области:

|  |  |  |
| --- | --- | --- |
| n | Погрешность | Отношение к предыдущей погрешности |
| 2 | 1,60E+01 |  |
| 4 | 4,00E+00 | 4 |
| 8 | 1,00E+00 | 4 |
| 16 | 2,50E-01 | 4 |
| 32 | 6,25E-02 | 4 |
| 64 | 1,56E-02 | 4 |

и по норме вектора погрешности:

|  |  |  |
| --- | --- | --- |
| n | Погрешность | Отношение к предыдущей погрешности |
| 2 | 1,60E+01 |  |
| 4 | 4,00E+00 | 4 |
| 8 | 1,00E+00 | 4 |
| 16 | 2,50E-01 | 4 |
| 32 | 6,25E-02 | 4 |
| 64 | 1,56E-02 | 4 |

По таблицам видно, что порядок сходимости схемы по пространству = 2.

1. **Тексты разработанных программ**
   1. **Файл HyperbolicalProblem.h с описанием всех необходимых функций и структур данных для решения гиперболической задачи**

#pragma once

#include <vector>

#include <fstream>

#include <iomanip>

#include "SLAE.h"

#include "Test.h"

using namespace std;

class HyperbolicalProblem

{

public:

vector<double> grid; // Сетка по пространству

vector<double> time\_grid; // Сетка по времени

int nodes\_count = 0; // Количество узлов

int elems\_count = 0; // Количество конечных элементов

vector<double> b; // Глобальный вектор правой части

vector<double> solution; // Полученное решение

vector<double> true\_solution; // Точное решение

vector<double> xk; // Вектор начального приближения для МСГ

vector<double> prev\_solution1; // Решение на предыдущей итерации по времени

vector<double> prev\_solution2; // Решение на предпредыдущей итерации по времени

// Вспомогательные векторы для итерации по времени

vector<double> Mqj\_1, Mqj\_2;

// Глобальные матрицы

Matrix A; // Матрица системы

Matrix G; // Матрица жесткости

Matrix M; // Матрица массы

SLAE slae; // СЛАУ

Test test; // Тестовая информация

HyperbolicalProblem()

{

}

// Функция считывания и формирования сетки по пространству

// из файла file\_name

void ReadFormGrid(const string& file\_name)

{

ifstream fin(file\_name);

int n; // Количество разбиений

double left, right; // Левая и правая границы расчетной области

fin >> left >> right >> n;

nodes\_count = n + 1;

double h = (right - left) / n; // Шаг разбиения

grid = vector<double>(nodes\_count);

// Формирование сетки по пространству

for(size\_t i = 0; i < nodes\_count; i++)

grid[i] = i \* h + left;

// Рачет количества конечных элементов

elems\_count = nodes\_count - 1;

fin.close();

}

// Функция считывания и формирования сетки по времени

// из файла file\_name

void ReadFormTimeGrid(const string& file\_name)

{

ifstream fin(file\_name);

int n; // Количество разбиений

double left, right; // Первая и последняя точки сетки по времени

fin >> left >> right >> n;

double h = (right - left) / n; // Шаг разбиения

time\_grid = vector<double>(n + 1);

// Формирование сетки по времени

for(size\_t i = 0; i < n + 1; i++)

time\_grid[i] = i \* h + left;

fin.close();

}

// Инициализация памяти под все необходимые векторы и матрицы

void InitializeMemory()

{

slae = SLAE(nodes\_count, 10000, 1e-20);

b = vector<double>(nodes\_count);

solution = vector<double>(nodes\_count);

true\_solution = vector<double>(nodes\_count);

prev\_solution1 = vector<double>(nodes\_count);

prev\_solution2 = vector<double>(nodes\_count);

Mqj\_1 = vector<double>(nodes\_count);

Mqj\_2 = vector<double>(nodes\_count);

A = Matrix(nodes\_count, nodes\_count - 1);

G = Matrix(nodes\_count, nodes\_count - 1);

