Московский государственный университет имени М.В. Ломоносова

Казахстанский филиал

Факультет вычислительной математики и кибернетики

Отчет по практикуму

Задача Дирихле для уравнения Пуассона

Составил: Шмаль А.С.

Проверил: Нетесов В.В.

Нур-Султан 2021

Содержание

1 Постановка задачи

В квадрате $G = \{0 \le x \le 1, 0 \le y \le 1\}$ с границей Γ требуется найти решение уравнения Пуассона

$$Lu = -f(x,y),\ (x,y) \in G, \tag{1}$$

Где L – оператор Лапласа:

$$Lu = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \tag{2}$$

1.1 Построение равномерной сетки

Разобьем отрезок $[0,1]$ на n равных частей. Обозначим $h = 1/n,\ x_i = ih, y_j = jh$, $0 \le i \le n,\ 0 \le j \le n$. Построим сетку узлов

$$\bar{\omega}_h = \{(x_i, y_j). \quad 0 \le i \le n, \quad 0 \le j \le n\}$$

Узлы $(x_i, y_j), \quad 1 \le i \le n-1, \quad 1 \le j \le n-1$ - внутренние, остальные, лежащие на границе квадрата, – граничные.

1.2 Разностная аппроксимация задачи Дирихле

Обозначим $u_{ij} \approx u(x_i, y_j)$. Заменяем оператор L во всех внутренних узлах разностным оператором L_h

$$L_h u_{ij} = \frac{u_{i+1j} + u_{i-1j} + u_{ij+1} + u_{ij-1} - 4u_{ij}}{h^2}, \tag{4}$$

$1 \le i \le n-1; \quad 1 \le j \le n-1$.

Если $u(x,y)$ имеет не менее четырех непрерывных ограниченных в рассматриваемой области производных по x и по y, то разностный оператор L_h аппроксимирует дифференциальный L со вторым порядком, т.е.

$$Lu - L_h u = O(|h|^2).$$

Учитывая соотношение (4), задаче (1), (2) ставим в соответствие разностную задачу: найти сеточную функцию, удовлетворяющую во внутренних узлах уравнениям

$$4u_{ij} - u_{i-1j} - u_{i+1j} - u_{ij-1} - u_{ij+1} = h^2 f_{ij}, \tag{5}$$

$$f_{ij} = f(x_i, y_j),\ 1 \le i \le n-1; \quad 1 \le j \le n-1$$

и принимающую в граничных узлах заданные значения

$$\begin{cases} u_{i0} = \mu(x_i, 0), & 0 \le i \le n \\ u_{in} = \mu(x_i, 1), & 0 \le i \le n \\ u_{0j} = \mu(0, y_j), & 1 \le j \le n-1 \\ u_{nj} = \mu(1, y_j), & 1 \le j \le n-1 \end{cases}$$

Запишем n уравнений, представленных формулой (5),одним матричным уравнением, объединяя неизвестные u_{ij} в длинном векторе размерности $(n-1)^2$. В частности, при $n = 4$ получаем вектор-столбец $u = [u_1, u_2, ..., u_9]$. При аналогичной нумерации правых частей f_{ij} уравнения (5) преобразуются к виду

$$\left[\begin{array}{ccc|ccc|ccc} 4 & -1 & & -1 & & & & & \\ -1 & 4 & -1 & & -1 & & & & \\ & -1 & 4 & & & -1 & & & \\ \hline -1 & & & 4 & -1 & & -1 & & \\ & -1 & & -1 & 4 & -1 & & -1 & \\ & & -1 & & -1 & 4 & & & -1 \\ \hline & & & -1 & & & 4 & -1 & \\ & & & & -1 & & -1 & 4 & -1 \\ & & & & & -1 & & -1 & 4 \end{array}\right] \left[\begin{array}{c} u_1 \\ u_2 \\ . \\ . \\ . \\ . \\ . \\ . \\ u_9 \end{array}\right] = h^2 \left[\begin{array}{c} f_1 \\ f_2 \\ . \\ . \\ . \\ . \\ . \\ . \\ f_9 \end{array}\right] \tag{6}$$

Минус единицы рядом с главной диагональю соответствуют вычитанию верхнего и нижнего соседей $-u_{ij-1} - u_{ij+1}$. Минус единицы, удаленные от главной диагонали, соответствуют вычитанию левого и правого соседей $-u_{i-1j} - u_{i+1j}$.

