**КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА**

**Лабораторна робота 2**

Зміст:

1. Постановка задачі.
2. Обґрунтування методів.
3. Результати.
4. Код програми.
5. Висновок.

1. **Постановка задачі**.

Знайти розв’язок системи лінійних рівнянь

Ax=b, де

А: b:

|1 2 0 0 0 0 0 0 …| |1|

|1 3 2 0 0 0 0 0 …| |2|

|0 1 3 2 0 0 0 0 …| |3|

|0 0 1 3 2 0 0 0 …| |4|

|0 0 0 1 3 2 0 0 …| |5|

|0 0 0 0 1 3 2 0 …| |6|

|0 0 0 0 0 1 3 2 …| |7|

|0 0 0 0 0 0 1 3 …| |8|

|…………………| |..|

Методами Якобі та Прогонки

Cx=b

C: b:

|1 1 0 0 0 0 0 0 …| |1|

|a1 b1 1 0 0 0 0 0 …| |2|

|0 a2 b2 1 0 0 0 0 …| |3|

|0 0 a3 b3 1 0 0 0 …| |4|

|0 0 0 a4 b4 1 0 0 …| |5|

|0 0 0 0 a5 b5 1 0 …| |6|

|0 0 0 0 0 a6 b6 1 …| |7|

|0 0 0 0 0 0 a7 b7 …| |8|

|…………………………...| |..|

(аі = 2+2/х; bi = 1/x)

Методами Зейделя та Прогонки

Для кожної системи знайти нев'язку та

Розв’язати з точністю до 0,001 і визначити кількість ітерацій.

2. Обґрунтування методів.

**Метод прогонки**

Метод прогонки или алгоритм Томаса ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Thomas algorithm*) используется для решения [систем линейных уравнений](http://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D0%BB%D0%B8%D0%BD%D0%B5%D0%B9%D0%BD%D1%8B%D1%85_%D0%B0%D0%BB%D0%B3%D0%B5%D0%B1%D1%80%D0%B0%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D1%85_%D1%83%D1%80%D0%B0%D0%B2%D0%BD%D0%B5%D0%BD%D0%B8%D0%B9) вида ~Ax=F, где *A* — [трёхдиагональная матрица](http://ru.wikipedia.org/wiki/%D0%A2%D1%80%D1%91%D1%85%D0%B4%D0%B8%D0%B0%D0%B3%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%82%D1%80%D0%B8%D1%86%D0%B0).

Описание метода

Система уравнений ~Ax=F равносильна соотношению

~A_{i}x_{i-1}+C_{i}x_{i}+B_{i}x_{i+1} = F_{i}.\qquad\qquad(1)

Метод прогонки основывается на предположении, что искомые неизвестные связаны рекуррентным соотношением:

x_i = \alpha_{i+1}x_{i+1} + \beta_{i+1},\,\! где ~i=n-1,n-2,\dots,1.\qquad\qquad(2)

Используя это соотношение, выразим xi-1 и xi через xi+1 и подставим в уравнение (1):

\left(A_i\alpha_i\alpha_{i+1} + C_i\alpha_{i+1} + B_i\right)x_{i+1} + A_i\alpha_i\beta_{i+1} + A_i\beta_i + C_i\beta_{i+1} - F_i = 0,

где *Fi* — правая часть *i*-го уравнения. Это соотношение будет выполняться независимо от решения, если потребовать

\begin{cases} A_i\alpha_i\alpha_{i+1} + C_i\alpha_{i+1} + B_i = 0\\ 
A_i\alpha_i\beta_{i+1} + A_i\beta_i + C_i\beta_{i+1} - F_i = 0 \end{cases}

Отсюда следует:

 \begin{cases} \alpha_{i+1} = \frac{-B_i}{A_i\alpha_i + C_i} \\
\beta_{i+1} = \frac{F_i - A_i\beta_i}{A_i\alpha_i + C_i}\end{cases}

Из первого уравнения получим:

\begin{cases} \alpha_2 = \frac{-B_1}{C_1} \\ 
\beta_2 = \frac{F_1}{C_1}\end{cases}

После нахождения прогоночных коэффициентов α и β, используя уравнение (2), получим решение системы. При этом,

x_i = {\alpha_{i+1}x_{i+1} + \beta_{i+1}},\!     i=n-1..1 \,\!

