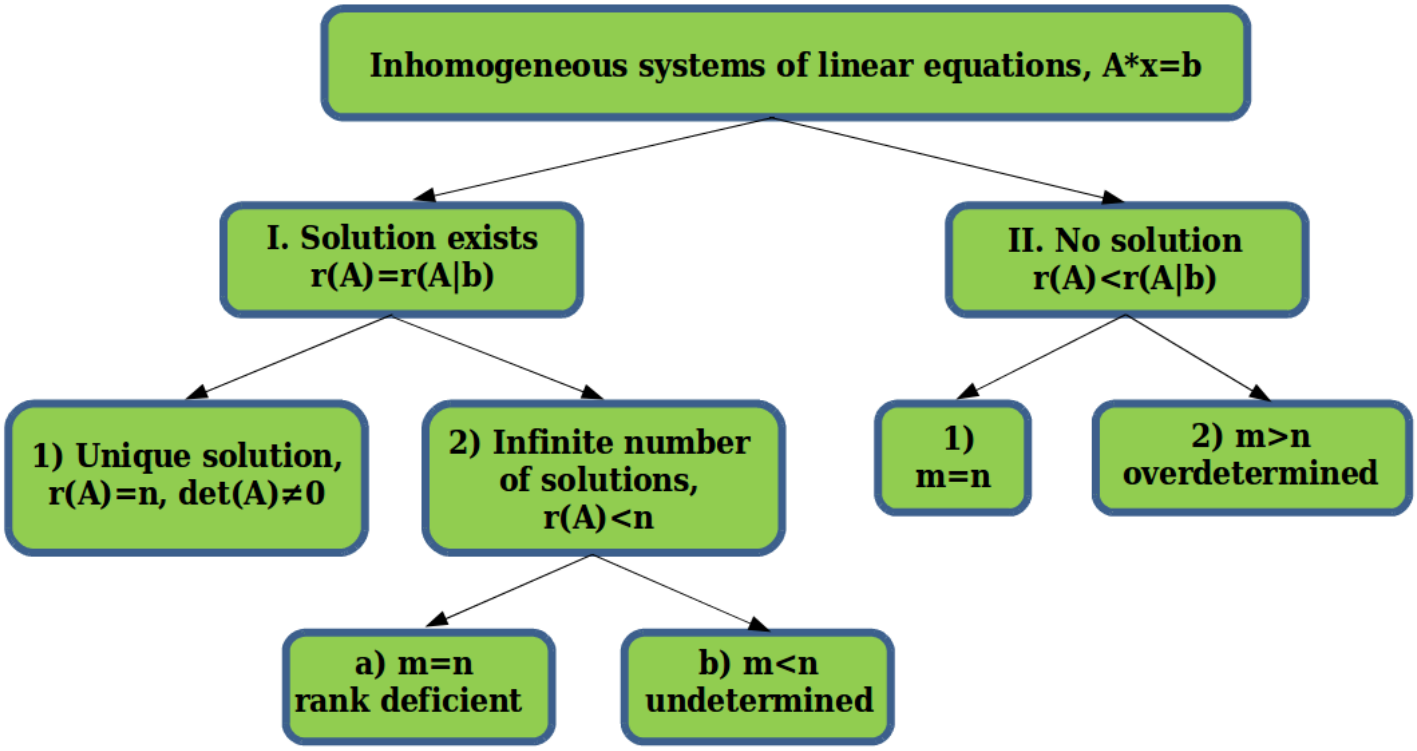
Systems of linear equations II.

During the previous practical, we looked at systems of linear equations which have either one unique solution, or an infinite number of solutions. In the latter case, we can still find a unique solution, namely if we prescribe the constrict of $||\boldsymbol{x}|| \rightarrow \min$, where ***x*** is the solution vector.



As we saw, the computation of the inverse of the coefficient matrix used to solve the system of equations can be very computationally expensive, especially as the size of the matrix increases. Moreover, depending on the algorithm used, the result of the algorithm can be numerically less precise, which in certain cases (for example, when calculating the inverse multiple times) can lead to unintended round-off errors. It is always recommended to use some kind of decomposition or factorization of the coefficient matrix when solving linear systems, however, the LU and Cholesky decompositions that we have studied only work on square matrices (with even more constrictions in the case of the Cholesky decomposition). If our matrix is rectangular, as in the case of underdetermined or overdetermined systems, we may use the QR factorization or the Singular Value Decomposition (SVD).

QR factorization

The QR factorization uses the fact that every matrix ***A***, independent of its shape, can be decomposed into the product of an orthonormal matrix ***Q*** and an upper rectangular matrix ***R***, such that $\boldsymbol{A} = \boldsymbol{Q} \cdot \boldsymbol{R}$. The orthonormality of matrix ***Q*** means that $\boldsymbol{Q}^T\boldsymbol{Q} = \boldsymbol{I}$, that is, the inverse of ***Q*** is equal to its transpose.

The original $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$ system becomes:

$$\boldsymbol{Q} \cdot \boldsymbol{R} \cdot \boldsymbol{x} = \boldsymbol{b}.$$

We can multiply from the left by $\boldsymbol{Q}^T$ (because **Q** is orthonormal), so we don't have to use invertion.

$$\boldsymbol{Q}^T \cdot \boldsymbol{Q} \cdot \boldsymbol{R} \cdot \boldsymbol{x} = \boldsymbol{Q}^T \cdot \boldsymbol{b}$$

$$\boldsymbol{R} \cdot \boldsymbol{x} = \boldsymbol{Q}^T \cdot \boldsymbol{b} = \boldsymbol{B}$$

The second step is solving the $\boldsymbol{R} \cdot \boldsymbol{x} = \boldsymbol{B}$ equation, but because **R** is an upper rectangular matrix, we can simply solve this by back-substitution.

There are multiple algorithms to compute this factorization, such as using the Gram-Schmidt process to find the orthonormal basis of the column vectors of ***A***, the Household-transformation and more. Calculating the QR factorization using the Gram-Schmidt process is the simplest way, however, it tends to be numerically unstable in certain cases and therefore usually other methods are implemented in numerical algorithms.