При этом компоненты вектора правой части вычисляются по формулам

$$f_i = \left(f_{i1} + \frac{\mu_{i0}}{h^2}, f_{i2}, f_{i3}, ..., f_{in-2}, f_{in-1} + \frac{\mu_{in}}{h^2}\right)^T,\ 1 \le i \le n-1 \tag{7}$$

Для решения системы воспользуемся следующими методами:

1) Метод матричной прогонки.

2) Метод сопряженных градиентов.

1.3 Расчетные формулы для методов

1.3.1 Метод матричной прогонки

Матричная прогонка относится к прямым методам решения разностныхуравнений. Она применяется к уравнениям, которые можно записать в виде системы векторных уравнений

$$-C_0 y_0 + B_0 y_1 = -F_0,$$

$$A_i y_{i-1} - C y_i + B_i y_{i+1} = -F_i,\ 1 \le i \le n-1,$$

$$A_n y_{n-1} - C_n y_n = -F_n,$$

в нашей задаче y_i – искомые векторы размерности $n-1$, f_i – заданные векторы, A_i, B_i, C_i – заданные квадратные матрицы порядка $n-1$. Решение системы ищут в виде

$$y_i = \alpha_{i+1} y_{i+1} + \beta_{i+1},\ i = n-1, n-2, ..., 0,$$

при этом $y_n = \beta_{n+1}$. В нашей задаче матрицы A_i и B_i совпадают и представляют собой блоки на сопутствующих диагоналях блочной матрицы (6), а C_i – диагональный блок той же матрицы. В этом случае расчетные формулы для прогоночных коэффициентов имеют вид

$$\alpha_{i+1} = (C_i - \alpha_i)^{-1},\ i = 1, 2, ...n-1,\ \alpha_1 = C_0^{-1},$$

$$\beta_{i+1} = \alpha_{i+1}(\beta_i + f_{i+1}),\ i = 1, 2, ...n,\ \beta_1 = \alpha_1 f_1.$$

Реализация метода на языке c++(dm есть тип Matrix<double>):

```
dm matr_sweep(const dm& c, const dm& f) {
        int n = c.row();
        vector<dm> alph;
        dm x(n), bet(n);
        alph.push_back(inv(c));
        bet[0] = alph[0] * f[0];
        for (int i = 0; i < n - 1; i++) {
                alph.push_back(inv(c - alph[i]));
                bet[i + 1] = alph[i + 1];
                bet[i+1] *= (bet[i] + f[i + 1]);
        }
        x[n - 1] = bet[n - 1];
        for (int i = n - 2; i >= 0; i--) {
                x[i] = alph[i] * x[i + 1] + bet[i];
        }
        return x;
}
```

1.3.2 Метод сопряженных градиентов

Рассмотрим следующую задачу оптимизации: $F(x) = \frac{1}{2}(Ax, x) - (f, x) \longrightarrow inf$, $x \in R^n$. Здесь A – симметричная положительно определённая матрица размера $n \times n$. Такая задача оптимизации называется квадратичной. Заметим, что $F'(x) = Ax - f$. Условие экстремума функции $F'(x) = 0$ эквивалентно системе $Ax - f = 0$. Функция F достигает своей нижней грани в единственной точке x_*, определяемой уравнением $Ax_* = f$. Таким образом, данная задача оптимизации сводится к решению системы линейных уравнений $Ax = f$.

Идея метода сопряжённых градиентов состоит в следующем: Пусть $\{p_k\}_{k=1}^n$ – базис

в R^n. Тогда для любой точки $x_0 \in R^n$ вектор $x_* - x_0$ раскладывается по базису $x_* - x_0 = \nu_1 p_1 + \ldots \nu_n p_n$. Таким образом, x_* представимо в виде $x_* = x_0 + \nu_1 p_1 + \ldots \nu_n p_n$. Каждое следующее приближение вычисляется по формуле: $x_k = x_0 + \nu_1 p_1 + \ldots \nu_k p_k$. Опишем способ построения базиса $\{p_k\}_{k=1}^n$ в методе сопряжённых градиентов. В качестве начального приближения x_0 выбираем произвольный вектор. На каждой итерации ν_k выбирается по правилу:

