

# Linear equations 1

## Linear algebra basics

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group  
Eindhoven University of Technology

Numerical Methods (6E5X0), 2020-2021

# Today's outline

- Introduction
- Matrix inversion
- Solving a linear system
- Towards larger systems
- Summary

# Today's outline

- Introduction
- Matrix inversion
- Solving a linear system
- Towards larger systems
- Summary

# Overview

## Goals

- Different ways of looking at a system of linear equations
- Determination of the inverse, determinant and the rank of a matrix
- The existence of a solution to a set of linear equations

# Different views of linear systems

- Separate equations:

$$x + y + z = 4$$

$$2x + y + 3z = 7$$

$$3x + y + 6z = 5$$

- Matrix mapping  $Mx = b$ :

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

- Linear combination:

$$x \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + y \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + z \begin{bmatrix} 1 \\ 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

# Different views of linear systems

- Separate equations:

$$x + y + z = 4$$

$$2x + y + 3z = 7$$

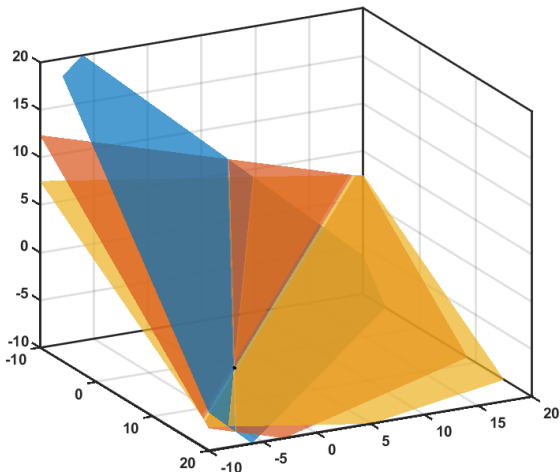
$$3x + y + 6z = 5$$

- Matrix mapping  $Mx = b$ :

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

- Linear combination:

$$x \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + y \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + z \begin{bmatrix} 1 \\ 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$



# Today's outline

- Introduction
- Matrix inversion
- Solving a linear system
- Towards larger systems
- Summary

# Inverse of a matrix

- The inverse  $M^{-1}$  is defined such that:

$$MM^{-1} = I \quad \text{and} \quad M^{-1}M = I$$

- Use the inverse to solve a set of linear equations:

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

$$M^{-1}M\mathbf{x} = M^{-1}\mathbf{b}$$

$$I\mathbf{x} = M^{-1}\mathbf{b}$$

$$\mathbf{x} = M^{-1}\mathbf{b}$$



# How to calculate the inverse?

- The inverse of an  $N \times N$  matrix can be calculated using the co-factors of each element of the matrix:

$$M^{-1} = \frac{1}{\det|M|} \begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{bmatrix}^T$$

- $\det|M|$  is the *determinant* of matrix  $M$ .
- $C_{ij}$  is the *co-factor* of the  $ij^{\text{th}}$  element in  $M$ .

# Computing the co-factors

Consider the following example matrix:  $M = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix}$

# Computing the co-factors

Consider the following example matrix:  $M = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix}$

A co-factor (e.g.  $C_{11}$ ) is the determinant of the elements left over when you cover up the row and column of the element in question, multiplied by  $\pm 1$ , depending on the position.

$$\begin{bmatrix} 1 & \times & \times \\ \times & 1 & 3 \\ \times & 1 & 6 \end{bmatrix}$$

# Computing the co-factors

Consider the following example matrix:  $M = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix}$

A co-factor (e.g.  $C_{11}$ ) is the determinant of the elements left over when you cover up the row and column of the element in question, multiplied by  $\pm 1$ , depending on the position.

$$\begin{bmatrix} 1 & \times & \times \\ \times & 1 & 3 \\ \times & 1 & 6 \end{bmatrix}$$

$$\begin{bmatrix} + & - & + \\ - & + & - \\ + & - & + \end{bmatrix}$$

# Computing the co-factors

Consider the following example matrix:  $M = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix}$

A co-factor (e.g.  $C_{11}$ ) is the determinant of the elements left over when you cover up the row and column of the element in question, multiplied by  $\pm 1$ , depending on the position.

$$\begin{bmatrix} 1 & \times & \times \\ \times & 1 & 3 \\ \times & 1 & 6 \end{bmatrix}$$

$$\begin{bmatrix} + & - & + \\ - & + & - \\ + & - & + \end{bmatrix}$$

$$C_{11} = +1 \cdot \det \begin{vmatrix} 1 & 3 \\ 1 & 6 \end{vmatrix} \\ = 6 \times 1 - 3 \times 1 = 3$$

# Computing the co-factors

Back to our example:

$$M^{-1} = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix}^{-1} = \frac{1}{\det|M|} \begin{bmatrix} 3 & -3 & -1 \\ -5 & 3 & 2 \\ 2 & -1 & -1 \end{bmatrix}^T$$

# Computing the co-factors

Back to our example:

$$M^{-1} = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix}^{-1} = \frac{1}{\det|M|} \begin{bmatrix} 3 & -3 & -1 \\ -5 & 3 & 2 \\ 2 & -1 & -1 \end{bmatrix}^T$$

- The determinant is very important
- If  $\det|M| = 0$ , the inverse does not exist (singular matrix)

# Calculating the determinant

Compute the determinant by multiplication of each element on a row (or column) by its cofactor and adding the results:

$$\det \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix} = +\det \begin{bmatrix} 1 & 3 \\ 1 & 6 \end{bmatrix} - \det \begin{bmatrix} 2 & 3 \\ 3 & 6 \end{bmatrix} + \det \begin{bmatrix} 2 & 1 \\ 3 & 1 \end{bmatrix} = -1$$



# Calculating the determinant

Compute the determinant by multiplication of each element on a row (or column) by its cofactor and adding the results:

$$\det \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix} = +\det \begin{bmatrix} 1 & 3 \\ 1 & 6 \end{bmatrix} - \det \begin{bmatrix} 2 & 3 \\ 3 & 6 \end{bmatrix} + \det \begin{bmatrix} 2 & 1 \\ 3 & 1 \end{bmatrix} = -1$$

$$\det \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix} = +\det \begin{bmatrix} 2 & 1 \\ 3 & 1 \end{bmatrix} - 3\det \begin{bmatrix} 1 & 1 \\ 3 & 1 \end{bmatrix} + 6\det \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix} = -1$$

# Today's outline

- Introduction
- Matrix inversion
- Solving a linear system
- Towards larger systems
- Summary

# Solving a linear system

- Our example:

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

# Solving a linear system

- Our example:

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

- The solution is:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = M^{-1}b = \frac{1}{-1} \begin{bmatrix} 3 & -5 & 2 \\ -3 & 3 & -1 \\ -1 & 2 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix} = \frac{1}{-1} \begin{bmatrix} -13 \\ 4 \\ 5 \end{bmatrix} = \begin{bmatrix} 13 \\ -4 \\ -5 \end{bmatrix}$$

# Solving a linear system

- Our example:

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

- The solution is:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = M^{-1}b = \frac{1}{-1} \begin{bmatrix} 3 & -5 & 2 \\ -3 & 3 & -1 \\ -1 & 2 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix} = \frac{1}{-1} \begin{bmatrix} -13 \\ 4 \\ 5 \end{bmatrix} = \begin{bmatrix} 13 \\ -4 \\ -5 \end{bmatrix}$$

- The inverse exists, because  $\det|M| = -1$ .

# Solving a linear system in Matlab using the inverse

- Create the matrix:

```
>> A = [1 1 1; 2 1 3; 3 1 6];
```

# Solving a linear system in Matlab using the inverse

- Create the matrix:

```
>> A = [1 1 1; 2 1 3; 3 1 6];
```

- Create solution vector:

```
>> b = [4; 7; 5];
```

# Solving a linear system in Matlab using the inverse

- Create the matrix:

```
>> A = [1 1 1; 2 1 3; 3 1 6];
```

- Create solution vector:

```
>> b = [4; 7; 5];
```

- Get the matrix inverse:

```
>> Ainv = inv(A);
```



# Solving a linear system in Matlab using the inverse

- Create the matrix:

```
>> A = [1 1 1; 2 1 3; 3 1 6];
```

- Create solution vector:

```
>> b = [4; 7; 5];
```

- Get the matrix inverse:

```
>> Ainv = inv(A);
```

- Compute the solution:

```
>> x = Ainv * b
```

# Solving a linear system in Matlab using the inverse

- Create the matrix:

```
>> A = [1 1 1; 2 1 3; 3 1 6];
```

- Create solution vector:

```
>> b = [4; 7; 5];
```

- Get the matrix inverse:

```
>> Ainv = inv(A);
```

- Compute the solution:

```
>> x = Ainv * b
```

- Matlab's internal direct solver:

```
>> x = A\b
```

## Exercise: performance of inverse computation

Create a script that generates matrices with random elements of various sizes  $N \times N$ . Compute the inverse of each matrix, and use `tic` and `toc` to see the computing time for each inversion. Plot the time as a function of the matrix size  $N$ .

## Exercise: performance of inverse computation

Create a script that generates matrices with random elements of various sizes  $N \times N$ . Compute the inverse of each matrix, and use `tic` and `toc` to see the computing time for each inversion. Plot the time as a function of the matrix size  $N$ .

```
% Generate random matrices of various sizes 's'.  
% Invert the matrices and store the time required  
% for the inversion. Plot the times vs 's'  
s = [10:10:90 100:100:1000 2000:1000:5000 10000]  
for n = 1:length(s)
```

## Exercise: performance of inverse computation

Create a script that generates matrices with random elements of various sizes  $N \times N$ . Compute the inverse of each matrix, and use `tic` and `toc` to see the computing time for each inversion. Plot the time as a function of the matrix size  $N$ .

```
% Generate random matrices of various sizes 's'.  
% Invert the matrices and store the time required  
% for the inversion. Plot the times vs 's'  
s = [10:10:90 100:100:1000 2000:1000:5000 10000]  
for n = 1:length(s)  
    s(n)  
    A = rand(s(n));
```

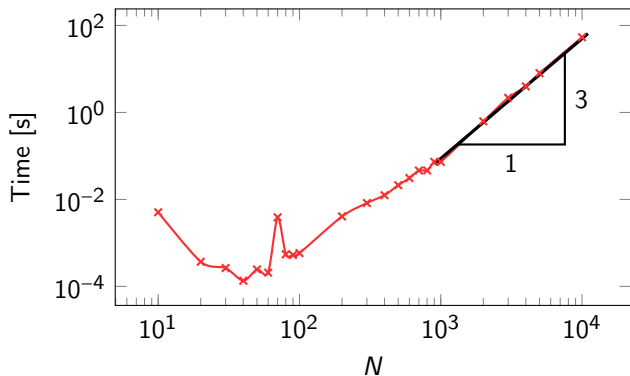
## Exercise: performance of inverse computation

Create a script that generates matrices with random elements of various sizes  $N \times N$ . Compute the inverse of each matrix, and use `tic` and `toc` to see the computing time for each inversion. Plot the time as a function of the matrix size  $N$ .

```
% Generate random matrices of various sizes 's'.  
% Invert the matrices and store the time required  
% for the inversion. Plot the times vs 's'  
s = [10:10:90 100:100:1000 2000:1000:5000 10000]  
for n = 1:length(s)  
    s(n)  
    A = rand(s(n));  
    tic;  
    Ainv = inv(A);  
    t_inv(n) = toc;  
end  
loglog(s,t_inv)  
xlabel('N')  
ylabel('Time [s]')
```

## Exercise: sample results

Each computer produces slightly different results because of background tasks, different matrices, etc. This is especially noticable for small systems.