M = Matrix(nodes\_count, nodes\_count - 1);

xk = vector<double>(nodes\_count);

}

// Функция формирования портрета матрицы

void FormPortrait(Matrix& m)

{

m.ind[0] = 0;

m.ind[1] = 0;

for(size\_t i = 1; i < nodes\_count; i++)

{

m.ind[i + 1] = m.ind[i] + 1;

m.columns\_ind[i - 1] = i - 1;

}

}

// Функция заполнения матриц жесткости и массы

void FillMatrices()

{

// Индекс очередного элемента на добавление в треугольник матрицы

int to\_add\_i\_tr = 0;

// Индекс очередного элемента на добавление в диагональ матрицы

int to\_add\_i\_di = 0;

// Цикл по конечным элементам

for(int elem\_i = 0; elem\_i < elems\_count; elem\_i++)

{

// Координаты узлов конечного элемента

double x0 = grid[elem\_i];

double x1 = grid[elem\_i + 1];

// Ширина конечного элемента

double hx = x1 - x0;

// Заполнение матрицы жесткости

G.diag[to\_add\_i\_di] += test.lambda / hx \* 1.0;

G.diag[to\_add\_i\_di + 1] += test.lambda / hx \* 1.0;

G.bot\_tr[to\_add\_i\_tr] += test.lambda / hx \* -1.0;

// Заполнение матрицы массы

M.diag[to\_add\_i\_di] += hx / 6.0 \* 2.0;

M.diag[to\_add\_i\_di + 1] += hx / 6.0 \* 2;

M.bot\_tr[to\_add\_i\_tr] += hx / 6.0 \* 1.0;

to\_add\_i\_di++;

to\_add\_i\_tr++;

}

// Заполнение верхних треуголников

// в силу симметричности матрицы

G.top\_tr = G.bot\_tr;

M.top\_tr = M.bot\_tr;

}

// Функция заполнения вектора правой части

void FillB(const double& t)

{

// Индекс очередного элемента на добавление

int to\_add\_i = 0;

// Цикл по конечным элементам

for(int elem\_i = 0; elem\_i < elems\_count; elem\_i++)

{

// Координаты узлов конечного элемента

double x0 = grid[elem\_i];

double x1 = grid[elem\_i + 1];

// Ширина конечного элемента

double hx = x1 - x0;

// Заполнение вектора точного решения

true\_solution[to\_add\_i] = test.u(x0, t);

true\_solution[to\_add\_i + 1] = test.u(x1, t);

// Заполнение вектора правой части системы

b[to\_add\_i] += hx / 6.0 \* (2.0 \* test.f(x0, t) + test.f(x1, t));

b[to\_add\_i + 1] += hx / 6.0 \* (test.f(x0, t) + 2 \* test.f(x1, t));

to\_add\_i++;

}

}

// Функция сборки глобальной матрицы системы

void AssembleGlobalMatrix(const double& c1, const double& c2)

{

// Сборка диагонали матрицы

for(size\_t i = 0; i < nodes\_count; i++)

A.diag[i] = G.diag[i] + c1 \* test.sigma \* M.diag[i] + c2 \* test.chi \* M.diag[i];

// Сборка нижнего треугольнка матрицы

for(size\_t i = 0; i < A.tr\_size; i++)

A.bot\_tr[i] = G.bot\_tr[i] + c1 \* test.sigma \* M.bot\_tr[i] + c2 \* test.chi \* M.bot\_tr[i];

A.top\_tr = A.bot\_tr;

}

// Функция учета краевых условий

void AccountBound(const double& t)

{

// Точное решение в вектор правой части на граничных узлах

b[0] = test.u(grid[0], t);

b[nodes\_count - 1] = test.u(grid[nodes\_count - 1], t);

// Единица на главной диагонали на граничных узлах

A.diag[0] = 1.0;

A.diag[nodes\_count - 1] = 1.0;

// Нули в строках матрицы на граничных узлах

A.top\_tr[0] = 0.0;