 x_n = \frac{F_n-A_n\beta_n}{C_n+A_n\alpha_n} 

Другим способом объяснения существа метода прогонки, более близким к терминологии конечно-разностных методов и объясняющим происхождение его названия, является следующий: преобразуем уравнение (1) к эквивалентному ему уравнению

~A' x=F'\qquad\qquad(1')

c наддиагональной матрицей

A' = \begin{pmatrix} C_1' & B_1 & 0   & 0   & \cdots & 0 & 0
                         \\ 0 & C_2' & B_2 & 0   & \cdots & 0 & 0
                         \\ 0   & 0 & C_3' & B_3 & \cdots & 0 & 0 
                         \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots 
                         \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots 
                         \\ \cdots & \cdots & \cdots & \cdots & \cdots & C'_{n-1} & B_{n-1}
                         \\ 0 & 0 & 0 & 0 & \cdots & 0 & C_{n}'
            \end{pmatrix}
  .

Вычисления проводятся в два этапа. На первом этапе вычисляются компоненты матрицы ~C_i' и вектора ~F', начиная с  ~i=2  до  ~i=n:

~C'_1=C_1;

C'_i=C_i-\frac{A_i B_{i-1}}{C'_{i-1}},

и

~F'_1=F_1;

F'_i=F_i-\frac{A_i F'_{i-1}}{C'_{i-1}}.

На втором этапе, для i=n, n-1, \dots, 1 вычисляется решение:

x_n=\frac{F'_n}{C'_n};

x_i=\frac{F'_i-B_i x_{i+1}}{C'_i}.

Такая схема вычисления объясняет также английский термин этого метода «shuttle».

Для применимости формул метода прогонки достаточно свойства [диагонального преобладания](http://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D0%BE%D0%B9%D1%81%D1%82%D0%B2%D0%BE_%D0%B4%D0%B8%D0%B0%D0%B3%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BF%D1%80%D0%B5%D0%BE%D0%B1%D0%BB%D0%B0%D0%B4%D0%B0%D0%BD%D0%B8%D1%8F_%D0%BC%D0%B0%D1%82%D1%80%D0%B8%D1%86%D1%8B) у матрицы *A*.

**Метод Якоби**

Возьмём систему линейных уравнений:

A\vec{x}=\vec{b}, где A=\left(
\begin{array}{ccc}
a_{11} & \ldots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{n1} & \ldots & a_{nn} 
\end{array} \right),\quad \vec{b}=\left(
\begin{array}{c}
b_1 \\
\vdots \\
b_n 
\end{array} \right)

Или \left\{
\begin{array}{rcl}
a_{11}x_1 + \ldots + a_{1n}x_n& = & b_{1} \\
\ldots ~~~~~~~~~~~~~~~~~~~~~ \\
a_{n1}x_1 + \ldots + a_{nn}x_n & = & b_{n} 
\end{array} \right.

## Описание метода

Для того, чтобы построить итеративную процедуру метода Якоби, необходимо провести предварительное преобразование системы уравнений A\vec{x}=\vec{b} к итерационному виду \vec{x}=B\vec{x}+\vec{g}. Оно может быть осуществлено по одному из следующих правил:

* B = E-D^{-1}A = D^{-1}(D - A),\quad \vec{g}=D^{-1}\vec{b};
* B = -D^{-1}(L + U) = -D^{-1}(A - D),\quad \vec{g}=D^{-1}\vec{b}
* D^{-1}_{ii} = 1 / D_{ii}, D_{ii} \neq 0,\, i = 1,2, ..., n\quad; 

где в принятых обозначениях D означает матрицу, у которой на главной диагонали стоят соответствующие элементы матрицы A, а все остальные нули; тогда как матрицы U и L содержат верхнюю и нижнюю треугольные части A, на главной диагонали которых нули, *E* — единичная матрица.

Тогда процедура нахождения решения имеет вид:

 \vec{x}^{(k+1)}  = B \vec{x}^{(k)}+\vec{g},

где *k* счётчик итерации.

В отличие от метода [Гаусса-Зейделя](http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%93%D0%B0%D1%83%D1%81%D1%81%D0%B0-%D0%97%D0%B5%D0%B9%D0%B4%D0%B5%D0%BB%D1%8F) мы не можем заменять x^{(k)}_i\, на x^{(k+1)}_i в процессе итерационной процедуры, т.к. эти значения понадобятся для остальных вычислений. Это наиболее значимое различие между методом Якоби и методом Гаусса-Зейделя решения [СЛАУ](http://ru.wikipedia.org/wiki/%D0%A1%D0%9B%D0%90%D0%A3). Таким образом на каждой итерации придётся хранить оба вектора приближений: старый и новый.