The time increases by 3 orders of magnitude, for every magnitude in  $N$ . The *computational complexity* of matrix inversion scales with  $\mathcal{O}(N^3)$ !

# Solving a linear system in Excel using the inverse

$$Ax = b \quad \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

- Create matrix **A** in 3 × 3 cells
- Create right hand side vector **b** in 3 vertical cells



# Solving a linear system in Excel using the inverse

$$Ax = b \quad \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

- Create matrix **A** in  $3 \times 3$  cells
- Create right hand side vector **b** in 3 vertical cells
- Compute the inverse **I** :
  - Select an empty area of  $3 \times 3$  cells
  - Type =MINVERSE( **B2:D4** ) (In Dutch Excel: INVERSEMAT)
  - Close with Ctrl+Shift+Enter

# Solving a linear system in Excel using the inverse

$$Ax = b \quad \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

- Create matrix **A** in  $3 \times 3$  cells
- Create right hand side vector **b** in 3 vertical cells
- Compute the inverse **I** :
  - Select an empty area of  $3 \times 3$  cells
  - Type =MINVERSE( **B2:D4** ) (In Dutch Excel: INVERSEMAT)
  - Close with Ctrl+Shift+Enter
- Solution:
  - Select 3 vertical cells
  - Type =MMULT( **H2:J4** ; **B6:B8** ) (In Dutch Excel: PRODUCTMAT. The semicolon may be a comma.)
  - Close with Ctrl+Shift+Enter

# Today's outline

- Introduction
- Matrix inversion
- Solving a linear system
- Towards larger systems
- Summary

# Towards larger systems

Computation of determinants and inverses of large matrices in this way is too difficult (slow), so we need other methods to solve large linear systems!

## Towards larger systems

- Determinant of upper triangular matrix:

$$\det |M_{\text{tri}}| = \prod_{i=1}^n a_{ii} \quad M = \begin{bmatrix} 5 & 3 & 2 \\ 0 & 9 & 1 \\ 0 & 0 & 1 \end{bmatrix} \Rightarrow \det |M| = 5 \times 9 \times 1 = 45$$

- Matrix multiplication:

$$\det |AM| = \det |A| \times \det |M|$$

- When  $A$  is an identity matrix ( $\det |A| = 1$ ):

$$\det |AM| = \det |A| \times \det |M| = 1 \times \det |M|$$

- With rules like this, we can use row-operations so that we can compute the determinant more cheaply.

# Solutions of linear systems

Rank of a matrix: the number of linearly independent columns (columns that can not be expressed as a linear combination of the other columns) of a matrix.

$$M = \begin{bmatrix} 5 & 3 & 2 \\ 0 & 9 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

- 3 independent columns
- In Matlab:

```
>> rank(M)
```

$$M = \begin{bmatrix} 1 & 2 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

- col 2 = 2 × col 1
- col 4 = col 3 - col 1
- 2 independent columns: rank = 2

# Solutions of linear systems

The solution of a system of linear equations may or may not exist, and it may or may not be unique. Existence of solutions can be determined by comparing the rank of the Matrix  $M$  with the rank of the augmented matrix  $M_a$ :

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

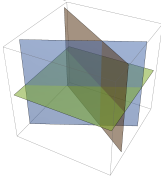
Our system:  $Mx = b$

$$M = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \Rightarrow M_a = \begin{bmatrix} M_{11} & M_{12} & M_{13} & b_1 \\ M_{21} & M_{22} & M_{23} & b_2 \\ M_{31} & M_{32} & M_{33} & b_3 \end{bmatrix}$$

# Existence of solutions for linear systems

For a matrix  $M$  of size  $n \times n$ , and augmented matrix  $M_a$ :

- $\text{Rank}(M) = n$ :  
Unique solution

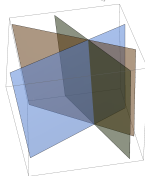
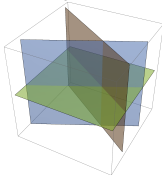




# Existence of solutions for linear systems

For a matrix  $M$  of size  $n \times n$ , and augmented matrix  $M_a$ :

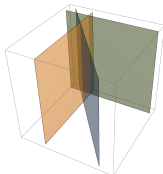
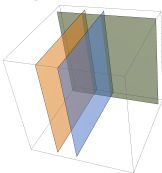
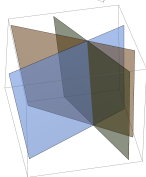
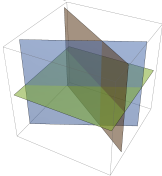
- $\text{Rank}(M) = n$ :  
Unique solution
- $\text{Rank}(M) = \text{Rank}(M_a) < n$ :  
Infinite number of solutions



# Existence of solutions for linear systems

For a matrix  $M$  of size  $n \times n$ , and augmented matrix  $M_a$ :

- $\text{Rank}(M) = n$ :  
Unique solution
- $\text{Rank}(M) = \text{Rank}(M_a) < n$ :  
Infinite number of solutions
- $\text{Rank}(M) < n$ ,  $\text{Rank}(M) < \text{Rank}(M_a)$ :  
No solutions



## Two examples

$$M = \begin{bmatrix} 1 & 1 & 2 \\ 0 & 3 & 1 \\ 0 & 0 & 2 \end{bmatrix} \quad b = \begin{bmatrix} 17 \\ 11 \\ 4 \end{bmatrix} \Rightarrow M_a = \begin{bmatrix} 1 & 1 & 2 & 17 \\ 0 & 3 & 1 & 11 \\ 0 & 0 & 2 & 4 \end{bmatrix}$$

$\text{rank}(M) = 3 = n \Rightarrow \text{Unique solution}$

## Two examples

$$M = \begin{bmatrix} 1 & 1 & 2 \\ 0 & 3 & 1 \\ 0 & 0 & 2 \end{bmatrix} \quad b = \begin{bmatrix} 17 \\ 11 \\ 4 \end{bmatrix} \Rightarrow M_a = \begin{bmatrix} 1 & 1 & 2 & 17 \\ 0 & 3 & 1 & 11 \\ 0 & 0 & 2 & 4 \end{bmatrix}$$

$\text{rank}(M) = 3 = n \Rightarrow$  Unique solution

$$M = \begin{bmatrix} 1 & 1 & 2 \\ 0 & 3 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad b = \begin{bmatrix} 17 \\ 11 \\ 0 \end{bmatrix} \Rightarrow M_a = \begin{bmatrix} 1 & 1 & 2 & 17 \\ 0 & 3 & 1 & 11 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$\text{rank}(M) = \text{rank}(M_a) = 2 < n \Rightarrow$  Infinite number of solutions

# Today's outline

- Introduction
- Matrix inversion
- Solving a linear system
- Towards larger systems
- Summary

# Summary

- Linear equations can be written as matrices
- Using the inverse, the solution can be determined
  - Inverse via cofactors
  - Inverse and solution in Matlab
  - Inverse and solution in Excel
- Introduced the concept of computational complexity: matrix inversion scales with  $N^3$
- A solution depends on the rank of a matrix

# Linear equations 2

## Direct methods

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group  
Eindhoven University of Technology

Numerical Methods (6E5X0), 2020-2021

# Today's outline

- Introduction
- Gauss elimination
- Partial Pivoting
- LU decomposition
- Summary



# Today's outline

- Introduction
- Gauss elimination
- Partial Pivoting
- LU decomposition
- Summary

# Overview

## Goals

Today we are going to write a program, which can solve a set of linear equations

- The first method is called Gaussian elimination
- We will encounter some problems with Gaussian elimination
- Then LU decomposition will be introduced

# Today's outline

- Introduction
- Gauss elimination
- Partial Pivoting
- LU decomposition
- Summary

# Define the linear system

Consider the system:

$$Ax = b$$

In general:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Desired solution:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ b'_3 \end{bmatrix}$$

# Using row operations

- Use row operations to simplify the system. Eliminate element  $A_{21}$  by subtracting  $A_{21}/A_{11} = d_{21}$  times row 1 from row 2.
- In this case, Row 1 is the pivot row, and  $A_{11}$  is the pivot element.

$$\left[ \begin{array}{ccc|c} A_{11} & A_{12} & A_{13} & b_1 \\ A_{21} & A_{22} & A_{23} & b_2 \\ A_{31} & A_{32} & A_{33} & b_3 \end{array} \right] \longrightarrow \left[ \begin{array}{ccc|c} A_{11} & A_{12} & A_{13} & b_1 \\ 0 & A'_{22} & A'_{23} & b'_2 \\ A_{31} & A_{32} & A_{33} & b_3 \end{array} \right]$$

# Using row operations

Eliminate element  $A_{21}$  using  $d_{21} = A_{21}/A_{11}$ .

$$\left[ \begin{array}{ccc|c} A_{11} & A_{12} & A_{13} & b_1 \\ A_{21} & A_{22} & A_{23} & b_2 \\ A_{31} & A_{32} & A_{33} & b_3 \end{array} \right] \longrightarrow \left[ \begin{array}{ccc|c} A_{11} & A_{12} & A_{13} & b_1 \\ 0 & A'_{22} & A'_{23} & b'_2 \\ A_{31} & A_{32} & A_{33} & b_3 \end{array} \right]$$

# Using row operations

Eliminate element  $A_{21}$  using  $d_{21} = A_{21}/A_{11}$ .

$$\left[ \begin{array}{ccc|c} A_{11} & A_{12} & A_{13} & b_1 \\ A_{21} & A_{22} & A_{23} & b_2 \\ A_{31} & A_{32} & A_{33} & b_3 \end{array} \right] \longrightarrow \left[ \begin{array}{ccc|c} A_{11} & A_{12} & A_{13} & b_1 \\ 0 & A'_{22} & A'_{23} & b'_2 \\ A_{31} & A_{32} & A_{33} & b_3 \end{array} \right]$$

- $d_{21} \rightarrow A_{21}/A_{11}$
- $A_{21} \rightarrow A_{21} - A_{11}d_{21}$
- $A_{22} \rightarrow A_{22} - A_{12}d_{21}$
- $A_{23} \rightarrow A_{23} - A_{13}d_{21}$
- $b_2 \rightarrow b_2 - b_1d_{21}$

# Using row operations

Eliminate element  $A_{21}$  using  $d_{21} = A_{21}/A_{11}$ .