A.bot\_tr[A.tr\_size - 1] = 0.0;

}

// Нахождение решения СЛАУ

void Solve()

{

slae.b = b;

cout << slae.ConjGradMethod(xk, solution, A) << endl;

}

// Базисная функция 1 на шаблонном конечном элементе

double phi1(const double& x)

{

return 1 - x;

}

// Базисная функция 2 на шаблонном конечном элементе

double phi2(const double& x)

{

return x;

}

// Вывод решения на временном слое t в поток fout

void PrintSolution(ofstream& fout, const double& t)

{

// Шапка таблицы

fout << "t = " << fixed << t << endl;

fout << setw(14) << "x";

fout << setw(14) << "prec" << setw(14) << "calc" << setw(14) << "diff" << setw(5) << "n" << endl;

double norm = 0, norm\_u = 0;

// Цикл по узлам сетки

for(int i = 0; i < nodes\_count; i++)

{

double prec = true\_solution[i]; // Точное решение в узле

double calc = solution[i]; // Полученное решение в узле

//if(i % 32 == 0)

{

fout << scientific;

fout << setw(14) << grid[i];

fout << setw(14) << prec;

fout << setw(14) << calc;

fout << setw(14) << abs(true\_solution[i] - solution[i]);

fout << fixed << setw(5) << i;

fout << endl;

// Накопление вектора точного решения

norm\_u += prec \* prec;

// Накопление вектора погрешности

norm += abs(prec - calc) \* abs(prec - calc);

}

}

//// Блок вывода для вывода решения в произвольной точке расчетной области

// double x = 0.5;

// vector<double> s = solution;

// double calc = phi1(x) \* s[0] + phi2(x) \* s[1];

// double prec = test.u(1.5, 2);

// fout << scientific;

// fout << setw(14) << 1.5;

// fout << setw(14) << prec;

// fout << setw(14) << calc;

// fout << setw(14) << abs(prec - calc);

// fout << fixed << setw(5) << "-";

// fout << " point" << endl;

// Расчет и вывод норм векторов относительной и абсолютной погрешности решения

fout << "||u-u\*||/||u\*|| = " << scientific << sqrt(norm) / sqrt(norm\_u) << endl;

fout << "||u-u\*||" << scientific << sqrt(norm) << endl;

}

// Итерация трехслойной неявной схемой по времени, вывод в fout

void IterateTime(ofstream& fout)

{

// Сборка матриц, которую можно провести один раз до начала итераций

FillMatrices();

// Получение векторов начального приближения на первых двух временных слоях

for(int x\_i = 0; x\_i < nodes\_count; x\_i++)

{

prev\_solution2[x\_i] = test.u(grid[x\_i], time\_grid[0]);

prev\_solution1[x\_i] = test.u(grid[x\_i], time\_grid[1]);

}

// Цикл по временным слоям

for(int time\_i = 2; time\_i < time\_grid.size(); time\_i++)

{

// Временные слои

double t2 = time\_grid[time\_i - 2];

double t1 = time\_grid[time\_i - 1];

double t0 = time\_grid[time\_i - 0];

// Вспомогательные переменные

double dt = t0 - t2;

double dt0 = t0 - t1;

double dt1 = t1 - t2;

// Числители дробей

double den2 = dt1 \* dt;

double den1 = dt1 \* dt0;

double den0 = dt \* dt0;

// Первые произодные

double n2 = dt0;

double n1 = dt;

double n0 = dt + dt0;

// Вторые производные

double o2 = 2;

double o1 = 2;

double o0 = 2;

// Обнуление веткора правой части

for(int i = 0; i < nodes\_count; i++)

b[i] = 0;

// Заполнение вектора правой части

FillB(t0);

M.MatVecMult(prev\_solution1, Mqj\_1, M.bot\_tr, M.top\_tr);

M.MatVecMult(prev\_solution2, Mqj\_2, M.bot\_tr, M.top\_tr);

