Linear equations 1

Linear algebra basics

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

Multiphase Reactors group
Eindhoven University of Technology

Advanced Numerical Methods for EngD (6PDPPD122), 2023-2024

Today's outline

- Introduction
- Solving a linear system
- Existence of solution
- Summary

Today's outline

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:

$$x + y + z = 4$$
$$2x + y + 3z = 7$$
$$3x + y + 6z = 5$$

Different views of linear systems

- Separate equations:

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

- Matrix mapping $Mx = b$:

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

Different views of linear systems

- Separate equations:

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

- 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:

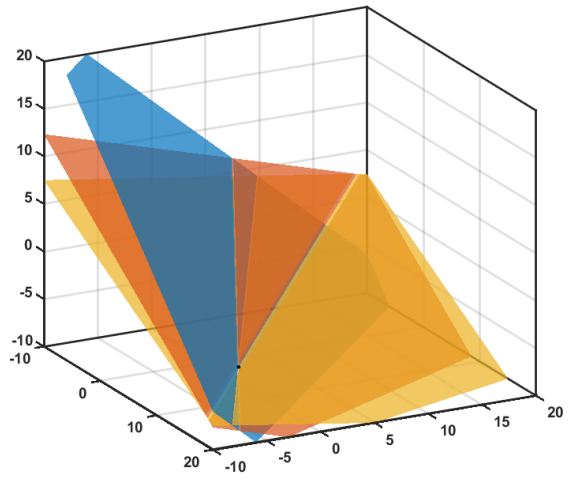
$$x + y + z = 4$$
$$2x + y + 3z = 7$$
$$3x + y + 6z = 5$$

- 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

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:

$$\begin{aligned} M\boldsymbol{x} &= \boldsymbol{b} \\ M^{-1}M\boldsymbol{x} &= M^{-1}\boldsymbol{b} \\ I\boldsymbol{x} &= M^{-1}\boldsymbol{b} \\ \boldsymbol{x} &= M^{-1}\boldsymbol{b} \end{aligned}$$

Solving a linear system in Python using the inverse

- Create the matrix:

```
1 >>> A = np.array([[1, 1, 1], [2, 1, 3], [3, 1, 6]])
```

Solving a linear system in Python using the inverse

- Create the matrix:

```
>>> A = np.array([[1, 1, 1], [2, 1, 3], [3, 1, 6]])
```

- Create solution vector:

```
>>> b = np.array([4, 7, 5])
```

Solving a linear system in Python using the inverse

- Create the matrix:

```
>>> A = np.array([[1, 1, 1], [2, 1, 3], [3, 1, 6]])
```

- Create solution vector:

```
>>> b = np.array([4, 7, 5])
```

- Get the matrix inverse:

```
>>> Ainv = np.linalg.inv(A)
```

Solving a linear system in Python using the inverse

- Create the matrix:

```
>>> A = np.array([[1, 1, 1], [2, 1, 3], [3, 1, 6]])
```

- Create solution vector:

```
>>> b = np.array([4, 7, 5])
```

- Get the matrix inverse:

```
>>> Ainv = np.linalg.inv(A)
```

- Compute the solution:

```
>>> x = np.dot(Ainv, b)
```

Solving a linear system in Python using the inverse

- Create the matrix:

```
>>> A = np.array([[1, 1, 1], [2, 1, 3], [3, 1, 6]])
```

- Create solution vector:

```
>>> b = np.array([4, 7, 5])
```

- Get the matrix inverse:

```
>>> Ainv = np.linalg.inv(A)
```

- Compute the solution:

```
>>> x = np.dot(Ainv, b)
```

- Python's internal direct solver:

```
>>> x = np.linalg.solve(A, b)
```

- These are black boxes! We are going over some methods later!

Exercise: performance of inverse computation

Create a script that generates matrices with random elements of various sizes $N \times N$ (e.g. values of $N \in \{10, 20, 50, 100, 200, \ldots, 5000, 10000\}$). Compute the inverse of each matrix, and use `import time` and `time.time()` to see the computing time for each inversion. Plot the time as a function of the matrix size N.

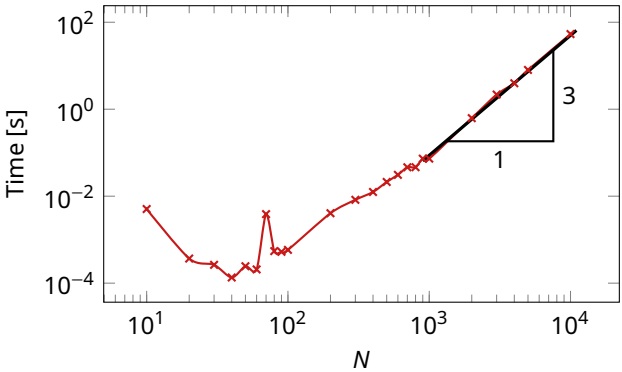
Exercise: performance of inverse computation

Create a script that generates matrices with random elements of various sizes $N \times N$ (e.g. values of $N \in \{10, 20, 50, 100, 200, \ldots, 5000, 10000\}$). Compute the inverse of each matrix, and use `import time` and `time.time()` to see the computing time for each inversion. Plot the time as a function of the matrix size N.

```
import numpy as np
import matplotlib.pyplot as plt
import time

# Generate random matrices of various sizes 's'.
# Invert the matrices and store the time required
# for the inversion. Plot the times vs 's'
s = np.array([10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000])
t_inv = []
for n in s:
   print(f'Working on size {n}')
   A = np.random.rand(n, n)
   start_time = time.time()
   Ainv = np.linalg.inv(A)
   t_inv.append(time.time() - start_time)

plt.loglog(s, t_inv)
plt.xlabel('N')
plt.ylabel('Time [s]')
plt.show()
```

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 $\mathcal{O}(N^3)$!

