

Faculty of Informatics

Department of Information Systems

P170B115 Numerical methods and algorithm Laboratory work 2

**Lecturer:** **Assoc. prof.  Andrius Kriščiūnas**

**Students: Arsenii Ziubin**

KAUNAS, 2025

**Part 1**

**Task**

Implement Simple Iteration, QR and Gaussian methods to solve follows systems of linear equations:

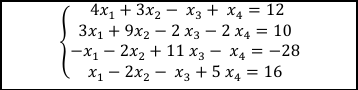


Figure 1 First system of linear equations

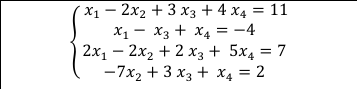


Figure 2 Second system of linear equations

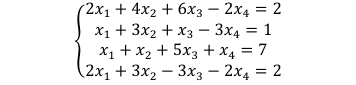


Figure 3 Third system of linear equations

**Simple Iteration method**

**Code of the method**

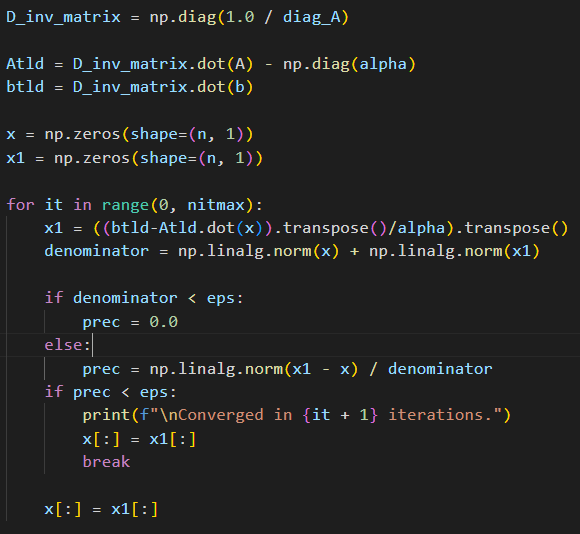


Figure 4 Simple Iteration method code

This code implements the Simple Iteration method, an iterative algorithm for solving Ax = b.First, the setup stage (D\_inv\_matrix, Atld, btld) rearranges the system into an iterative form, which allows calculating a new guess for x from an old one.Then, the iteration loop starts with an initial guess (x). In each step, it calculates a new, refined guess (x1) and measures the relative difference (prec) between the new and old guesses. If this difference is smaller than a tiny tolerance (eps), the solution has converged, and the loop stops. If not, the new guess becomes the current guess, and the process repeats until convergence or the maximum iteration limit (nitmax) is reached.

**First system of linear equations**

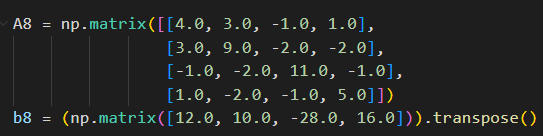
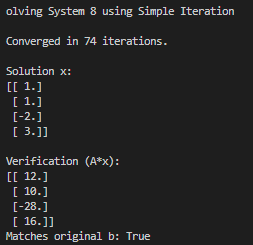


Figure 8

Results



**Second system of linear equations**

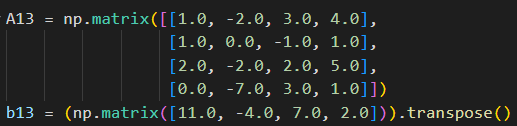
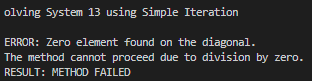


Figure 13

Results



**Third system of linear equations**

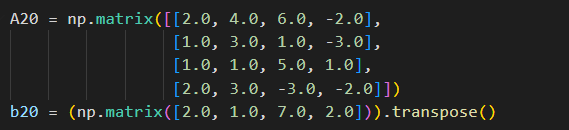
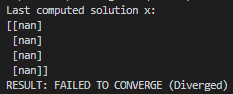


