Московский авиационный институт
(Национальный исследовательский университет)
Факультет прикладной математики и физики
Кафедра математической кибернетики

Лабораторная работа № 1

по курсу «Численные методы»

Тема: численные методы решения СЛАУ.

Студент: Мукин Ю. Д.
Группа: 80-304Б-18
Преподаватель: Гидаспов В.Ю.

Москва, 2021

Задание 1

1) Постановка задачи: Реализовать алгоритм LU - разложения матриц (с выбором главного элемента) в виде программы. Используя разработанное программное обеспечение, решить систему линейных алгебраических уравнений (СЛАУ). Для матрицы СЛАУ вычислить определитель и обратную матрицу.

Вариант 12:

$$\begin{cases} -x_1-8x_2+0x_3+5x_4=-60 \\ 6x_1-6x_2+2x_3+4x_4=-10 \\ -5x_1+0x_2-9x_3+4x_4=65 \\ 9x_1-5x_2-6x_3+x_4=18 \end{cases}$$

2) Теория:

LU – разложение матрицы A представляет собой разложение матрицы A в произведение нижней и верхней треугольных матриц, т.е. A = LU

где L - нижняя треугольная матрица, U- верхняя треугольная матрица. Чтобы получить LU разложение, необходимо выполнить прямой ход метода Гаусса. Рассмотрим k-ый шаг метода, на котором осуществляется обнуление поддиагональных элементов k-го столбца матрицы A. С этой целью используется следующая операция:

$$a_{ij}^{(k)}=a_{ij}^{(k-1)}-\mu_i^{(k)}a_{kj}^{(k-1)}, \quad \mu_i^{(k)}=\frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}}, \quad i=\overline{k+1,n}, \quad j=\overline{k,n}$$

В результате прямого хода метода Гаусса получим $A^{(n-1)}=U$, $L=M_1^{-1}M_2^{-1.}..M_{(n-1)}^{-1}$. Таким образом, искомое разложение A = LU получено.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -\mu_{k+1}^{(k)} & 1 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & -\mu_n^{(k)} & 0 & 0 & 1 \end{pmatrix} \quad M_k^{-1}=\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \mu_{k+1}^{(k)} & 1 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \mu_n^{(k)} & 0 & 0 & 1 \end{pmatrix}$$

3) Результат работы программы:

P
2 0 3 1
U
9.000000 -5.000000 -6.000000 4.000000
0.000000 -8.555556 -0.666667 5.444444
0.000000 0.000000 -12.116883 1.454545
0.000000 0.000000 0.000000 0.381565
L
1.000000 0.000000 0.000000 0.000000
-0.111111 1.000000 0.000000 0.000000
-0.555556 0.324675 1.000000 0.000000
0.666667 0.311688 -0.512326 1.000000
inverse matrix
-0.028090 -0.042135 0.095506 -0.073034
-0.904494 1.643258 -0.724719 0.848315
-0.123596 0.314607 -0.179775 0.078652
-1.252809 2.620787 -1.140449 1.342697
determinant of matrix A 356.000000
enter vector B: -60 -10 65 18