$$\nu_k = \underset{\nu_k}{argmin} F(x_{k-1} + \nu_k p_k).$$

Базисные вектора p_k вычисляются по формулам:

$$p_1 = -F'(x_0),$$

$$p_{k+1} = -F'(x_k) + \mu_k p_k.$$

Коэффициент μ_k выбирается так, чтобы векторы p_k и p_{k+1} были сопряжёнными относительно A (собственно, отсюда вытекает название самого метода):

$$\mu_k = \frac{(F'(x_k), Ap_k)}{(Ap_k, p_k)}.$$

Если обозначить за $r_k = f - Ax_k = -F'(x_k)$, то после нескольких упрощений получим окончательные формулы, используемые при применении метода сопряжённых градиентов на практике:

$$r_1 = f - Ax_0,$$

$$p_1 = r_1,$$

$$z = Ap_k,$$

$$\nu_k = \frac{(r_k, r_k)}{(z, p_k)},$$

$$x_{k+1} = x_k + \nu_k p_k,$$

$$r_{k+1} = r_k - \nu_k z,$$

$$\mu_k = \frac{(r_{k+1}, r_{k+1})}{(r_k, r_k)},$$

$$p_{k+1} = r_{k+1} + \mu_k p_k.$$

Вычисления прекращаются, как только число $|||r_k||_2/||r_1||_2$ становится достаточно малым, либо, не достигнув нужной точности, проводим вычисления по изначально заданному числу итераций. В данном алгоритме z, p_k, r_k, x_k – векторы, но в нашей задаче мы "сворачиваем" их в матрицы. A – та же матрица, имеющая вид (6), а f – набор тех же векторов, подчиненных формулам (7).

Реализация метода на языке c++(dm есть тип Matrix<double>, при реализации использовался тип возвращаемого объекта **tuple** для возврата функцией с помощью метода **make_tuple** четырех данных: признак достижения точности, достигнутой точности, число итераций, матрица полученных решений):

```
tuple<int, double, int, dm>
CG(const dm& f, double eps, int max_iter) {
        int n = f.row();
        dm z(n), x(n), p(n), r(n), matr;
        p = r = f;
        double sr1 = matr.scalar(r, r);
        double nf = sqrt(sr1);
        if (!nf) nf = 1.0;
        double res = 1.0; //  res = |r| / nf
        if (res <= eps) {
                eps = res;
                max_iter = 0;
                return make_tuple(0, eps, max_iter, x);
        }
        for (int k = 1; k < max_iter; k++) {
                z = Ax(p);
                double v = sr1 / matr.scalar(p, z);
                x += p * v;
                r -= z * v;
```

```
                double sr2 = matr.scalar(r, r);
                res = sqrt(sr2) / nf;
                if (res <= eps) {
                        eps = res;
                        max_iter = k;
                        return
                        make_tuple(0,eps,max_iter,x);
                }
                double m = sr2 / sr1;
                p = r + p * m;
                sr1 = sr2;
        }
        eps = res;
        return make_tuple(1, eps, max_iter, x);
}
```