$$\left[ \begin{array}{ccc|c} A_{11} & A_{12} & A_{13} & b_1 \\ A_{21} & A_{22} & A_{23} & b_2 \\ A_{31} & A_{32} & A_{33} & b_3 \end{array} \right] \longrightarrow \left[ \begin{array}{ccc|c} A_{11} & A_{12} & A_{13} & b_1 \\ 0 & A'_{22} & A'_{23} & b'_2 \\ A_{31} & A_{32} & A_{33} & b_3 \end{array} \right]$$

- $d_{21} \rightarrow A_{21}/A_{11}$
- $A_{21} \rightarrow A_{21} - A_{11}d_{21}$
- $A_{22} \rightarrow A_{22} - A_{12}d_{21}$
- $A_{23} \rightarrow A_{23} - A_{13}d_{21}$
- $b_2 \rightarrow b_2 - b_1d_{21}$

```
d21 = A(2,1)/A(1,1);  
A(2,1) = A(2,1) - A(1,1)*d21;  
A(2,2) = A(2,2) - A(1,2)*d21;  
A(2,3) = A(2,3) - A(1,3)*d21;  
b(2) = b(2) - b(1)*d21;
```



# Using row operations

Eliminate element  $A_{31}$  using  $d_{31} = A_{31}/A_{11}$ .

$$\left[ \begin{array}{ccc|c} A_{11} & A_{12} & A_{13} & b_1 \\ 0 & A'_{22} & A'_{23} & b'_2 \\ A_{31} & A_{32} & A_{33} & b_3 \end{array} \right] \longrightarrow \left[ \begin{array}{ccc|c} A_{11} & A_{12} & A_{13} & b_1 \\ 0 & A'_{22} & A'_{23} & b'_2 \\ 0 & A'_{32} & A'_{33} & b'_3 \end{array} \right]$$

# Using row operations

Eliminate element  $A_{31}$  using  $d_{31} = A_{31}/A_{11}$ .

$$\left[ \begin{array}{ccc|c} A_{11} & A_{12} & A_{13} & b_1 \\ 0 & A'_{22} & A'_{23} & b'_2 \\ A_{31} & A_{32} & A_{33} & b_3 \end{array} \right] \longrightarrow \left[ \begin{array}{ccc|c} A_{11} & A_{12} & A_{13} & b_1 \\ 0 & A'_{22} & A'_{23} & b'_2 \\ 0 & A'_{32} & A'_{33} & b'_3 \end{array} \right]$$

- $d_{31} \rightarrow A_{31}/A_{11}$
- $A_{31} \rightarrow A_{31} - A_{11}d_{31}$
- $A_{32} \rightarrow A_{32} - A_{12}d_{31}$
- $A_{33} \rightarrow A_{33} - A_{13}d_{31}$
- $b_3 \rightarrow b_3 - b_1d_{31}$

```
d31 = A(3,1)/A(1,1);  
A(3,1) = A(3,1) - A(1,1)*d31;  
A(3,2) = A(3,2) - A(1,2)*d31;  
A(3,3) = A(3,3) - A(1,3)*d31;  
b(3) = b(3) - b(1)*d31;
```

## Using row operations

Eliminate element  $A_{32}$  using  $d_{32} = A_{32}/A'_{22}$ . Note that now the second row has become the pivot row.

$$\left[ \begin{array}{ccc|c} A_{11} & A_{12} & A_{13} & b_1 \\ 0 & A'_{22} & A'_{23} & b'_2 \\ 0 & A_{32} & A_{33} & b_3 \end{array} \right] \longrightarrow \left[ \begin{array}{ccc|c} A_{11} & A_{12} & A_{13} & b_1 \\ 0 & A'_{22} & A'_{23} & b'_2 \\ 0 & 0 & A''_{33} & b''_3 \end{array} \right]$$

# Using row operations

Eliminate element  $A_{32}$  using  $d_{32} = A_{32}/A'_{22}$ . Note that now the second row has become the pivot row.

$$\left[ \begin{array}{ccc|c} A_{11} & A_{12} & A_{13} & b_1 \\ 0 & A'_{22} & A'_{23} & b'_2 \\ 0 & A_{32} & A_{33} & b_3 \end{array} \right] \longrightarrow \left[ \begin{array}{ccc|c} A_{11} & A_{12} & A_{13} & b_1 \\ 0 & A'_{22} & A'_{23} & b'_2 \\ 0 & 0 & A''_{33} & b''_3 \end{array} \right]$$

- $d_{32} \rightarrow A_{32}/A'_{22}$
- $A_{32} \rightarrow A_{32} - A'_{22}d_{32}$
- $A_{33} \rightarrow A_{33} - A'_{23}d_{32}$
- $b_3 \rightarrow b_3 - b'_2d_{32}$

```
d32 = A(3,2)/A(2,2);  
A(3,2) = A(3,2) - A(2,2)*d32;  
A(3,3) = A(3,3) - A(2,3)*d32;  
b(3) = b(3) - b(2)*d32;
```

## Using row operations

Eliminate element  $A_{32}$  using  $d_{32} = A_{32}/A'_{22}$ . Note that now the second row has become the pivot row.

$$\left[ \begin{array}{ccc|c} A_{11} & A_{12} & A_{13} & b_1 \\ 0 & A'_{22} & A'_{23} & b'_2 \\ 0 & A_{32} & A_{33} & b_3 \end{array} \right] \longrightarrow \left[ \begin{array}{ccc|c} A_{11} & A_{12} & A_{13} & b_1 \\ 0 & A'_{22} & A'_{23} & b'_2 \\ 0 & 0 & A''_{33} & b''_3 \end{array} \right]$$

- $d_{32} \rightarrow A_{32}/A'_{22}$
- $A_{32} \rightarrow A_{32} - A'_{22}d_{32}$
- $A_{33} \rightarrow A_{33} - A'_{23}d_{32}$
- $b_3 \rightarrow b_3 - b'_2d_{32}$

```
d32 = A(3,2)/A(2,2);  
A(3,2) = A(3,2) - A(2,2)*d32;  
A(3,3) = A(3,3) - A(2,3)*d32;  
b(3) = b(3) - b(2)*d32;
```

The matrix is now a triangular matrix — the solution can be obtained by back-substitution.

# Backsubstitution

The system now reads:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & 0 & A''_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \end{bmatrix}$$

# Backsubstitution

The system now reads:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & 0 & A''_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \end{bmatrix}$$

Start at the last row  $N$ , and work upward until row 1.

$$x_3 = b''_3 / A''_{33}$$

$$x_2 = (b'_2 - A'_{23}x_3) / A'_{22}$$

$$x_1 = (b_1 - A_{12}x_2 - A_{13}x_3) / A_{11}$$

# Backsubstitution

The system now reads:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & 0 & A''_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \end{bmatrix}$$

Start at the last row  $N$ , and work upward until row 1.

$$x_3 = b''_3 / A''_{33}$$

$$x_2 = (b'_2 - A'_{23}x_3) / A'_{22}$$

$$x_1 = (b_1 - A_{12}x_2 - A_{13}x_3) / A_{11}$$

$$\begin{aligned} x(3) &= b(3) / A(3,3) \\ x(2) &= (b(2) - A(2,3)*x(3)) / A(2,2) \\ x(1) &= (b(1) - A(1,2)*x(2) - A(1,3)*x(3)) / A(1,1) \end{aligned}$$

In general:

$$x_N = \frac{b_N}{A_{NN}} \quad x_i = \frac{b_i - \sum_{j=i+1}^N A_{ij}x_j}{A_{ii}}$$



# Writing the program

- Create a function that provides the framework: take matrix  $A$  and vector  $b$  as an input, and return the solution  $x$  as output:

```
function [x,A,b] = GaussianEliminate(A,b)
```

- We will use *for-loops* instead of typing out each command line.
- Useful Matlab shortcuts:
  - $A(1,:) = [A_{11}, A_{12}, A_{13}]$
  - $A(:,2) = [A_{12}, A_{22}, A_{32}]^T$
  - $A(1,2:end) = [A_{12}, A_{13}]$
- A row operation could look like:

```
A(i,:) = A(i,:) - d*A(1,:)
```

# The program: elimination

```
function [x,A,b] = GaussianEliminate(A,b)

% Perform elimination to obtain an upper triangular matrix
N = length(b);
for column=1:(N-1) % Select pivot
    for row=(column+1):N % Loop over subsequent rows (below pivot)
        d=A(row,column)/A(column,column);
        A(row,:)=A(row,:)-d*A(column,:);
        b(row)= b(row)-d*b(column);
    end
end
```

# The program: Backsubstitution

```
% Assign b to x
x=b;

% Perform backsubstitution
for row=N:-1:1
    x(row) = b(row);
    for i =(row+1):N
        x(row)=x(row)-A(row,i)*x(i);
    end
    x(row)=x(row)/A(row,row);
end
```

$$x_N = \frac{b_N}{A_{NN}} \quad x_i = \frac{b_i - \sum_{j=i+1}^N A_{ij}x_j}{A_{ii}}$$

## Exercise: Gaussian Elimination

- The function we just made can be found on Canvas
- Use `help GaussianEliminate` to find out how it works
- Solve the following system of equations:

$$\begin{bmatrix} 9 & 9 & 5 & 2 \\ 6 & 7 & 1 & 3 \\ 6 & 4 & 3 & 5 \\ 2 & 6 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 7 \\ 4 \\ 10 \\ 1 \end{bmatrix}$$

- Compare your solution with `A\b`

# Today's outline

- Introduction
- Gauss elimination
- **Partial Pivoting**
- LU decomposition
- Summary

# Partial pivoting

- Now try to run the algorithm with the following system:

$$\begin{bmatrix} 0 & 2 & 1 \\ 3 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \\ 10 \end{bmatrix}$$

# Partial pivoting

- Now try to run the algorithm with the following system:

$$\begin{bmatrix} 0 & 2 & 1 \\ 3 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \\ 10 \end{bmatrix}$$

- It does not work! Division by zero, due to  $A_{11} = 0$ .
- Solution: Swap rows to move largest element to the diagonal.

# Partial pivoting: implementing row swaps

- Find maximum element row below pivot in current column

```
[dummy, index] = max(abs(A(column:end, column)));  
Index = index + column - 1;
```



# Partial pivoting: implementing row swaps

- Find maximum element row below pivot in current column
- Store current row

```
[dummy, index] = max(abs(A(column:end, column)));  
Index = index + column - 1;
```

```
temp = A(column, :);
```

# Partial pivoting: implementing row swaps

- Find maximum element row below pivot in current column
- Store current row
- Swap pivot row and desired row in A

```
[dummy, index] = max(abs(A(column:end, column)));  
Index = index + column - 1;
```

```
temp = A(column, :);
```

```
A(column, :) = A(index, :);  
A(index, :) = temp;
```

# Partial pivoting: implementing row swaps

- Find maximum element row below pivot in current column
- Store current row
- Swap pivot row and desired row in A
- Do the same for b: store and swap

```
[dummy, index] = max(abs(A(column:end, column)));  
Index = index + column - 1;
```

```
temp = A(column, :);
```

```
A(column, :) = A(index, :);  
A(index, :) = temp;
```

```
temp = b(column);  
b(column) = b(index);  
b(index) = temp;
```

# Improve the program by using re-usable functions

```
function [x] = GaussianEliminate(A,b)
% GaussianEliminate(A,b): solves x in Ax=b
N = length(b);
for c=1:(N-1)
    [dummy,index]=max(abs(A(c:end,c)));
    index=index+c-1;
    A = SWAP(A,c,index); % Created swap function
    b = SWAP(b,c,index);
    for row=(column+1):N
        d=A(row,column)/A(column,column);
        A(row,:)=A(row,:)-d*A(column,:);
        b(row)= b(row)-d*b(column);
    end
end
x = backwardSub(A,b); % Created BS function
return
```

This function is also provided (named GaussianEliminate\_v2 and GaussianEliminate\_v3 on Canvas).

