МИНИСТЕРСТВО НАУКИ
И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Новосибирский государственный технический
университет»

Новосибирский
государственный
технический университет
НЭТИ

Кафедра прикладной математики

Курсовая работа
по дисциплине «Уравнения математической физики»

Факультет:	ПМИ
Группа:	ПМ-71
Студенты:	Востриков Вячеслав
Вариант:	41, 2
Преподаватели:	Персова Марина Геннадьевна, Патрушев Илья Игоревич
Дата:	9.06.2020

Новосибирск
2020

1. Условие задачи:

МКЭ для двумерной начально-краевой задачи для параболического уравнения с неявной трехслойной схемой аппроксимации по времени в декартовой системе координат с билинейными базисными функциями на прямоугольниках, с краевыми условиями всех типов и с разложением коэффициента диффузии по билинейным базисным функциям.

2. Метод решения:

Дано уравнение параболического вида с начально - краевыми условиями:

$$\sigma\frac{du}{dt} - div\big(\lambda grad(u)\big) = f$$
$$u|_{S_1} = u_g$$
$$\lambda\frac{\partial u}{\partial n}|_{S_2} = \Theta$$
$$\left(\lambda\frac{\partial u}{\partial n} + \beta\big(u - u_\beta\big)\right)|_{S_3} = 0$$
$$u(t)|_{t=t_0} = u_0$$

Рассмотрим аппроксимацию по неявной 3-х слойной схеме:

$$u(x,t) = u_{j-2}(x)\cdot\ \eta_2^j(t) + \ u_{j-1}(x)\cdot\ \eta_1^j(t) + \ u_j(x)\cdot\ \eta_0^j(t),$$

где $u_{j-2}(x)$ – это решение u на временном слое j-2, $u_{j-1}(x)$ – это решение на j-1 временном слое, $u_j(x)$ – это решение на j временном слое, а η_k^j - это базисные квадратичные полиномы Лагранжа, которые вычисляются следующим образом:

$$\eta_2^j(t) = \frac{1}{\Delta t_1 \Delta t}\cdot(t - \ t_{j-1})\cdot(t - \ t_j)^1$$
$$\eta_1^j(t) = \ -\frac{1}{\Delta t_1 \Delta t_0}\cdot(t - \ t_{j-2})\cdot(t - \ t_j)$$
$$\eta_0^j(t) = \frac{1}{\Delta t \Delta t_0}\cdot(t - \ t_{j-2})\cdot(t - \ t_{j-1})$$
$$\Delta t = t_j - \ t_{j-2}\,, \qquad \Delta t_0 = t_j - \ t_{j-1}\,, \qquad \Delta t_1 = t_{j-1} - \ t_{j-2}$$

Исходя из такого представления решения u преобразуем уравнение:

[1] Метод конечных элементов для решеня скаларных и векторных задач : учеб. пособие / Ю. Г Соловейчик, М.Э. Рояк, М. Г. Персова – Новосибирск: Изд-во НГТУ, 2007. – 896с. (“Учебники НГТУ”) стр. 368

$$\sigma \cdot \frac{\partial u(x,t)}{\partial t} = \sigma \cdot \frac{\partial}{\partial t} \cdot \left(u_{j-2}(x) \cdot \eta_2^j(t) + u_{j-1}(x) \cdot \eta_1^j(t) + u_j(x) \cdot \eta_0^j(t) \right)$$

Отыскание производной u по dt сводится к вычислению производных базисных полиномов Лагранжа:

$$\left. \frac{d\eta_2^j(t)}{dt} \right|_{t=t_j} = \frac{\Delta t_0}{\Delta t_1 \Delta t}$$

$$\left. \frac{d\eta_1^j(t)}{dt} \right|_{t=t_j} = -\frac{\Delta t}{\Delta t_1 \Delta t_0}$$

$$\left. \frac{d\eta_0^j(t)}{dt} \right|_{t=t_j} = \frac{\Delta t + \Delta t_0}{\Delta t \Delta t_0}$$