2 Листинги программ

Воспользуемся тем же классом матриц, реализованном в файле "Matrix.h", а также функцией **linspace**, которые были описаны в первом отчете. Дополним лишь тот класс следующими операторами перегрузки:

```
template<class F>
Matrix<F> operator+(const Matrix<F>& A,const Matrix<F>& B){
        try {
                if (A.rows != B.rows || A.cols != B.cols){
                        throw "Error!";
                }
        }
        catch (const char* str) {cout << str << endl;}
        Matrix<F> Res(A.rows, A.cols);
        for (int i = 0; i < Res.rows; i++) {
                for (int j = 0; j < Res.cols; j++) {
                        Res(i, j) = A(i, j) + B(i, j);
                }
        }
        return Res;
}
template<class F>
vector<F> operator*(const Matrix<F>& A,const vector<F>& v){
        try {
                if (A.cols != v.size()) throw "Error!";
        }
        catch (const char* str) {cout << str << endl;}
        vector<F> res(A.rows);
        for (int i = 0; i < A.rows; i++) {
                        res[i] = 0;
                        for (int j = 0; j < A.cols; j++) {
                                res[i] += A(i, j) * v[j];
                        }
        }
```

```
        return res;
}
template<class F>
Matrix<F>operator*(const Matrix<F>& A, const F& a) {
        Matrix<F> Res(A.rows, A.cols);
        for (int i = 0; i < Res.rows; i++) {
                for (int j = 0; j < Res.cols; j++) {
                        Res(i, j) = A(i, j) * a;
                }
        }
        return Res;
}
template<class T>Matrix<T>
Matrix<T>::operator+=(const Matrix& B) {
        try {
                if (rows != B.rows || cols != B.cols) {
                        throw "Error!";
                }
        }
        catch (const char* str) {cout << str << endl;}
        for (int i = 0; i < rows; i++) {
                for (int j = 0; j < cols; j++) {
                        matrix[i][j] += B(i, j);
                }
        }
        return *this;
}
```

```
template<class T>Matrix<T>
Matrix<T>::operator-=(const Matrix& B) {
        try {
                if (rows != B.rows || cols != B.cols) {
                        throw "Error!";
                }
        }
        catch (const char* str) {cout << str << endl;}
        for (int i = 0; i < rows; i++) {
                for (int j = 0; j < cols; j++) {
                        matrix[i][j] -= B(i, j);
                }
        }
        return *this;
}
template<class T>
vector<T>& Matrix<T>::operator[](int i) {
        return matrix[i];
}
template<class T>
const vector<T>& Matrix<T>::operator[](int i) const {
        return matrix[i];
}
```

Также реализуем вспомогательные фунции. Для метода матричной прогонки функцию **inv** обращения матриц, а для метода сопряженных градиентов функцию **Ax** формирования матрицы z.

```
dm inv(dm a) {
        int n = a.row();
        dm e(n), x(n);
        for (int i = 0; i < n; i++) e(i, i) = 1;
        double eps = 1.0e-100;
        for (int k = 0; k < n - 1; k++) {
                double max = fabs(a(k, k));
                int pos = k;
                for (int t = k + 1; t < n; t++) {
                        if (fabs(a(t, k)) > max) {
                                max = fabs(a(t, k));
                                pos = t;
                        }
                }
                try {
                        if (max < eps) throw "Error!";
                }
                catch (const char* str){cout<< str <<endl;}
                if (pos != k) {
                        for (int j = k; j < n; j++) {
                                swap(a(pos, j), a(k, j));
                        }
                        for (int s = 0; s < n; s++) {
                                swap(e(pos, s), e(k, s));
                        }
                }
```

```
                for (int i = k + 1; i < n; i++) {
                        double tmp = a(i, k) / a(k, k);
                        a(i, k) = 0;
                        for (int j = k + 1; j < n; j++) {
                                a(i, j) -= tmp * a(k, j);
                        }
                        for (int s = 0; s < n; s++) {
                                e(i, s) -= tmp * e(k, s);
                        }
                }
        }
        for (int k = 0; k < n; k++) {
                for (int i = n - 1; i >= 0; i--) {
                        x(i, k) = e(i, k);
                        double sum = 0 ;
                        for (int j = n - 1; j > i; j--) {
                                sum += a(i, j) * x(j, k);
                        }
                        x(i, k) = (x(i, k) - sum)/a(i, i);
                }
        }
        return x;
}
```

```
dm Ax(const dm& x) {
        int n = x.row(); dm matr(n);
        for (int i = 0; i < n; i++) {
                matr(0, i) = 4 * x(0, i) - x(1, i);
                if (i > 0) matr(0, i) -= x(0, i - 1);
                if (i < n - 1) matr(0, i) -= x(0, i + 1);
        }
        for (int j = 1; j < n - 1; j++) {
                for (int i = 0; i < n; i++) {
                        matr(j, i) = 4 * x(j, i);
                        matr(j, i) -= x(j - 1, i);
                        matr(j, i) -= x(j + 1, i);
                        if (i > 0) {
                                matr(j, i) -= x(j, i - 1);
                        }
                        if (i < n - 1) {
                                matr(j, i) -= x(j, i + 1);
                        }
                }
        }
        for (int i = 0; i < n; i++) {
                matr(n - 1, i) = 4 * x(n - 1, i);
                matr(n - 1, i) -= x(n - 2, i);
                if (i > 0) {
                        matr(n - 1, i) -= x(n - 1, i - 1);
                }
```

```
                if (i < n - 1) {
                        matr(n - 1, i) -= x(n - 1, i + 1);
                }
        }
        return matr;
}
```