## Условие сходимости

Приведем достаточное условие сходимости метода.

|  |  |  |
| --- | --- | --- |
| [Logo arte.jpg](http://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Logo_arte.jpg) | [**Теорема**](http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D0%BE%D0%B9_%D0%B8%D1%82%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%B8)**.** Пусть \| B \| < 1\!. Тогда при любом выборе начального приближения \vec{x}^{(0)}\!:   1. метод сходится; 2. скорость сходимости метода равна скорости сходимости [геометрической прогрессии](http://ru.wikipedia.org/wiki/%D0%93%D0%B5%D0%BE%D0%BC%D0%B5%D1%82%D1%80%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B5%D1%81%D1%81%D0%B8%D1%8F) со знаменателем q= \|B\|\!; 3. верна оценка погрешности: \|\vec{x}^{(k)}-\vec{x}\|=q^k \, \|\vec{x}^{(0)}-\vec{x}\|\!. |  |

## Условие остановки

Условие окончания итерационного процесса при достижении точности \varepsilon в упрощённой форме имеет вид:

\parallel x^{(k+1)}-x^{(k)}\parallel \le \varepsilon

(Существует более точное условие окончания итерационного процесса, которое более сложно и требует дополнительных вычислений)

**Метод Зейделя**

Возьмём систему: A\vec{x}=\vec{b}, где

A=\left(
\begin{array}{ccc}
a_{11} & \ldots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{n1} & \ldots & a_{nn} 
\end{array} \right),\quad \vec{b}=\left(
\begin{array}{c}
b_1 \\
\vdots \\
b_n 
\end{array} \right)

Или \left\{
\begin{array}{rcl}
a_{11}x_1 + \ldots + a_{1n}x_n& = & b_{1} \\
& &\\
a_{n1}x_1 + \ldots + a_{nn}x_n & = & b_{n} 
\end{array} \right.

И покажем, как её можно решить с использованием метода Гаусса-Зейделя.

## Метод

Чтобы пояснить суть метода, перепишем задачу в виде:

\left\{
\begin{array}{lcr}
a_{11}x_1 & = &-a_{12}x_2 - a_{13}x_3 -\ldots - a_{1n}x_n  +  b_1\\
a_{21}x_1 + a_{22}x_2 & = & -a_{23}x_3 - \ldots - a_{2n}x_n  +  b_2\\
\ldots & &\\
a_{(n-1)1}x_1 + a_{(n-1)2}x_2 +\ldots+ a_{(n-1)(n-1)}x_{n-1} & = & -a_{(n-1)n}x_n + b_{n-1}\\
a_{n1}x_1 + a_{n2}x_2 +\ldots+ a_{n(n-1)}x_{n-1}+ a_{nn}x_n & = & b_n
\end{array} \right.

Здесь в *j*-м уравнении мы перенесли в правую часть все члены, содержащие *xi* , для *i* > *j*. Эта запись может быть представлена:

(\mathrm{L} + \mathrm{D} )\vec{x} = -\mathrm{U} \, \vec{x} + \vec{b},

где в принятых обозначениях D означает матрицу, у которой на главной диагонали стоят соответствующие элементы матрицы A, а все остальные нули; тогда как матрицы U и L содержат верхнюю и нижнюю треугольные части A, на главной диагонали которых нули.

Итерационный процесс в методе Гаусса-Зейделя строится по формуле (\mathrm{L} + \mathrm{D} )\vec{x}^{(k+1)} = -\mathrm{U} \vec{x}^{(k)} + \vec{b} ,\quad k = 0, 1, 2, \ldots после выбора соответствующего начального приближения \vec{x}^{(0)}.