С учетом этих поправок:

$$\sigma \cdot \frac{\partial u(x,t)}{\partial t} = \sigma \cdot \left(\frac{\Delta t_0}{\Delta t_1 \Delta t} \cdot u_{j-2}(x) - \frac{\Delta t}{\Delta t_1 \Delta t_0} \cdot u_{j-1}(x) + \frac{\Delta t + \Delta t_0}{\Delta t \Delta t_0} \cdot u_j(x) \right)$$

$$u(x,t)|_{t=t_1} = u(x,t_1)$$

Для $j-$ того элемента:

$$-div\left(\lambda grad(u_j)\right) + \sigma \cdot \frac{\Delta t_0}{\Delta t_1 \Delta t} \cdot u_{j-2}(x) - \sigma \cdot \frac{\Delta t}{\Delta t_1 \Delta t_0} \cdot u_{j-1}(x) + \sigma \cdot \frac{\Delta t + \Delta t_0}{\Delta t \Delta t_0} \cdot u_j(x) = f_j$$

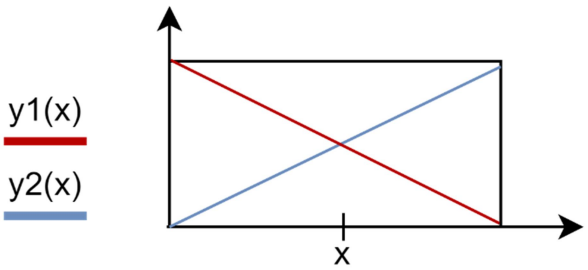
Тем самым получаем СЛАУ:

$$\left(\frac{\Delta t + \Delta t_0}{\Delta t \Delta t_0} \cdot C_\sigma + B \right) \cdot u_j(x) = F_j - \frac{\Delta t_0}{\Delta t_1 \Delta t} \cdot C_\sigma \cdot u_{j-2}(x) + \frac{\Delta t}{\Delta t_1 \Delta t_0} \cdot C_\sigma \cdot u_{j-1}(x)$$

2.1. Метод построения базисных функций

Ранее было определено, что область исследования разбивается на подобласти, которыми, по условию задачи, являются прямоугольные элементы. Функции ψ_i будем строить таким образом: они имеют свои ненулевые значения только на определенном элементе, на всех других элементах они равны нулю. Изобразим один из элементов. На каждом элементе мы построим 4 билинейных функции из тех соображений, что в каждом узле только одна функция принимает значение 1, а другие 0. В итоге мы получим 4 локальных функции. Запишем полученные соотношения, на конечном элементе. Сначала покажем как будут строиться билинейные функции.

Построим одномерные билинейные функции, они имеют вид:



Каждая из двух функций равна 1 в одном из узлов и в другом узле равна 0. Уравнения данных функций имеют вид:

$$\begin{cases} y_1(x) = 1 - x \\ y_2(x) = x \end{cases}$$

Билинейные функции строятся как произведение функций X и Y, которые являются в свою очередь одномерными функциями, представленными выше, тогда локальные функции ψ имеют вид:

$$\psi_{11} = X_1(x)Y_1(y) \qquad \psi_{12} = X_1(x)Y_2(y)$$
$$\psi_{21} = X_2(x)Y_1(y) \qquad \psi_{22} = X_2(x)Y_2(y)$$

2.2. Построение матриц жесткости и масс

Начнем с построения матрицы жесткости. Матрица жесткости получается после подстановки функций ψ в соответствующие градиенты. Матрица жесткости будет иметь размер 4x4, в силу построения локальных функций ψ. Каждый элемент матрицы жесткости представляет собой интеграл от скалярного произведения с параметром λ. Поэтому распишем этот интеграл в общем виде, после чего подставим в этот вид все локальные функции.

$$\int_{\Omega} \lambda grad(\psi_{ij}) grad(\psi_{km}) d\Omega = \int_{\Omega} \lambda grad(X_i Y_j) grad(X_k Y_m) d\Omega =$$

$$\int_{\Omega} \lambda \left(\frac{\partial}{\partial x}(X_i) Y_j, \frac{\partial}{\partial y}(Y_j) X_i \right) \left(\frac{\partial}{\partial x}(X_k) Y_m, \frac{\partial}{\partial y}(Y_m) X_k \right) d\Omega =$$

$$\int_{\Omega} \lambda \left(\frac{\partial}{\partial x}(X_i) Y_j \frac{\partial}{\partial x}(X_k) Y_m + \frac{\partial}{\partial y}(Y_j) X_i \frac{\partial}{\partial y}(Y_m) X_k \right) d\Omega =$$

$$\int_{\Omega} \lambda \left(\frac{\partial}{\partial x}(X_i) \frac{\partial}{\partial x}(X_k) Y_m Y_j + \frac{\partial}{\partial y}(Y_j) \frac{\partial}{\partial y}(Y_m) X_k X_i \right) d\Omega =$$