Let's solve the example from the previous practical:

$$\begin{aligned} 7x_1 + 2x_2 + 2x_4 &= 1 \\ x_1 + 8x_2 + x_3 + 8x_4 &= 2 \end{aligned}$$

$$\boldsymbol{A} = \begin{bmatrix} 7 & 2 & 0 & 2 \\ 1 & 8 & 1 & 8 \end{bmatrix}, \qquad \boldsymbol{b} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Remember that the example has infinitely many solutions as the rank of the coefficient matrix is only 2. As the coefficient matrix is rectangular, we have to use the QR factorization to decompose it.

```
clear all; close all; format short
A = [7, 2, 0, 2; 1, 8, 1, 8]; b = [1; 2];
rA = rank(A) % rank of A
```

```
rA = 2
```

```
rank([A b])
```

```
ans = 2
```

```
xa = A'*inv(A*(A'))*b
```

```
xa = 4×1
    0.0745
    0.1195
    0.0127
    0.1195
```

```
xb = pinv(A)*b
```

```
xb = 4×1
    0.0745
    0.1195
    0.0127
    0.1195
```

```
xc = A\b
```

```
xc = 4×1
    0.0741
    0.2407
         0
         0
```

```
norm(xa), norm(xb), norm(xc)
```

```
ans = 0.1852
ans = 0.1852
ans = 0.2519
```

```
[Q R] = qr(A)
```

```
Q = 2×2
   -0.9899   -0.1414
   -0.1414    0.9899
R = 2×4
   -7.0711   -3.1113   -0.1414   -3.1113
         0    7.6368    0.9899    7.6368
```

```
norm(A-Q*R)    % check the decomposition
```

```
ans = 8.8818e-16
```

```
Q'*Q     % Q is orthonormal if the Q'*Q product is an identity matrix
```

```
ans = 2×2
    1.0000    0.0000
    0.0000    1.0000
```

Calculate the new right side of the system:

```
B=Q'*b
```

```
B = 2×1
   -1.2728
    1.8385
```

Solve the second step using the fact that ***R*** is an upper rectangular matrix:

```
opt.UT=true;
x=linsolve(R,B,opt)
```

```
x = 4×1
    0.0741
    0.2407
         0
         0
```

```
norm(A*x-b)
```

```
ans = 0
```

The solution is correct, but its norm is not the same as the norm of the solution from the end of the previous practical (0.1852). The solution by QR factorization does not give us the solution with the least norm, rather it

gives us the solution with the most zero entries. Depending on the concrete problem, one or the other might be more suitable.

Singluar Value Decomposition (SVD)

The SVD is another form of decomposition that can be used arbitrarily shaped matrices. It is very similar to the eigendecomposition which is used in the case of square matrices:

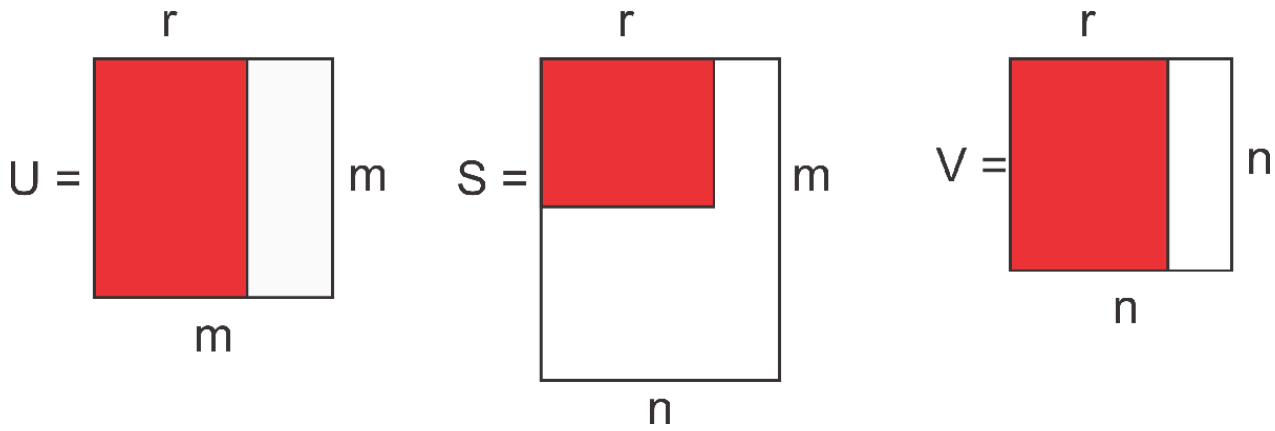
- Eigenvalues: matrix ***A*** (with size m x m) has an eigenvalue λ is we can find a vector ***v*** ($v = 0$) such, that $A \cdot v = \lambda \cdot v$. In this case, the vector ***v*** is called an eigenvector of ***A*** corresponding to the eigenvalue λ. The eigenvalues can be found using the characteristic polinomial of the matrix $|A - \lambda I| = 0$. In MATLAB, we can use the `eig` function: `[V, D] = eig(A)`.
- Singular values: the singular values of a matrix ***A*** (with size m x n) are the squares of the non-zero eigenvalues of the product $A^T A$ ($\sigma_i = \sqrt{\lambda_i}$, where σ_i is the i-th singular value and λ_i is the i-th eigenvalue of the product $A^T A$). In MATLAB, we can compute the SVD by using the `svd` function: `[U, S, V] = svd(A)`.

The SVD decomposes the matrix ***A*** (m x n) into the product of three matrices:

$$A = U \cdot S \cdot V^T$$

where:

- ***U*** is an orthonormal matrix ($U^T U = I$) with size m x m, the first *r* columns of which are the eigenvectors of the product AA^T.
- ***S*** is a diagonal matrix with size m x n that contains the singular values of ***A*** on its diagonal.
- ***V*** is a orthonormal matrix ($V^T V = I$) with size n x n, the first *r* columns of which are the eigenvectors of the product $A^T A$.



When using the SVD of a matrix to solve a linear system of equations, the great thing about it is that the inverse of matrix ***A*** can be very easily computed because of the structure of the resulting matrices:

$$A^{-1} = U^{-1} \cdot S^{-1} \cdot (V^T)^{-1}$$

Because of the orthogonality of ***U*** and *V*:

$$A^{-1} = U^T \cdot S^{-1} \cdot V$$, which can be rewritten as $$A^{-1} = V \cdot S^{-1} \cdot U^T$$.