# Alternatives to this program

- MATLAB can compute the solution to  $Ax=b$  with its own solvers (more efficient)  $A \setminus b$
- Too many loops. Loops make MATLAB slow.
- There are fundamental problems with Gaussian elimination

# Alternatives to this program

- MATLAB can compute the solution to  $Ax=b$  with its own solvers (more efficient)  $A \setminus b$
- Too many loops. Loops make MATLAB slow.
- There are fundamental problems with Gaussian elimination
  - You can add a counter to the algorithm to see how many subtraction and multiplication operations it performs for a given size of matrix  $A$ .
  - The number of operations to perform Gaussian elimination is  $\mathcal{O}(2N^3)$  (where  $N$  is the number of equations)
  - Exercise: verify this for our script

# Alternatives to this program

- MATLAB can compute the solution to  $Ax=b$  with its own solvers (more efficient)  $A \setminus b$
- Too many loops. Loops make MATLAB slow.
- There are fundamental problems with Gaussian elimination
  - You can add a counter to the algorithm to see how many subtraction and multiplication operations it performs for a given size of matrix  $A$ .
  - The number of operations to perform Gaussian elimination is  $\mathcal{O}(2N^3)$  (where  $N$  is the number of equations)
  - Exercise: verify this for our script
  - LU decomposition takes  $\mathcal{O}(2N^3/3)$  flops, 3 times less!
  - Forward and backward substitution each take  $\mathcal{O}(N^2)$  flops (both cases)

# Today's outline

- Introduction
- Gauss elimination
- Partial Pivoting
- **LU decomposition**
- Summary



# LU Decomposition

Suppose we want to solve the previous set of equations, but with several right hand sides:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} \vdots & \vdots & \vdots \\ x_1 & x_2 & x_3 \\ \vdots & \vdots & \vdots \end{bmatrix} = \begin{bmatrix} \vdots & \vdots & \vdots \\ b_1 & b_2 & b_3 \\ \vdots & \vdots & \vdots \end{bmatrix}$$

# LU Decomposition

Suppose we want to solve the previous set of equations, but with several right hand sides:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} \vdots & \vdots & \vdots \\ x_1 & x_2 & x_3 \\ \vdots & \vdots & \vdots \end{bmatrix} = \begin{bmatrix} \vdots & \vdots & \vdots \\ b_1 & b_2 & b_3 \\ \vdots & \vdots & \vdots \end{bmatrix}$$

Factor the matrix A into two matrices, L and U, such that  $A = LU$ :

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \times & 1 & 0 \\ \times & \times & 1 \end{bmatrix} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \end{bmatrix}$$

Now we can solve for each right hand side, using only a forward followed by a backward substitution!

# Substitutions

- Define a lower and upper matrix  $L$  and  $U$  so that  $A = LU$
- Therefore  $LUx = b$
- Define a new vector  $y = Ux$  so that  $Ly = b$
- Solve for  $y$ , use  $L$  and forward substitution
- Then we have  $y$ , solve for  $x$ , use  $Ux = y$
- Solve for  $x$ , use  $U$  and backward substitution
- But how to get  $L$  and  $U$ ?

# Decomposing the matrix (1)

When we eliminate the element  $A_{21}$  we can keep multiplying by a matrix that undoes this row operations, so that the product remains equal to  $A$ .

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ d_{21} & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A_{22} - d_{21}A_{12} & A_{23} - d_{21}A_{13} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

## Decomposing the matrix (2)

When we eliminate the element  $A_{31}$  we can keep multiplying by a matrix that undoes this row operations, so that the product remains equal to  $A$ .

$$A = \begin{bmatrix} 1 & 0 & 0 \\ d_{21} & 1 & 0 \\ d_{31} & 0 & 1 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A'_{22} = A_{22} - d_{21}A_{12} & A'_{23} = A_{23} - d_{21}A_{13} \\ 0 & A'_{32} = A_{32} - d_{31}A_{12} & A'_{33} = A_{33} - d_{31}A_{21} \end{bmatrix}$$

## Decomposing the matrix (3)

When we eliminate the element  $A_{32}$  we can keep multiplying by a matrix that undoes this row operations, so that the product remains equal to  $A$ .

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ d_{21} & 1 & 0 \\ d_{31} & d_{32} & 1 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & A'_{32} & A''_{33} = A'_{33} - d_{32}A'_{23} \end{bmatrix}$$

## Decomposing the matrix (3)

When we eliminate the element  $A_{32}$  we can keep multiplying by a matrix that undoes this row operations, so that the product remains equal to  $A$ .

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ d_{21} & 1 & 0 \\ d_{31} & d_{32} & 1 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & A'_{32} & A''_{33} \end{bmatrix} \quad A''_{33} = A'_{33} - d_{32}A'_{23}$$

This finishes the LU decomposition!

## Pivoting during decomposition

Suppose we have arrived at the situation below, where  $A'_{32} > A'_{22}$ :

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ d_{21} & 1 & 0 \\ d_{31} & 0 & 1 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & A'_{32} & A'_{33} \end{bmatrix}$$



## Pivoting during decomposition

Suppose we have arrived at the situation below, where  $A'_{32} > A'_{22}$ :

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ d_{21} & 1 & 0 \\ d_{31} & 0 & 1 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & A'_{32} & A'_{33} \end{bmatrix}$$

Exchange rows 2 and 3 to get the largest value on the main diagonal. Use a permutation matrix to store the swapped rows:

## Pivoting during decomposition

Suppose we have arrived at the situation below, where  $A'_{32} > A'_{22}$ :

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ d_{21} & 1 & 0 \\ d_{31} & 0 & 1 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & A'_{32} & A'_{33} \end{bmatrix}$$

Exchange rows 2 and 3 to get the largest value on the main diagonal. Use a permutation matrix to store the swapped rows:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ d_{31} & 0 & 1 \\ d_{21} & 1 & 0 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A'_{32} & A'_{33} \\ 0 & A'_{22} & A'_{23} \end{bmatrix}$$

## Pivoting during decomposition

Suppose we have arrived at the situation below, where  $A'_{32} > A'_{22}$ :

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ d_{21} & 1 & 0 \\ d_{31} & 0 & 1 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & A'_{32} & A'_{33} \end{bmatrix}$$

Exchange rows 2 and 3 to get the largest value on the main diagonal. Use a permutation matrix to store the swapped rows:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ d_{31} & 0 & 1 \\ d_{21} & 1 & 0 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A'_{32} & A'_{33} \\ 0 & A'_{22} & A'_{23} \end{bmatrix}$$

Multiplying with a permutation matrix will swap the rows of a matrix. The permutation matrix is just an identity matrix, whose rows have been interchanged.

# Recipe for LU decomposition

When decomposing matrix  $A$  into  $A = LU$ , it may be beneficial to swap rows to get the largest values on the diagonal of  $U$  (pivoting). A permutation matrix  $P$  is used to store row swapping such that:

$$PA = LU$$

- Write down a permutation matrix and the linear system
- Promote the largest value in the column diagonal
- Eliminate all elements below diagonal
- Move on to the next column and move largest elements to diagonal
- Eliminate elements below diagonal
- Repeat 5 and 6
- Write down  $L, U$  and  $P$

# LU decomposition example (1)

Write down a permutation matrix, which starts as the identity matrix, and the linear system:

$$PA = LU$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 1 \\ 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 1 \\ 1 & 2 & 0 \end{bmatrix}$$

# LU decomposition example (1)

Write down a permutation matrix, which starts as the identity matrix, and the linear system:

$$PA = LU$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 1 \\ 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 1 \\ 1 & 2 & 0 \end{bmatrix}$$

Promote the largest value into the diagonal of column 1 — swap row 1 and 2:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 1 \\ 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 2 & 0 \end{bmatrix}$$

## LU decomposition example (2)

Eliminate all **elements below the diagonal** — row 2 already contains a zero in column 1, row 3 = row 3 - 0.5 row 1. Record the **multiplier 0.5** in  $L$ :

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 1 \\ 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.5 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1.5 & -0.5 \end{bmatrix}$$

## LU decomposition example (2)

Eliminate all **elements below the diagonal** — row 2 already contains a zero in column 1, row 3 = row 3 - 0.5 row 1. Record the **multiplier 0.5** in  $L$ :

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 1 \\ 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \mathbf{0.5} & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 \\ \mathbf{0} & 1 & 1 \\ 0 & 1.5 & -0.5 \end{bmatrix}$$

Elimination of column 1 is done. Step to the next column, and move the largest value below/on the diagonal to the diagonal ( **swap rows 2 and 3** ). Adjust  $P$  and **lower triangle of  $L$**  accordingly:

$$\begin{bmatrix} 0 & 1 & 0 \\ \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \mathbf{1} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 0 \\ 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \mathbf{0.5} & 1 & 0 \\ \mathbf{0} & \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 \\ \mathbf{0} & \mathbf{1.5} & \mathbf{-0.5} \\ \mathbf{0} & \mathbf{1} & \mathbf{1} \end{bmatrix}$$



## LU decomposition example (3)

Eliminate all elements below the diagonal —

row 3 = row 3 -  $\frac{2}{3}$ row 2. Record the multiplier  $\frac{2}{3}$  in L:

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 0 \\ 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0 & \frac{2}{3} & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 \\ 0 & 1.5 & -0.5 \\ 0 & 0 & \frac{4}{3} \end{bmatrix}$$

## LU decomposition example (3)

Eliminate all elements below the diagonal —  
row 3 = row 3 -  $\frac{2}{3}$ row 2. Record the multiplier  $\frac{2}{3}$  in L:

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 0 \\ 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0 & \frac{2}{3} & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 \\ 0 & 1.5 & -0.5 \\ 0 & 0 & \frac{4}{3} \end{bmatrix}$$

We have obtained the matrices from  $PA = LU$ :

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0 & \frac{2}{3} & 1 \end{bmatrix} \quad U = \begin{bmatrix} 2 & 1 & 1 \\ 0 & 1.5 & -0.5 \\ 0 & 0 & \frac{4}{3} \end{bmatrix}$$

Proceed with solving for  $x$ .