Теперь подставим величину $\lambda = \sum_{i=1}^{9} \varphi_i \lambda_i$ и запишем полученный интеграл.

$$\int_{\Omega} \sum_{l=1}^{9} \varphi_l \lambda_l \left(\frac{\partial}{\partial x}(X_i) \frac{\partial}{\partial x}(X_k) Y_m Y_j \right) d\Omega + \int_{\Omega} \sum_{l=1}^{9} \varphi_l \lambda_l \left(\frac{\partial}{\partial y}(Y_j) \frac{\partial}{\partial y}(Y_m) X_k X_i \right) d\Omega =$$

$$\sum_{l=1}^{9} \lambda_l \left[\int_x \int_y \varphi_l \left(\frac{\partial}{\partial x}(X_i) \frac{\partial}{\partial x}(X_k) Y_m Y_j \right) dxdy + \int_x \int_y \varphi_l \left(\frac{\partial}{\partial y}(Y_j) \frac{\partial}{\partial y}(Y_m) X_k X_i \right) dxdy \right]$$

$x \in [0,1], y \in [0,1]$

Принадлежность областей интегрирования отрезку [0;1] подразумевает линейную замену величин x и y, стоящих под дифференциалами.
Запишем данные матрицы:

Матрица жесткости:

$$b_{11}=\frac{1}{360h_xh_y}\begin{bmatrix}-h_x^{\,2}\lambda_9+12h_x^{\,2}\lambda_2+48h_x^{\,2}\lambda_5-4h_x^{\,2}\lambda_6+9h_x^{\,2}\lambda_7+36h_x^{\,2}\lambda_4+9h_x^{\,2}\lambda_1-h_x^{\,2}\lambda_3+12h_x^{\,2}\lambda_8+\\ 9h_y^{\,2}\lambda_1+h_y^{\,2}\lambda_3+36h_y^{\,2}\lambda_2-h_y^{\,2}\lambda_7+12h_y^{\,2}\lambda_6+12h_y^{\,2}\lambda_4+48h_y^{\,2}\lambda_5-4h_y^{\,2}\lambda_8-h_y^{\,2}\lambda_9\end{bmatrix}$$

Из-за очень больших размеров элементов матрицы жесткости, матрица не приведена. Сумма по строке в матрице жесткости равна 0.

Построим матрицу масс:

$$C:=\begin{bmatrix}\frac{1}{9}hyhx & \frac{1}{18}hyhx & \frac{1}{18}hyhx & \frac{1}{36}hyhx\\ \frac{1}{18}hyhx & \frac{1}{9}hyhx & \frac{1}{36}hyhx & \frac{1}{18}hyhx\\ \frac{1}{18}hyhx & \frac{1}{36}hyhx & \frac{1}{9}hyhx & \frac{1}{18}hyhx\\ \frac{1}{36}hyhx & \frac{1}{18}hyhx & \frac{1}{18}hyhx & \frac{1}{9}hyhx\end{bmatrix}$$

После построения матриц масс и жесткости остался не построенный локальный вектор правой части, построим его.

Функцию f представим в виде разложения по базисным функциям с весами f_{mk}, то есть вектор правой части представляет собой интеграл от произведения линейной комбинации базисных функций на одну из базисных функций. Тогда вектор F будет иметь следующую структуру:

$$F=\begin{bmatrix}F_1\\F_2\\F_3\\F_4\end{bmatrix}$$

и соответствующие элементы векторов F_1, F_2, F_3, F_4 находятся по формуле:

$$F_j=\sum_{k=1}^{4}f_k\int_x\int_y\psi_k\psi_j\,dxdy$$

2.3. Сборка глобальной матрицы

После нахождения локальных матриц и локального вектора правой части возникает задача сборки глобальной матрицы, займемся этой проблемой.