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 Python:

```
>>> numpy.linalg.matrix_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:

```
>>> numpy.linalg.matrix_rank(A)
>>> numpy.linalg.matrix_rank(np.column_stack((A,b))) # Concatenated matrices
```

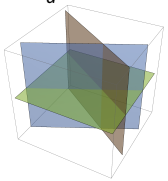
Our system: $Mx = b$

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

Existence of solutions for linear systems

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

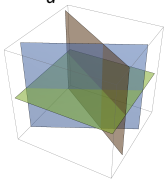
- Rank$(M) = n$:
  Unique solution



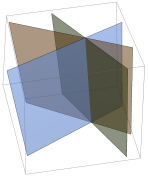
Existence of solutions for linear systems

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

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



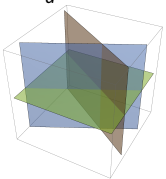
- $\text{Rank}(M) = \text{Rank}(M_a) < n$:
  Infinite number of solutions



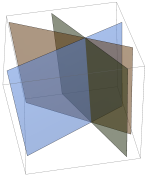
Existence of solutions for linear systems

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

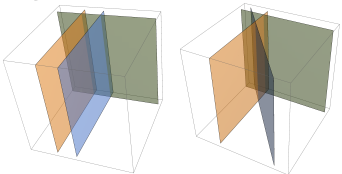
- $\text{Rank}(M) = n$:
  Unique solution



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



- $\text{Rank}(M) < n$, $\text{Rank}(M) < \text{Rank}(M_a)$:
  No solutions



Two examples

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

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

Summary

- Linear equations can be written as matrices
- Using the inverse, the solution can be determined
- Introduced the concept of computational complexity: matrix inversion scales with N^3
- Existence of a solution depends on the rank of a matrix

Linear equations 2

Direct methods

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

Multiphase Reactors group
Eindhoven University of Technology

Advanced Numerical Methods for EngD (6PDPPD122), 2023-2024

Today's outline

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

Today's outline

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

Define the linear system

Consider the system:

$$Ax = b$$

In general:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}$$

Desired solution:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \end{bmatrix}$$

Using row operations

- Use row operations to simplify the system. Eliminate element A_{10} by subtracting $A_{10}/A_{00} = d_{10}$ times row 1 from row 2.
- In this case, Row 1 is the pivot row, and A_{00} is the pivot element.

$$\left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ A_{10} & A_{11} & A_{12} & b_1 \\ A_{20} & A_{21} & A_{22} & b_2 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ 0 & A'_{11} & A'_{12} & b'_1 \\ A_{20} & A_{21} & A_{22} & b_2 \end{array}\right]$$

Using row operations

Eliminate element A_{10} using $d_{10} = A_{10}/A_{00}$.

$$\left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ A_{10} & A_{11} & A_{12} & b_1 \\ A_{20} & A_{21} & A_{22} & b_2 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ 0 & A'_{11} & A'_{12} & b'_1 \\ A_{20} & A_{21} & A_{22} & b_2 \end{array}\right]$$

Using row operations

Eliminate element A_{10} using $d_{10} = A_{10}/A_{00}$.

$$\left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ A_{10} & A_{11} & A_{12} & b_1 \\ A_{20} & A_{21} & A_{22} & b_2 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ 0 & A'_{11} & A'_{12} & b'_1 \\ A_{20} & A_{21} & A_{22} & b_2 \end{array}\right]$$

- $d_{10} \longrightarrow A_{10}/A_{00}$
- $A_{10} \longrightarrow A_{10} - A_{00}d_{10}$
- $A_{11} \longrightarrow A_{11} - A_{01}d_{10}$
- $A_{12} \longrightarrow A_{12} - A_{02}d_{10}$
- $b_1 \longrightarrow b_1 - b_0 d_{10}$

Using row operations

Eliminate element A_{10} using $d_{10} = A_{10}/A_{00}$.

$$\left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ A_{10} & A_{11} & A_{12} & b_1 \\ A_{20} & A_{21} & A_{22} & b_2 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ 0 & A'_{11} & A'_{12} & b'_1 \\ A_{20} & A_{21} & A_{22} & b_2 \end{array}\right]$$

- $d_{10} \longrightarrow A_{10}/A_{00}$
- $A_{10} \longrightarrow A_{10} - A_{00}d_{10}$
- $A_{11} \longrightarrow A_{11} - A_{01}d_{10}$
- $A_{12} \longrightarrow A_{12} - A_{02}d_{10}$
- $b_1 \longrightarrow b_1 - b_0 d_{10}$

```
d10 = A[1,0] / A[0,0]

A[1,0] = A[1,0] - A[0,0] * d10
A[1,1] = A[1,1] - A[0,1] * d10
A[1,2] = A[1,2] - A[0,2] * d10

b[1] = b[1] - b[0] * d10
```

Using row operations

Eliminate element A_{20} using $d_{20} = A_{20}/A_{00}$.

$$\left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ 0 & A'_{11} & A'_{12} & b'_1 \\ A_{20} & A_{21} & A_{22} & b_2 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ 0 & A'_{11} & A'_{12} & b'_1 \\ 0 & A'_{21} & A'_{22} & b'_2 \end{array}\right]$$

Using row operations

Eliminate element A_{20} using $d_{20} = A_{20}/A_{00}$.

$$\left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ 0 & A'_{11} & A'_{12} & b'_1 \\ A_{20} & A_{21} & A_{22} & b_2 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ 0 & A'_{11} & A'_{12} & b'_1 \\ 0 & A'_{21} & A'_{22} & b'_2 \end{array}\right]$$

- $d_{20} \longrightarrow A_{20}/A_{00}$
- $A_{20} \longrightarrow A_{20} - A_{00}d_{20}$
- $A_{21} \longrightarrow A_{21} - A_{01}d_{20}$
- $A_{22} \longrightarrow A_{22} - A_{02}d_{20}$
- $b_2 \longrightarrow b_2 - b_0 d_{20}$

```
d20 = A[2, 0] / A[0, 0]