Figure 20

Results



**Gaussian method**

**Code of the method**

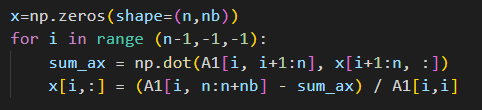
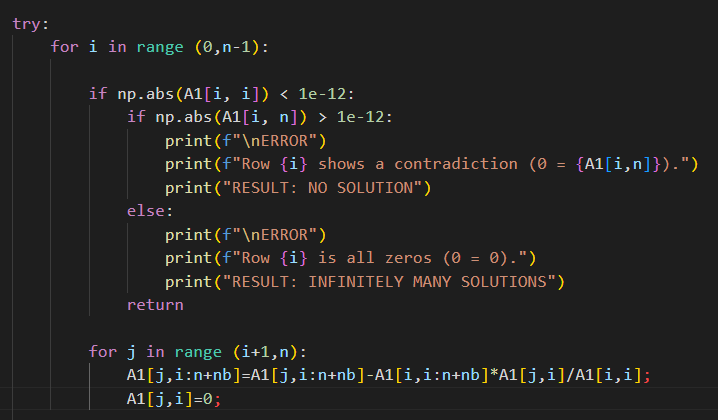


Figure 9 Gaussian method

This function solves the system Ax = b using Gaussian Elimination. It first performs forward elimination by creating an augmented matrix A1 (combining A and b) and applying row operations to transform it into an upper triangular form. During this process, it strictly checks pivot elements. If a zero pivot is found, it determines if there is NO SOLUTION (a contradiction like 0x = 5) or INFINITELY MANY SOLUTIONS (a redundancy like 0x = 0).If a unique solution exists, the code performs Backward Substitution, iterating from the last row upwards. It solves for each xi using the already-computed values from the rows below. Finally, it prints the solution x and verifies it.

Results

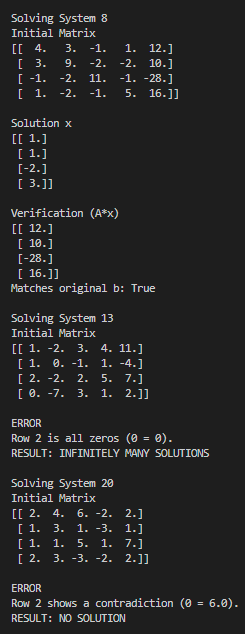
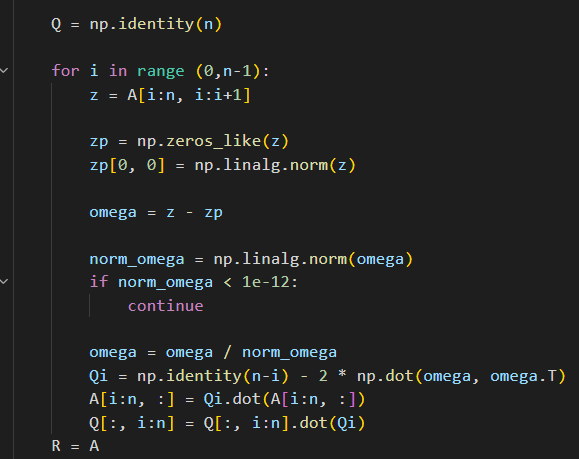


Figure 11 Results for first, second and third matrices

We can see from the result that the 1 system was solved correctly, second has infinite solutions and third has no solutions because of the division by 0.

**QR method**

**Code of the method**



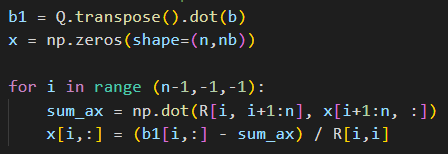
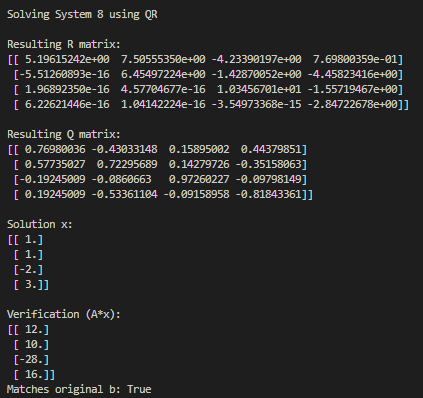
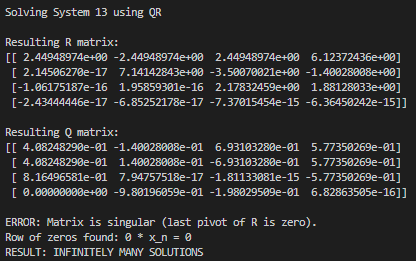


Figure 12 QR

This function solves Ax = b using QR Decomposition. It iteratively decomposes the matrix A into an orthogonal matrix Q and an upper-triangular matrix R. This transforms the original system into the equivalent, easily solvable system Rx = Q^Tb. The code then performs Backward Substitution on this new system to find the solution vector x and includes a singularity check to handle non-unique solutions.