Поэтому локальные базисные функции пронумеруем в зависимости от ненулевого узла. Тогда в глобальной нумерации локальные функции конечного элемента имеют следующие номера, показанные на рисунке, где параметр n есть число узлов на горизонтальной линии всех конечных элементов прямоугольной области. Величина i определяется из номера конечного элемента в глобальной нумерации конечных элементов (не узлов).

Таким образом номера функций в глобальной нумерации однозначно получаются из нумерации функций в локальной нумерации путем формул:

$$\psi_1 = \Psi_i$$

$$\psi_2 = \Psi_{i+1}$$

…

$$\psi_{49} = \Psi_{i+6\,n+6}$$

После определения глобальной нумерации функций теперь можно написать матрицу индексов для глобальной матрицы, которая получается путем умножения вектора индексов локальных функции на этот же, но транспонированный, что соответствует локальному перебору локальных функций.

2.4. Сборка глобальной матрицы

Краевые условия первого рода будем учитывать с поправкой на то, что на границе, где указаны краевые условия функции ψ равные нулю. Реализовать данные условия можно путем внесения в глобальную матрицу на главную диагональ в соответствующий номер этой диагонали (номер узла в глобальной нумерации, для которого в данный момент учитывается краевое условие первого рода) ставится число $\sigma\square A_{ij}$, то есть число намного большее, чем элементы глобальной матрицы. В вектор правой части ставится число σU_g, которое соответствует произведению точного значения функции в данном узле на относительно большое число, что приводит нас к

решению обычного линейного уравнения, решением которого является значение функции в данном узле.

Краевые условия второго рода на рёбрах $S_{i,j}$ вносит вклад только в правую часть СЛАУ.

$$b^{S_{i,j}} = \frac{h_{i,j}}{6}\begin{pmatrix} 2\theta_1^{S_{i,j}} + \theta_2^{S_{i,j}} \\ \theta_1^{S_{i,j}} + 2\theta_2^{S_{i,j}} \end{pmatrix},$$

где i,j – координаты ребра и коэффициенты разложение параметра $\theta^{S_{i,j}}$ по линейному базису. Фактически эти коэффициенты равны значениям $\theta^{S_{i,j}}$ в узлах ребра.

При учете третьих краевых условий формируются локальная матрица и вектор правой части, которые заносятся в СЛАУ аналогично локальной матрице конечного элемента и локального вектора правой части конечного элемента.

$$A^{S_{i,j}} = \frac{\beta^{S_{i,j}} h_{i,j}}{6}\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$$

$$b^{S_{i,j}} = \frac{\beta^{S_{i,j}} h_{i,j}}{6}\begin{pmatrix} 2u_{\beta 1}^{S_{i,j}} + u_{\beta 2}^{S_{i,j}} \\ u_{\beta 1}^{S_{i,j}} + 2u_{\beta 2}^{S_{i,j}} \end{pmatrix},$$

где i,j – координаты ребра, β и коэффициенты разложения параметра $u^{S_{i,j}}{}_{\beta 1}$ по линейному базису. Фактически эти коэффициенты равны значениям $u^{S_{i,j}}{}_{\beta 1}$ в узлах ребра.

Для решения СЛАУ используется ЛОС с неполной факторизацией.

3. Входные данные

На вход программе подаются следующие входные файлы:

- Area.txt
- GridX.txt
- GridY.txt
- time.txt

Содержание файлов:

- Area.txt: [количество узлов по X] [количество узлов по Y] [Количество временных слоев + начальный]
- time.txt: массив узлов времени
- GridX.txt, GridY.txt: содержат массивы координат узлов.

4. Тестирование:

Тест №1:

$u = x + y + \cos(t)$, σ = 1, λ= 1, $t_j = 1$

В файле T.txt значения от 0 до 1 с шагом 0,1

X.txt = 1 2 3

Y.txt = 1 2 3

№ узла	U	U*	||U - U*||
1	2,5403023058611	2,54030230586814	6,9997E-12
2	3,5403023058621	3,54030230586814	6,0396E-12
3	4,54030230586832	4,54030230586814	1,7941E-13
4	3,54030230586854	3,54030230586814	4,0012E-13
5	4,54030230586842	4,54030230586814	2,7978E-13
6	5,54030230586120	5,54030230586814	6,9402E-12
7	4,54030230586821	4,54030230586814	7,0166E-14
8	5,54030230586121	5,54030230586814	6,9305E-12
9	6,54030230586321	6,54030230586814	4,9303E-12