# Substitutions

$$Ax = b \Rightarrow PAx = Pb \equiv d$$

$$PA = LU \Rightarrow LUx = d$$

- Define a new vector  $y = Ux$ 
  - $Ly = b \Rightarrow Ly = d$
  - Solve for  $y$ , forward substitution:

$$y_1 = \frac{d_1}{L_{11}}$$

$$y_i = \frac{d_i - \sum_{j=1}^{i-1} L_{ij}y_j}{L_{ii}}$$

- Then solve  $Ux = y$ :
  - Solve for  $x$ , backward substitution:

$$x_N = \frac{y_N}{U_{NN}}$$

$$x_i = \frac{y_i - \sum_{j=i+1}^{N-1} U_{ij}x_j}{U_{ii}}$$

# How to use the solver in Matlab

```
A = rand(5,5);           % Get random matrix
[L, U, P] = lu(A);        % Get L, U and P
b = rand(5,1);           % Random b vector
d = P*b;                 % Permute b vector
y = forwardSub(L,d);      % Can also do y=L\d
x = backwardSub(U,y);     % Can also do x=U\y
rnorm = norm(A*x - b);   % Residual

% Compare results to internal Matlab solver
x = A\b
```

# How to use the solver in Matlab

```
A = rand(5,5);           % Get random matrix
[L, U, P] = lu(A);        % Get L, U and P
b = rand(5,1);           % Random b vector
d = P*b;                 % Permute b vector
y = forwardSub(L,d);      % Can also do y=L\d
x = backwardSub(U,y);     % Can also do x=U\y
rnorm = norm(A*x - b);   % Residual

% Compare results to internal Matlab solver
x = A\b
```

- Use this as a basis to create a function that takes  $A$  and  $b$ , and returns  $x$ .
- Use the function to check the performance for various matrix sizes and inspect the performance.

# Today's outline

- Introduction
- Gauss elimination
- Partial Pivoting
- LU decomposition
- Summary

# Summary

- This lecture covered direct methods using elimination techniques.
- Gaussian elimination can be slow ( $\mathcal{O}(N^3)$ )
- Back substitution is often faster ( $\mathcal{O}(N^2)$ )
- LU decomposition means that we don't have to do Gaussian elimination every time (saves time and effort), but the matrix has to stay the same.
- Matlab has build in routines for solving linear equations (backslash operator `\`) and LU decomposition (`lu`).
- Advanced techniques such as (preconditioned) conjugate gradient or biconjugate gradient solvers are also available.
- Next part covers iterative approaches

# Linear equations 3

## Iterative methods

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group  
Eindhoven University of Technology

Numerical Methods (6E5X0), 2020-2021



# Today's outline

- Introduction
- Sparse matrices
- Laplace's equation
- Creating a sparse system
- Iterative methods
- Summary

# Today's outline

- Introduction
- Sparse matrices
- Laplace's equation
- Creating a sparse system
- Iterative methods
- Summary

# Sparse matrices

- In many engineering cases, we deal with sparse matrices (as opposed to dense matrices)
- A matrix is sparse when it mostly consists of zeros
- Linear systems where equations depend on a limited number of variables (e.g. spatial discretization)
- Storing zeros is not very efficient:

```
>> A = eye(10000);  
>> whos A  
>> S = sparse(A);  
>> whos S
```

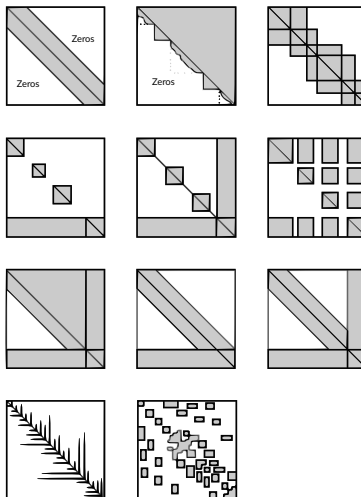
- Can you think of a way to achieve this?
- Sparse matrix formats: Yale, CRS, CCS

# Sparse matrix storage format

- Example: Yale storage format, storing 3 vectors:
  - $A = [5 \ 8 \ 3 \ 6]$
  - $IA = [0 \ 1 \ 2 \ 3 \ 4]$
  - $JA = [0 \ 1 \ 2 \ 1]$
- $A$  stores the non-zero values
- $IA$  stores the index in  $A$  of the first non-zero in row  $i$
- $JA$  stores the column index
- Note: zero-based indices are used here!

$$A = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{bmatrix}$$

# Sparse matrix layout examples



# Today's outline

- Introduction
- Sparse matrices
- Laplace's equation
- Creating a sparse system
- Iterative methods
- Summary

# Laplace's equation

$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T$$

$T$  = Temperature

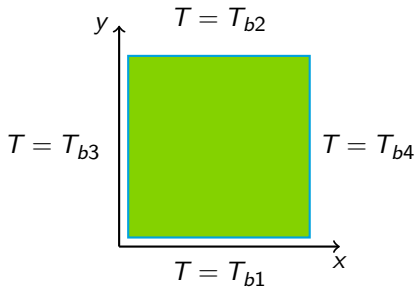
$\alpha$  = Thermal diffusivity

# Laplace's equation

$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T$$

$T$  = Temperature

$\alpha$  = Thermal diffusivity





# Laplace's equation

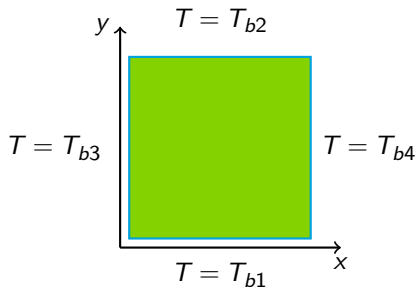
$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T$$

$T$  = Temperature

$\alpha$  = Thermal diffusivity

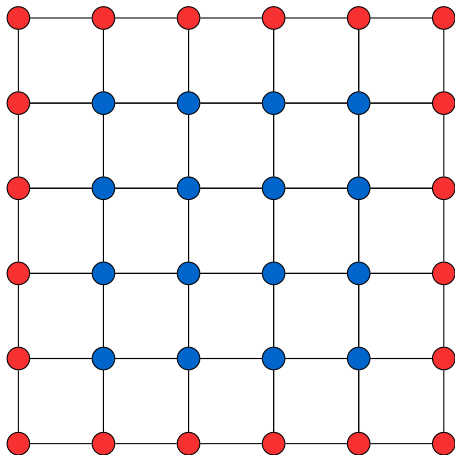
In steady state:

$$\nabla^2 T = 0$$



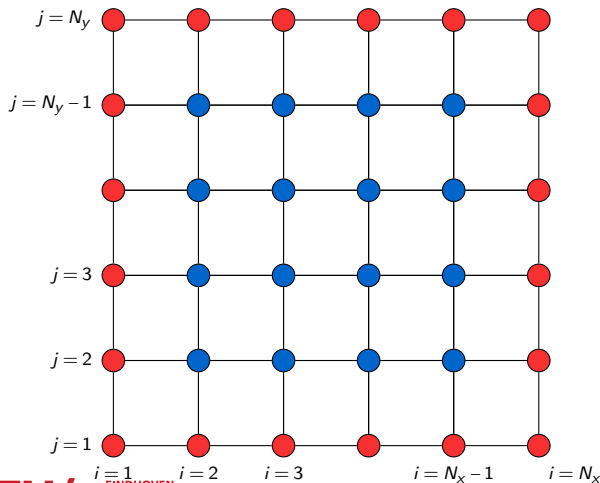
$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

# Discretization of Laplace's equation (I)



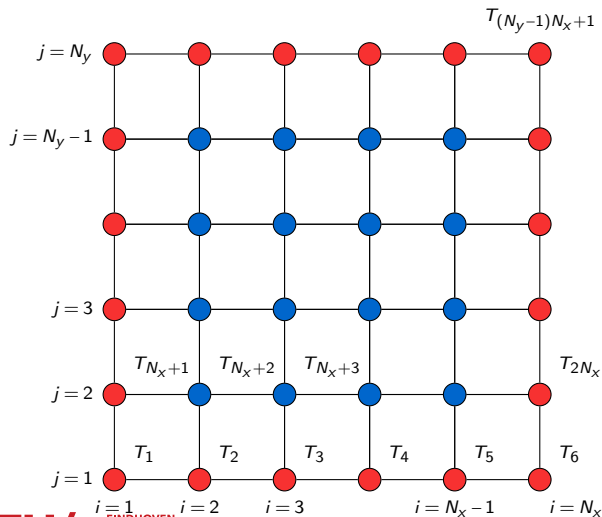
- Define a grid of points in  $x$  and  $y$

# Discretization of Laplace's equation (I)



- Define a grid of points in  $x$  and  $y$
- Index of the grid points using 2D coordinates  $i$  and  $j$

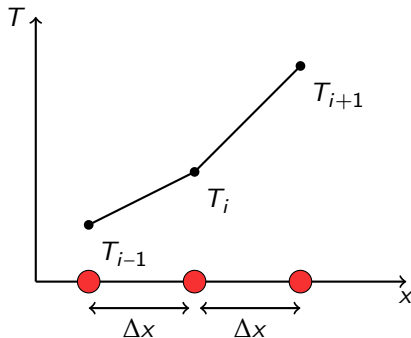
# Discretization of Laplace's equation (I)



- Define a grid of points in  $x$  and  $y$
- Index of the grid points using 2D coordinates  $i$  and  $j$
- Set up the equations using a 1D index system:  
$$T_{i,j} = T_{i+N_x(j-1)}$$

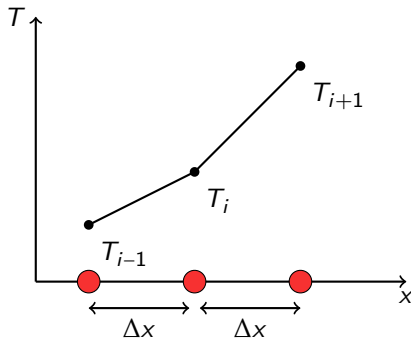
# Discretization of Laplace's equation (II)

Estimate the second-order differentials: assume a piece-wise linear profile in the temperature:



## Discretization of Laplace's equation (II)

Estimate the second-order differentials: assume a piece-wise linear profile in the temperature:



$$\begin{aligned}\frac{\partial^2 T}{\partial x^2} &\approx \frac{\left. \frac{\partial T}{\partial x} \right|_{i+\frac{1}{2}} - \left. \frac{\partial T}{\partial x} \right|_{i-\frac{1}{2}}}{\Delta x} \\ &\approx \frac{\frac{(T_{i+1,j} - T_{i,j})}{\Delta x} - \frac{(T_{i,j} - T_{i-1,j})}{\Delta x}}{\Delta x} \\ &= \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{(\Delta x)^2}\end{aligned}$$

## Discretization of Laplace's equation (III)

The  $y$ -direction is derived analogously, so that the 2D Laplace's equation is discretized as:

$$\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{(\Delta x)^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{(\Delta y)^2} = 0$$

## Discretization of Laplace's equation (III)

The  $y$ -direction is derived analogously, so that the 2D Laplace's equation is discretized as:

$$\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{(\Delta x)^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{(\Delta y)^2} = 0$$