// Сборка глобальной матрицы системы

AssembleGlobalMatrix(n0 / den0, o0 / den0);

// Сборка вектора правой части для трехслойной неявной схемы

for(int i = 0; i < nodes\_count; i++)

b[i] += - n2 / den2 \* test.sigma \* Mqj\_2[i] + n1 / den1 \* test.sigma \* Mqj\_1[i]

- o2 / den2 \* test.chi \* Mqj\_2[i] + o1 / den1 \* test.chi \* Mqj\_1[i];

// Учет первых краевых условий

AccountBound(t0);

// Решение полученной системы

Solve();

// Вывод решения

//if(time\_i == time\_grid.size() - 1)

{

PrintSolution(fout, t0);

fout << endl;

}

// Обновление векторов решения на предыдущих итерациях

prev\_solution2 = prev\_solution1;

prev\_solution1 = solution;

}

}

};

* 1. **Файл SLAE.h с описанием всех необходимых функций и структур данных для решения СЛАУ в разреженном формате методом сопряженных градиентов**

#pragma once

#include "Vector.h"

#include "Matrix.h"

using namespace std;

class SLAE

{

public:

int maxiter; // Максимальное количество итераций

double eps; // Велечина требуемой относительной невязки

vector<double> b; // Вектор правой части

vector<double> t; // Вспомогательный вектор для МСГ

vector<double> tt; // Вспомогательный вектор для МСГ

vector<double> rk1; // Вектор невязки на перд. итерации МСГ

vector<double> zk1; // Вектор спуска на пред. итерации МСГ

vector<double> AtAzk1; // Вспомогательный вектор для МСГ

SLAE(int size, int \_maxiter, double \_eps)

{

maxiter = \_maxiter;

eps = \_eps;

b.resize(size);

t.resize(size);

tt.resize(size);

rk1.resize(size);

zk1.resize(size);

AtAzk1.resize(size);

}

SLAE()

{

}

// Метод сопряженных градиентов, возвращает количество итераций

int ConjGradMethod(vector<double>& xk1, vector<double>& res, Matrix& mat)

{

for(int i = 0; i < mat.size; i++)

res[i] = xk1[i] = 0;

mat.MatVecMult(xk1, t, mat.bot\_tr, mat.top\_tr); // t = A \* x0

mat.MatVecMult(b - t, rk1, mat.top\_tr, mat.bot\_tr); // r0 = AT(f - A \* x0)

zk1 = rk1;

int k = 1;

while(k < maxiter)

{

mat.MatVecMult(zk1, t, mat.bot\_tr, mat.top\_tr); // t = A \* zk-1

mat.MatVecMult(t, AtAzk1, mat.top\_tr, mat.bot\_tr); // AtAzk1 = At \* A \* zk-1

double ak = (rk1 \* rk1) / (AtAzk1 \* zk1);

xk1 = xk1 + ak \* zk1;

double bk = rk1 \* rk1;

rk1 = rk1 - ak \* AtAzk1;

bk = (rk1 \* rk1) / bk;

zk1 = rk1 + bk \* zk1;

double disc = Norm(rk1) / Norm(b); // Относительная невязка

if(disc < eps)

break;

else

k++;

}

res = xk1;

return k;

}

// Метод сопряженных градиентов с предобусловденной неполной

// факторизацией матрицей, возвращает количество итераций

int ConjGradPredMethod(vector<double>& xk1, vector<double>& res, Matrix& mat, SLAE& fac\_slae, Matrix& fac\_mat)

{

for(int i = 0; i < mat.size; i++)

res[i] = xk1[i] = 0;

mat.MatVecMult(xk1, t, mat.bot\_tr, mat.top\_tr); // t = A \* x0

mat.MatVecMult(b - t, rk1, mat.top\_tr, mat.bot\_tr); // r0 = AT(f - A \* x0)

// Решаем z0 = M-1 \* r0

fac\_slae.b = rk1;

fac\_slae.ConjGradMethod(t, zk1, fac\_mat);

int k = 1;

while(k < maxiter)