Тест №2:

$u = x + y + \cos(t)$, σ = 1, λ= 1, $t_j = 1$

В файле T.txt значения от 0 до 1 с шагом 0,05

X.txt = 1 2 3

Y.txt = 1 2 3

№ узла	U	U*	||U - U*||
1	2,54030230586524	2,54030230586814	2,9E-12
2	3,54030230586511	3,54030230586814	3,03E-12
3	4,54030230586822	4,54030230586814	7,99E-14
4	3,54030230586832	3,54030230586814	1,8E-13
5	4,54030230586802	4,54030230586814	1,2E-13
6	5,54030230586868	5,54030230586814	5,4E-13
7	4,54030230586811	4,54030230586814	3,02E-14
8	5,54030230586792	5,54030230586814	2,2E-13
9	6,54030230586785	6,54030230586814	2,9E-13

Тест №3:

$u = x + y + \cos(t)$, σ = 1, λ= 1, $t_j = 1$

В файле T.txt значения от 0 до 1 с шагом 0,025

X.txt = 1 2 3

Y.txt = 1 2 3

№ узла	U	U*	||U - U*||
1	2,54030230586753	2,54030230586814	6,09734E-13
2	3,54030230586783	3,54030230586814	3,09974E-13
3	4,54030230586782	4,54030230586814	3,20632E-13
4	3,54030230586828	3,54030230586814	1,40332E-13
5	4,54030230586813	4,54030230586814	9,76996E-15
6	5,54030230586819	5,54030230586814	4,9738E-14
7	4,54030230586811	4,54030230586814	3,01981E-14
8	5,54030230586807	5,54030230586814	7,01661E-14
9	6,54030230586805	6,54030230586814	9,05942E-14

5. Вывод:

$k = log_2 \frac{\|u_h - u^*\|}{\left\|u_{\frac{h}{2}} - u^*\right\|}$ k- порядок сходимости

Для теста **1**, **2** k = 2,14861604323185

Для теста **2**, **3** k = 2,17979802272317

Порядок сходимости для 3-х слойной неявной схемы равен 2, что совпадает с теоретической величиной.

