3.2.4 Conjugate gradient method

The conjugate gradient (CG) method is the most popular iterative method for solving large systems of linear equations. CG is effective for systems of the form $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is a known, square, symmetric, positive-definite (or positive-indefinite) matrix. These systems arise in many important settings, such as finite difference and finite element methods for solving partial differential equations, structural analysis, circuit analysis, and math homework. The main idea of the CG algorithm is to solve the linear equations by minimizing the quadratic form $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{Ax} - \mathbf{b}^T\mathbf{x}$. For further information see Wikipedia.

Class design

Design a new class `CGSolver`. The properties of a `CGSolver` object are the associated matrix $\mathbf{A}$, the vector $\mathbf{b}$ and a tolerance level. The class `CGSolver` has a constructor with a matrix object as parameter. Specify a method to solve a system of linear equations $\mathbf{Ax} = \mathbf{b}$ using the conjugate gradient method. Create the UML diagram of the `CGSolver` class. In the next step develop the Nassi-Schneiderman diagram for the conjugate gradient algorithm. Use an appropriate loop statement with a termination condition using the defined tolerance level. Compare your Nassi-Schneiderman diagram with the diagram in Fig. 3.1.

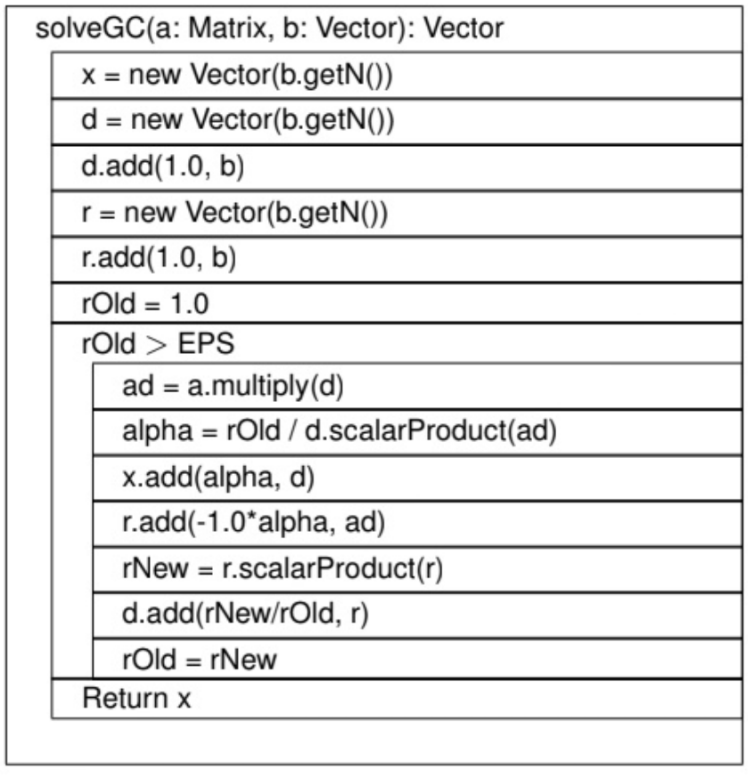


Figure 3.1: Nassi-Sheidermann diagram for a CG solver

Implementation

Implement class `CGSolver` in Java by using your or the given Nassi-Schneiderman diagram. Use the `add`, `scalarProduct`, and `multiply` methods of the classes `Vector` and `Matrix`. Develop some test cases (test programs, test units) to verify the results of the implemented conjugate gradient algorithm. As an example, note that the solution of $\begin{bmatrix} 4 & 1 \\ 1 & 3 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ is $\mathbf{x}^* = [1/11, 7/11]^T$.

In mathematics, the **conjugate gradient method** is an algorithm for the numerical solution of particular systems of linear equations, namely those whose matrix is positive-definite. The conjugate gradient method is often implemented as an iterative algorithm, applicable to sparse systems that are too large to be handled by a direct implementation or other direct methods such as the Cholesky decomposition. Large sparse systems often arise when numerically solving partial differential equations or optimization problems.