Метод Гаусса-Зейделя можно рассматривать как модификацию [метода Якоби](http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%AF%D0%BA%D0%BE%D0%B1%D0%B8). Основная идея модификации состоит в том, что новые значения \vec{x}^{(i)} используются здесь сразу же по мере получения, в то время как в [методе Якоби](http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%AF%D0%BA%D0%BE%D0%B1%D0%B8) они не используются до следующей итерации:

\left\{\begin{array}{ccccccccccc}
{x_{1}}^{(k+1)} &=& c_{12}{x_2^{(k)}} &+& c_{13}x_3^{(k)}&+& {\ldots}&+& c_{1n}{x_n}^{(k)} &+& d_1 \\
{x_{2}}^{(k+1)} &=& c_{21}{x_1^{(k+1)}} &+& c_{23}x_3^{(k)}&+& {\ldots}&+& c_{2n}{x_n}^{(k)} &+& d_2 \\
\ldots & & & & & & & & & & \\
{x_{n}}^{(k+1)} &=& c_{n1}{x_1^{(k+1)}} &+& c_{n2}{x_2^{(k+1)}}&+& {\ldots}&+& c_{n(n-1)}{x_{n-1}}^{(k+1)} &+& d_n
\end{array}\right.,

где c_{ij}=-\frac{a_{ij}}{a_{ii}},\quad d_i=\frac{b_i}{a_{ii}},\quad i=1,\ldots,n

Таким образом i-тая компонента (*k* + 1)-го приближения вычисляется по формуле:

x_i^{(k+1)}=\sum_{j=1}^{i-1}c_{ij}x_{j}^{(k+1)}+\sum_{j=i+1}^{n}c_{ij}x_{j}^{(k)}+d_i, \quad i=1,\ldots,n

## Условие сходимости

Приведём достаточное условие сходимости метода.

|  |  |  |
| --- | --- | --- |
| [Logo arte.jpg](http://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Logo_arte.jpg) | [**Теорема**](http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D0%BE%D0%B9_%D0%B8%D1%82%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%B8)**.** Пусть \| \mathrm{A}_2 \| < 1\!, где \mathrm{A}_2 = -(\mathrm{L} + \mathrm{D})^{-1} \, \mathrm{U}, \quad (\mathrm{L} + \mathrm{D})^{-1}\! – матрица, обратная к (\mathrm{L} + \mathrm{D})\!. Тогда при любом выборе начального приближения \vec{x}^{(0)}\!:   1. метод Гаусса-Зейделя сходится; 2. скорость сходимости метода равна скорости сходимости [геометрической прогрессии](http://ru.wikipedia.org/wiki/%D0%93%D0%B5%D0%BE%D0%BC%D0%B5%D1%82%D1%80%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B5%D1%81%D1%81%D0%B8%D1%8F) со знаменателем q = \|\mathrm{A}_2\|\!; 3. верна оценка погрешности: \|\vec{x}^{(k)}-\vec{x}\|=q^k \, \|\vec{x}^{(0)}-\vec{x}\|\!. |  |

## Условие окончания

Условие окончания итерационного процесса Зейделя при достижении точности \varepsilon в упрощённой форме имеет вид:

\parallel x^{(k+1)}-x^{(k)}\parallel \le \varepsilon

Более точное условие окончания итерационного процесса имеет вид

\parallel Ax^{(k)}-b\parallel \le \varepsilon

и требует больше вычислений. Хорошо подходит для [разреженных матриц](http://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D1%80%D0%B5%D0%B6%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%82%D1%80%D0%B8%D1%86%D0%B0).

**Підрахування констант:**

**3. Результати.**

**4. Код програми.**

#include <iostream>

#include <math.h>

#define MaxN 100

#define eps 0.001

using namespace std;

/// our matrix

double A[MaxN][MaxN];

/// vector x - we must find it

double x[MaxN];

/// first answer

double x0[MaxN];

/// vector of free elements

double b[MaxN];

/// nevyazka

double nev[MaxN];

/// matrix range

int n = 0;

///have an answer

bool haveanswer = false;

///second - ???

double second = 0;

/// init matrix for first lab

void Init1()

{

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

{

for (int j = 0; j < n; j++)

{

A[i][j] = 0;

}

A[i][i] = 3;

if (i == 0)

{

A[0][0] = 1;

}

else

{

A[i][i-1] = 1;

}

if (i <= n-1)

{

A[i][i+1] = 2;

}

b[i] = i + 1;

}

haveanswer = false;

}

///help functions

double a(double x)

{

return 2+2/x;

}

double p(double x)

{

return 1/x;

}

/// init matrix for second lab

void Init2()

{

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

{

for (int j = 0; j < n; j++)