Use a single index counter  $k = i + N_x(j-1)$ , so that the equation becomes:

$$\frac{T_{k+1} - 2T_k + T_{k-1}}{(\Delta x)^2} + \frac{T_{k+N_x} - 2T_k + T_{k-N_x}}{(\Delta y)^2} = 0$$



## Discretization of Laplace's equation (III)

The y-direction is derived analogously, so that the 2D Laplace's equation is discretized as:

$$\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{(\Delta x)^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{(\Delta y)^2} = 0$$

Use a single index counter  $k = i + N_x(j-1)$ , so that the equation becomes:

$$\frac{T_{k+1} - 2T_k + T_{k-1}}{(\Delta x)^2} + \frac{T_{k+N_x} - 2T_k + T_{k-N_x}}{(\Delta y)^2} = 0$$

For an equal spaced grid  $\Delta x = \Delta y = 1$ :

$$T_{k-N_x} + T_{k-1} - 4T_k + T_{k+1} + T_{k+N_x} = 0$$

$$\Rightarrow AT = b$$

# Today's outline

- Introduction
- Sparse matrices
- Laplace's equation
- **Creating a sparse system**
- Iterative methods
- Summary

# Creating the linear system

$$T_{k-N_x} + T_{k-1} - 4T_k + T_{k+1} + T_{k+N_x} = 0$$

Create a *banded* matrix  $A$ : the main diagonal  $k$  contains  $-4$ , whereas the bands at  $k-1$ ,  $k+1$ ,  $k-N_x$  and  $k+N_x$  contain a  $1$ . Boundary cells just contain a  $1$  on the main diagonal so that the temperature is equal to  $T_b$  (e.g.  $T_1 = 1T_b$ ).

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \dots & 1 & \dots & 1 & -4 & 1 & \dots & 1 & \ddots & 0 \\ 0 & \dots & 1 & \dots & 1 & -4 & 1 & \dots & 1 & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_k \\ T_{k+1} \\ \vdots \\ T_{(N_y-1)N_x} \\ T_{(N_y-1)N_x+1} \end{bmatrix} = \begin{bmatrix} T_b \\ T_b \\ \vdots \\ 0 \\ 0 \\ \vdots \\ T_b \\ T_b \end{bmatrix}$$

# Creating the linear system

$$T_{k-N_x} + T_{k-1} - 4T_k + T_{k+1} + T_{k+N_x} = 0$$

Create a *banded* matrix  $A$  in Matlab, by setting the coefficients for the internal cells:

```
Nx=5; %number of points along x direction
Ny=5; %number of points in the y direction
Nc=Nx*Ny; % Total number of points

e = ones(Nc,1);
A = spdiags([e,e,-4*e,e,e],[-Nx,-1,0,1,Nx],Nc,Nc);
b = zeros(Nc,1);
```

The function `spdiags` uses the following arguments:

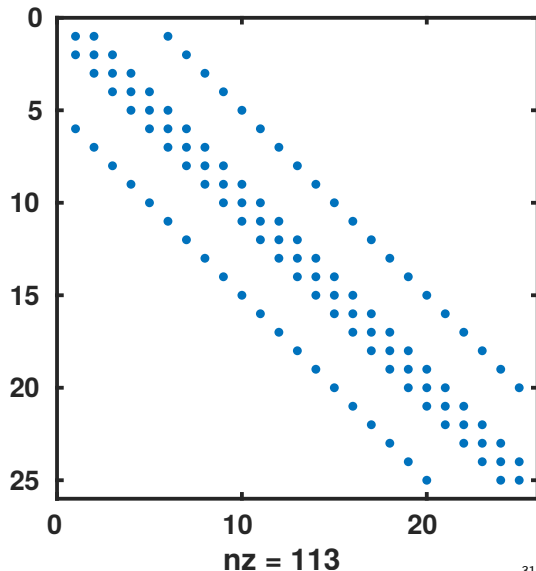
- The coefficients that have to be put on the diagonals arranged as columns in a matrix
- The position of the bands with respect to the main diagonal
- Size of the resulting matrix (in our case square  $N_x N_y \times N_x N_y$ )

# Matrix sparsity

- Let's check the matrix layout:

```
>> spy(A)
```

- This command shows the non-zero values of a matrix
- Apart from the main diagonal, there are offset bands!



# About boundary conditions

- For the nodes on the boundary, we have a simple equation:

$$T_{k,\text{boundary}} = \text{Some fixed value}$$

- However, we have set all nodes to be a function of their neighbors...
- Find the boundary node indices using  $k = i + Nx(j - 1)$ 
  - $i = 1, j = 1:Ny$
  - $i = Nx, j = 1:Ny$
  - $j = 1, i = 1:Nx$
  - $j = Ny, i = 1:Nx$
- Reset the row in  $A$  to zeros, set  $A_{kk} = 1$
- Set value in rhs:  $b_k = T_{k,\text{boundary}}$
- Boundary conditions are often more elaborate to implement! See `setBoundaryConditions.m`.

# Partial implementation of the boundary conditions

See `setBoundaryConditions.m`.

```
function [A,b] = setBoundaryConditions(A,b,Tb,Nx,Ny)

% Set boundary conditions over x-direction
for i=1:Nx
    j = 1;
    ind = i + Nx * (j-1);
    A(ind,:) = 0;    % Reset matrix for boundary cells
    A(ind,ind) = 1;  % Add a 1 on the diagonal
    b(ind) = Tb(1);
    j = Ny;
    ind = i + Nx * (j-1);
    A(ind,:) = 0;    % Reset matrix for boundary cells
    A(ind,ind) = 1;  % Add a 1 on the diagonal
    b(ind) = Tb(2);
end

%% Repeat for y-direction
```

# How applying boundary conditions affects the linear system

```
function [A,b] = setBoundaryConditions(A,b,Tb,Nx,Ny)
```



# How applying boundary conditions affects the linear system

```
function [A,b] = setBoundaryConditions(A,b,Tb,Nx,Ny)
```

- Make sure that matrix  $A$  and right hand side vector  $b$  are in your workspace, as well as  $N_x$  and  $N_y$
- Create a vector that holds the temperature at each boundary:

```
>> T = [10 20 30 40];
```

- Call the function, store  $A$  and  $b$  in new variables:

```
>> [A2,b2] = setBoundaryConditions(A,b,T,Nx,Ny);
```

- Check the new structure of the matrix and the right hand side:

```
>> subplot(1,2,1); spy(A2);  
>> subplot(1,2,2); spy(b2);
```

# A full program, including solver

The program and auxiliary functions are on Canvas (`solveLaplaceEq.m`)

```
function [x,y,T,A] = solveLaplaceEq(Nx,Ny)
% Solves the steady-state Laplace equation

Tb = [10 20 30 40]; % Fixed boundary temperatures

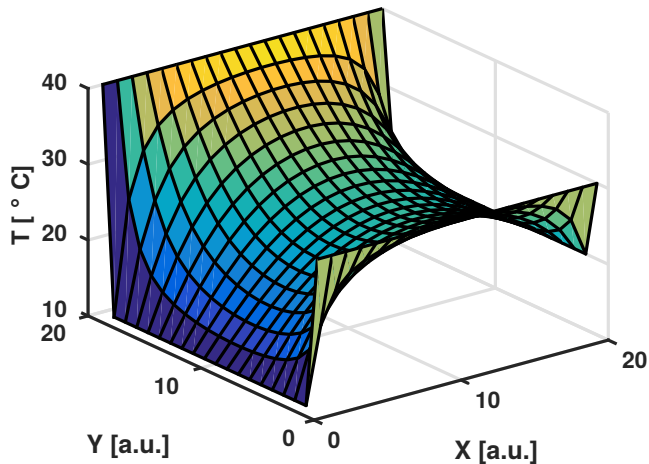
% Fill sparse matrix with [1 1 -4 1 1]
e = ones(Nx*Ny,1);
A = spdiags([e,e,-4*e,e,e],[-Nx,-1,0,1,Nx],Nx*Ny,Nx*Ny);
b = zeros(Nx*Ny,1);

[A,b] = setBoundaryConditions(A,b,Tb,Nx,Ny);

T = A\b; % Solve matrix
Tc = reshape(T,[Nx,Ny]); % Reshape x-vec to mat Nx,Ny
[xc yc] = meshgrid(1:Nx,1:Ny); % Get position arrays
surf(xc,yc,Tc); % Surface plot
```

## Sample results

Solved for a  $20 \times 20$  system with  $T_b = [10 \ 20 \ 30 \ 40]$ .



## Exercise: Verify the numerical solution using Fourier-series

A Fourier-series expansion for the steady-state heat conduction in a flat plate is given for a domain:  $x, y \in [0, 1]$ , with fixed-temperature boundaries  $T|_{x=0} = T|_{x=1} = T|_{y=0} = 0$  and  $T|_{y=1} = 1$ :

$$T = \frac{4}{\pi} \sum_{n=1}^{\infty} \frac{\sin(m\pi x) \sinh(m\pi y)}{m \sinh(m\pi)} \quad \text{with} \quad m = 2n - 1$$

Compute and plot the exact temperature profile in the 2D plate, and compare it with the numerical solution:

---

Hints:

- Use meshgrid to create a mesh in  $x$  and  $y$
- Compute the temperature using the Fourier series, use vectorised computations over  $x$  and  $y$  so that only 1 loop (over  $n$ ) is required.
- Solve the numerics for the same problem (note the boundary conditions)
- Compare the numerical and exact solutions (e.g. a plot).