X:
6.002659 15.308510 -4.470744 13.694148

4) Код программы

```
int get_LU_with_main_element(matrix *matA, matrix *matL, matrix *matU, int P[])
{
    int n = (*matA).columns; int p=0;
    int t;
    matrix tmp1, tmp2;
    matrix C = create_matrix(n,n);
    for(int i=0; i<n; i++)
        for(int j=0; j<n; j++)
            *get_element(&C, i, j) = (*get_element(&(*matA), i, j));
    for( int i = 0; i < n; i++ )
    {
        double pivotValue = 0;
        int pivot = -1;
        for( int row = i; row < n; row++ )
        {
            if( fabs((*get_element(&C, row, i))) > pivotValue )
            {
                pivotValue = fabs((*get_element(&C, row, i)));
                pivot = row;
            }
        }
        if( pivotValue != 0 )
        {
            p++;
            tmp1 = get_matrix_line(C, i);
            tmp2 = get_matrix_line(C, pivot);
            insert_matrix_line(&C, tmp1, pivot);
            insert_matrix_line(&C, tmp2, i);
            t = P[i]; P[i] = P[pivot]; P[pivot] = t;
            for( int j = i+1; j < n; j++ )
            {
                *get_element(&C, j, i) /= (*get_element(&C, i, i));
                for( int k = i+1; k < n; k++ )
                    *get_element(&C, j, k) -= (*get_element(&C, j, i)) * (*get_element(&C, i, k));
```

```
            }
        }
    }
    for(int i=0; i<n; i++)
        for(int j=0; j<n; j++)
        {
            if(i<j)
                *get_element(&(*matU), i, j) = *get_element(&C, i, j);
            if(i==j)
            {
                *get_element(&(*matU), i, j) = *get_element(&C, i, j);
                *get_element(&(*matL), i, j) = 1;
            }
            if(i>j)
                *get_element(&(*matL), i, j) = *get_element(&C, i, j);
        }
    return p;
}

double get_det(matrix matU, int p)
{
    double det = 1;
    for (int i=0;i<matU.columns; i++)
        det *= (*get_element(&matU, i, i));
    return pow(-1,p)*det;
}

matrix get_inverse_matrix(matrix L, matrix U, int P[])
{
    int n = L.columns;
    matrix res = create_matrix(n,n);
    matrix res1 = create_matrix(n,n);
    matrix E = create_singular_matrix(n);
    for(int i=0; i<n; i++)
    {
        matrix y = solve_by_LU(L, E, get_matrix_line(E,i));
        matrix x = solve_by_LU(E, U, y);
        for(int j=0; j<n; j++)
            *get_element(&res, j, i) = *get_element(&x, 0, j);
    }
    for(int i=0; i<n; i++)
        insert_matrix_column(&res1, get_matrix_column(res,i), P[i]);
    return res1;
}

matrix solve_by_Seidel(matrix alpha, matrix betta, double epsilon)
{
    int n = alpha.rows;
    matrix x = create_matrix(n, 1); matrix B = create_matrix(n, n); matrix C = create_matrix(n, n);
    matrix pre_x = create_matrix(n, 1);
    for(int i=0; i<n; i++)
            *get_element(&x, i, 0) = *get_element(&betta, i, 0);
    double epsilon_i, tmp;
    int k = 0;
    do
    {
        for(int i=0; i<n; i++)
            *get_element(&pre_x, i, 0) = *get_element(&x, i, 0);
        for(int i=0; i<n; i++)
        {
            tmp = 0;
            for(int j=0; j<n; j++)
                tmp += *get_element(&alpha, i, j) *(*get_element(&x, j, 0));
```

```
            *get_element(&x, i, 0) = *get_element(&betta, i, 0) + tmp;
        }
        epsilon_i = get_norm(pre_x, x);
        k++;
    }
    while (epsilon_i > epsilon && k<40);
    printf("\n%d iterations have been done",k);
    return x;
}

void LU_method()
{
    matrix matA, matL, matU, B;
    int n, perm;
    printf("enter the dimension of the matrix: ");
    scanf("%d", &n);
    int *P = malloc(n*sizeof(int));
    printf("enter the matrix in one line: ");
    matA = create_matrix(n,n); matL = create_matrix(n,n); matU = create_matrix(n,n); B = create_matrix(1,n);
    for (int i=0; i<matA.columns; i++)
    {
        P[i] = i;
        for (int j=0; j<matA.columns; j++)
            scanf("%lf", &*get_element(&matA, i, j));
    }
    perm = get_LU_with_main_element(&matA, &matL, &matU, P);
    printf("\n%s \n%d %d %d %d ", "P", P[0], P[1], P[2], P[3]);
    printf("\n%s ", "U");
    print_matrix(&matU);
    printf("\n%s ", "L");
    print_matrix(&matL);
    matrix matR = get_inverse_matrix(matL, matU, P);
    printf("\n%s ", "inverse matrix");
    print_matrix(&matR);
    printf("\n%s %lf", "determinant of matrix A", get_det(matU, perm));
    printf("\nenter vector B: ");
    for (int i=0; i<B.rows; i++)
        for (int j=0; j<B.columns; j++)
            scanf("%lf", &*get_element(&B, i, P[j]));
    matrix x = solve_by_LU(matL, matU, B);
    printf("\nX: ");
    print_matrix(&x);
}
```

5) Выводы: Метод Гаусса применяется как для аналитического решения СЛАУ, так и для нахождения обратной матрицы. Он позволяет получить наиболее точное решение и является менее трудоемким методом для матриц ограниченного размера.

Задание 2

1) Постановка задачи: Реализовать метод прогонки в виде программы, задавая в качестве входных данных ненулевые элементы матрицы системы и вектор правых частей. Используя разработанное программное обеспечение, решить СЛАУ с трехдиагональной матрицей.