Пользуясь вышереализованными "инструментами", программой **gnuplot**, а также программой на языке Си из первого отчета, реализуем уже саму программу:

```
double u_a(double x, double y) {
        return //some function;
}
double fun(double x, double y) {
        return //some function;
}
double north(double x) {return u_a(x, 1.0);}
double south(double x) {return u_a(x, 0.0);}
double east(double y) {return u_a(1.0, y);}
double west(double y) {return u_a(0.0, y);}

typedef Matrix<double> dm;
typedef vector<double> dv;
```

```
int main(void) {
        int n;
        cout << "Number of nodes ";
        cin >> n;
        int nm1 = n - 1;
        int nm2 = n - 2;
        int np1 = n + 1;
        double h = 1.0 / n;
        double h2 = h * h;
        dv x = linspace(0.0, 1.0, np1);
        dv y = x;
        dm f(nm1), u(nm1), ua(nm1), c(nm1);
        for (int j = 0; j < nm1; j++) {
                for (int i = 0; i < nm1; i++) {
                        f(j, i) += fun(x[i + 1],y[j + 1]);
                        f(j, i) *= h2;
                        if (!j) f(0, i) += south(x[i + 1]);
                        else if (j == nm2) {
                                f(nm2, i) += north(x[i+1]);
                        }
                }
                f(j, 0) += west(y[j + 1]);
                f(j, nm2) += east(y[j + 1]);
        }
        int opt, flag, iter, max_iter = 100;
        double eps = 1.e-3;
        cout << "Number of method ";
        cin >> opt;
```

```
auto start = high_resolution_clock::now();
switch (opt) {
        case 1:
                for (int i = 0; i < nm1; i++) {
                        c(i, i) = 4.0;
                        if (i > 0) {
                                c(i - 1, i) = -1.0;
                        }
                        if (i < nm1 - 1) {
                                c(i + 1, i) = -1.0;
                        }
                }
                u = matr_sweep(c, f);
                break;
        case 2:
                tie(flag, eps, iter, u) =
                                CG(f, eps, max_iter);
                break;
        default:
                cout << "Error!" << endl;
                return 1;
}
auto stop = high_resolution_clock::now();
auto d = duration_cast<microseconds>(stop - start);
d *= 1.0e-6;
cout << d.count() << "␣sec" << endl;
double dlt = 0.0;
double I, J;
```

```
        for (int j = 0; j < nm1; j++) {
                for (int i = 0; i < nm1; i++) {
                        ua(j, i) = u_a(x[i + 1],y[j + 1]);
                        double t = fabs(u(j,i) - ua(j,i));
                        if (t > dlt) {
                                I = i; J = j; dlt = t;
                        }
                }
        }
        if (opt == 2) {
                cout << "flag = " << flag;
                cout << ", eps = " << eps;
                cout<<", number of iterations = " << iter;
                cout << endl << endl;
                cout << "dlt = " << dlt << ", x[i]= ";
                cout <<  x[I] << ", y[j]= " << y[J];
                cout << endl << endl;
                ofstream ofs("data.txt");
        for (int j = 0; j < nm1; j++) {
                for (int i = 0; i < nm1; i++) {
                        ofs << x[i + 1] << ' ' << y[j + 1];
                        ofs << ' ' << u(j, i) << endl;
                }
        }
        if (execlp("./plot", "./plot", NULL) < 0) {
                perror( "failed"); return 1;
        }
        ofs.close(); return 0;
}
```

3 Результаты расчетов

Метод матричной прогонки.