Порядок аппроксимации = 2

6. Приложение:

```
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <locale.h>
#include <memory.h>
#include <vector>
#include "math.h"

using namespace std;

#define STR 15
#define EPS 1e-15
#define PI 3.141592653589793
#define MAXITER 1000
#define MEMORY     200000
#define N_MAX 100

typedef double real;

real* global;
real* GridX;
real* GridY;
real* di;
real* ggl;
real* ggu;
real* r;
real* x;
real* f;
real* p;
real* z;
real* q;
real* L;
real* U;
real* diag;
real* s;
real* sout;
real  eps;
real delta_time;

int* ig;
int* jg;
int   N;
int    Nx;
int   Ny;
int count_t;
int stat_read = 0;

int    LU();
void  SetupDataMetrics(vector<real> prev_resh, real);
void  GetInfo();
void  ReadGrid();

void AddToMatrix(int, int, real);
real GetLambda(real, real);
real GetGamma(real, real);
real GetF(real, real, real);
real GetIdeal(real, real, real);
```

```
void GaussL(real*, real*);
void GaussU(real*, real*);
void MultMatrixOnVector(real*, real*);
real ScalarMult(real*, real*);
real MultLU(int, int);
void InitFunc1(real);
void Solver();
void Result(real);

ofstream output("Res.txt");


real GetLambda(real x, real y) {
	return 1.;
}

real GetGamma(real x, real y) {
	return 1;
}

real GetF(real x, real y, real t) {
	return -12 * x * x - 12 * y * y + pow(t, 2) * GetLambda(x, y) - 2 * t *
GetGamma(x, y);
}

real GetIdeal(real x, real y, real t) {
	return pow(x, 4) + pow(y, 4) + pow(t, 2);
}


void InitFunc1(vector<real>& a, real time) {
	int i, k;
	global = new double[MEMORY];
	memset(global, 0, MEMORY * sizeof(double));
	GetInfo();
	ig = (int*)global;
	jg = (int*)(global + N + 1);

	int istep = 0;
	for (i = 0; i < N + 1; i++) {
		ig[i] = istep;
		istep += i;
	}

	istep = 0;
	for (i = 0; i < N; i++)
		for (k = 0; k < i; k++) {
			jg[istep] = k;
			istep++;
		}

	ggl = global + N + 1 + ig[N];
	ggu = global + 2 * (ig[N]) + N + 1;
	di = global + 3 * (ig[N]) + N + 1;

	f = di + N;
	r = f + N;
	z = r + N;
	p = z + N;
	q = p + N;
	diag = q + N;
```

```
	L = diag + N;
	U = L + ig[N];
	x = U + ig[N];
	s = x + N;
	sout = s + N;
	GridX = sout + N;
	GridY = GridX + Nx;

	ReadGrid();

	for (int i = 0; i < Ny; i++)
		for (int j = 0; j < Nx; j++)
			a.push_back(GetIdeal(GridX[j], GridY[i], time));

	SetupDataMetrics(a, time);
}

void ReinitFunc(vector<real> prev_resh, real time) {
	int i, k;
	memset(global, 0, MEMORY * sizeof(double));

	ig = (int*)global;
	jg = (int*)(global + N + 1);

	int istep = 0;
	for (i = 0; i < N + 1; i++) {
		ig[i] = istep;
		istep += i;
	}

	istep = 0;
	for (i = 0; i < N; i++)
		for (k = 0; k < i; k++) {
			jg[istep] = k;
			istep++;
		}

	ggl = global + N + 1 + ig[N];
	ggu = global + 2 * (ig[N]) + N + 1;
	di = global + 3 * (ig[N]) + N + 1;

	f = di + N;
	r = f + N;
	z = r + N;
	p = z + N;
	q = p + N;
	diag = q + N;
	L = diag + N;
	U = L + ig[N];
	x = U + ig[N];
	s = x + N;
	sout = s + N;
	GridX = sout + N;
	GridY = GridX + Nx;

	ReadGrid();
	SetupDataMetrics(prev_resh, time);
}

void GetInfo() {
	ifstream file("Area.txt");
```

```
	file >> Nx >> Ny >> count_t >> delta_time;
	N = Nx * Ny;
	file.close();
}

void ReadGrid() {
	int i;

	ifstream fileX("GridX.txt");
	for (i = 0; i < Nx; i++)
		fileX >> GridX[i];
	fileX.close();

	ifstream fileY("GridY.txt");
	for (i = 0; i < Ny; i++)
		fileY >> GridY[i];
	fileY.close();
}

void SetupDataMetrics(vector<real> prev_resh, real time) {
	int i, k, i1, k1;

	int	Index[4];
	real	lambda, gamma, px, y, xp, yp, hx, hy;
	real	tmp, hx2, hy2, ud, u1, u2, u3;

	real	fv[4];
	real	B[4][4];
	real	C[4][4];
	real	F[4];

	for (k = 0; k < Ny - 1; k++)
		for (i = 0; i < Nx - 1; i++)	{
			px = GridX[i];
			y = GridY[k];

			xp = GridX[i + 1];
			yp = GridY[k + 1];