Вариант 12:

$$\begin{cases} -11\cdot x_1 + 9\cdot x_2 = -114 \\ x_1 - 8\cdot x_2 + x_3 = 81 \\ -2\cdot x_2 - 11\cdot x_3 + 5\cdot x_4 = -8 \\ 3\cdot x_3 - 14\cdot x_4 + 7\cdot x_5 = -38 \\ 8\cdot x_4 + 10\cdot x_5 = 144 \end{cases}$$

2) Теория: Метод прогонки является одним из эффективных методов решения СЛАУ с трех - диагональными матрицами. Рассмотрим СЛАУ:

$$a_1 = 0 \begin{cases} b_1x_1 + c_1x_2 = d_1 \\ a_2x_1 + b_2x_2 + c_2x_3 = d_2 \\ \qquad a_3x_2 + b_3x_3 + c_3x_4 = d_3 \\ \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots \\ \qquad\qquad a_{n-1}x_{n-2} + b_{n-1}x_{n-1} + c_{n-1}x_n = d_{n-1} \\ \qquad\qquad\qquad a_nx_{n-1} + b_nx_n = d_n, \qquad c_n = 0, \end{cases}$$

Решение которой будем искать в виде $x_i = P_i x_{i+1} + Q_i$, где P и Q – прогоночные коэффиценты, которые вычисляются по формулам:

$$P_i = \frac{-c_i}{b_i + a_iP_{i-1}}, \quad Q_i = \frac{d_i - a_iQ_{i-1}}{b_i + a_iP_{i-1}}.$$

$$P_1 = \frac{-c_1}{b_1}, \quad Q_1 = \frac{d_1}{b_1}$$

$$P_n = 0, \text{ т.к. } c_n = 0, \ Q_n = \frac{d_n - a_nQ_{n-1}}{b_n + a_nP_{n-1}}, \quad i = n.$$

Обратный ход метода прогонки осуществляется в соответствии с выражением:

$$\begin{cases} x_n = P_nx_{n+1} + Q_n = 0\cdot x_{n+1} + Q_n = Q_n \\ x_{n-1} = P_{n-1}x_n + Q_{n-1} \\ x_{n-1} = P_{n-2}x_{n-1} + Q_{n-2} \\ \dots\dots\dots\dots\dots\dots \\ x_1 = P_1x_2 + Q_1. \end{cases}$$

Достаточное условие сходимости метода прогонки:

$a_i \neq 0$, $c_i \neq 0$, $i = \overline{2, n-1}$

$|b_i| \geq |a_i| + |c_i|$, $i = \overline{1, n}$,

3) Результат работы программы:

```
A
0.000000 1.000000 -2.000000 3.000000 8.000000
B
-11.000000 -8.000000 11.000000 -14.000000 10.000000
C
9.000000 1.000000 5.000000 7.000000 0.000000
enter vector D: -114 81 -8 -38 144

P
0.818182 0.139241 -0.466352 0.454573 0.000000
Q
10.363636 -9.835443 -2.580874 1.964885 9.407113
X
1.690850 -10.600072 -5.491426 6.241108 9.407113
```