A[2, 0] = A[2, 0] - A[0, 0] * d20
A[2, 1] = A[2, 1] - A[0, 1] * d20
A[2, 2] = A[2, 2] - A[0, 2] * d20
b[2] = b[2] - b[0] * d20
```

Using row operations

Eliminate element A'_{21} using $d_{21} = A'_{21}/A'_{11}$. Note that now the second row has become the pivot row.

$$\left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ 0 & A'_{11} & A'_{12} & b'_1 \\ 0 & A'_{21} & A'_{22} & b'_2 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ 0 & A'_{11} & A'_{12} & b'_1 \\ 0 & 0 & A''_{22} & b''_2 \end{array}\right]$$

Using row operations

Eliminate element A'_{21} using $d_{21} = A'_{21}/A'_{11}$. Note that now the second row has become the pivot row.

$$\left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ 0 & A'_{11} & A'_{12} & b'_1 \\ 0 & A'_{21} & A'_{22} & b'_2 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ 0 & A'_{11} & A'_{12} & b'_1 \\ 0 & 0 & A''_{22} & b''_2 \end{array}\right]$$

- $d_{21} \longrightarrow A_{21}/A'_{11}$
- $A_{21} \longrightarrow A_{21} - A'_{11}d_{21}$
- $A_{22} \longrightarrow A_{22} - A'_{12}d_{21}$
- $b_2 \longrightarrow b_2 - b'_2 d_{21}$

```
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
b[2] = b[2] - b[1] * d21
```

Using row operations

Eliminate element A'_{21} using $d_{21} = A'_{21}/A'_{11}$. Note that now the second row has become the pivot row.

$$\left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ 0 & A'_{11} & A'_{12} & b'_1 \\ 0 & A'_{21} & A'_{22} & b'_2 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ 0 & A'_{11} & A'_{12} & b'_1 \\ 0 & 0 & A''_{22} & b''_2 \end{array}\right]$$

- $d_{21} \longrightarrow A_{21}/A'_{11}$
- $A_{21} \longrightarrow A_{21} - A'_{11} d_{21}$
- $A_{22} \longrightarrow A_{22} - A'_{12} d_{21}$
- $b_2 \longrightarrow b_2 - b'_2 d_{21}$

```
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
b[2] = b[2] - b[1] * d21
```

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

Backsubstitution

The system now reads:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ 0 & A'_{11} & A'_{12} \\ 0 & 0 & A''_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_0 \\ b'_1 \\ b''_2 \end{bmatrix}$$

Backsubstitution

The system now reads:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ 0 & A'_{11} & A'_{12} \\ 0 & 0 & A''_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_0 \\ b'_1 \\ b''_2 \end{bmatrix}$$

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

$$x_2 = b''_2 / A''_{22}$$
$$x_1 = (b'_1 - A'_{12} x_2) / A'_{11}$$
$$x_0 = (b_0 - A_{01} x_1 - A_{02} x_2) / A_{00}$$

Backsubstitution

The system now reads:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ 0 & A'_{11} & A'_{12} \\ 0 & 0 & A''_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_0 \\ b'_1 \\ b''_2 \end{bmatrix}$$

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

$$x_2 = b''_2 / A''_{22}$$
$$x_1 = (b'_1 - A'_{12} x_2) / A'_{11}$$
$$x_0 = (b_0 - A_{01} x_1 - A_{02} x_2) / A_{00}$$

```
x = np.empty_like(b)
x[2] = b[2] / A[2,2]
x[1] = (b[1] - A[1,2] * x[2]) / A[1,1]
x[0] = (b[0] - A[0,1] * x[1] - A[0,2] * x[2]) / A[0,0]
```

In general:

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

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:

```
def gaussian_eliminate(A, b):
    pass # Your implementation here
```

- We will use *for-loops* instead of typing out each command line.
- Useful Python (with NumPy) shortcuts:
  - `A[0, :]` = $[A_{00}, A_{01}, A_{02}]$
  - `A[:, 1]` = $[A_{01}, A_{11}, A_{21}]$
  - `A[0, 1:]` = $[A_{01}, A_{02}]$
- A row operation could look like:

```
A[i, :] = A[i, :] - d * A[0, :]
```

The program: elimination step

An initial draft could look like:

```
def gaussian_eliminate_draft(A,b):
    """Perform elimination to obtain an upper triangular matrix"""
    A = np.array(A,dtype=np.float64)
    b = np.array(b,dtype=np.float64)

    assert A.shape[0] == A.shape[1], "Coefficient matrix should be square"

    N = len(b)
    for col in range(N-1): # Select pivot
        for row in range(col+1,N): # Loop over rows below pivot
            d = A[row,col] / A[col,col] # Choose elimination factor
            for element in range(row,N): # Elements from diagonal to right
                A[row,element] = A[row,element] - d * A[col,element]
            b[row] = b[row] - d * b[col]

    return A,b
```

The program: elimination step

Employing some of the row operations to create `gaussian_eliminate_v1`:

```
for element in range(row,N):
   A[row,element] = A[row,element] - d * A[col,element]
```

```
A[row,:] = A[row,:] - d * A[col,:]
```

The program: elimination step

Employing some of the row operations to create `gaussian_eliminate_v1`:

```
for element in range(row,N):
   A[row,element] = A[row,element] - d * A[col,element]
```

```
A[row,:] = A[row,:] - d * A[col,:]
```

```
def gaussian_eliminate_v1(A,b):
   A = np.array(A,dtype=np.float64)
   b = np.array(b,dtype=np.float64)

   assert A.shape[0] == A.shape[1], "Coefficient matrix should be square"

   N = len(b)
   for col in range(N-1):
      for row in range(col+1,N):
         d = A[row,col] / A[col,col]
         A[row,:] = A[row,:] - d * A[col,:]
         b[row] = b[row] - d * b[col]

   return A,b
```

Testing

Let's try to eliminate our linear system! If you create/downloaded our file `gaussjordan.py`, you can access the functions by importing them. The file should be stored where your own code/notebook is:

```
from gaussjordan import gaussian_eliminate_draft,gaussian_eliminate_v1
import numpy as np

A = np.array([[1, 1, 1], [2, 1, 3], [3, 1, 6]])
b = np.array([4, 7, 5])