Функция: $u(x, y) = sin(\pi x)cos(\pi y)$

Lu: $-2\pi^2 sin(\pi x)cos(\pi y)$

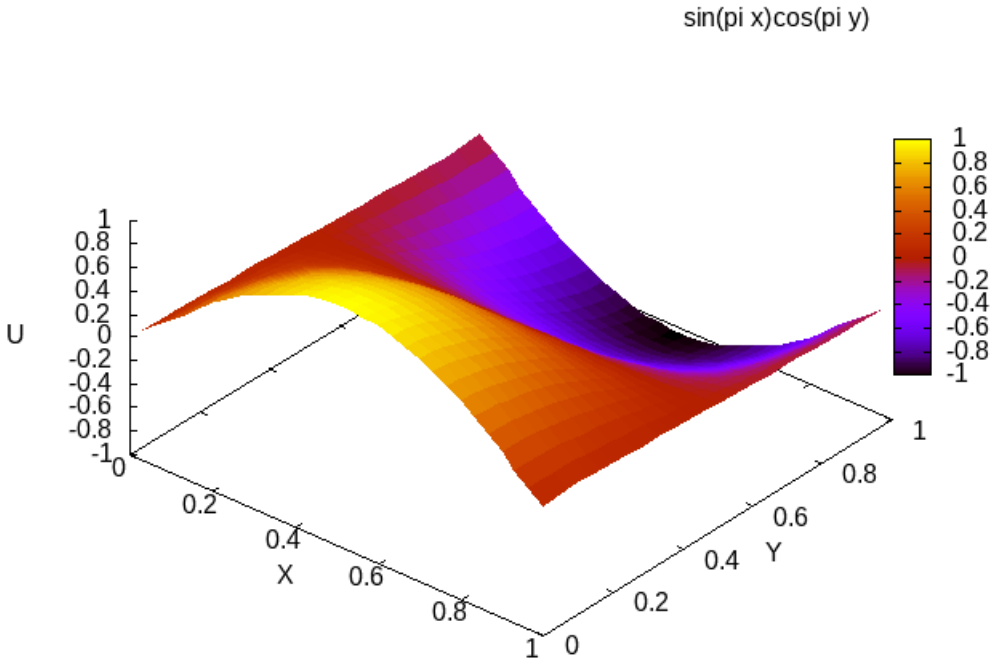
Число узлов: 64

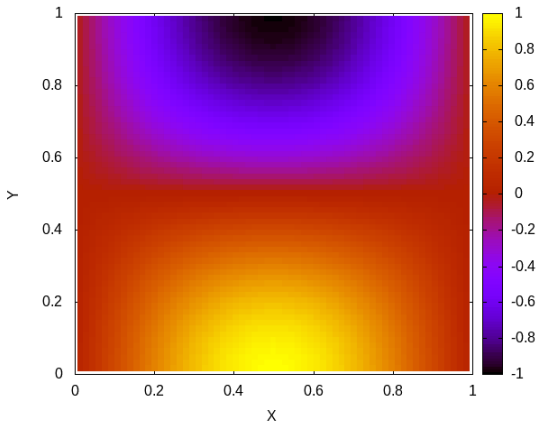
Максимальная погрешность и в какой точке она достигается:

dlt = 6.7693e-05, x[i]= 0.484375, y[j]= 0.203125

Время выполнения: 0.251187 секунд

Графики функции:





Метод сопряженных градиентов.

Точность: 10^{-5}

Функция: $u(x, y) = sin(\pi x)cos(\pi y)$

Lu: $-2\pi^2 sin(\pi x)cos(\pi y)$

Число узлов: 64

Признак достижения точности, точность и пройденное число итераций:

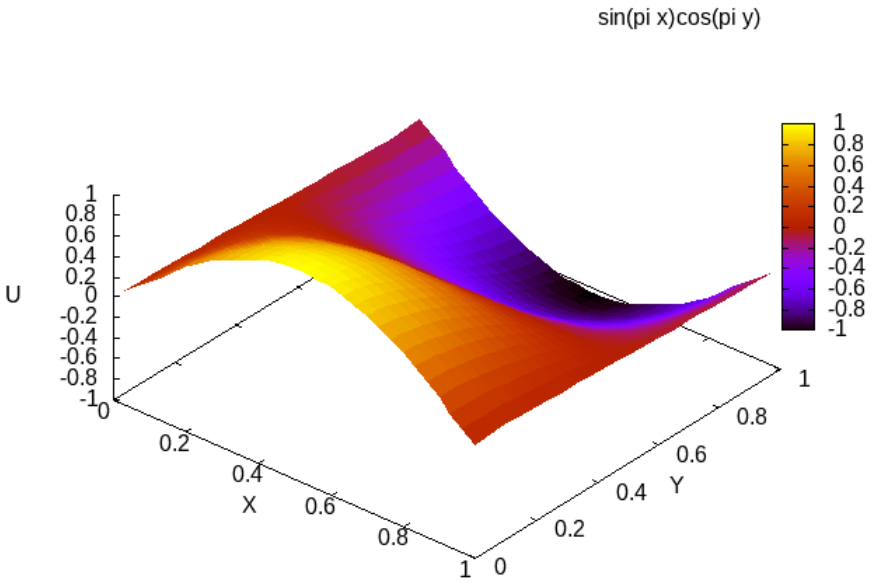
flag = 0, eps = 5.87775e-06, число итераций = 38