4) Код программы

```
void get_ABC(matrix mat, matrix *A, matrix *B, matrix *C)
{
   int n = mat.columns;
   *get_element(&(*A), 0, 0) = 0;
   *get_element(&(*B), 0, 0) = *get_element(&mat, 0, 0);
   *get_element(&(*C), 0, 0) = *get_element(&mat, 0, 1);
   for(int i=1; i<n-1; i++)
   {
      *get_element(&(*A), 0, i) = *get_element(&mat, i, i-1);
      *get_element(&(*B), 0, i) = *get_element(&mat, i, i);
      *get_element(&(*C), 0, i) = *get_element(&mat, i, i+1);
   }
   *get_element(&(*A), 0, n-1) = *get_element(&mat, n-1, n-2);
   *get_element(&(*B), 0, n-1) = *get_element(&mat, n-1, n-1);
   *get_element(&(*C), 0, n-1) = 0;
}

void get_PQ(matrix A, matrix B, matrix C, matrix D, matrix *P, matrix *Q)
{
   int n = A.columns;
   *get_element(&(*P), 0, 0) = -( *get_element(&C, 0, 0))/(*get_element(&B, 0, 0));
   *get_element(&(*Q), 0, 0) = ( *get_element(&D, 0, 0))/(*get_element(&B, 0, 0));
   for(int i=1; i<n-1; i++)
   {
         *get_element(&(*P), 0, i) = -( *get_element(&C, 0, i))/((*get_element(&B, 0, i))+(*get_element(&A, 0,
i))*(*get_element(&(*P), 0, i-1)));
         *get_element(&(*Q), 0, i) = ((*get_element(&D, 0, i))-(*get_element(&A, 0, i)*(*get_element(&(*Q), 0,
i-1))))/((*get_element(&B, 0, i))+(*get_element(&A, 0, i)*(*get_element(&(*P), 0, i-1))));
   }
   *get_element(&(*P), 0, n-1) = 0;
    *get_element(&(*Q), 0, n-1) = ((*get_element(&D, 0, n-1))-(*get_element(&A, 0, n-1)*(*get_element(&(*Q), 0, n-
2))))/((*get_element(&B, 0, n-1))+(*get_element(&A, 0, n-1)*(*get_element(&(*P), 0, n-2))));
}

matrix solve_by_run(matrix P, matrix Q)
{
   int n = P.columns;
```

```
    matrix x = create_matrix(1,n);
    *get_element(&x, 0, n-1) = *get_element(&Q, 0, n-1);
    for(int i=2; i<=n; i++)
        *get_element(&x, 0, n-i) = (*get_element(&P, 0, n-i)) * (*get_element(&x, 0, n-i+1)) + *get_element(&Q, 0, n-i);
    return x;
}

void run_method()
{
    matrix mat, A, B, C, P, Q, D;
    int n;
    printf("enter the dimension of the matrix: ");
    scanf("%d", &n);
    printf("enter the matrix in one line: ");
    mat = create_matrix(n,n); A = create_matrix(1,n); B = create_matrix(1,n); C = create_matrix(1,n); D = create_matrix(1,n); P = create_matrix(1,n); Q = create_matrix(1,n);
    for (int i=0; i<mat.columns; i++)
        for (int j=0; j<mat.columns; j++)
            scanf("%lf", &*get_element(&mat, i, j));
    get_ABC(mat, &A, &B, &C);
    printf("\n%s ", "A");
    print_matrix(&A);
    printf("\n%s ", "B");
    print_matrix(&B);
    printf("\n%s ", "C");
    print_matrix(&C);
    printf("\nenter vector D: ");
    for (int i=0; i<D.rows; i++)
        for (int j=0; j<D.columns; j++)
            scanf("%lf", &*get_element(&D, i, j));
    get_PQ(A,B,C,D,&P,&Q);
    printf("\n%s ", "P");
    print_matrix(&P);
    printf("\n%s ", "Q");
    print_matrix(&Q);
    matrix x = solve_by_run(P,Q);
    printf("\n%s ", "X");
    print_matrix(&x);
}
```

5) Выводы: Метод прогонки отлично подходит для решения СЛАУ с трехдиагональной матрицей, так как был разработан исключительно для этого. Является итерационным методом.

Задание 3

1) Постановка задачи: Реализовать метод простых итераций и метод Зейделя в виде программ, задавая в качестве входных данных матрицу системы, вектор правых частей и точность вычислений. Используя разработанное программное обеспечение, решить СЛАУ. Проанализировать количество итераций, необходимое для достижения заданной точности.

Вариант 12:

$$\begin{cases} 14\cdot x_1 - 4\cdot x_2 - 2\cdot x_3 + 3\cdot x_4 = 38 \\ -3\cdot x_1 + 23\cdot x_2 - 6\cdot x_3 - 9\cdot x_4 = -195 \\ -7\cdot x_1 - 8\cdot x_2 + 21\cdot x_3 - 5\cdot x_4 = -27 \\ -2\cdot x_1 - 2\cdot x_2 + 8\cdot x_3 + 18\cdot x_4 = 142 \end{cases}$$

2.1) Теория (метод простых итераций): Методы последовательных приближений, в которых при вычислении последующего приближения решения используются предыдущие, уже известные приближенные решения, называются итерационными. Рассмотрим СЛАУ:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n \end{cases}$$

Приведем СЛАУ к эквивалентному виду:

$$\begin{cases} x_1 = \beta_1 + \alpha_{11}x_1 + \alpha_{12}x_2 + \ldots + \alpha_{1n}x_n \\ x_2 = \beta_2 + \alpha_{21}x_1 + \alpha_{22}x_2 + \ldots + \alpha_{2n}x_n \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ x_n = \beta_n + \alpha_{n1}x_1 + \alpha_{n2}x_2 + \ldots + \alpha_{nn}x_n \end{cases}$$