{

// Решаем tt = M-1 \* rk-1

fac\_slae.b = rk1;

fac\_slae.ConjGradMethod(t, tt, fac\_mat);

mat.MatVecMult(zk1, t, mat.bot\_tr, mat.top\_tr); // t = A \* zk-1

mat.MatVecMult(t, AtAzk1, mat.top\_tr, mat.bot\_tr); // AtAzk1 = At \* A \* zk-1

double ak = (tt \* rk1) / (AtAzk1 \* zk1);

xk1 = xk1 + ak \* zk1;

double bk = tt \* rk1;

rk1 = rk1 - ak \* AtAzk1;

// Решаем tt = M-1 \* rk

fac\_slae.b = rk1;

fac\_slae.ConjGradMethod(t, tt, fac\_mat);

bk = (tt \* rk1) / bk;

zk1 = tt + bk \* zk1;

double disc = Norm(rk1) / Norm(b);

if(disc < eps)

break;

else

k++;

}

res = xk1;

return k;

}

};

* 1. **Файл Matrix.h с описанием структур данных для хранения СЛАУ в разреженном формате методом**

#pragma once

#include <vector>

#include <fstream>

using namespace std;

class Matrix

{

public:

int size = 0; // Размер матрицы

int tr\_size = 0; // Количество элементов в треугольнике

vector<int> ind; // Указатели начала строк

vector<int> columns\_ind; // Номера столбцов внедиагональных элементов

vector<double> top\_tr; // Верхний треугольник

vector<double> bot\_tr; // Нижний треугольник

vector<double> diag; // Диагональ

// Конструктор для матрицы с известным количеством элементов в треуголниках

Matrix(const int& t\_size, const int& t\_tr\_size) : size(t\_size), tr\_size(t\_tr\_size)

{

top\_tr = vector<double>(tr\_size);

bot\_tr = vector<double>(tr\_size);

columns\_ind = vector<int>(tr\_size);

diag = vector<double>(size);

ind = vector<int>(size + 1);

}

Matrix(const Matrix& mat)

{

size = mat.size;

tr\_size = mat.tr\_size;

top\_tr = mat.top\_tr;

bot\_tr = mat.bot\_tr;

diag = mat.diag;

ind = mat.ind;

columns\_ind = mat.columns\_ind;

}

Matrix()

{

}

// Получение диагональной факторизации матрицы

void DiagFact(Matrix& fact)

{

fact.diag = diag;

for (int i = 0; i < size + 1; i++)

fact.ind[i] = 0;

}

// Функция умножения матрицы на вектор vec, результат в res

void MatVecMult(const vector<double>& vec, vector<double>& res,

const vector<double>& bot\_tr, const vector<double>& top\_tr)

{

for (int i = 0; i < size; i++)

res[i] = 0;

for (int i = 0; i < size; i++)

{

res[i] += vec[i] \* diag[i];

int prof\_len = ind[i + 1] - ind[i];

for (int k = 0; k < prof\_len; k++)

{

int i\_in\_gg = ind[i] + k;

int j = columns\_ind[i\_in\_gg];

res[i] += vec[j] \* bot\_tr[i\_in\_gg];

res[j] += vec[i] \* top\_tr[i\_in\_gg];

}

}

}

// Функция умножения матрицы на вектор vec, результат в res

void MatVecMult(const vector<double>& vec, vector<double>& res)

{

MatVecMult(vec, res, bot\_tr, top\_tr);

}

// Зануление всех элементов матрицы

void ResetValues()

{

for(int i = 0; i < size; i++)

diag[i] = 0;

for(int i = 0; i < tr\_size; i++)

{

bot\_tr[i] = 0;

top\_tr[i] = 0;

}

}

// Умножение матрицы на число

void Mult(const double& val)

{

for(int i = 0; i < size; i++)

diag[i] \*= val;

for(int i = 0; i < tr\_size; i++)

{

bot\_tr[i] \*= val;

top\_tr[i] \*= val;