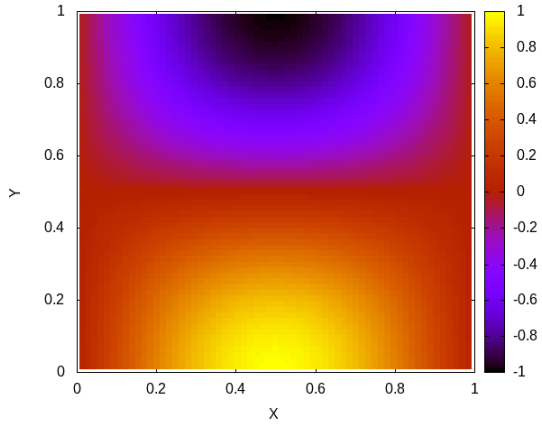
Максимальная погрешность и в какой точке она достигается:

dlt = 6.785e-05, x[i]= 0.484375, y[j]= 0.203125

Время выполнения: 0.038864 секунд

Графики функции:





Метод матричной прогонки.

Функция: $u(x, y) = yx^3 + xy^2$

Lu: $2x(3y + 1)$

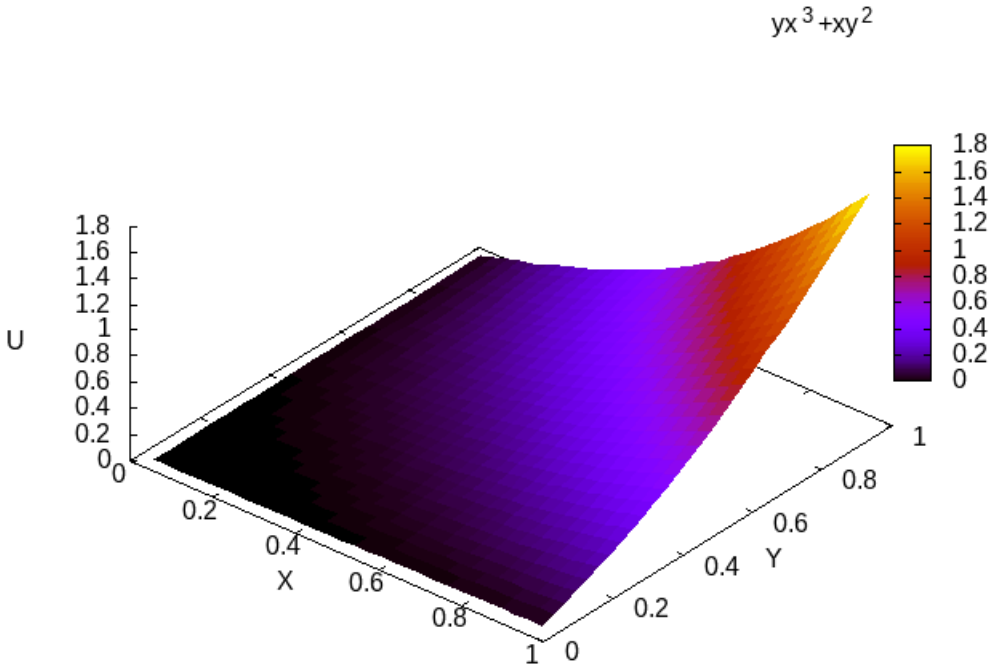
Число узлов: 32

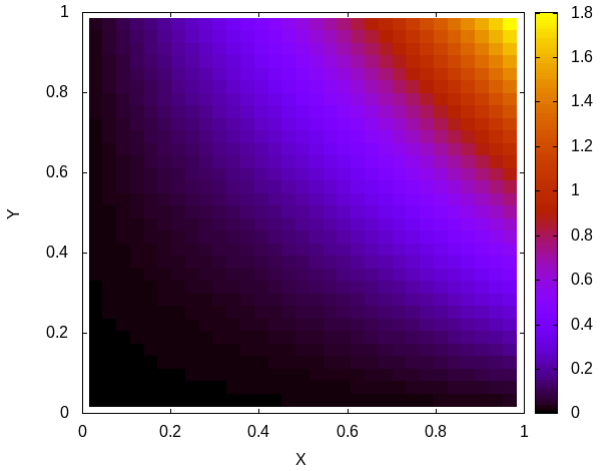
Максимальная погрешность и в какой точке она достигается:

dlt = 3.10862e-15, x[i]= 0.65625, y[j]= 0.71875

Время выполнения: 0.02613 секунд

Графики функции:





Метод сопряженных градиентов.