			hx = xp - px;
			hy = yp - y;
			hx2 = hx * hx;
			hy2 = hy * hy;

			lambda = GetLambda(px + hx / 2.0, y + hy / 2.0);
			gamma = GetGamma(px + hx / 2.0, y + hy / 2.0);

			fv[0] = GetF(px, y, time);
			fv[1] = GetF(xp, y, time);
			fv[2] = GetF(px, yp, time);
			fv[3] = GetF(xp, yp, time);

			tmp = 1.0 / (hx * hy);

			ud = (hx2 + hy2) * tmp / 3;
			u1 = (hx2 - 2 * hy2) * tmp / 6;
			u2 = -(2 * hx2 - hy2) * tmp / 6;
			u3 = -(hx2 + hy2) * tmp / 6;


			B[0][0] = ud;
```

```
B[0][1] = u1;
B[0][2] = u2;
B[0][3] = u3;

B[1][0] = u1;
B[1][1] = ud;
B[1][2] = u3;
B[1][3] = u2;

B[2][0] = u2;
B[2][1] = u3;
B[2][2] = ud;
B[2][3] = u1;

B[3][0] = u3;
B[3][1] = u2;
B[3][2] = u1;
B[3][3] = ud;

ud = hx * hy / 9.0;
u1 = hx * hy / 18.0;
u2 = u1;
u3 = hx * hy / 36.0;

C[0][0] = ud;
C[0][1] = u1;
C[0][2] = u2;
C[0][3] = u3;

C[1][0] = u1;
C[1][1] = ud;
C[1][2] = u3;
C[1][3] = u2;

C[2][0] = u2;
C[2][1] = u3;
C[2][2] = ud;
C[2][3] = u1;

C[3][0] = u3;
C[3][1] = u2;
C[3][2] = u1;
C[3][3] = ud;

F[0] = C[0][0] * fv[0] + C[0][1] * fv[1] +
       C[0][2] * fv[2] + C[0][3] * fv[3];
F[1] = C[1][0] * fv[0] + C[1][1] * fv[1] +
       C[1][2] * fv[2] + C[1][3] * fv[3];
F[2] = C[2][0] * fv[0] + C[2][1] * fv[1] +
       C[2][2] * fv[2] + C[2][3] * fv[3];
F[3] = C[3][0] * fv[0] + C[3][1] * fv[1] +
       C[3][2] * fv[2] + C[3][3] * fv[3];


Index[0] = Nx * k + i;
Index[1] = Nx * k + i + 1;
Index[2] = Nx * (k + 1) + i;
Index[3] = Nx * (k + 1) + i + 1;

for (i1 = 0; i1 < 4; i1++)      {
```

```
                        for (k1 = 0; k1 < 4; k1++)
                              AddToMatrix(Index[i1], Index[k1], lambda * B[i1][k1]
+ gamma * C[i1][k1] / delta_time);

                        real val = 0;
                        for (int z = 0; z < 4; z++)
                              val += C[i1][z] * prev_resh[z];
                        f[Index[i1]] += F[i1] + 1 / delta_time * val;
                  }
            }

      for (i = 0; i < Nx; i++)
      {
            px = GridX[i];
            y = GridY[0];
            di[i] = 1.0e+50;
            f[i] = 1.0e+50 * GetIdeal(px, y, time);
            y = GridY[Ny - 1];
            di[Nx * (Ny - 1) + i] = 1.0e+50;
            f[Nx * (Ny - 1) + i] = 1.0e+50 * GetIdeal(px, y, time);
      }

      for (k = 0; k < Ny; k++)
      {
            y = GridY[k];
            px = GridX[0];
            di[k * Nx] = 1.0e+50;
            f[k * Nx] = 1.0e+50 * GetIdeal(px, y, time);
            px = GridX[Nx - 1];
            di[(k + 1) * Nx - 1] = 1.0e+50;
            f[(k + 1) * Nx - 1] = 1.0e+50 * GetIdeal(px, y, time);
      }
}

void AddToMatrix(int i, int j, real el)
{
      int k;
      if (i == j) di[i] += el;
      else  {
            if (i > j)  {
                  for (k = ig[i]; k < ig[i + 1]; k++)
                        if (jg[k] == j) ggl[k] += el;
            }
            else  {
                  for (k = ig[j]; k < ig[j + 1]; k++)
                        if (jg[k] == i) ggu[k] += el;
            }
      }
}


int LU()
{
      int i, j;
      for (i = 0; i < N; i++)  {
            for (j = ig[i]; j < ig[i + 1]; j++)   {
                  L[j] = (ggl[j] - MultLU(i, jg[j]));
                  U[j] = (ggu[j] - MultLU(jg[j], i)) / diag[jg[j]];
            }
            diag[i] = di[i] - MultLU(i, i);
      }
```

```
	return 0;
}

void GaussL(real* in, real* out)
{
	int i, j;
	real result;
	for (i = 0; i < N; i++)
	{
		result = 0;
		for (j = ig[i]; j < ig[i + 1]; j++)
		{
			result += L[j] * out[jg[j]];
		}
		out[i] = (in[i] - result) / diag[i];
	}
}

void GaussU(real* in, real* out)
{
	int i, j;
	for (i = 0; i < N; i++) out[i] = in[i];
	for (i = N - 1; i >= 0; i--)
	{
		for (j = ig[i]; j < ig[i + 1]; j++)