Results:





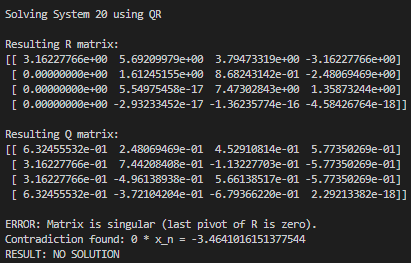


Figure 14 Results for first, second and third soles

From the result we see that first matrix was correctly calculated, second has infinite number of solutions and third has no solution because includes zeros.

**Part 2**

**Task**

Implement the Newton method to solve the system of non-linear equations.

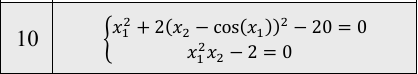


Figure 15 System of non-linear equations

**Graphical solution**

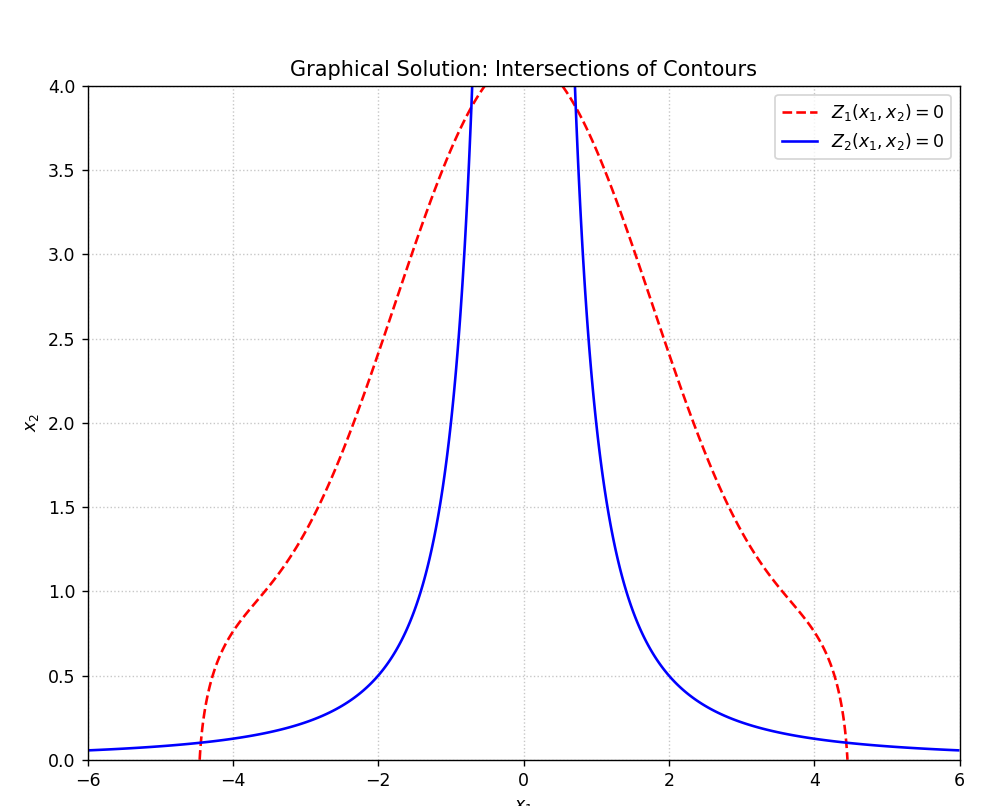


Figure 16 Graphical solution for system of non-linear equations

Red line: first equation, Blue line: second equation.  
the results are   
[-4.44160772 0.10137937]

[-0.71851577 3.8739801 ]

[4.44160772 0.10137937]

[0.71851577 3.87398007]

**Mesh of initial guesses**

To draw the mesh of initial guesses I have used Newton Method.