Точность: 10^{-5}

Функция: $u(x, y) = yx^3 + xy^2$

Lu: $2x(3y + 1)$

Число узлов: 32

Признак достижения точности, точность и пройденное число итераций:

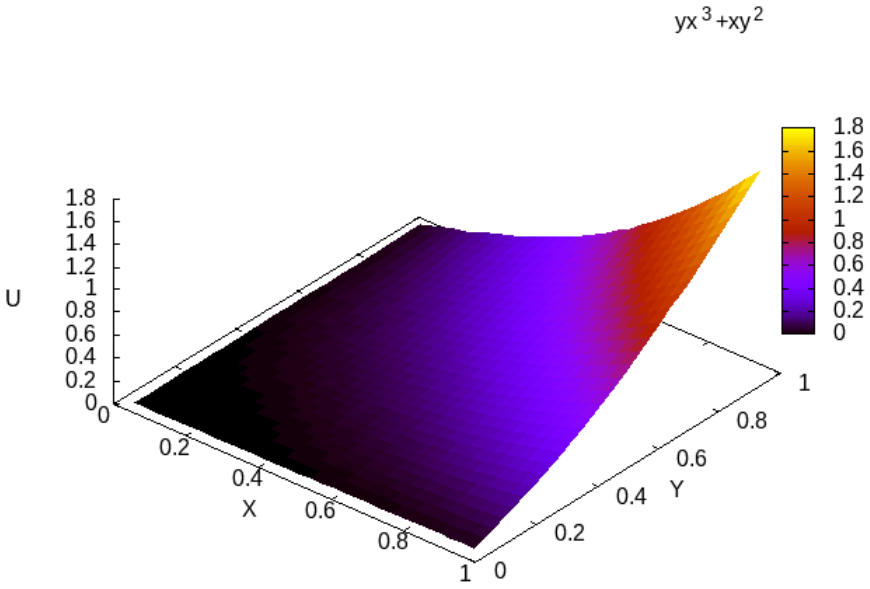
flag = 0, eps = 8.46515e-06, число итераций = 76

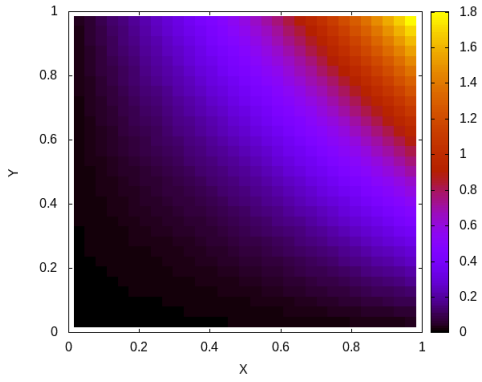
Максимальная погрешность и в какой точке она достигается:

dlt = 9.47254e-06, x[i]= 0.84375, y[j]= 0.25

Время выполнения: 0.018891 секунд

Графики функции:





Для функции $sin(\pi x)cos(\pi y)$ наблюдаемая максимальная погрешность почти одна и та же на одной и той же сетке и в методе матричной прогонки, и в методе сопряженных градиентов, однако последний дает результат почти в 10 раз быстрее.

Для функции $yx^3 + xy^2$ время выполнения едва ли отличается в двух методах, но наблюдаемая максимальная погрешность в методе матричной прогонки почти в 10^{10} раз меньше, чем в методе сопряженных градиентов.

Список литературы

[1] Самарский А.А., Гулин А.В. Численные методы математической физики. – М.: Научный мир, 2000. – 316 с.

[2] Деммель Дж. Вычислительная линейная алгебра. Теория и приложения. Пер. с англ. — М.: Мир, 2001. — 430 с, ил.