## Exercise: Verify the numerical solution using Fourier-series

```
% Generate the mesh  
Nx = 35; Ny = 35;  
[x y] = meshgrid(linspace(0,1,Nx),linspace(0,1,Ny));  
T = zeros(size(x));
```

# Exercise: Verify the numerical solution using Fourier-series

```
% Generate the mesh
Nx = 35; Ny = 35;
[x y] = meshgrid(linspace(0,1,Nx),linspace(0,1,Ny));
T = zeros(size(x));
% Fourier series expansion
for n = 1:100
    m = 2*n-1;
    T = T + (sin(m*pi*x).*sinh(m*pi*y))./(m*sinh(m*pi));
end
Tex = T*4/pi;
```

# Exercise: Verify the numerical solution using Fourier-series

```
% Generate the mesh
Nx = 35; Ny = 35;
[x y] = meshgrid(linspace(0,1,Nx),linspace(0,1,Ny));
T = zeros(size(x));
% Fourier series expansion
for n = 1:100
    m = 2*n-1;
    T = T + (sin(m*pi*x).*sinh(m*pi*y))./(m*sinh(m*pi));
end
Tex = T*4/pi;
% Compute numerical solution and post-process

% First plot is created inside solveLaplaceEq, which also returns Tnum
figure; subplot(1,3,1)
[xc,yc,Tnum] = solveLaplaceEq(Nx,Ny)

% Plot exact (Fourier)
subplot(1,3,2); surf(x,y,Tex);
xlabel('x'); ylabel('y'); zlabel('T')

% Plot difference
subplot(1,3,3); surf(x,y,Tex-Tnum);
xlabel('x'); ylabel('y'); zlabel('T')
```

# LU decomposition of a sparse matrix

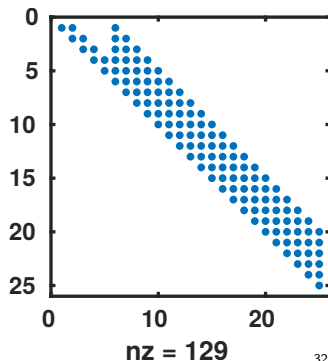
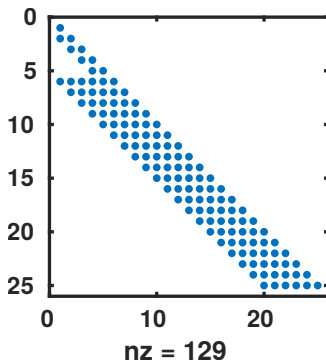
```
>> [L,U,P] = lu(A)
>> subplot(1,2,1)
>> spy(L)
>> subplot(1,2,2)
>> spy(U)
```



# LU decomposition of a sparse matrix

- With LU decomposition we produce matrices that are less sparse than the original matrix.
- Sparse storage often required, and also numerical techniques that fully utilizes this!

```
>> [L,U,P] = lu(A)
>> subplot(1,2,1)
>> spy(L)
>> subplot(1,2,2)
>> spy(U)
```



# LU decomposition

- LU decomposition and Gaussian elimination on a matrix like  $A$  requires more memory (with 3D problems, the offset in the diagonal would even be bigger!)
- In general extra memory allocation will not be a problem for MATLAB
- MATLAB is clever, in that sense that it attempts to reorder equations, to move elements closer to the diagonal)

# LU decomposition

- LU decomposition and Gaussian elimination on a matrix like  $A$  requires more memory (with 3D problems, the offset in the diagonal would even be bigger!)
- In general extra memory allocation will not be a problem for MATLAB
- MATLAB is clever, in that sense that it attempts to reorder equations, to move elements closer to the diagonal)

## Alternatives for elimination methods

- Use iterative methods when systems are large and sparse.
- Often such systems are encountered when we want to solve PDE's of higher dimensions

# Today's outline

- Introduction
- Sparse matrices
- Laplace's equation
- Creating a sparse system
- Iterative methods
- Summary

# Examples of iterative methods

- Jacobi method
- Gauss-Seidel method
- Successive over relaxation
  
- bicg — Bi-conjugate gradient method
- pcg — preconditioned conjugate gradient method
- gmres — generalized minimum residuals method
- bicgstab — Bi-conjugate gradient method

# The Jacobi method

- In our example we derived the following equation:

$$T_{k-N_x} + T_{k-1} - 4T_k + T_{k+1} + T_{k+N_x} = 0$$

- Rearranging gives:

$$T_k = \frac{T_{k-N_x} + T_{k-1} + T_{k+1} + T_{k+N_x}}{4}$$

# The Jacobi method

- In our example we derived the following equation:

$$T_{k-N_x} + T_{k-1} - 4T_k + T_{k+1} + T_{k+N_x} = 0$$

- Rearranging gives:

$$T_k = \frac{T_{k-N_x} + T_{k-1} + T_{k+1} + T_{k+N_x}}{4}$$

- In the Jacobi scheme the iteration proceeds as follows:
  - ① Start with an initial guess for the values of  $T$  at each node

# The Jacobi method

- In our example we derived the following equation:

$$T_{k-N_x} + T_{k-1} - 4T_k + T_{k+1} + T_{k+N_x} = 0$$

- Rearranging gives:

$$T_k = \frac{T_{k-N_x} + T_{k-1} + T_{k+1} + T_{k+N_x}}{4}$$

- In the Jacobi scheme the iteration proceeds as follows:
  - ① Start with an initial guess for the values of  $T$  at each node
  - ② Compute updated values and store a new vector:

$$T_k^{\text{new}} = \frac{T_{k-N_x}^{\text{old}} + T_{k-1}^{\text{old}} + T_{k+1}^{\text{old}} + T_{k+N_x}^{\text{old}}}{4}$$



# The Jacobi method

- In our example we derived the following equation:

$$T_{k-N_x} + T_{k-1} - 4T_k + T_{k+1} + T_{k+N_x} = 0$$

- Rearranging gives:

$$T_k = \frac{T_{k-N_x} + T_{k-1} + T_{k+1} + T_{k+N_x}}{4}$$

- In the Jacobi scheme the iteration proceeds as follows:
  - ① Start with an initial guess for the values of  $T$  at each node
  - ② Compute updated values and store a new vector:

$$T_k^{\text{new}} = \frac{T_{k-N_x}^{\text{old}} + T_{k-1}^{\text{old}} + T_{k+1}^{\text{old}} + T_{k+N_x}^{\text{old}}}{4}$$

- ③ Do this for all nodes

# The Jacobi method

- In our example we derived the following equation:

$$T_{k-N_x} + T_{k-1} - 4T_k + T_{k+1} + T_{k+N_x} = 0$$

- Rearranging gives:

$$T_k = \frac{T_{k-N_x} + T_{k-1} + T_{k+1} + T_{k+N_x}}{4}$$

- In the Jacobi scheme the iteration proceeds as follows:
  - ① Start with an initial guess for the values of  $T$  at each node
  - ② Compute updated values and store a new vector:

$$T_k^{\text{new}} = \frac{T_{k-N_x}^{\text{old}} + T_{k-1}^{\text{old}} + T_{k+1}^{\text{old}} + T_{k+N_x}^{\text{old}}}{4}$$

- ③ Do this for all nodes
- ④ Repeat the procedure until converged

# Jacobi method for Laplace's equation

See `laplace_jacobi.m` (from Canvas)

```
% Grid size  
nx = 40; ny = 40;
```

# Jacobi method for Laplace's equation

See `laplace_jacobi.m` (from Canvas)

```
% Grid size
nx = 40; ny = 40;
% The temperature field + boundaries at old and new times
T = zeros(nx,ny);
T(1,:) = 40; % Left
T(nx,:) = 60; % Right
T(:,1) = 20; % Bottom
T(:,ny) = 30; % Top
```

# Jacobi method for Laplace's equation

See `laplace_jacobi.m` (from Canvas)

```
% Grid size
nx = 40; ny = 40;
% The temperature field + boundaries at old and new times
T = zeros(nx,ny);
T(1,:) = 40; % Left
T(nx,:) = 60; % Right
T(:,1) = 20; % Bottom
T(:,ny) = 30; % Top
Tnew = T;
```

# Jacobi method for Laplace's equation

See `laplace_jacobi.m` (from Canvas)

```
% Grid size
nx = 40; ny = 40;
% The temperature field + boundaries at old and new times
T = zeros(nx,ny);
T(1,:) = 40; % Left
T(nx,:) = 60; % Right
T(:,1) = 20; % Bottom
T(:,ny) = 30; % Top
Tnew = T;
% For plotting
[x y] = meshgrid(1:nx, 1:ny);
```

# Jacobi method for Laplace's equation

See `laplace_jacobi.m` (from Canvas)

```
% Grid size
nx = 40; ny = 40;
% The temperature field + boundaries at old and new times
T = zeros(nx,ny);
T(1,:) = 40; % Left
T(nx,:) = 60; % Right
T(:,1) = 20; % Bottom
T(:,ny) = 30; % Top
Tnew = T;
% For plotting
[x y] = meshgrid(1:nx, 1:ny);
for iter = 1:1000
    for i = 2:nx-1
        for j = 2:ny-1
            Tnew(i,j) = (T(i-1,j)+T(i+1,j)+T(i,j-1)+T(i,j+1))/4.0;
        end
    end
end
```

# Jacobi method for Laplace's equation

See `laplace_jacobi.m` (from Canvas)

```
% Grid size
nx = 40; ny = 40;
% The temperature field + boundaries at old and new times
T = zeros(nx,ny);
T(1,:) = 40; % Left
T(nx,:) = 60; % Right
T(:,1) = 20; % Bottom
T(:,ny) = 30; % Top
Tnew = T;
% For plotting
[x y] = meshgrid(1:nx, 1:ny);
for iter = 1:1000
    for i = 2:nx-1
        for j = 2:ny-1
            Tnew(i,j) = (T(i-1,j)+T(i+1,j)+T(i,j-1)+T(i,j+1))/4.0;
        end
    end
    surf(x,y,Tnew);
    title(['Iteration: ' num2str(iter)]);
    drawnow
    T = Tnew; % Update T
end
```



## About the straightforward implementation

- The method as implemented works fine for a simple Laplace equation
- For generic systems of linear equations, the implementation cannot be used.

## About the straightforward implementation

- The method as implemented works fine for a simple Laplace equation
- For generic systems of linear equations, the implementation cannot be used.

We will now introduce the Jacobi method so it can be used for generic systems of linear equations.

# The Jacobi method with matrices

We can split our (banded) matrix  $A$  into a diagonal matrix  $D$  and a remainder  $R$ :

$$A = D + R$$

$$\begin{bmatrix} \times & \times & & & & & & & & \\ \times & \times & \times & & & & & & & \\ & \times & \times & \times & & & & & & \\ & & \times & \times & \times & & & & & \\ & & & \times & \times & \times & & & & \\ & & & & \times & \times & \times & & & \\ & & & & & \times & \times & \times & & \\ \times & & & & & & \times & \times & \times & \\ & \times & & & & & & \times & \times & \times \\ & & \times & & & & & & \times & \times \end{bmatrix} = \begin{bmatrix} \times & & & & & & & & & \\ & \times & & & & & & & & \\ & & \times & & & & & & & \\ & & & \times & & & & & & \\ & & & & \times & & & & & \\ & & & & & \times & & & & \\ & & & & & & \times & & & \\ & & & & & & & \times & & \\ & & & & & & & & \times & \\ & & & & & & & & & \times \end{bmatrix} + \begin{bmatrix} & \times & & & & & & & & \\ \times & & \times & & & & & & & \\ & \times & \times & \times & & & & & & \\ & & \times & \times & \times & & & & & \\ & & & \times & \times & \times & & & & \\ & & & & \times & \times & \times & & & \\ & & & & & \times & \times & \times & & \\ \times & & & & & & \times & \times & \times & \\ & \times & & & & & & \times & \times & \times \\ & & \times & & & & & & \times & \times \end{bmatrix}$$

# Jacobi method: solving a system

- We can solve  $AT = b$ , now written generally as  $Ax = b$ , by:

$$Ax = b$$

$$(D + R)x = b$$

$$Dx = b - Rx$$

$$Dx^{\text{new}} = b - Rx^{\text{old}}$$

$$x^{\text{new}} = D^{-1}(b - Rx^{\text{old}})$$

- Using the  $n$  and  $n + 1$  notation for old and new time steps, we find in general:

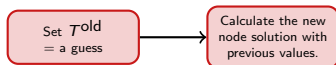
$$x^{n+1} = D^{-1}(b - Rx^n)$$

$$x_i^{n+1} = \frac{1}{A_{ii}} \left( b_i - \sum_{j \neq i} A_{ij} x_j^n \right)$$