Aprime,bprime = gaussian_eliminate_draft(A,b)
print(Aprime)
print(bprime)
```

The program: Backsubstitution

Now we have elimination working, let's create a back substitution algorithm too. Recall:

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

```
def backsubstitution_draft(A, b):
    """Back substitutes an upper triangular matrix to
          find x in Ax=b"""
    x = np.copy(b)
    N = len(b)

    for row in range(N-1, -1, -1):
        for i in range(row+1, N):
            x[row] = x[row] - A[row, i] * x[i]
        x[row] = x[row] / A[row, row]

    return x
```

The program: Backsubstitution

Now we have elimination working, let's create a back substitution algorithm too. Recall:

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

```python
def backsubstitution_draft(A, b):
    """Back substitutes an upper triangular matrix to
          find x in Ax=b"""
    x = np.copy(b)
    N = len(b)

    for row in range(N-1, -1, -1):
        for i in range(row+1, N):
            x[row] = x[row] - A[row, i] * x[i]
        x[row] = x[row] / A[row, row]

    return x
```

```python
def backsubstitution_v1(A,b):
    """Back substitutes an upper triangular matrix to find x in Ax=b"""
    x = np.empty_like(b)
    N = len(b)

    for row in range(N)[::-1]:
        x[row] = (b[row] - np.sum(A[row,row+1:] * x[row+1:])) / A[row,row]

    return x
```

A full Gauss Elimination solver

- The functions we just built are distributed in a Python module
- 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 `np.linalg.solve(A,b)`

Today's outline

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 in A below pivot in current column, store the index

```
index = np.argmax(np.abs(A[col:, col])) + col
```

Partial pivoting: implementing row swaps

- Find maximum element row in A below pivot in current column, store the index

```
index = np.argmax(np.abs(A[col:, col])) + col
```

- Swap rows through indexing (i.e. select rows by their index).

```
A[[i,col]] = A[[col,i]]
```

Partial pivoting: implementing row swaps

- Find maximum element row in A below pivot in current column, store the index

```
index = np.argmax(np.abs(A[col:, col])) + col
```

- Swap rows through indexing (i.e. select rows by their index).

```
A[[i,col]] = A[[col,i]]
```

- Do the same for `b` — swap through indexing

```
b[[i,col]] = b[[col,i]]
```

Adding the partial pivoting rules

```python
def gaussian_eliminate_partial_pivot(A,b):
    A = np.array(A,dtype=np.float64)
    b = np.array(b,dtype=np.float64)

    assert A.shape[0] == A.shape[1], "Coefficient matrix should be square"

    N = len(b)
    for col in range(N-1):
        index = np.argmax(np.abs(A[col:, col])) + col
        A[[i,col]] = A[[col,i]]
        b[[i,col]] = b[[col,i]]

        for row in range(col+1,N):
            d = A[row,col] / A[col,col]
            A[row,:] = A[row,:] - d * A[col,:]
            b[row] = b[row] - d * b[col]

    return A,b
```

Alternatives to this program

- Python can compute the solution to Ax=b with `scipy.linalg.solve` or `numpy.linalg.solve` solvers (more efficient)
- Too many loops. Loops make Python slow.
- There are fundamental problems with Gaussian elimination

Alternatives to this program

- Python can compute the solution to Ax=b with `scipy.linalg.solve` or `numpy.linalg.solve` solvers (more efficient)
- Too many loops. Loops make Python 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

- Python can compute the solution to Ax=b with `scipy.linalg.solve` or `numpy.linalg.solve` solvers (more efficient)
- Too many loops. Loops make Python 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

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 L and U such that $A = LU$
- Therefore $Ax = LUx = b$
- 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?
  - Gaussian elimination
  - Doolittle's decomposition
  - Crout's decomposition

Recipe for LU decomposition

LU decomposition can be done in Python using `scipy.linalg.lu`. Row swapping is done to get the largest values on the main diagonal of U (pivoting). A permutation matrix P is used to store row swapping such that:

$$PA = LU$$

Matrix P is returned in addition to L and U:

```
from scipy.linalg import lu
A = np.array([[1,1,1],[2,1,3],[3,1,6]])
P,L,U = lu(A)
with np.printoptions(precision=2, suppress=True,
      threshold=5):
   print(P,L,U,sep='\n\n')
```

```
[[0. 1. 0.]
 [0. 0. 1.]
 [1. 0. 0.]]

[[1. 0. 0. ]
 [0.33 1. 0. ]
 [0.67 0.5 1. ]]

[[ 3. 1. 6. ]
 [ 0. 0.67 -1. ]
 [ 0. 0. -0.5 ]]
```

Substitutions

$$Ax = b \quad \Rightarrow \quad PAx = Pb \equiv d$$
$$PA = LU \quad \Rightarrow \quad LUx = d$$

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

$$y_0 = \frac{d_0}{L_{00}}$$
$$y_i = \frac{d_i - \sum_{j=0}^{i} L_{ij} y_j}{L_{ii}}$$

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

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

How to use the solver in Python

```python
import numpy as np
from scipy.linalg import lu
from gaussjordan import backsubstitution_v1 as backwardSub
from gaussjordan import forwardsubstitution as forwardSub

# Example usage
A = np.random.rand(5, 5) # Get random matrix
P, L, U = lu(A) # Get L, U and P
b = np.random.rand(5) # Random b vector
d = P @ b # Permute b vector
y = forwardSub(L, d) # Can also do y=np.linalg.solve(L,d)
x = backwardSub(U, y) # Can also do x=np.linalg.solve(U,y)
rnorm = np.linalg.norm(A @ x - b) # Residual
```

How to use the solver in Python

```python
import numpy as np
from scipy.linalg import lu
from gaussjordan import backsubstitution_v1 as backwardSub
from gaussjordan import forwardsubstitution as forwardSub