We can see, that only the matrix ***S*** has to be inverted, which is really simple as ***S*** is a diagonal matrix:

$$S = \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \sigma_r & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \rightarrow S^{-1} = \begin{bmatrix} \frac{1}{\sigma_1} & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \frac{1}{\sigma_2} & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \frac{1}{\sigma_r} & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Let's solve the previous example using the SVD:

```
[U,S,V] = svd(A)
```

```
U = 2×2
   -0.3979    0.9174
   -0.9174   -0.3979
S = 2×4
   12.1209         0         0         0
         0    6.3312         0         0
V = 4×4
   -0.3055    0.9515    0.0368   -0.0015
   -0.6712   -0.2130   -0.0933   -0.7039
   -0.0757   -0.0629    0.9943   -0.0406
   -0.6712   -0.2130   -0.0356    0.7092
```

```
norm(A-U*S*V') % checking the decomposition
```

```
ans = 1.4589e-15
```

Check the resulting properties of the matrices:

```
norm(U'*U-eye(size(A,1))) % U mxm orthonormal matrix, U'=inv(U)
```

```
ans = 4.5347e-16
```

```
norm(V'*V-eye(size(A,2))) % V nxn orthonormal matrix, V'=inv(V)
```

```
ans = 5.3885e-16
```

```
diag(S), sqrt(eig(A'*A))  % singular values are the eigenvalues oft he matrix A'*A
```

```
ans = 2×1
   12.1209
    6.3312
ans = 4×1 complex
   0.0000 + 0.0000i
   0.0000 + 0.0000i
   6.3312 + 0.0000i
```

```
  12.1209 + 0.0000i
```

Compute the inverse using the results of the SVD:

```
invS=(1./S)' % inverse of S
```

```
invS = 4×2
    0.0825       Inf
       Inf    0.1579
       Inf       Inf
       Inf       Inf
```

```
invS(invS==Inf)=0 % where there's an Inf element we exchange it to 0
```

```
invS = 4×2
    0.0825         0
         0    0.1579
         0         0
         0         0
```

```
invS*S % check
```

```
ans = 4×4
    1.0000         0         0         0
         0    1.0000         0         0
         0         0         0         0
         0         0         0         0
```

```
invA=V*invS*U' % pseudoinverse of matrix A
```

```
invA = 4×2
    0.1479   -0.0367
   -0.0088    0.0642
   -0.0066    0.0097
   -0.0088    0.0642
```

```
invA*A % check
```

```
ans = 4×4
    0.9986    0.0024   -0.0367    0.0024
    0.0024    0.4958    0.0642    0.4958
   -0.0367    0.0642    0.0097    0.0642
    0.0024    0.4958    0.0642    0.4958
```

Solve the system of equations:

```
x=invA*b
```

```
x = 4×1
    0.0745
    0.1195
    0.0127
    0.1195
```

```
norm(A*x-b)
```

```
ans = 4.9651e-16
```

The norm of the solution is the same as norm from the last practical, in other words, the SVD solution calculates the least norm solution as well.

The inverse matrix calculated with the SVD is also called the pseudo-inverse or generalized inverse. When the matrix is not singular ($\det(\boldsymbol{A}) \neq 0$), the generalized inverse is the same as the normal inverse, but in the case of a singular matrix ($\det(\boldsymbol{A}) = 0$), only the pseudo-inverse can be calculated.

Apart from solving systems of linear equations, the SVD is a widely used method in applied mathematics, computer vision, statistics and signal analysis. Some of its applications include data approximation (when working with very big matrices), dimensionality reduction, trend analysis and forecasting (recommendation services for companies like Netflix, Amazon etc.).

Built-in functions for solving systems of linear equations with rectangular coefficient matrix

The built-in `pinv` command uses the SVD of the matrix to calculate its generalized inverse when the matrix is rectangular. If the matrix is square and non-singular, the `pinv` function shouldn't be used to calculate the inverse (it is computationally expensive).

```
A\b
```

```
ans = 4×1
    0.0741
    0.2407
         0
         0
```

```
pinv(A)*b
```

```
ans = 4×1
    0.0745
    0.1195
    0.0127
    0.1195
```

The results are the same as from the 'manual' calculation.

The other built-in functions for solving systems of linear equations:

- The `\` operator (`mldivide`) uses the LU or the Cholesky decomposition for square matrices and the QR factorization for rectangular matrices.
- The `linsolve` uses the same methods as the `\` operator, but it can be parametrized using preliminary knowledge about the structure of the matrix (upper/lower triangular nature).

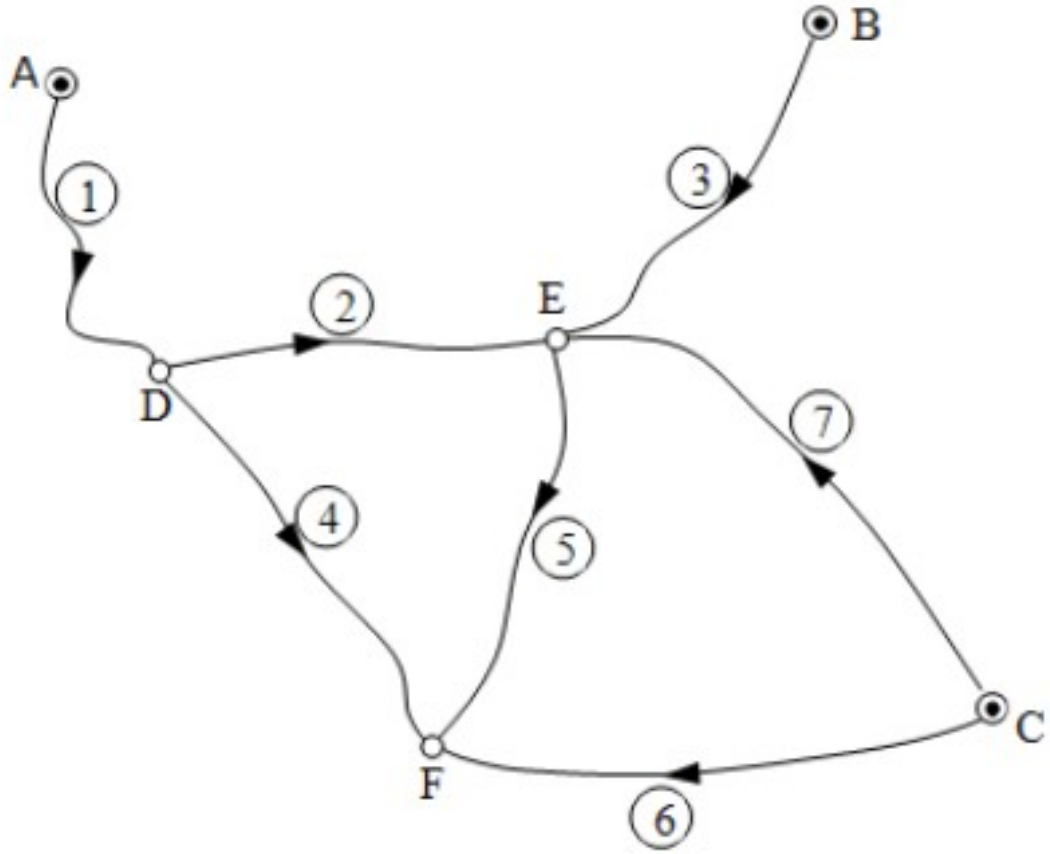
Overdetermined System (no solution exists)