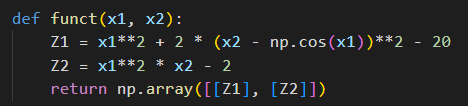
****

Figure 17 Code of system of non-linear equations

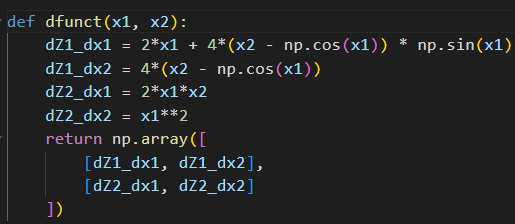
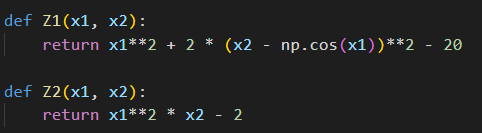


Figure 18 Code of Jacobian matrix



**Figure 19**

I divided the system into two separate functions, Z1 and Z2, for plotting the 2D contour graphs.

The Newton-Raphson method is an approach to find the solutions (or "roots") of an equation. It starts with an initial guess. It then looks at the function's slope at that guess to find a new, better guess that should be closer to the actual solution. It repeats this process, getting progressively closer with each step, until the guess is so good that the function's value is almost zero.

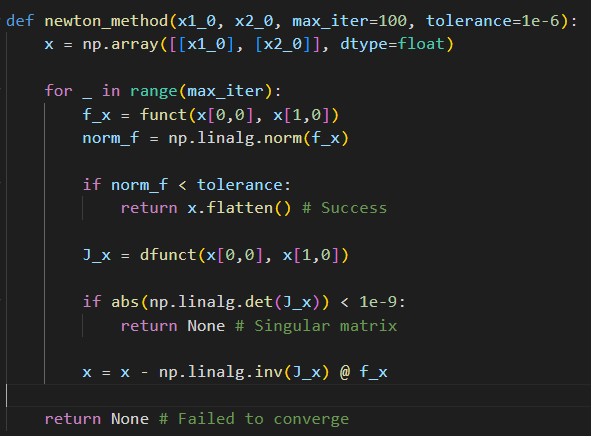


Figure 20 Code of the Newton method

Inside the loop, it first calls funct to check the current error—that is, how far the guess is from being a solution. If this error is smaller than the tolerance, the guess is considered a success and is returned.

If the error is still too large, the function calls dfunct to get the Jacobian matrix, which is the system's equivalent of a slope. It checks if this Jacobian is valid (not singular). If it isn't, the method can't continue and stops.

If the Jacobian is valid, the code uses it along with the current error to calculate the next, improved guess. This process repeats until a solution is found or the loop hits its maximum iterations, in which case it reports failure.