		{
			out[jg[j]] -= U[j] * out[i];
		}
	}
}

void MultMatrixOnVector(real* in, real* out)
{
	int i, j;
	real* out1;
	out1 = new real[N];
	for (i = 0; i < N; i++)
	{
		out1[i] = di[i] * in[i];
		for (j = ig[i]; j < ig[i + 1]; j++)
		{
			out1[i] += ggl[j] * in[jg[j]];
			out1[jg[j]] += ggu[j] * in[i];
		}
	}

	for (i = 0; i < N; i++)
		out[i] = out1[i];
	delete[] out1;
}

real ScalarMult(real* v1, real* v2) {
	int i;
	real result;
	result = 0;
	for (i = 0; i < N; i++) {
		result += v1[i] * v2[i];
	}
	return result;
}
```

```
real MultLU(int i, int j)
{
	int k, l, find;
	real result;
	result = 0.0;
	if (i == j)  {
		for (k = ig[i]; k < ig[i + 1]; k++)
			result += U[k] * L[k];
	}
	else  {

		if (i > j)   {
			for (k = ig[j]; k < ig[j + 1]; k++)    {
				find = 0;
				for (l = ig[i]; l < ig[i + 1] && find == 0; l++){
					if (jg[l] == jg[k]){
						result += U[k] * L[l];
						find = 1;
					}
				}
			}
		}

		else  {
			for (l = ig[i]; l < ig[i + 1]; l++)    {
				find = 0;
				for (k = ig[j]; k < ig[j + 1] && find == 0; k++)    {
					if (jg[l] == jg[k]){
						result += U[k] * L[l];
						find = 1;
					}
				}
			}
		}

	}
	return result;
}

void Solver()
{
	int iter;
	int i, check, stop;
	real alpha, alphazn, alphach, beta, betach, betazn, CheckExit;

	check = LU();
	if (check != 0) cout << "Cant do factorization" << check + 1 << endl;

	stop = 0;
	for (i = 0; i < N; i++) x[i] = 0;
	GaussL(f, r);
	GaussU(r, z);
	MultMatrixOnVector(z, q);
	GaussL(q, p);

	for (iter = 0; iter < MAXITER && stop == 0; iter++)
	{
		alphach = ScalarMult(p, r);
		alphazn = ScalarMult(p, p);
		alpha = alphach / alphazn;
		for (i = 0; i < N; i++) x[i] += alpha * z[i];
```

```
            for (i = 0; i < N; i++) r[i] -= alpha * p[i];
            GaussU(r, s);
            MultMatrixOnVector(s, sout);
            GaussL(sout, q);
            betazn = ScalarMult(p, p);
            betach = ScalarMult(p, q);
            beta = -betach / betazn;
            for (i = 0; i < N; i++) z[i] = beta * z[i] + s[i];
            for (i = 0; i < N; i++) p[i] = beta * p[i] + q[i];
            CheckExit = ScalarMult(r, r);
            if (CheckExit < EPS) stop = 1;
        }
}

void Result(real time)
{
        int i, k, num = 0;
        real px, y, func, res = 0.0, norm = 0.0, tmp;


        output << "X" << setw(STR + 8) << "Y" << setw(STR + 8)
                << "U" << setw(STR + 8) << "U*" << setw(STR + 8) << "U* - U" << endl;

        for (k = 0; k < Ny; k++)
                for (i = 0; i < Nx; i++)
                {
                        px = GridX[i];
                        y = GridY[k];

                        func = GetIdeal(px, y, time);
                        tmp = fabs(x[num] - func);

                        res += tmp * tmp;
                        norm += func * func;

                        output << setprecision(STR) << px
                                << setw(STR + 8) << y
                                << setw(STR + 8) << x[num]
                                << setw(STR + 8) << func;
                        output << setw(STR + 8) << tmp << endl;
                        num++;
                }
        output << "\n ||U-U*|| / ||U*|| = " << scientific << sqrt(res / norm) << endl;

}

int main() {
        vector <vector<real>> solutionVector(N_MAX);
        InitFunc1(solutionVector[0], 0);

        for (int i = 1; i < count_t; i++) {
                Solver();
                Result(i * delta_time);
                for (int j = 0; j < N; j++)
                        solutionVector[i].push_back(x[j]);
                ReinitFunc(solutionVector[i - 1], i * delta_time);
        }

        output.close();
        return 0;
}
```