В векторно-матричной форме:

$$x = \beta + \alpha x .$$

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix},\ \beta = \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_n \end{pmatrix},\ \alpha = \begin{pmatrix} \alpha_{11} & \cdots & \alpha_{1n} \\ \vdots & \cdots & \vdots \\ \alpha_{n1} & \ldots & \alpha_{nn} \end{pmatrix}.$$

Достаточное условие сходимости (преобладание диагональных элементов):

$$|a_{ii}| > \sum_{j=1, i\neq j}^{n} |a_{ij}| \quad \forall i$$

2.2) Теория (метод Зейделя): Является модификацией метода простых итераций. Для вычисления следующего компонента вектора неизвестных используются предыдущие компоненты, которые уже вычислены на данной итерации. Итерационная формула имеет вид:

$$x^{k+1} = (E-B)^{-1}\, Cx^k + (E-B)^{-1}\beta.$$

3) Результат работы программы:

alpha
0.000000 0.285714 0.142857 -0.214286
0.130435 0.000000 0.260870 0.391304
0.333333 0.380952 0.000000 0.238095
0.111111 0.111111 -0.444444 0.000000
betta
2.714286
-8.478261
-1.285714
7.888889
enter precision: 0.01

7 iterations have been done
X iterative method
-0.999324
-6.000211
-1.999959
7.999742
5 iterations have been done
X Seidel
-1.000725
-6.000407
-2.000370
8.000039

4) Код программы

```
void get_alpha_betta(matrix matA, matrix B, matrix *alpha, matrix *betta)
{
   int n = matA.columns;
   for(int i=0; i<n; i++)
      for(int j=0; j<n; j++)
      {
         *get_element(&(*betta), i, 0) = *get_element(&B, i, 0) /(*get_element(&matA, i, i));
         if(i != j)
            *get_element(&(*alpha), i, j) = -(*get_element(&matA, i, j))/(*get_element(&matA, i, i));
         else
            *get_element(&(*alpha), i, j) = 0;
      }
}


matrix solve_by_iterative_method(matrix alpha, matrix betta, double epsilon)
{
   matrix x = betta;
   matrix pre_x;
   double epsilon_i;
   int k = 0;
   do
   {
```

```
      pre_x = x;
      x =  addition_matrix(betta, multiply_matrix(alpha, pre_x));
      epsilon_i = get_norm(pre_x, x);
      k++;
   }
   while (epsilon_i > epsilon && k<100);
   printf("\n%d iterations have been done",k);
   return x;
}

matrix solve_by_Seidel(matrix alpha, matrix betta, double epsilon)
{
   int n = alpha.rows;
   matrix x = create_matrix(n, 1); matrix B = create_matrix(n, n); matrix C = create_matrix(n, n);
   matrix pre_x = create_matrix(n, 1);
   for(int i=0; i<n; i++)
         *get_element(&x, i, 0) = *get_element(&betta, i, 0);
   double epsilon_i, tmp;
   int k = 0;
   do
   {
      for(int i=0; i<n; i++)
         *get_element(&pre_x, i, 0) = *get_element(&x, i, 0);
      for(int i=0; i<n; i++)
      {
         tmp = 0;
         for(int j=0; j<n; j++)
            tmp += *get_element(&alpha, i, j) *(*get_element(&x, j, 0));
         *get_element(&x, i, 0) = *get_element(&betta, i, 0) + tmp;
      }
      epsilon_i = get_norm(pre_x, x);
      k++;
   }
   while (epsilon_i > epsilon && k<40);
   printf("\n%d iterations have been done",k);
   return x;
}

void Seidel_iterative_methods()
```

```
{
    matrix matA, B, alpha, betta;
    double eps;
    int n;
    printf("enter the dimension of the matrix: ");
    scanf("%d", &n);
    printf("enter the matrix in one line: ");
    matA = create_matrix(n,n); alpha = create_matrix(n,n); B = create_matrix(n,1); betta = create_matrix(n,1);
    for (int i=0; i<matA.columns; i++)
        for (int j=0; j<matA.columns; j++)
            scanf("%lf", &*get_element(&matA, i, j));
    printf("\nenter vector B: ");
    for (int i=0; i<B.rows; i++)
        for (int j=0; j<B.columns; j++)
            scanf("%lf", &*get_element(&B, i, j));
    get_alpha_betta(matA, B, &alpha, &betta);
    printf("\n%s ", "alpha");
    print_matrix(&alpha);
    printf("\n%s ", "betta");
    print_matrix(&betta);
    printf("\nenter precision: ");
    scanf("%lf", &eps);
    matrix x1 = solve_by_iterative_method(alpha, betta, eps);
    printf("\n%s ", "X iterative method");
    print_matrix(&x1);
    matrix x2 = solve_by_Seidel(alpha, betta, eps);
    printf("\n%s ", "X Seidel");
    print_matrix(&x2);
}
```