(in red) for minimizing a quadratic function associated with a given linear system. Conjugate gradient, assuming exact arithmetic, converges in at most *n* steps, where *n* is the size of the matrix of the system (here *n* = 2).

Suppose we want to solve the system of linear equations

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

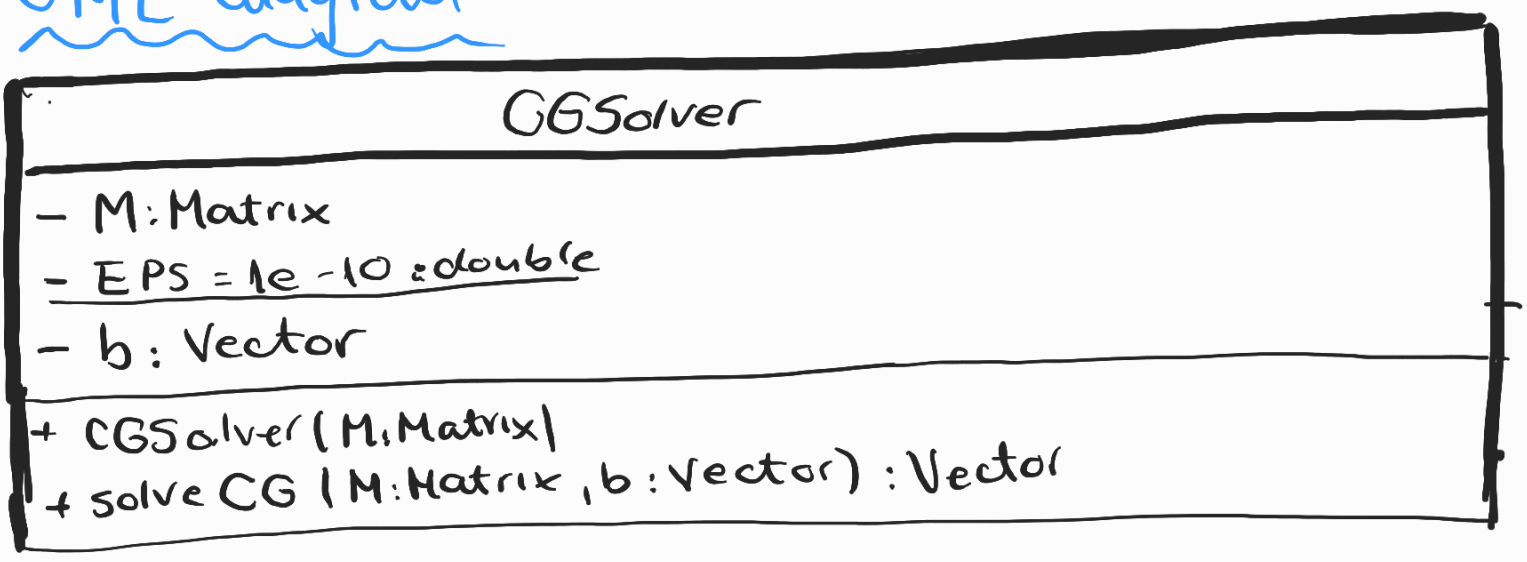
for the vector $\mathbf{x}$, where the known $n \times n$ matrix $\mathbf{A}$ is symmetric (i.e., $\mathbf{A}^{\mathsf{T}} = \mathbf{A}$), positive-definite (i.e. $\mathbf{x}^{\mathsf{T}}\mathbf{A}\mathbf{x} > 0$ for all non-zero vectors $\mathbf{x}$ in $\mathbf{R}^n$), and real, and $\mathbf{b}$ is known as well. We denote the unique solution of this system by $\mathbf{x}_*$.

The algorithm is detailed below for solving **Ax** = **b** where $\mathbf{A}$ is a real, symmetric, positive-definite matrix. The input vector $\mathbf{x}_0$ can be an approximate initial solution or **0**. It is a different formulation of the exact procedure described above.