In most engineering problems, we are using some kind of measurement and calculating parameters values using some form of mathematical model of the given physical phenomenon. It often happens that in order to minimize the effect of measurements errors (at least those, that cannot be eliminated by choosing a specific configuration), we make more measurements than there are unknowns in the mathematical model. If the model

is linear, then what we end up with is an overdetermined system of linear equations where we have more equations than unknowns.

As the measurements are never perfect and there are always random fluctuations in the values, we have very little hope of finding an exact solution. However, we can still find a solution which minimizes the sum of the squares of the discrepancies ($\sum_i e_i^2 \to \min$, where e_i is difference between the i-th unknown computed value and its theoretical true value (also called error). This is more commonly called the least-squares solution.

Take a look at an example from surveying. In surveying we almost always have more measurements than what is absolutely necessary, in order to compensate for measurements errors and eliminate blunders. Consider the following levelling network:



We measured the height differences along levelling lines 1-7. We also have three benchmarks (points with known elevations) in the area, A, B and C. Our goal is to calculate the elevation of points D, E and F using all the available measurements. As we have 7 measurements and 3 uknown elevations (m > n in matrix ***A***), we can only find an optimal solution when the sum of the squared errors of the elevations are minimalized. (The arrows in the figure show the direction of rise in the lines.)

Our system of equations looks like the following:

$$
\begin{aligned}
&\text{1st line}: && H_D - H_A = H_D - 183.506 = +6.135\\
&\text{2nd line}: && H_E - H_D = +8.343\\
&\text{3rd line}: && H_E - H_B = H_E - 192.353 = +5.614\\
&\text{4th line}: && H_F - H_D = +1.394\\
&\text{5th line}: && H_F - H_E = -6.969\\
&\text{6th line}: && H_F - H_C = H_F - 191.880 = -0.930\\
&\text{7th line}: && H_E - H_C = H_E - 191.880 = +6.078
\end{aligned}
$$

In matrix form:

$$
\boldsymbol{A} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \boldsymbol{x} = \begin{bmatrix} H_D \\ H_E \\ H_F \end{bmatrix} \quad \boldsymbol{b} = \begin{bmatrix} 6.135 + 183.506 \\ 8.343 \\ 5.614 + 192.353 \\ 1.394 \\ -6.969 \\ -0.930 + 191.880 \\ 6.078 + 191.880 \end{bmatrix} = \begin{bmatrix} 189.641 \\ 8.343 \\ 197.967 \\ 1.394 \\ -6.969 \\ 190.950 \\ 197.958 \end{bmatrix}
$$

Instead of typing the matrix and the vector into MATLAB, we can use the already existing `levelling.txt` file that contains the values:

```
clear all
Ab = load('levelling.txt'), A = Ab(:,1:end-1), b = Ab(:,end)
```

```
Ab = 7×4
    1.0000         0         0  189.6410
   -1.0000    1.0000         0    8.3430
         0    1.0000         0  197.9670
   -1.0000         0    1.0000    1.3940
         0   -1.0000    1.0000   -6.9690
         0         0    1.0000  190.9500
         0    1.0000         0  197.9580
A = 7×3
     1     0     0
    -1     1     0
     0     1     0
    -1     0     1
     0    -1     1
     0     0     1
     0     1     0
b = 7×1
  189.6410
    8.3430
  197.9670
    1.3940
   -6.9690
  190.9500
  197.9580
```

Check the existence of the solution:

```
rank(A), rank(Ab) % 5, 5
```

```
ans = 3
```

```
ans = 4
```

The rank of ***A*** is smaller than the rank of the augmented matrix, which means that no exact solution exits. As we have an overdetermined system, what we can be looking for, is the solution with least error, that is, the least-squares solution. Mathematically this is a solution with the constraint $\|\boldsymbol{A} \cdot \boldsymbol{x} - \boldsymbol{b}\| \rightarrow \min$. It can be derived, that this solution is the following:

$$\boldsymbol{x} = (\boldsymbol{A}^T \cdot \boldsymbol{A})^{-1} \cdot \boldsymbol{A}^T \cdot \boldsymbol{b}$$

In MATLAB:

```
xa = inv(A'*A)*A'*b
```

```
xa = 3×1
  189.6153
  197.9588
  190.9830
```

```
norm(A*xa-b)
```

```
ans = 0.0506
```

As calculating the inverse is computationally expensive, this method is not commonly used in numerical solutions. We can instead use the QR factorization and the SVD decomposition:

Solution using the QR decomposition:

```
[Q R] = qr(A)
```

```
Q = 7×7
   -0.5774   -0.1741   -0.3077    0.6243    0.2670   -0.0164   -0.2835
    0.5774   -0.3482    0.0615    0.0684    0.5971    0.2659   -0.3312
         0   -0.5222   -0.2462   -0.3761   -0.1927   -0.5809   -0.3882
    0.5774    0.1741   -0.3693    0.5559   -0.3301   -0.2823    0.0478
         0    0.5222   -0.4308   -0.2879    0.5782   -0.3442    0.0776
         0         0   -0.6770   -0.2680   -0.2481    0.6265   -0.1254
         0   -0.5222   -0.2462    0.0198    0.1737   -0.0292    0.7970
R = 7×3
   -1.7321    0.5774    0.5774
         0   -1.9149    0.6963
         0         0   -1.4771
         0         0         0
         0         0         0
         0         0         0
         0         0         0
```

```
norm(A-Q*R)    % check the decomposition
```

```
ans = 7.0008e-16
```

```
Q'*Q     % Q is orthonormal if the Q'*Q product is an identity matrix
```

```
ans = 7×7
    1.0000         0   -0.0000   -0.0000   -0.0000   -0.0000         0
         0    1.0000    0.0000   -0.0000   -0.0000   -0.0000    0.0000
   -0.0000    0.0000    1.0000    0.0000    0.0000    0.0000    0.0000
   -0.0000   -0.0000    0.0000    1.0000    0.0000    0.0000   -0.0000
   -0.0000   -0.0000    0.0000    0.0000    1.0000    0.0000   -0.0000
```

```
   -0.0000   -0.0000    0.0000    0.0000    0.0000    1.0000    0.0000
         0    0.0000    0.0000   -0.0000   -0.0000    0.0000    1.0000
```

```
B=Q'*b
```

```
B = 7×1
 -103.8676
 -246.0788
 -282.1006
    0.0344
    0.0083
   -0.0357
   -0.0050
```

```
opt.UT=true;
x=linsolve(R,B,opt)
```

```
x = 3×1
  189.6153
  197.9588
  190.9830
```

```
norm(A*x-b)
```

```
ans = 0.0506
```