}

}

};

* 1. **Файл Vector.h с описанием перегрузок операторов для векторов из стандартной библиотеки c++**

#pragma once

#include <vector>

using namespace std;

// Умножение вектора на число

vector<double> operator \* (const double& val, vector<double> vec)

{

for (size\_t i = 0; i < vec.size(); i++)

vec[i] \*= val;

return vec;

}

// Сложение векторов

vector<double> operator + (vector<double> vec1, const vector<double>& vec2)

{

for (size\_t i = 0; i < vec1.size(); i++)

vec1[i] += vec2[i];

return vec1;

}

// Вычитание векторов

vector<double> operator - (vector<double> vec1, const vector<double>& vec2)

{

for(size\_t i = 0; i < vec1.size(); i++)

vec1[i] -= vec2[i];

return vec1;

}

// Скалярное произведение векторов

double ScalarMult(const vector<double>& vec1, const vector<double>& vec2)

{

double res = 0;

for(int i = 0; i < vec1.size(); i++)

res += vec1[i] \* vec2[i];

return res;

}

// Скалярное произведение векторов

double operator \* (const vector<double>& vec1, const vector<double>& vec2)

{

return ScalarMult(vec1, vec2);

}

// Норма вектора

double Norm(const vector<double>& vec)

{

return sqrt(ScalarMult(vec, vec));

}

* 1. **Файл Test.h с описанием функций для проведения тестов**

#pragma once

#pragma once

using namespace std;

class Test

{

public:

int test\_n = 0; // Номер теста

double lambda = 1;

double sigma = 0;

double chi = 0;

Test(const int& t\_N) : test\_n(t\_N) {};

Test() { };

double f(const double& x, const double& t)

{

return -1 \* divlambdagrad(x, t) +

sigma \* dudt(x, t) + chi \* d2udt2(x, t);

//return -1 \* divlambdagrad(x, t) + sigma \* u(x, t);

}

// Точное решение

double u(const double& x, const double& t)

{

switch(test\_n)

{

case 0: return 2.0;

case 1: return x;

case 2: return x \* x;

case 3: return x \* x \* x;

case 4: return x \* x \* x \* x;

case 5: return t;

case 6: return t \* t;

case 7: return t \* t \* t;

case 8: return t \* t \* t \* t;

};

}

double divlambdagrad(const double& x, const double& t)

{

switch(test\_n)

{

case 0: return 0;

case 1: return 0;

case 2: return 2;

case 3: return 6 \* x;

case 4: return 12 \* x \* x;

case 5: return 0;

case 6: return 0;

case 7: return 0;

case 8: return 0;

};

}

// Производная точного решения по t

double dudt(const double& x, const double& t)

{

switch(test\_n)

{

case 0: return 0;

case 1: return 0;

case 2: return 0;

case 3: return 0;

case 4: return 0;

case 5: return 1;

case 6: return 2 \* t;

case 7: return 3 \* t \* t;

case 8: return 4 \* t \* t \* t;

};

}

// Вторая производная точного решения по t

double d2udt2(const double& x, const double& t)

{

switch(test\_n)

{

case 0: return 0;

case 1: return 0;

case 2: return 0;

case 3: return 0;

case 4: return 0;

case 5: return 0;

case 6: return 2;

case 7: return 6 \* t;

case 8: return 12 \* t \* t;

};

}

};

* 1. **Файл main.cpp, выполняющий последовательность вызова функций для решения поставленной задачи**

#include <iostream>

#include "HyperbolicalProblem.h"

int main()

{

HyperbolicalProblem hp;

hp.ReadFormGrid("grid.txt");

hp.ReadFormTimeGrid("time\_grid.txt");

hp.InitializeMemory();

hp.test = Test(4);

hp.FormPortrait(hp.A);

hp.FormPortrait(hp.M);

hp.FormPortrait(hp.G);

ofstream fout("result.txt");

hp.IterateTime(fout);

fout.close();

}