Figure 21 Mesh of initial guesses code

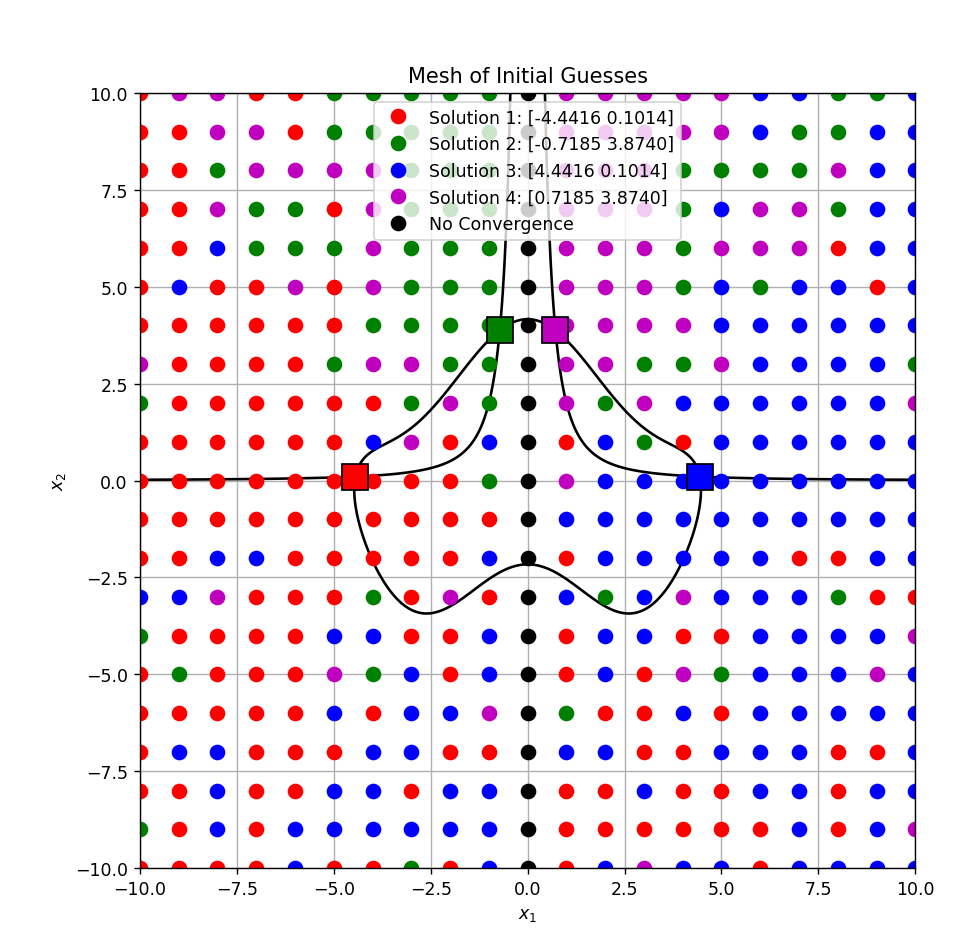


Figure 22 Mesh of initial guesses

From the picture 22 we can see that Newton method correctly found 4 different solutions to the system. The black points show where Newton’s method could not find a solution, meaning the method did not converge for those values.

**Calculated solutions of the system in a table**

**A screenshot of a computer screen

AI-generated content may be incorrect.**

Figure 22 Table with calculated solutions and initial guesses

**Verifying the obtained results using python fsolve()**

Obtained results were verified with python function fsolve() from scipy.optimize library.

A screen shot of a computer code

AI-generated content may be incorrect.

Figure 23 Python code for fsolve() verification

**A screenshot of a computer screen

AI-generated content may be incorrect.**

Figure 24 fsolve() results

**Part 3**

**Task description**

Find the best location for new stores in the city square so that the total cost of building them is as low as possible. The cost of each store depends on two factors: Distance from other stores – stores that are two close increase the cost and distance from the city boundary – stores outside the city cause a boundary penalty. The optimization problem is solved with Gradient Descent method which gradually adjusts the positions of new stores until the total cost becomes minimal.

**Objective Function**

The function of total cost is :

where is the cost based on distances to other stores and

is the cost related to city boundaries.

The distance cost between two stores and is:

and the boundary cost is:

The aim is to minimize the total cost for all new stores.

**Method Used**

The Gradient Descent algorithm was used to find the local minimum of the cost function. In each iteration, the positions of the new stores were updated according to the negative gradient direction:

Where is the step size and is the gradient of the cost function,

The algorithm stops when the chang in total cost becomes smaller than a given tolerance or the maximum number of iterations is reached.

**Method parameters**

Step size : 0.1

Maximum iterations: 500

Tolerance: 1e-6

Number of existing stores n: 8

Number of new sores m: 5

The initial coordinates of stores were generated randomly.

**Results and Visualization**

The figure below shows the initial and final positions of the stores. Blue circles represent existing stores, yellow squares represent the initial positions of new stores, red stars represent the optimized positions after gradient descend and dashed red lines show the

movement of each new store during optimization.

A screen shot of a computer program

AI-generated content may be incorrect.

Figure 25 Python code for plotting store locations

A diagram of a store location

AI-generated content may be incorrect.

Figure 26 Graph of final store locations

The next figure shows the total cost decreased (objective function) with each iteration of the gradient descend method, proving that algorithm successfully minimized the cost.

A screen shot of a computer code

AI-generated content may be incorrect.

Figure 27 Python code for cost history

A graph with a blue line

AI-generated content may be incorrect.

Figure 28 Graph of cost history

From the results, we can conclude that the gradient descent method successfully minimized the total cost of stores, new stores moved away from each other, stayed inside the city boundary and avoided additional penalties. The total cost decreased steadily until convergence, proving that the optimization was effective.