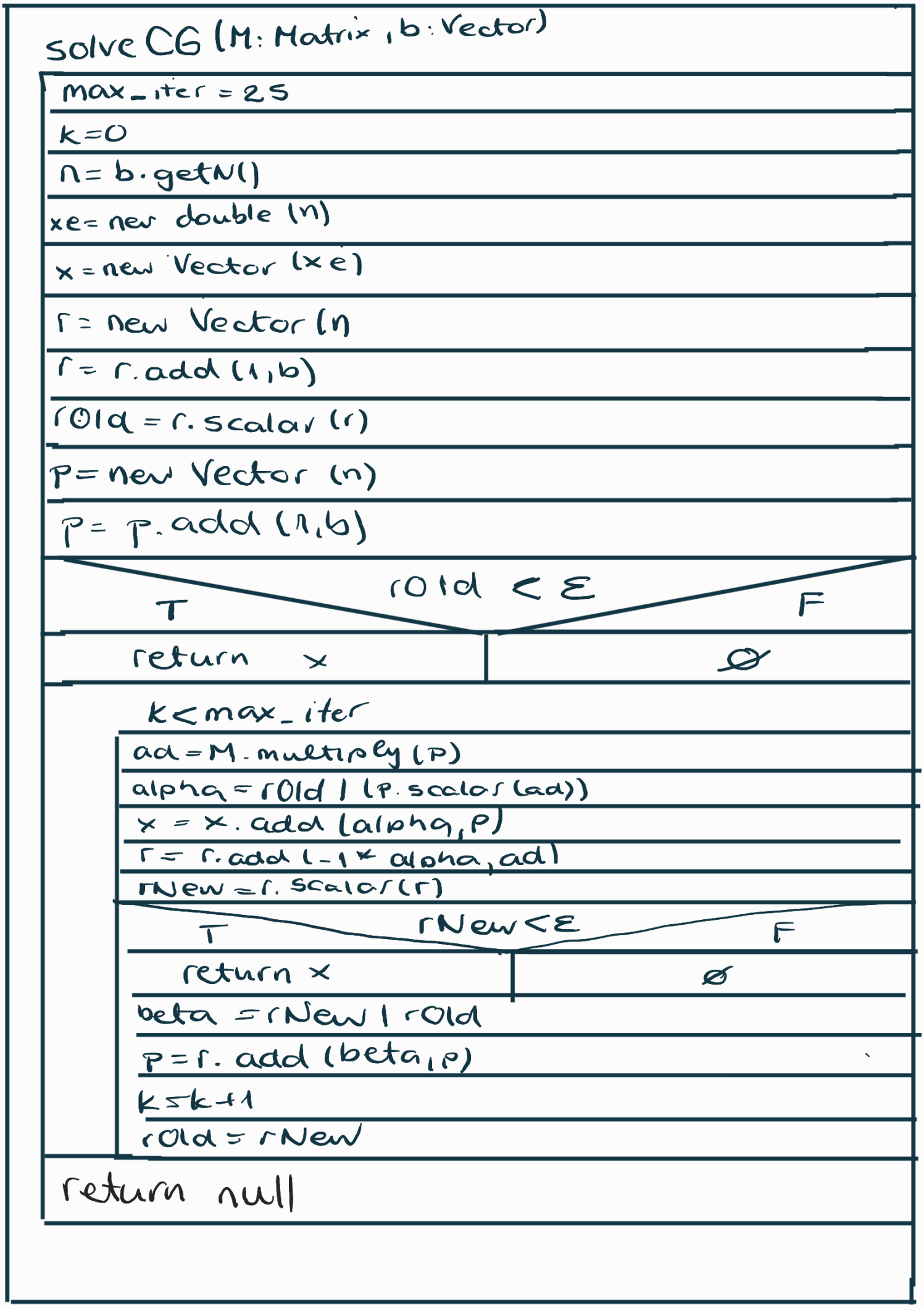
$$\begin{aligned}
&\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0 \\
&\text{if } \mathbf{r}_0 \text{ is sufficiently small, then return } \mathbf{x}_0 \text{ as the result} \\
&\mathbf{p}_0 := \mathbf{r}_0 \\
&k := 0 \\
&\text{repeat} \\
&\quad \alpha_k := \frac{\mathbf{r}_k^{\mathsf{T}} \mathbf{r}_k}{\mathbf{p}_k^{\mathsf{T}} \mathbf{A} \mathbf{p}_k} \\
&\quad \mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k \\
&\quad \mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{A} \mathbf{p}_k \\
&\quad \text{if } \mathbf{r}_{k+1} \text{ is sufficiently small, then exit loop} \\
&\quad \beta_k := \frac{\mathbf{r}_{k+1}^{\mathsf{T}} \mathbf{r}_{k+1}}{\mathbf{r}_k^{\mathsf{T}} \mathbf{r}_k} \\
&\quad \mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k \\
&\quad k := k + 1 \\
&\text{end repeat} \\
&\text{return } \mathbf{x}_{k+1} \text{ as the result}
\end{aligned}$$