# Example usage
A = np.random.rand(5, 5) # Get random matrix
P, L, U = lu(A) # Get L, U and P
b = np.random.rand(5) # Random b vector
d = P @ b # Permute b vector
y = forwardSub(L, d) # Can also do y=np.linalg.solve(L,d)
x = backwardSub(U, y) # Can also do x=np.linalg.solve(U,y)
rnorm = np.linalg.norm(A @ x - b) # Residual
```

- 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

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.
- Python's libraries have built in routines for solving linear equations and LU decomposition.
- 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, Dr.ir. Maike Baltussen, Prof.dr.ir. Martin van Sint Annaland

Multiphase Reactors group
Eindhoven University of Technology

Advanced Numerical Methods for EngD (6PDPPD122), 2023-2024

Today's outline

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

Today's outline

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:

```
import numpy as np
from scipy.sparse import csr_matrix

A = np.eye(10000)
print(A.nbytes)

S = csr_matrix(A)
print(S.data.nbytes)
```

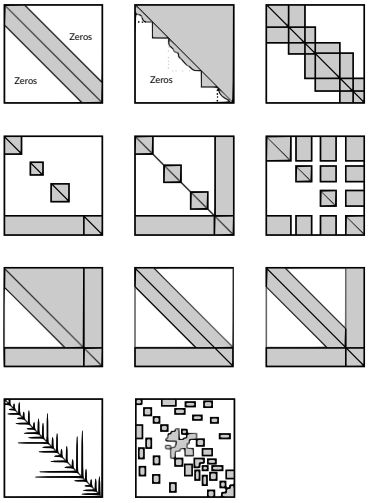
- Can you think of a way to achieve this?
- Sparse matrix formats: Yale, CRS, CCS

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

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

Sparse matrix layout examples



Today's outline

Laplace's equation

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

T = Temperature

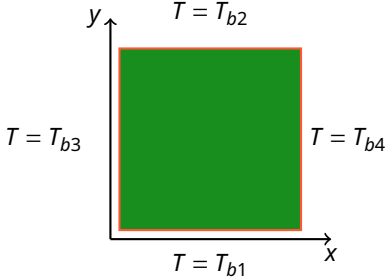
α = Thermal diffusivity

Laplace's equation

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

T = Temperature

α = Thermal diffusivity



Laplace's equation

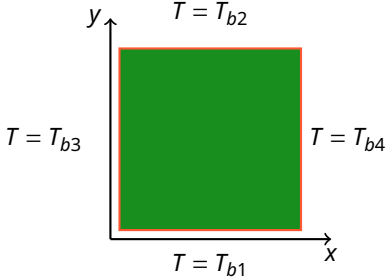
$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T$$

T = Temperature

α = Thermal diffusivity

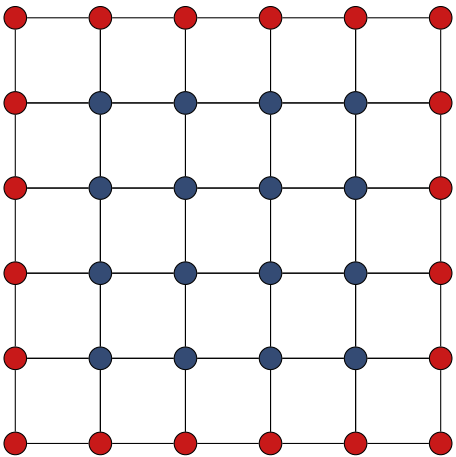
In steady state:

$$\nabla^2 T = 0$$



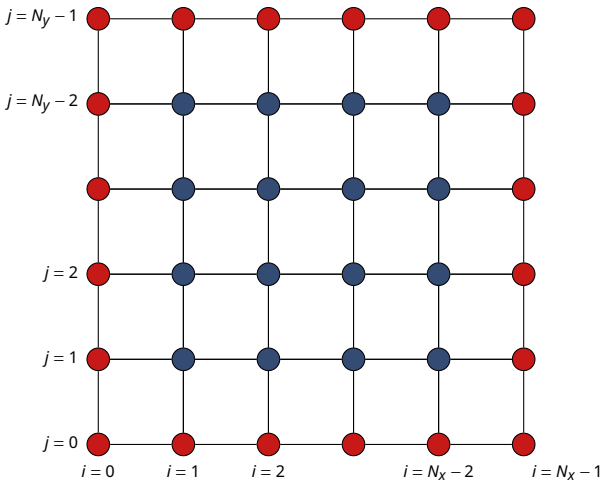
$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

Discretization of Laplace's equation (I)



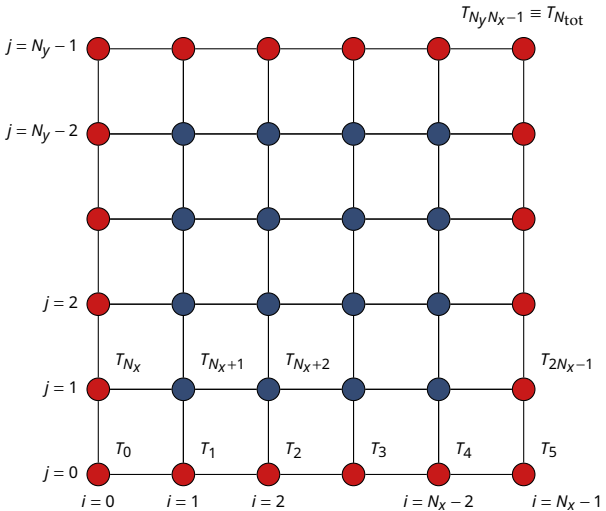
- Define a grid of points in x and y

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

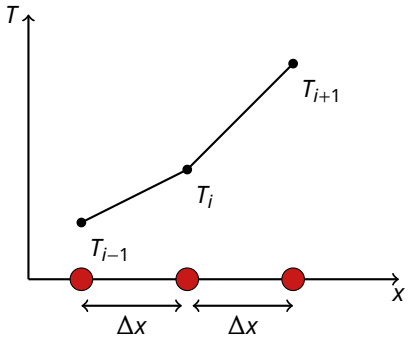
Discretization of Laplace's equation (I)



- 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: $T_{i,j} \rightarrow T_{i+jNx}$

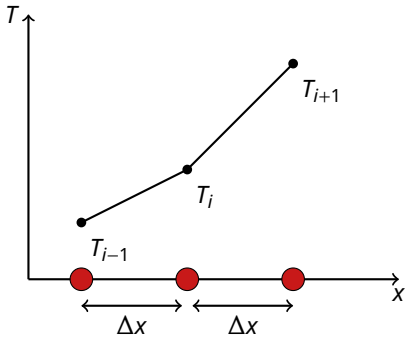
Discretization of Laplace's equation (II)

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:



$$\frac{\partial^2 T}{\partial x^2} \approx \frac{\left.\frac{\partial T}{\partial x}\right|_{i+\frac{1}{2}} - \left.\frac{\partial T}{\partial x}\right|_{i-\frac{1}{2}}}{\Delta x}$$

$$\approx \frac{\frac{(T_{i+1,j}-T_{i,j})}{\Delta x} - \frac{(T_{i,j}-T_{i-1,j})}{\Delta x}}{\Delta x}$$

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

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

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. $T_1 = 1T_b$).

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \cdots & 1 & \cdots & 1 & -4 & 1 & \cdots & 1 & \ddots & 0 \\ 0 & \cdots & 1 & \cdots & 1 & -4 & 1 & \cdots & 1 & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} T_0 \\ T_1 \\ \vdots \\ T_k \\ T_k + 1 \\ \vdots \\ T_{N_y N_x - 2} \\ T_{N_y N_x - 1} \end{bmatrix} = \begin{bmatrix} T_b \\ T_b \\ \vdots \\ 0 \\ 0 \\ \vdots \\ T_b \\ T_b \end{bmatrix}$$

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 Python, by setting the coefficients for the internal cells:

```
import numpy as np
from scipy.sparse import diags

Nx,Ny = 50,50 # Number of grid points along x,y direction
Nc = Nx*Ny # Total number of points

e = np.ones(Nc)
A = diags([e, e, -4*e, e, e], [-Nx, -1, 0, 1, Nx], shape=(Nc,Nc))
b = np.zeros(Nc)
```

The function `diags` uses the following arguments:

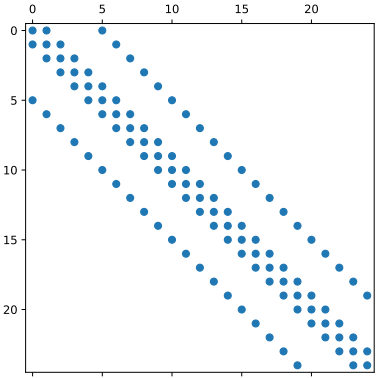
- The coefficients that have to be put on the diagonals arranged as columns in a matrix
- The position of the bands with respect to the main diagonal
- Size of the resulting matrix (in our case square $N_xN_y \times N_xN_y$)

Matrix sparsity

- Let's check the matrix layout by adding:

```
print(A)
plt.spy(A, marker='o',markersize=6)
```

- The *sparse* structure stores/prints only the nonzero elements
- `spy` shows the location of the nonzero values in the 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
- Solution: Determine the boundary node indices k and set the coefficients accordingly

```
bnd_bottom = np.arange(Nx)
bnd_left = np.arange(Ny) * Nx
bnd_right = bnd_left + Nx - 1
bnd_top = bnd_bottom + Nx*(Ny-1)
```

- Reset each row k in A to zeros, then set element $A_{kk} = 1$
- Set values in rhs: $b_k = T_{\text{boundary}}$
- Boundary conditions are often more elaborate to implement!

Implementation of the boundary conditions

A (shortened) version of the `set_boundary_conditions(A,b,Tb,Nx,Ny)` function:

```
def set_boundary_conditions(A, b, Tb, Nx, Ny):

    A = lil_matrix(A) # Required for efficient modification of the sparsity

    # Select nodes that lie at the boundaries
    bnd_bottom = np.arange(Nx)
    bnd_left = np.arange(Ny) * Nx
    bnd_right = bnd_left + Nx - 1
    bnd_top = bnd_bottom + Nx*(Ny-1)

    bnd_all = np.unique(np.concatenate((bnd_bottom,bnd_left,bnd_right,bnd_top)))

    # Reset the coefficient row to zero, add a 1 only on the main diagonal
    A[bnd_all,:] = 0
    A[bnd_all,bnd_all] = 1

    b[bnd_bottom] = Tb['bottom']
    b[bnd_left] = Tb['left']
    b[bnd_right] = Tb['right']
    b[bnd_top] = Tb['top']

    return A.tocsr(), b
```

How applying boundary conditions affects the linear system

Using the functions provided in `laplace_demo.py`:

```
Nx = Ny = 5 # number of internal grid cells over x/y-direction

T_boundary = {'bottom': 300, 'left': 1000, 'right': 1000, 'top': 500}

A,b = create_laplace_coefficient_matrix(Nx,Ny)
A,b = set_boundary_conditions(A, b, T_boundary, Nx, Ny)
```

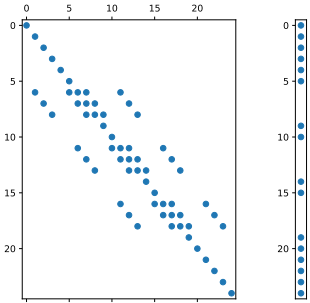
How applying boundary conditions affects the linear system

Using the functions provided in `laplace_demo.py`:

```
Nx = Ny = 5 # number of internal grid cells over x/y-direction

T_boundary = {'bottom': 300, 'left': 1000, 'right': 1000, 'top': 500}

A,b = create_laplace_coefficient_matrix(Nx,Ny)
A,b = set_boundary_conditions(A, b, T_boundary, Nx, Ny)
```

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