The heights of the unknown points are:

- $H_D = 189.6153\,\mathrm{m}$
- $H_E = 197.9588\,\mathrm{m}$
- $H_F = 190.9830\,\mathrm{m}$

This means that the errors (random or otherwise) in our measurements only permit us to have a solution where the sum of the squared errors in the solution is around 5 cm. If we had to submit our results, we couldn't really vouch for a precision of millimeters, as far as the heights of the unknown points are considered.

Solution using SVD:

```
[U,S,V] = svd(A)
```

```
U = 7×7
   -0.1494   -0.3536    0.5577    0.6243    0.2670   -0.0164   -0.2835
    0.5577    0.3536   -0.1494    0.0684    0.5971    0.2659   -0.3312
    0.4082   -0.0000    0.4082   -0.3761   -0.1927   -0.5809   -0.3882
    0.0000    0.7071    0.0000    0.5559   -0.3301   -0.2823    0.0478
   -0.5577    0.3536    0.1494   -0.2879    0.5782   -0.3442    0.0776
   -0.1494    0.3536    0.5577   -0.2680   -0.2481    0.6265   -0.1254
    0.4082   -0.0000    0.4082    0.0198    0.1737   -0.0292    0.7970
S = 7×3
    2.1753         0         0
         0    2.0000         0
         0         0    1.1260
         0         0         0
         0         0         0
         0         0         0
         0         0         0
V = 3×3
   -0.3251   -0.7071    0.6280
```

```
    0.8881   -0.0000    0.4597
   -0.3251    0.7071    0.6280
```

```
norm(A-U*S*V') % checking the decomposition
```

```
ans = 1.1752e-15
```

```
norm(U'*U-eye(size(A,1))) % U mxm orthonormal matrix, U'=inv(U)
```

```
ans = 1.0213e-15
```

```
norm(V'*V-eye(size(A,2))) % V nxn orthonormal matrix, V'=inv(V)
```

```
ans = 5.9396e-16
```

```
diag(S), sqrt(eig(A'*A))  % singular values are the eigenvalues oft he matrix A'*A
```

```
ans = 3×1
    2.1753
    2.0000
    1.1260
ans = 3×1
    1.1260
    2.0000
    2.1753
```

```
invS=(1./S)' % inverse of S
```

```
invS = 3×7
    0.4597       Inf       Inf       Inf       Inf       Inf       Inf
       Inf    0.5000       Inf       Inf       Inf       Inf       Inf
       Inf       Inf    0.8881       Inf       Inf       Inf       Inf
```

```
invS(invS==Inf)=0 % where there's an Inf element we exchange it to 0
```

```
invS = 3×7
    0.4597         0         0         0         0         0         0
         0    0.5000         0         0         0         0         0
         0         0    0.8881         0         0         0         0
```

```
invS*S % check
```

```
ans = 3×3
     1     0     0
     0     1     0
     0     0     1
```

```
invA=V*invS*U' % pseudoinverse of matrix A
```

```
invA = 3×7
    0.4583   -0.2917    0.1667   -0.2500    0.0417    0.2083    0.1667
    0.1667    0.1667    0.3333    0.0000   -0.1667    0.1667    0.3333
    0.2083   -0.0417    0.1667    0.2500    0.2917    0.4583    0.1667
```

```
invA*A % check
```

```
ans = 3×3
    1.0000    0.0000    0.0000
   -0.0000    1.0000   -0.0000
   -0.0000   -0.0000    1.0000
```

```
x=invA*b
```

```
x = 3×1
  189.6153
  197.9588
  190.9830
```

```
norm(A*x-b)
```

```
ans = 0.0506
```

The solution and the error are exactly the same, the difference can be found in the computational time. If we had to solve the problem for example a 1000 times, the QR factorization would be reasonably faster.

Solution using built-in functions

Let's solve the system 10.000 times using the \ operator and the SVD and time the results:

```

```

Solution using the SVD:

```

```

We can see that the time it takes to find the solution using `pinv` is significantly longer (and gets even worse with larger matrices).

Checking the solutions:

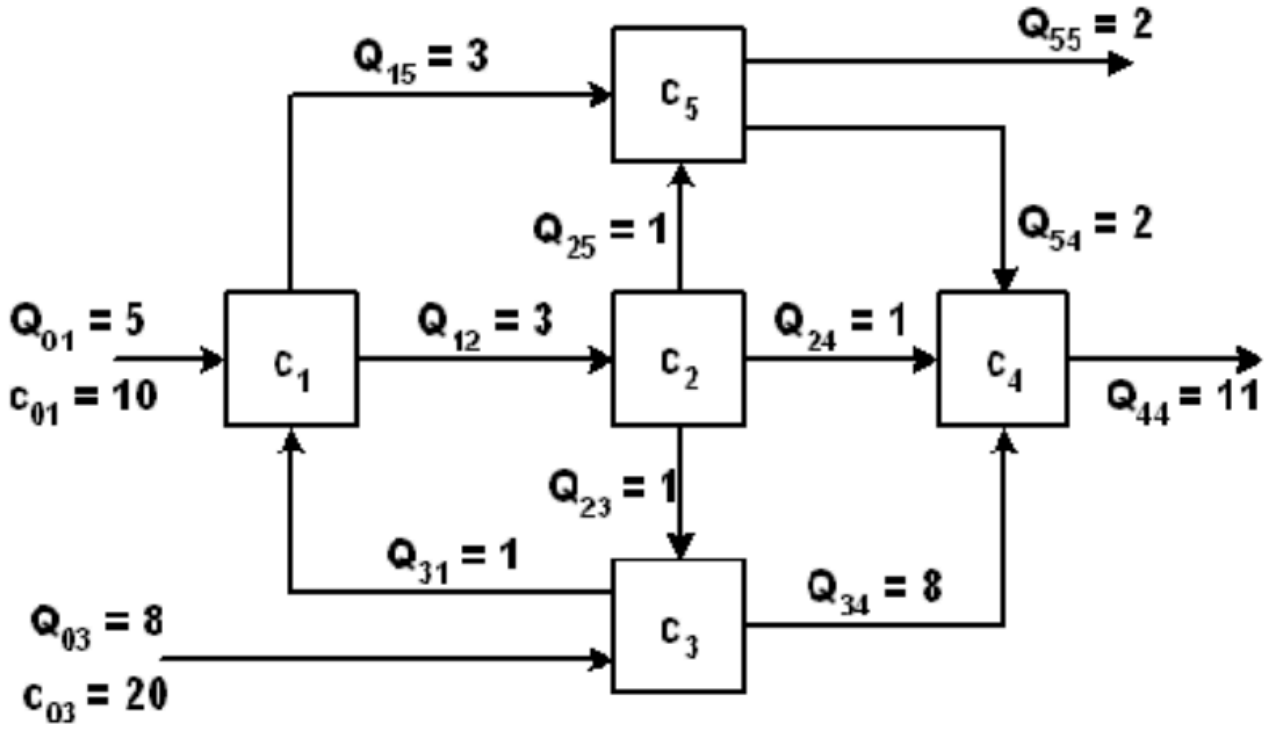
```

```

Both solutions give the same errors of course.

Iterative methods of solving systems of linear equations

Apart from direct solutions, we can also employ iterative methods to solve systems of linear equations. In many cases, especially with sparse coefficient matrices, these solutions can be significantly more efficient. Let's see an example for these types of solutions:

We have a system of 5 reactors and we want to determine the amount of concentration in each reactor. The figure of the connection of the reactors is the following:



As a simplification, the rate of amalgamation in the reactors is considered perfect (the concentration is perfectly equal in every part of the reactor). We have volumetric flow (*Q*) into and out of every reactor and as the amalgamation is perfect, we can use the fact that the sum of the mass flow (concentration times volumetric flow) into a reactor and out of a reactor is zero. We will use the convention that the outflow is positive. Consider the first reactor for example on the left side: We have 5 units of inflow with concentration of 10 units from an outside source and 1 unit of inflow from reactor 3 with concentration c_3. We also have an outflow of 3 units with concentration c_1 to reactor two and an outflow of 3 units with the same concentration c_1 to reactor 5. The sum of the incoming and outgoing mass flow is zero (with the outgoing being positive as convention now):

$$3 \cdot c_1 + 3 \cdot c_1 - 1 \cdot c_3 - 5 \cdot 10 = 0 \rightarrow 6c_1 - c_3 = 50$$

Similarly, the whole system can be written as the following:

$$\begin{aligned} 6c_1 - c_3 &= 50 \\ -3c_1 + 3c_2 &= 0 \\ -c_2 + 9c_3 &= 160 \\ -c_2 - 8c_3 + 11c_4 - 2c_5 &= 0 \\ -3c_1 - c_2 + 4c_5 &= 0 \end{aligned}$$

In matrix form:

$$\boldsymbol{A} = \begin{bmatrix} 6 & 0 & -1 & 0 & 0 \\ -3 & 3 & 0 & 0 & 0 \\ 0 & -1 & 9 & 0 & 0 \\ 0 & -1 & -8 & 11 & -2 \\ -3 & -1 & 0 & 0 & 4 \end{bmatrix} \quad \boldsymbol{c} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{bmatrix} \quad \boldsymbol{b} = \begin{bmatrix} 50 \\ 0 \\ 160 \\ 0 \\ 0 \end{bmatrix}$$

$$\boldsymbol{A} \cdot \boldsymbol{c} = \boldsymbol{b}$$

The coefficient matrix ***A*** contains many zero entries and therefore can be considered a sparse matrix. Iterative methods are typically used when the coefficient matrix is sparse and the entries are mostly located along the diagonal and close to it. These type of matrices are called diagonally dominant.

Iterative methods work differently than direct methods as we first have to supply the algorithms with initial guesses for the unknown which values are then iteratively changed as the solutions start to converge. (If they converge.) As an example, let's see the following system:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{12}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned} \rightarrow \begin{aligned} x_1 &= \frac{b_1 - a_{12}x_2 - a_{13}x_3}{a_{11}} \\ x_2 &= \frac{b_2 - a_{21}x_1 - a_{23}x_3}{a_{22}} \\ x_3 &= \frac{b_3 - a_{31}x_1 - a_{32}x_2}{a_{33}} \end{aligned}$$

The initial guesses can be substituted into the right-hand side of the equations and we can calculate a new solution. These are then substituted in the second iteration and so on... We can stop when the changes in the solutions (the differences between two iterations) become adequately small (smaller than some tolerance). The formula of iteration in the example above would be the following:

$$x_i = \frac{1}{a_{\mathrm{ii}}} \cdot \left(b_i - \left(\sum_{j=1, j \neq i}^{j=n} a_{\mathrm{ij}} \cdot x_j \right) \right)$$

This is basically the equation of the Jacobi iterative method.

We take a look at two iterative methods:

- Jacobi method
- Gauss-Seidel method

The main difference between the two methods is that the Jacobi method changes the values of all the unknowns at the end of an iteration, while the Gauss-Seidel method changes the value of each unknown right after they are calculated and uses these new values inside the same iteration. This means, that in the second method, when we refresh the value of x_1, we use this new value to calculate the new $x_2, x_3, \cdots$ values in the same iteration. We can derive the iterative formulas using matrix notation starting from the well known form $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$.