Задание 4

1) Постановка задачи: Реализовать метод вращений в виде программы, задавая в качестве входных данных матрицу и точность вычислений. Используя разработанное программное обеспечение, найти собственные значения и собственные векторы симметрических матриц. Проанализировать зависимость погрешности вычислений от числа итераций.

Вариант 12:

$$\begin{pmatrix} 7 & 3 & -1 \\ 3 & -7 & -8 \\ -1 & -8 & -2 \end{pmatrix}$$

2) Теория: Метод вращений Якоби применим только для симметрических матриц и решает полную проблему собственных значений и векторов. Он основан на нахождении матрицы U, которая позволяет осуществить преобразование подобия для исходной матрицы и на выходе получить диагональную матрицу с собственными значениями на главной диагонали.

Алгоритм метода следующий:

- Выбираем максимальный по модулю недиагональный элемент
- Строим матрицу U таким образом, чтобы в результате преобразования подобия выбранный элемент обнулился. Для этого берем матрицу вращения, имеющую следующий вид:

$$U^k = \begin{array}{cc}
\begin{array}{ccccccccc} & & & i & & & j & & \end{array} & \\
\left(\begin{array}{ccccccccc}
1 & & & \vdots & & & \vdots & & \\
& \ddots & & \vdots & & & \vdots & 0 & \\
& & 1 & \vdots & & & \vdots & & \\
\cdots & \cdots & \cdots & \cos\varphi^{(k)} & \cdots & \cdots & -\sin\varphi^{(k)} & \cdots & \cdots \\
& & & \vdots & 1 & & \vdots & & \\
& & & \vdots & & \ddots & \vdots & & \\
& & & \vdots & & 1 & \vdots & & \\
\cdots & \cdots & \cdots & \sin\varphi^{(k)} & \cdots & \cdots & \cos\varphi^{(k)} & \cdots & \cdots \\
& & & \vdots & & & \vdots & 1 & \\
& 0 & & \vdots & & & \vdots & \ddots & \\
& & & \vdots & & & \vdots & & 1
\end{array}\right) &
\begin{array}{c} \\ \\ \\ i \\ \\ \\ \\ j \\ \\ \\ \\ \end{array}
\end{array}$$

Угол вращения определяется из условия равенства нулю выбранного элемента.

- выполняется преобразование подобия

В качестве критерия окончания итерационного процесса берется условие малости суммы квадратов внедиагональных элементов. Координатными столбцами собственных векторов будут столбцы матрицы

$$U = U^{(0)}U^{(1)}...U^{(k)}$$

3) Результат работы программы:

```
enter the desired precision: 0.01

5 iterations have been done
X
0.868494 -0.064321 0.491509
0.349851 0.781987 -0.515850
-0.351174 0.619967 0.701654
lambda
8.575333
-12.999218
2.423885
```