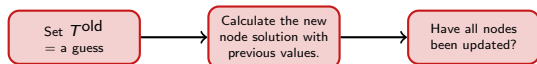
# Diagram of the Jacobi method

Set  $T^{\text{old}}$   
= a guess

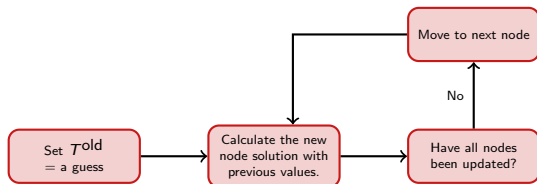
# Diagram of the Jacobi method



# Diagram of the Jacobi method

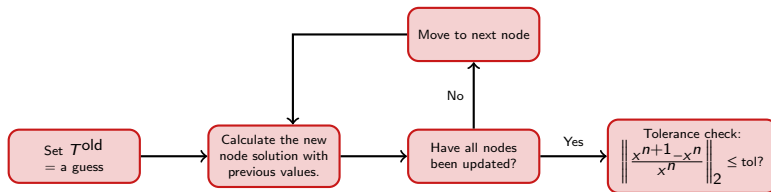


# Diagram of the Jacobi method

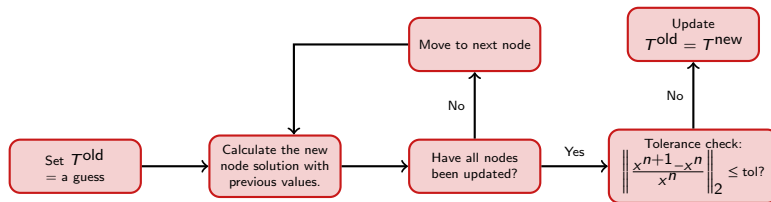




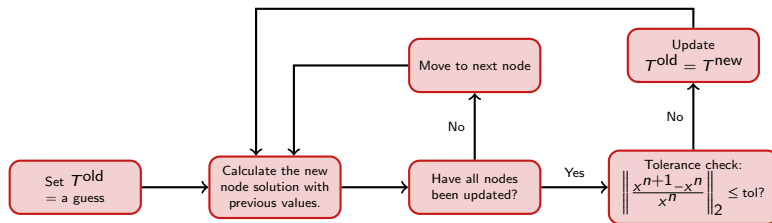
# Diagram of the Jacobi method



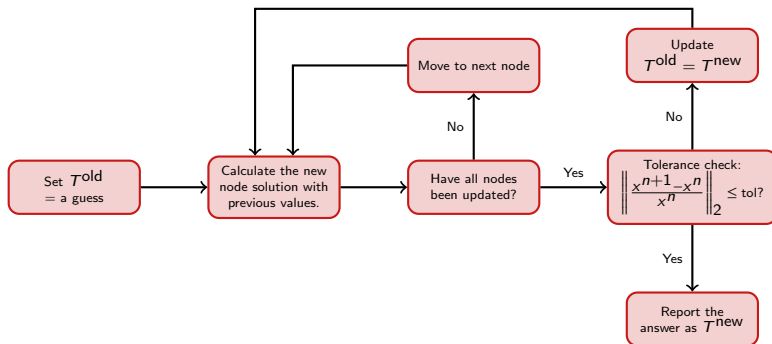
# Diagram of the Jacobi method



# Diagram of the Jacobi method



# Diagram of the Jacobi method



# The core of the solver

The full file is on Canvas, `solveJacobi.m`.

```
1  while ( xDiff > tol && it_jac < 1000 )
2      x_old = x;
3      for i=1:N
4          s = 0;
5          for j = 1:N
6              if (j ~= i)
7                  s = s+A(i,j)*x_old(j);
8              end
9          end
10         x(i) = (b(i)-s)/A(i,i);
11     end
12     it_jac = it_jac+1;
13     xDiff = norm((x-x_old)./x,2);
14 end
15 it_jac
```

# The core of the solver

The full file is on Canvas, `solveJacobi.m`.

```
1  while ( xDiff > tol && it_jac < 1000 )
2      x_old = x;
3      for i=1:N
4          s = 0;
5          for j = 1:N
6              if (j ~= i)
7                  s = s+A(i,j)*x_old(j);
8              end
9          end
10         x(i) = (b(i)-s)/A(i,i);
11     end
12     it_jac = it_jac+1;
13     xDiff = norm((x-x_old)./x,2);
14 end
15 it_jac
```

Try to call it from the `solveLaplaceEq.m` file, instead of using `\`.

# A few details on this algorithm

- The while loop holds two aspects
  - A convergence criterion (`norm((x-x_old)./x,2)> tol`). Some considerations are:
    - $L_1$ -norm (sum)
    - $L_2$ -norm (Euclidian distance)
    - $L_\infty$ -norm (max)
  - Protection against infinite loops (no convergence)

# A few details on this algorithm

- The while loop holds two aspects
  - A convergence criterion (`norm((x-x_old)./x,2)> tol`). Some considerations are:
    - $L_1$ -norm (sum)
    - $L_2$ -norm (Euclidian distance)
    - $L_\infty$ -norm (max)
  - Protection against infinite loops (no convergence)
- Reset the sum for each row, before summing for the new unknown node



## A few details on this algorithm

- The while loop holds two aspects
  - A convergence criterion (`norm((x-x_old)./x,2)> tol`). Some considerations are:
    - $L_1$ -norm (sum)
    - $L_2$ -norm (Euclidian distance)
    - $L_\infty$ -norm (max)
  - Protection against infinite loops (no convergence)
- Reset the sum for each row, before summing for the new unknown node
- Start vector  $x$  is not shown in the example, but should be there!
- It can have huge impact on performance!
- The for-loops also have a large performance penalty!

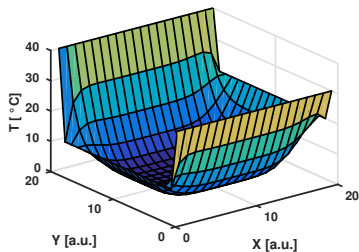
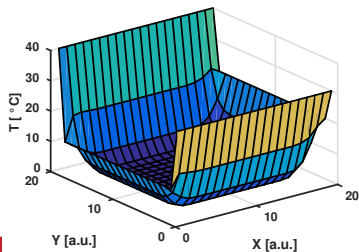
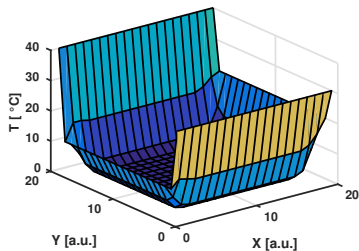
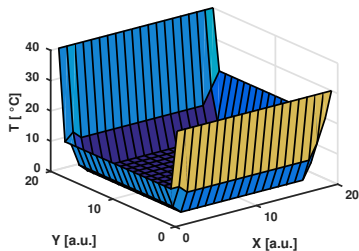
# The solver using array indices

Make a copy of the Jacobian solver, and replace the for-loop by a vector-operation:

```
% While not converged or max_it not reached
while ( xDiff > tol && it_jac < 1000 )
    x_old = x;
    for i=1:N
        % Sum off-diagonal*x_old
        offDiagonalIndex = [1:(i-1) (i+1):N];
        Aij_Xj = A(i,offDiagonalIndex)*x_old(offDiagonalIndex);

        % Compute new x value
        x(i) = (b(i)-Aij_Xj)/A(i,i);
    end
    it_jac = it_jac+1;
    xDiff = norm((x-x_old)./x,2);
end
```

# Iterations 1, 2, 3 and 10



# Gauss-Seidel method

The Gauss-Seidel method is quite similar to Jacobi method

- The only difference is that the new estimate  $x^{\text{new}}$  is returned to the solution  $x^{\text{old}}$  as soon as it is completed
- For following nodes, the updated solution is used immediately

# Gauss-Seidel method

The Gauss-Seidel method is quite similar to Jacobi method

- The only difference is that the new estimate  $x^{\text{new}}$  is returned to the solution  $x^{\text{old}}$  as soon as it is completed
- For following nodes, the updated solution is used immediately
- Our straightforward script (from the Jacobi method) is therefore changed easily:
  - Do not create a `Tnew` array (save memory!)
  - Do not store the solution in `Tnew`, but simply in `T`
  - Do not perform the update step `T=Tnew`
  - See `laplace_gaussseidel.m` for the algorithm.

# Gauss-Seidel method

The Gauss-Seidel method is quite similar to Jacobi method

- The only difference is that the new estimate  $x^{\text{new}}$  is returned to the solution  $x^{\text{old}}$  as soon as it is completed
- For following nodes, the updated solution is used immediately
- Our straightforward script (from the Jacobi method) is therefore changed easily:
  - Do not create a `Tnew` array (save memory!)
  - Do not store the solution in `Tnew`, but simply in `T`
  - Do not perform the update step `T=Tnew`
  - See `laplace_gaussseidel.m` for the algorithm.
- The straightforward script works well for the current Laplace equation, but we define the generic Gauss-Seidel algorithm on the following slides.

# Gauss-Seidel method

- Define a lower and strictly upper triangular matrix, such that  $A = L + U$
- Now we can solve  $AT=b$  by:

$$(L + U)T = b$$

$$LT = b - UT$$

$$LT^{\text{new}} = b - UT^{\text{old}}$$

$$T^{\text{new}} = L^{-1}(b - UT^{\text{old}})$$

- Using the  $n$  and  $n + 1$  notation for old and new time steps, we find in for the general Gauss-Seidel method:

$$x^{n+1} = L^{-1}(b - Ux^n)$$

$$x_i^{n+1} = \frac{1}{A_{ii}} \left( b_i - \sum_{j < i} A_{ij} x_j^{n+1} - \sum_{j > i} A_{ij} x_j^n \right)$$

# Today's outline

- Introduction
- Sparse matrices
- Laplace's equation
- Creating a sparse system
- Iterative methods
- **Summary**



# Summary

- Partial differential equations can be discretized into sparse systems of linear equations
- Sparse matrices can be stored in memory efficiently using specialised formats (e.g. compressed row storage)
- The Jacobi and Gauss–Seidel methods were introduced as iterative methods; other methods are based on the same principle (successive over-relaxation method, for example)
- Various implementation issues were discussed, e.g. vectorised computing, convergence tolerances

# Direct methods vs. Iterative methods

- Iterative methods converge *gradually* to a solution while direct methods (possibly with partial pivoting) factorise a (set of) matrix(ces) which allow to compute the solution by *substitution*.
- Direct methods generally use more memory, since they need to store also the result matrices.
- A strictly (or irreducibly) diagonally dominant matrix is a prerequisite for convergence of the Jacobi and Gauss-Seidel method.
- For real-life situations; 1D problems are generally solved with direct methods (LU decomposition). If you have systems of more than 1 dimension, a direct method still can be used, if there are no memory issues, otherwise an iterative method would be more attractive.