Add $\boldsymbol{B} \cdot \boldsymbol{x}$ and subtract $\boldsymbol{B} \cdot \boldsymbol{x}$ from the left-hand side of the equation. Depending on the choice of matrix ***B***, we will have the different methods:

$$\boldsymbol{B} \cdot \boldsymbol{x} + \boldsymbol{A} \cdot \boldsymbol{x} - \boldsymbol{B} \cdot \boldsymbol{x} = \boldsymbol{B} \cdot \boldsymbol{x} + (\boldsymbol{A} - \boldsymbol{B}) \cdot \boldsymbol{x} = \boldsymbol{b}$$

Rearrange the equation:

$$\boldsymbol{B} \cdot \boldsymbol{x} = -(\boldsymbol{A} - \boldsymbol{B}) \cdot \boldsymbol{x} + \boldsymbol{b}$$

Multiply from the left side by B^{-1}:

$$x = -B^{-1}(A - B) \cdot x + B^{-1} \cdot b = B^{-1} \cdot B \cdot x - B^{-1} \cdot A \cdot x + B^{-1} \cdot b$$

Group the coefficients of ***x*** on the right-hand side:

$$x = (I - B^{-1}A)x + B^{-1}b$$

This gives us the formula to compute the k+1-th iteration of vector ***x***:

$$x^{(k+1)} = (I - B^{-1}A)x^{(k)} + B^{-1}b$$

Let ***Ai*** denote $(I - B^{-1}A)$ and ***bi*** denote $(B^{-1}b)$, then we can write the equation in a more concise form:

$$x^{(k+1)} = Ai \cdot x^{(k)} + bi$$

We can create a separate function which handles the iterative solution in the above form given some matrix ***Ai***, some vector ***bi***, a vector of initial guesses, a tolerance value for the stop condition and a value for the maximum number of iterations. Find this function in `iterative.m`.

Solution using the Jacobi method

In case of the Jacobi method, the matrix ***B*** contains the diagonal elements of matrix ***A***, which means that it is very simple to calculate the inverse of ***B***. We can use a vector of ones as the initial guess. First, let's load the data and check if the solution exists:

```
Ab = load('waterplant.txt')
```

```
Ab = 5×6
     6     0    -1     0     0    50
    -3     3     0     0     0     0
     0    -1     9     0     0   160
     0    -1    -8    11    -2     0
    -3    -1     0     0     4     0
```

```
A = Ab(:,1:end-1), b = Ab(:,end)
```

```
A = 5×5
     6     0    -1     0     0
    -3     3     0     0     0
     0    -1     9     0     0
     0    -1    -8    11    -2
    -3    -1     0     0     4
b = 5×1
    50
     0
   160
     0
     0
```

```
rank(A), rank(Ab) % 5, 5
```

```
ans = 5
ans = 5
```

The solution exits and as the matrix ***A*** is full-rank, we have a unique solution. We first compute the matrix ***B***, which is now the diagonal elements of ***A***:

```
x0=ones(5,1)
```

```
x0 = 5×1
     1
     1
     1
     1
     1
```

```
% Jacobi method:  AI and bI input
B = diag(diag(A)) % diagonal elements in matrix A
```

```
B = 5×5
     6     0     0     0     0
     0     3     0     0     0
     0     0     9     0     0
     0     0     0    11     0
     0     0     0     0     4
```

```
Ai=eye(5)-inv(B)*A
```

```
Ai = 5×5
         0         0    0.1667         0         0
    1.0000         0         0         0         0
         0    0.1111         0         0         0
         0    0.0909    0.7273         0    0.1818
    0.7500    0.2500         0         0         0
```

```
bi = inv(B)*b
```

```
bi = 5×1
    8.3333
         0
   17.7778
         0
         0
```

```
% Solution via iteration
X=iterativ(Ai,bi,x0,1E-6,100)
```

```
X = 5×15
    1.0000   11.3148   11.3148   11.4537   11.5058   11.5058   11.5084   11.5094 ···
    1.0000    8.5000   11.3148   11.3148   11.4537   11.5058   11.5058   11.5084
    1.0000   17.8889   18.7222   19.0350   19.0350   19.0504   19.0562   19.0562
    1.0000   13.2828   14.9874   16.5741   16.9295   16.9610   16.9904   16.9970
    1.0000    6.6250   10.6111   11.3148   11.4190   11.4928   11.5058   11.5078
```

```
% Solution is in the last coloumn of x
x=X(:,end)
```

```
x = 5×1
   11.5094
   11.5094
   19.0566
   16.9983
   11.5094
```

```
% iteration number
iter = size(X,2) % 15
```

```
iter = 15
```

Seems like it took 20 iterations for the algorithm to reach the tolerance. Check the results:

```
% Check the result (relative error)
norm(A*x-b)/norm(b)
```

We can see, that the check only gives zero with the tolerance specified. Plot the convergence of the solutions:

```
%  Visualization
figure(1);
plot(X','*-')
title('Jacobi methods convergence')
```

Solution using the Gauss-Seidel method

In order to solve the system using the Gauss-Seidel method, we only have to change the matrix ***B***. In this case, it becomes the lower triangular part of matrix ***A***.

```
% Gauss-Seidel method: AI and  bI input
B=tril(A) % % lower triangular elements in matrix A
```

```
B = 5×5
     6     0     0     0     0
    -3     3     0     0     0
     0    -1     9     0     0
     0    -1    -8    11     0
    -3    -1     0     0     4
```

```
Ai=eye(5)-inv(B)*A
```

```
Ai = 5×5
         0         0    0.1667         0         0
         0         0    0.1667         0         0
         0         0    0.0185         0         0
         0         0    0.0286         0    0.1818
         0         0    0.1667         0         0
```

```
bi = inv(B)*b
```

```
bi = 5×1
    8.3333
    8.3333
   18.7037
   14.3603
    8.3333
```

```
% Solution via iteration
X=iterativ(Ai,bi,x0,1E-6,100)
```

```
X = 5×7
    1.0000   11.4537   11.5084   11.5094   11.5094   11.5094   11.5094
    1.0000   11.4537   11.5084   11.5094   11.5094   11.5094   11.5094
    1.0000   19.0504   19.0565   19.0566   19.0566   19.0566   19.0566
    1.0000   16.4415   16.9880   16.9981   16.9983   16.9983   16.9983
```