4) Код программы

```
double get_angle(matrix matA, int i, int j)
{
    if ((*get_element(&matA, i, i))==(*get_element(&matA, j, j)))
        return PI / 4;
    else
        return 0.5 * atan(2*(*get_element(&matA, i, j))/((*get_element(&matA, i, i))-(*get_element(&matA, j, j))));
}

double get_square_summ(matrix matA)
{
    int n = matA.columns;
    double sum = 0;
    for(int i=1; i<n; i++)
        for(int j=0; j<i; j++)
            sum+=(*get_element(&matA, i, j))*(*get_element(&matA, i, j));
    return sqrt(sum);
}

matrix jakobi_method(matrix matA, double epsilon, matrix *lambda)
{
    int n = matA.columns;
    int a[2] = {0,0};
    matrix X = create_singular_matrix(n);
    double p;
    int iter=0;
    double angle;
```

```
    do
    {
        get_max(matA, a);
        int i = a[0]; int j = a[1];
        angle = get_angle(matA, i, j);
        matrix matU = create_singular_matrix(n);
        *get_element(&matU, i, i) = cos(angle);
        *get_element(&matU, i, j) = -sin(angle);
        *get_element(&matU, j, i) = sin(angle);
        *get_element(&matU, j, j) = cos(angle);
        X = multiply_matrix(X,matU);

        matrix tmp = create_matrix(n,n);
        for(int i1=0; i1<n; i1++)
            for(int j1=0; j1<n; j1++)
                *get_element(&tmp, i1, j1) = *get_element(&matU, i1, j1);
        *get_element(&tmp, i, j) = sin(angle);
        *get_element(&tmp, j, i) = -sin(angle);

        matrix tmp1 =  multiply_matrix(tmp, matA);
        matA = multiply_matrix(tmp1, matU);

        p= get_square_summ(matA);
        iter++;
    } while (epsilon < p && angle != 0 && iter < 100);

    for (int k=0; k<n; k++)
        *get_element(&(*lambda), k, 0) = *get_element(&matA, k, k);
    printf("\n%d iterations have been done", iter);
    return X;
}

void rotation_method()
{
    matrix matA, lambda;
    int n; double eps = 0.001;
    printf("enter the dimension of the matrix: ");
    scanf("%d", &n);
    printf("enter the matrix in one line: ");
    matA = create_matrix(n,n); lambda = create_matrix(n,1);
    for (int i=0; i<matA.columns; i++)
        for (int j=0; j<matA.columns; j++)
            scanf("%lf", &*get_element(&matA, i, j));
    printf("enter the desired precision: ");
    scanf("%lf", &eps);
    matrix X = jakobi_method(matA, eps, &lambda);
```

```
    printf("\n%s ", "X");
    print_matrix(&X);
    printf("\n%s ", "lambda");
    print_matrix(&lambda);
}
```

Задание 5

1) Постановка задачи: Реализовать алгоритм QR – разложения матриц в виде программы. На его основе разработать программу, реализующую QR – алгоритм решения полной проблемы собственных значений произвольных матриц, задавая в качестве входных данных матрицу и точность вычислений. С использованием разработанного программного обеспечения найти собственные значения матрицы.

Вариант 12:

$$\begin{pmatrix} 5 & -1 & -2 \\ -4 & 3 & -3 \\ -2 & -1 & 1 \end{pmatrix}$$

2) Теория: QR-алгоритм позволяет находить как вещественные, так и комплексные собственные значения. В основе алгоритма лежит разложение исходной матрицы на произведение ортогональной матрицы Q и верхней треугольной матрицы R. Алгоритм использует следующий итерационный процесс:

$A^{(0)} = A$,

$A^{(0)} = Q^{(0)}R^{(0)}$ - производится QR - разложение,

$A^{(1)} = R^{(0)}Q^{(0)}$ - производится перемножение матриц,

....

$A^{(k)} = Q^{(k)}R^{(k)}$ - разложение,

$A^{(k+1)} = R^{(k)}Q^{(k)}$ перемножение.

Если в ходе итерационного процесса прослеживается комплексно-сопряженная пара собственных значений, то начиная с некоторой итерации они будут отличаться незначительно. Поэтому в качестве окончания итерационного процесса можно взять условие $|\lambda^{(k)} - \lambda^{(k-1)}| \le \varepsilon$

Само QR разложение выполняется при помощи матрицы Хаусхолдера, имеющей следующий вид: $H=E-\frac{2}{v^T v} v v^T$, где вектор v определяется таким образом, чтобы в результате преобразования Ai = HiAi-1 обнулить поддиагональные элементы. Повторив данное преобразование n-1 раз, получим QR разложение, где $Q=(H_{n-1}H_{n-2}\dots H_{n-0})^T=H_1 H_2\dots H_{n-0},\ R=A_{n-1}$

3) Результат работы программы:

```
enter the desired precision: 0.01

A0:
6.333333 0.304290 -3.129904
-0.486864 3.888889 1.142879
-0.544331 -1.242260 -1.222222

A1:
6.588846 1.339249 2.238798
-0.424240 3.503220 -2.450824
-0.098179 -0.346388 -1.092067

A2:
6.481323 1.302295 -2.634769
-0.198469 3.776216 1.860139
-0.018779 -0.115791 -1.257538

A3:
6.452790 1.462309 2.626223
-0.122052 3.759086 -1.926279
-0.003511 -0.035698 -1.211876

A4:
6.425036 1.488110 -2.680183
-0.070588 3.802602 1.830044
-0.000668 -0.011330 -1.227639

A5:
6.409429 1.524644 2.694880
-0.042139 3.813209 -1.815243
-0.000127 -0.003579 -1.222638

A6:
6.399517 1.539134 -2.708337
-0.025099 3.824735 1.792801
-0.000024 -0.001137 -1.224252

A7:
6.393549 1.550011 2.714862
-0.015031 3.830187 -1.783657
-0.000005 -0.000361 -1.223736

A8:
6.389924 1.555773 -2.719199
-0.009012 3.833977 1.776795
-0.000001 -0.000115 -1.223901
lambda0: 6.389924
lambda1: 3.833977
lambda2: -1.223901
```