This is the most commonly used algorithm. The same formula for β_k is also used in the Fletcher–Reeves nonlinear conjugate gradient method.

UML diagram


CGSolver
- M: Matrix
- EPS = 1e-10 : double
- b : Vector
+ CGSolver(M: Matrix)
+ solveCG(M: Matrix, b: Vector) : Vector

Nassi - Shneiderman



Java code

```
public class CGSolver {
    private Matrix M;
    private final static double EPS = 1e-10;
    private Vector b;

    public CGSolver (Matrix M) {    this.M=M;   }
```

```
public Vector solveCG (Matrix M, Vector b) {

    int max_iter = 25;
    int k=0;
    int n = b.getN();

    //Creating the solution vector x
    double [] xe = new double [n];

    //Initializing x
    Vector x = new Vector (n);

    // If we use following expressions for r and p it is the same as if we set the initial x, i.e. x0 to be all zeros
    //Initializing the first r vector
    Vector r = new Vector (n);
    r = r.add(1, b);
    double rOld=r.scalar(r);

    //Initializing the first vector p
    Vector p = new Vector (n);
    p=p.add(1, b);

    //We are checking if x with all zeros is the solution
    if (rOld<EPS) {
        return x;
    }
```

```
//Declaring variables we will use in the while loop
double alpha;
Vector ad;

//Setting up maximum number of iterations
while (k<max_iter ) {

//Finding the vector which is the result of M*p
ad = M.multiply(p);

//Calculating alpha
alpha = rOld/(p.scalar(ad));

//Calculating new x and r
x = x.add(alpha, p);
r=r.add(-1*alpha, ad);

//Checking if the residual r is small enough, i.e. the solution is approximate enough
double rNew=r.scalar(r);

if (rNew<EPS) {
    System.out.println("\n"+"Solution has been found.");
    x.print("Solution is the vector :");
    return x;}
```

```
    // IF the solution has not yet been found, we have to calculate new vector p for our next point/iteration
    double beta = rNew/rOld;
    p = r.add(beta, p);

    //number of iteration increases
    k++;

    //We use our next r for next iteration
    rOld=rNew;
    }
    System.out.println("The solution has not been found for the maximum number of iterations. x from the last iteration is:" + "\n");
    return x;
    }

public static void main (String [] args) {

    double [] [] x = {{0.744, -0.5055, -0.0851},{-0.5055, 3.4858, 0.0572}, {-0.0851,0.0572, 0.4738 }};
    Matrix M = new Matrix (x);
    Vector b = new Vector (-0.0043, 2.2501, 0.2798);
    M.print("Matrica M");
    b.print("b");
    CGSolver A = new CGSolver(M);
    Vector l = A.solveCG(M,b);
}
```