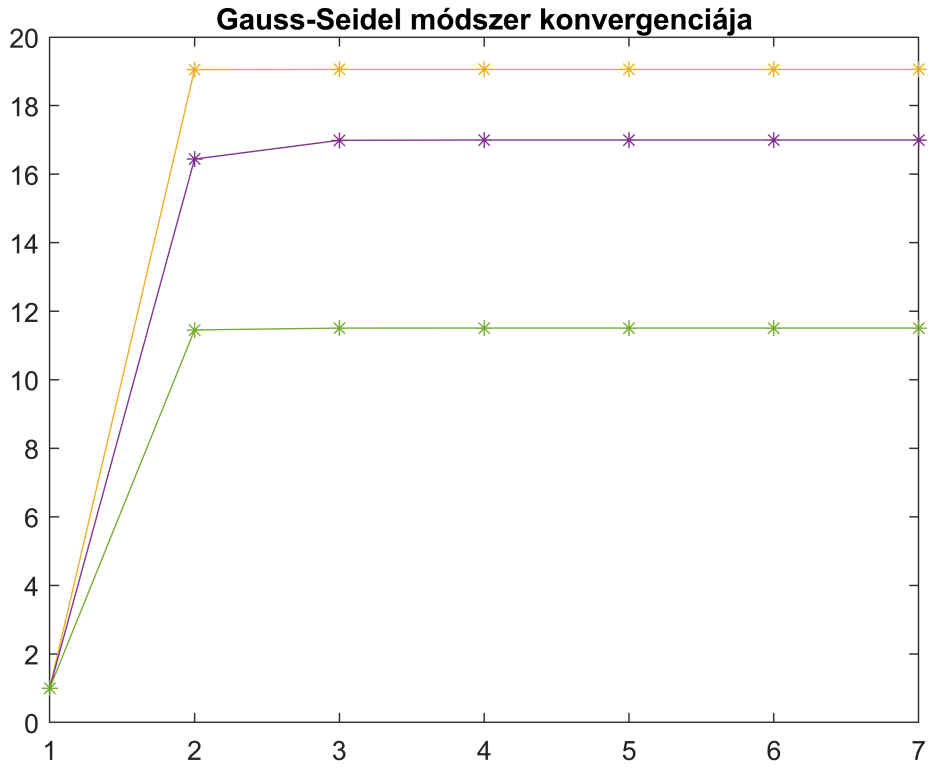
```
    1.0000   11.4537   11.5084   11.5094   11.5094   11.5094   11.5094
```

```
% Solution is in the last coloumn of x
x=X(:,end)
```

```
x = 5×1
   11.5094
   11.5094
   19.0566
   16.9983
   11.5094
```

```
% Iteration number
iter=size(X,2) % 7
```

```
iter = 7
ans = 7.6867e-11
```



Now, it only took 9 iterations to reach at least the desired tolerance. The convergence of the Gauss-Seidel method is significantly better than the Jacobi method. Plot the convergence:

```
% Check the result (relative error)
norm(A*x-b)/norm(b)
```

We can see from the figure, that after 3 iterations, the changes to the solutions become very small. If we want to use the built-in function to solve the system iteratively, we can use the `gmres` command:

```
% Visualization
figure(2);
plot(X','*-')
```

```
title('Gauss-Seidel módszer konvergenciája')
```

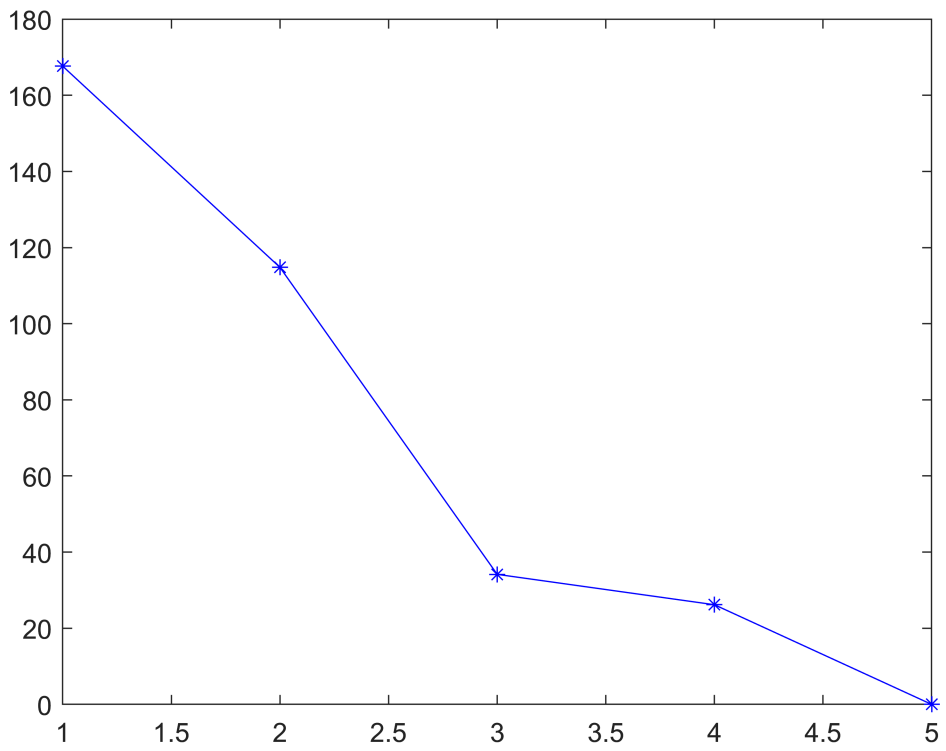
The `gmres` function can give us information about the solution method: `flags` store the exit code of the algorithm (if it is not zero, the algorithm couldn't reach an adequate solution), `relres` returns the residual norm, `iter` returns the number of outer and inner iterations and the `resvec` returns the amount of residual norms at each inner iteration.

If we are working with large sparse matrices, it is better to use the `sparse` function to efficiently store the matrix.

```
[x, flags, relres, iter, resvec] = gmres(A,b)
```

```
x = 5×1
   11.5094
   11.5094
   19.0566
   16.9983
   11.5094
flags = 0
relres = 6.7729e-17
iter = 1×2
     1     4
resvec = 5×1
  167.6305
  114.7905
   34.1596
   26.1821
    0.0000
```

```
figure(3); plot(resvec,'b*-') % errors along the solution
```



```
AS=sparse(A)
```

```
AS =
   (1,1)        6
   (2,1)       -3
   (5,1)       -3
   (2,2)        3
   (3,2)       -1
   (4,2)       -1
   (5,2)       -1
   (1,3)       -1
   (3,3)        9
   (4,3)       -8
   (4,4)       11
   (4,5)       -2
   (5,5)        4
```

```
function X=iterativ(Ai,bi,x0,e,imax)
X=x0; i=0;
x1=Ai*x0+bi; % első iteráció
while i<=imax && norm(x1-x0)>e
  x0=x1;
  x1=Ai*x0+bi;
  X=[X x1];
  i=i+1;
end
end
```