4) Код программы

```
double square_sum_column(matrix mat, int column_number, int first_index)
{
   int n = mat.columns;
   double sum =0;
   for(int i=first_index; i<n;i++)
      sum+= (*get_element(&mat,i, column_number)) * (*get_element(&mat,i, column_number));
   return sqrt(sum);
}

int is_end(matrix mat, double eps)
{
   int n = mat.columns; int z;
   double sum1, sum2;
   for(int j=0; j<n;j++)
   {
      sum1 = square_sum_column(mat, j, j+1);
      sum2 = square_sum_column(mat, j, j+2);
      if(sum2 > eps)
         return 0;
      else if(sum1 <= eps)
      {
         printf("\nlambda%d: %lf",j ,*get_element(&mat,j, j));
      }
      else if(sum1 > eps)
      {
         double aii = *get_element(&mat, j, j);
         double ajj = *get_element(&mat, j+1, j+1);
         double aij = *get_element(&mat, j, j+1);
         double aji = *get_element(&mat, j+1, j);
         double x = (aii + ajj) / 2;
         double D = (-(aii+ajj)*(aii+ajj) + 4*(aii*ajj - aij*aji));
         if (D<0)
            return 0;
         double y = sqrt(D) / 2;
         printf("\nlambda%d: %lf + %lfi", j, x, y);
         printf("\nlambda%d: %lf - %lfi", j+1, x, y);
         j++;
```

```
        }
    }
    return 1;
}

void get_QR(matrix matA, matrix *matQ, matrix *matR)
{
    int n = matA.columns;
    double ab, bb, norm;
    matrix c, a, b, e;
    a = create_matrix(n,1);
    b = create_matrix(n,n);
    e = create_matrix(n,1);
    c = create_matrix(n-1,1);
    for (int i=0; i<n; i++)
    {
        if (i==0)
        {
            insert_matrix_column(&b,get_matrix_column(matA, i),i);
            norm = scalar_product(get_matrix_column(b, i),get_matrix_column(b, i));
            for(int j =0;j<n; j++)
                *get_element(&e, j, 0) = (*get_element(&b, j, 0))/sqrt(norm);
        }
        else
        {
            a = get_matrix_column(matA, i);
            for(int l=0; l<i; l++)
            {
                ab = scalar_product(a,get_matrix_column(b, l));
                bb = scalar_product(get_matrix_column(b, l),get_matrix_column(b, l));
                *get_element(&c, l, 0) = ab/bb;
            }
            printf("\n");
            for(int k=0; k<i;k++)
                for(int j =0;j<n; j++)
                    *get_element(&a, j, 0) -= (*get_element(&b, j, k))*(*get_element(&c, k, 0));
            insert_matrix_column(&b,a,i);
            norm = scalar_product(a,a);
            for(int j =0;j<n; j++)
```

```
            *get_element(&e, j, 0) = (*get_element(&b, j, i))/sqrt(norm);
        }
        insert_matrix_column(&(*matQ),e, i);
    }
    (*matR) = multiply_matrix(transpose_matrix(*matQ), matA);
}

void GR_method()
{
    matrix matA, matR, matQ;
    int n, k;
    double eps;
    k=0;
    printf("enter the dimension of the matrix: ");
    scanf("%d", &n);
    printf("enter the matrix in one line: ");
    matA = create_matrix(n,n); matR = create_matrix(n,n); matQ = create_matrix(n,n);
    for (int i=0; i<matA.columns; i++)
        for (int j=0; j<matA.columns; j++)
            scanf("%lf", &*get_element(&matA, i, j));
    printf("enter the desired precision: ");
    scanf("%lf", &eps);
    while(is_end(matA, eps)==0 && k<200)
    {
        get_QR(matA, &matQ, &matR);
        matA = multiply_matrix(matR, matQ);
        printf("\nA%d:",k);
        print_matrix(&matA);
        k++;
    }
}
```