```
plt.subplot(121); plt.spy(A2);
plt.subplot(122); plt.spy(b[:,None]);
```

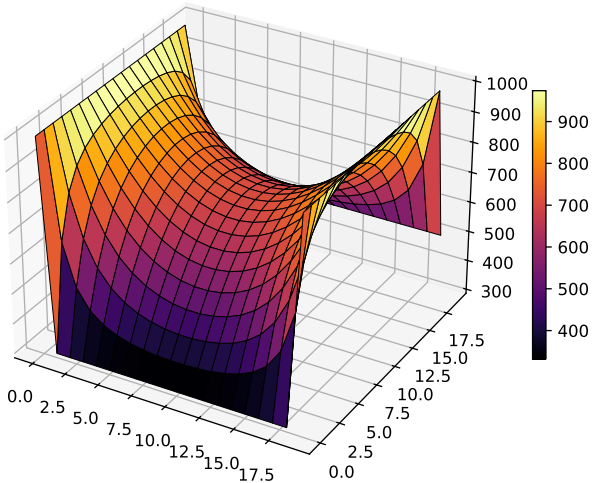


A full program, including solver

The program and auxiliary functions are on Canvas (`laplace_demo.py`)

```
import numpy as np
from scipy.sparse.linalg import spsolve
from matplotlib import cm
import matplotlib.pyplot as plt

Nx = Ny = 20

T_boundary = {'bottom': 300, 'left': 1000, 'right': 1000, 'top': 500}

A,b = create_laplace_coefficient_matrix(Nx,Ny)
A,b = set_boundary_conditions(A, b, T_boundary, Nx, Ny)

T = spsolve(A,b).reshape((Nx,Ny))

fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
x,y = np.meshgrid(np.arange(Nx),np.arange(Ny))
surf = ax.plot_surface(x,y,T,cmap=cm.inferno)
fig.colorbar(surf, shrink=0.5)
plt.show()
```

Sample results



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 $T\big|_{x=0} = T\big|_{x=1} = T\big|_{y=0} = 0$ and $T\big|_{y=1} = 1$:

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

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

Full script in `solveLaplaceEqAndFourier.py`

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

Nx = Ny = 20

xf,yf = np.meshgrid(np.linspace(0,1,Nx),np.linspace(0,1,Ny))
term = np.zeros_like(x)
N = 100

for m in range(1,N,2):
   term = term + (np.sin(m*np.pi*xf)*np.sinh(m*np.pi*yf)) / (m*np.sinh(m*np.pi))

sol = term * 4 / np.pi
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
surf = ax.plot_surface(x,y,sol,cmap=cm.inferno)
fig.colorbar(surf, shrink=0.5)
plt.show()
```

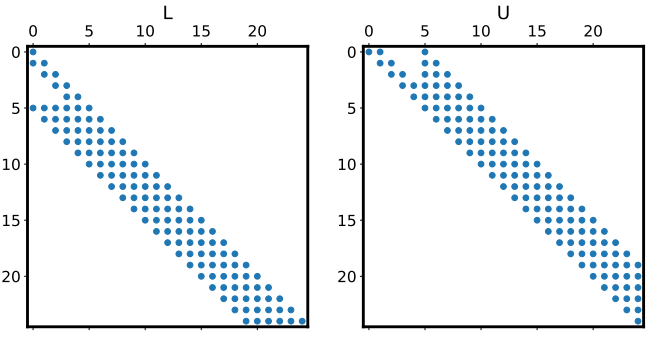
LU decomposition of a sparse matrix

```
import numpy as np
from scipy.linalg import lu
import matplotlib.pyplot as plt
from laplace_demo import
      create_laplace_coefficient_matrix

A,b = create_laplace_coefficient_matrix(5,5)

# Perform LU decomposition
# Note: lu does not work on sparse arrays,
# so we map to a full array
P,L,U = lu(A.toarray())

# Plot the sparsity patterns of L and U
plt.subplot(121)
plt.spy(L)
plt.title('L')
plt.subplot(122)
plt.spy(U)
plt.title('U')
plt.tight_layout()
```

LU decomposition of a sparse matrix

```python
import numpy as np
from scipy.linalg import lu
import matplotlib.pyplot as plt
from laplace_demo import
      create_laplace_coefficient_matrix

A,b = create_laplace_coefficient_matrix(5,5)

# Perform LU decomposition
# Note: lu does not work on sparse arrays,
# so we map to a full array
P,L,U = lu(A.toarray())

# Plot the sparsity patterns of L and U
plt.subplot(121)
plt.spy(L)
plt.title('L')
plt.subplot(122)
plt.spy(U)
plt.title('U')
plt.tight_layout()
```

- 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!



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 Python
- Python 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 Python
- Python 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

Examples of iterative methods

- Jacobi method
- Gauss-Seidel method
- Succesive 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.py` for animation included (from Canvas)

```python
import numpy as np
import matplotlib.pyplot as plt

# Set grid resolution
nx = 40
ny = 40

# Set old solution array
T = np.zeros((nx,ny))

# Set boundary conditions
T[0,:] = 40 # Left
T[nx-1,:] = 60 # Right
T[:,0] = 20 # Bottom
T[:,ny-1] = 30 # Top

# Set new solution array (inc bnd conditions)
Tnew = T.copy()
```

```python
# Create grid for plotting
x,y = np.meshgrid(np.arange(1,nx+1), np.arange(1,ny+1))

# Perform iterations
for iter in range(1,1001):
   for i in range(1,nx-1):
      for j in range(1,ny-1):
         # Calculate new solution
         Tnew[i,j] = \
            (T[i-1,j]+T[i+1,j]+T[i,j-1]+T[i,j+1])/4.0
   T = Tnew.copy()
```

Jacobi method for Laplace's equation

See `laplace_jacobi.py` for animation included (from Canvas)

```python
import numpy as np
import matplotlib.pyplot as plt

# Set grid resolution
nx = 40
ny = 40

# Set old solution array
T = np.zeros((nx,ny))

# Set boundary conditions
T[0,:] = 40 # Left
T[nx-1,:] = 60 # Right
T[:,0] = 20 # Bottom
T[:,ny-1] = 30 # Top

# Set new solution array (inc bnd conditions)
Tnew = T.copy()
```

```python
# Create grid for plotting
x,y = np.meshgrid(np.arange(1,nx+1), np.arange(1,ny+1))

# Perform iterations
for iter in range(1,1001):
   for i in range(1,nx-1):
      for j in range(1,ny-1):
         # Calculate new solution
         Tnew[i,j] = \
            (T[i-1,j]+T[i+1,j]+T[i,j-1]+T[i,j+1])/4.0
   T = Tnew.copy()
```