{

A[i][j] = 0;

}

A[i][i] = a(i+1);

if (i > 0)

{

A[i][i-1] = p(i);

}

if (i <= n-1)

{

A[i][i+1] = 1;

}

b[i] = i + 1;

}

haveanswer = false;

}

///print Ax=b;

/// if we don't know x than print xi else numbers

void print()

{

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

{

cout << "| ";

for (int j = 0; j < n; j++)

{

cout << A[i][j] << " ";

}

cout << (i==(int)(n/2)?"|x|":"| |") << (haveanswer?"":"x") << (haveanswer?x[i]:i) << (i==(int)(n/2)?"|=|":"| |") << b[i] << "|" << endl;

}

}

/// reset vector x

void ResetX()

{

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

{

x[i] = 0;

}

}

/// method Jacobi

void Jacobi()

{

cout << "Method Jacobi starts..." << endl;

double norm;

do {

for (int i = 0; i < n; i++) {

x0[i] =- b[i];

for (int g = 0; g < n; g++) {

if (i != g)

x0[i] += A[i][g] \* x[g];

}

x0[i] /= -A[i][i];

}

norm = fabs(x[0] - x0[0]);

for (int h = 0; h < n; h++) {

if (fabs(x[h] - x0[h]) > norm)

norm = fabs(x[h] - x0[h]);

x[h] = x0[h];

}

} while (norm > eps);

haveanswer = true;

}

/// method Sweep

void Sweep()

{

cout << "Method Sweep starts..." << endl;

/// 3-diagonal to 2-diagonal

A[0][1] = A[0][1] / A[0][0];

b[0] = b[0] / A[0][0];

A[0][0] = 1;

for(int i = 1; i < n-1; i++)

{

A[i][i+1] = A[i][i+1] / (A[i][i] - A[i][i-1] \* A[i-1][i]);

b[i] = (b[i] - A[i][i-1] \* b[i-1])/(A[i][i] - A[i][i-1] \* A[i-1][i]);

A[i][i-1] = 0;

A[i][i] = 1;

}

b[n-1] = (b[n-1] - A[n-1][n-2] \* b[n-2])/(A[n-1][n-1] - A[n-1][n-2] \* A[n-2][n-1]);

A[n-1][n-2] = 0;

A[n-1][n-1] = 1;

/// find answer:

x[n-1] = b[n-1];

for(int i = n-2; i >= 0; i--)

{

x[i] = b[i] - x[i+1] \* A[i][i+1];

}

haveanswer = true;

}

///help function

bool converge(double \*xk, double\* xkp)

{

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

{

if (fabs(xk[i]-xkp[i]) >= eps)

return false;

}

return true;

}

/// method Zeidel

void Zeidel()

{

cout << "Method Zeidel starts..." << endl;

do

{

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

{

double var = 0;

for(int j = 0; j < n; j++)

if(j != i) var += (A[i][j]\*x[j]);

x0[i] = x[i];

x[i]=(b[i] - var)/A[i][i];

}

}

while(!converge(x,x0));

haveanswer = true;

}

/// print nevyazka

void Print\_Nevyazka()

{

cout << "Nevyazka:" << endl;

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

{

nev[i] = 0;

for(int j = 0; j < n; j++)

{

nev[i] += A[i][j] \* x[i];

}

nev[i] = b[i] - nev[i];

cout << "| " << nev[i] << " |" << endl;

}

}

/// print don't remember what :(

void Print\_Second()

{

cout << "Second:" << endl;

/// calculate it

/// print it

cout << second << end;

}

/// main function

int main()

{

cout << "Input n:" << endl;

cin >> n;

Init1();

print();

ResetX();

Jacobi();

print();

Init1();

Print\_Nevyazka();

Print\_Second();

ResetX();

Sweep();

print();

Init1();

Print\_Nevyazka();

Print\_Second();

cout<<"-----------------------------------------------------"<<endl;

Init2();

print();

ResetX();

Zeidel();

print();

Print\_Nevyazka();

Print\_Second();

Init2();

ResetX();

Sweep();

print();

Init2();

Print\_Nevyazka();

Print\_Second();

system("pause");

return 0;

}

**5. Висновок.**

Методи зійшлися, отже зроблені вірно. Похибка для таких методів апріорі відома: 0.001 – адже заздалегідь ми задаємо точність підрахунків.