Example 1 (Test case above)

```
Iteration: 1      x = [ 0.0261  1.8702 -2.1522]    residual = 4.3649
Iteration: 2      x = [-0.5372  0.5115 -0.3009]    residual = 0.7490
Iteration: 3      x = [ 0.5488  0.7152  0.6028]    residual = 0.0000

Solution:         x = [ 0.5488  0.7152  0.6028]
```

My result

```
Defined matrix is Matrica M:
|       0.744,          -0.5055,    -0.0851    |
|       -0.5055,              3.4858,    0.0572    |
|       -0.0851,              0.0572,    0.4738    |
b= (-0.0043,2.2501,0.2798)^T
Solution has been found.
Solution is the vector := (0.5491368078415264,0.7152467015189033,0.6028269966661729)^T
```

Example 2:

```
Iteration: 1    x = [ 6.0798  0.2932 -3.6765  1.9076 -2.1460  5.5981]
                residual = 5.2930
Iteration: 2    x = [ 1.5966 -0.3208 -0.2777 -0.5521  1.3176  1.3223]
                residual = 1.1933
Iteration: 3    x = [ 0.6074  0.2616  0.2860  0.7581  0.8068  0.6812]
                residual = 0.3795
Iteration: 4    x = [ 0.5343  0.6703  0.6294  0.5496  0.4109  0.6765]
                residual = 0.0346
Iteration: 5    x = [ 0.5477  0.7172  0.6058  0.5464  0.4280  0.6519]
                residual = 0.0047
Iteration: 6    x = [ 0.5488  0.7152  0.6028  0.5449  0.4237  0.6459]
                residual = 0.0000

Solution:       x = [ 0.5488  0.7152  0.6028  0.5449  0.4237  0.6459]
The solution is correct.
```

Text case

```
    public static void main (String [] args) {

        double [] [] x = {{ 3.4430, -0.3963,  2.5012,  0.9525,  0.6084, -1.2728},
                          {-0.3963,  0.6015, -0.4108, -0.1359, -0.0295,  0.2630},
                          {2.5012, -0.4108,  2.5927,  0.7072,  0.5587, -1.0613},
                          {0.9525, -0.1359,  0.7072,  1.1634,  0.1920, -0.4344},
                          {0.6084, -0.0295,  0.5587,  0.1920,  0.7636, -0.3261},
                          {-1.2728,  0.2630, -1.0613, -0.4344, -0.3261,  1.0869}};
        Matrix M = new Matrix (x);
        Vector b = new Vector (3.0685,  0.0484,  2.5783,  1.2865,  0.8671, -0.8230);
        M.print("Matrica M");
        b.print("b");
        CGSolver A = new CGSolver(M);
        Vector l = A.solveCG(M,b);
    }
}
```

My results:

Problems @ Javadoc Declaration Console Call Hierarchy

<terminated> CGSolver [Java Application] C:\Users\User\Desktop\Programming\Eclipse\jre1.8.0_201\jre1.8.0_201\jre1.8.0_201\bin\javaw.exe (8. dec 2022. 19:32:13 – 19:32:34) [pid: 6364]

```
Defined matrix is Matrica M:
|       3.443,          -0.3963,    2.5012,    0.9525,    0.6084,    -1.2728    |
|       -0.3963,                0.6015,    -0.4108,    -0.1359,    -0.0295,    0.263    |
|       2.5012,         -0.4108,    2.5927,    0.7072,    0.5587,    -1.0613    |
|       0.9525,         -0.1359,    0.7072,    1.1634,    0.192,    -0.4344    |
|       0.6084,         -0.0295,    0.5587,    0.192,    0.7636,    -0.3261    |
|       -1.2728,                0.263,    -1.0613,    -0.4344,    -0.3261,    1.0869    |
b= (3.0685,0.0484,2.5783,1.2865,0.8671,-0.823)^T
Solution has been found.
Solution is the vector := (0.5488252073455604,0.7152045853125403,0.6027868107664425,0.5448522879830898,0.4236962375161299,0.6459055453349904)^T
```