→ Try to modify this script so that 1 cell/block of cells in the center is kept at 100 degrees

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:

$$A = D + R$$

Jacobi method: solving a system

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

$$\begin{aligned} Ax &= b \\ (D+R)x &= b \\ Dx &= b - Rx \\ Dx^{\text{new}} &= b - Rx^{\text{old}} \\ x^{\text{new}} &= D^{-1}(b - Rx^{\text{old}}) \end{aligned}$$

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

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

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

Diagram of the Jacobi method

Set T^{old} = a guess

Diagram of the Jacobi method

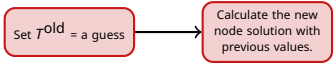


Diagram of the Jacobi method

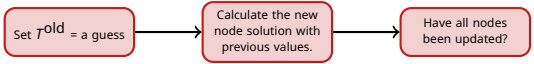


Diagram of the Jacobi method

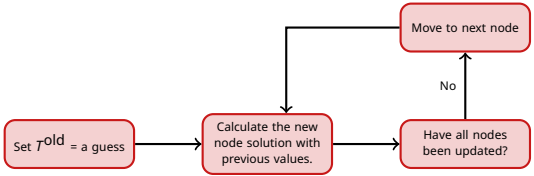


Diagram of the Jacobi method

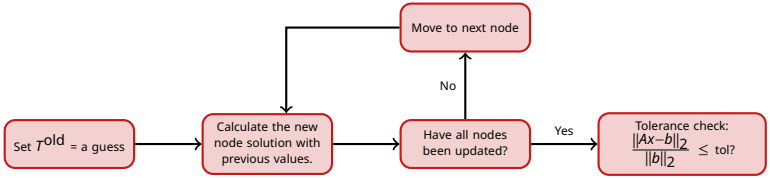


Diagram of the Jacobi method

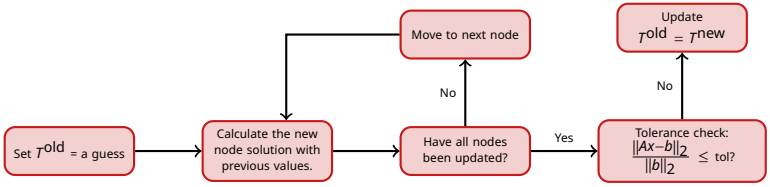


Diagram of the Jacobi method

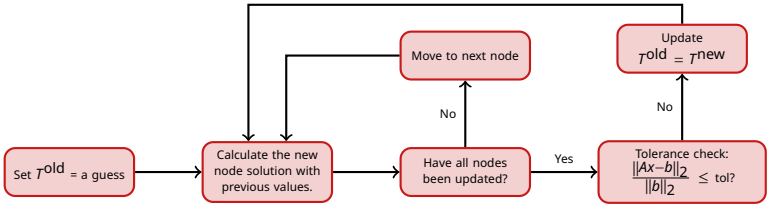
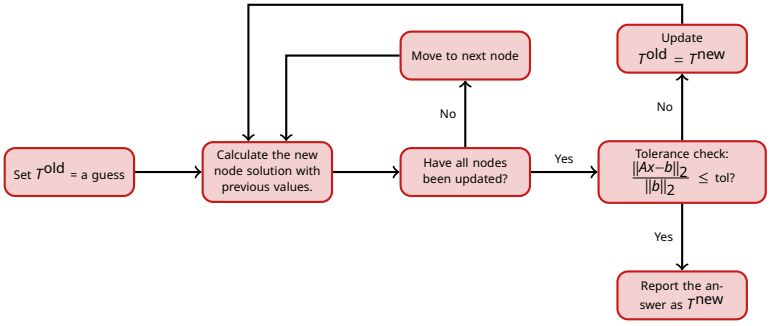


Diagram of the Jacobi method



The core of the solver

The full function `jacobi(A, b, tol=1e-2)` is on Canvas, see `it_methods.py`. The gist is:

```
# While not converged or max_it not reached
while (x_diff > tol and it_jac < 1000):
   x_old = x.copy()
   for i in range(N):
      s = 0
      for j in range(N):
         if j != i:
            # Sum off-diagonal*x_old
            s += A[i,j] * x_old[j]
      # Compute new x value
      x[i] = (b[i] - s) / A[i,i]

   # Increase number of iterations
   it_jac += 1
   x_diff = norm(A@x - b)/norm(b)
```

The core of the solver

The full function `jacobi(A, b, tol=1e-2)` is on Canvas, see `it_methods.py`. The gist is:

```
# While not converged or max_it not reached
while (x_diff > tol and it_jac < 1000):
   x_old = x.copy()
   for i in range(N):
      s = 0
      for j in range(N):
         if j != i:
            # Sum off-diagonal*x_old
            s += A[i,j] * x_old[j]
      # Compute new x value
      x[i] = (b[i] - s) / A[i,i]

   # Increase number of iterations
   it_jac += 1
   x_diff = norm(A@x - b)/norm(b)
```

Try to call it from the `laplace_demo.py` file, instead of using `spsolve`.

A few details on this algorithm

- The while loop holds two aspects
  - A convergence criterion (`norm(A@x - b)/norm(b)> tol`). Some considerations are:
    - L_1-norm (sum)
    - L_2-norm (Euclidian distance)
    - L_∞-norm (max)
  - Protection against infinite loops (no convergence)

A few details on this algorithm

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

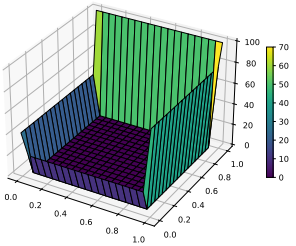
- Start vector x is not shown in the example, but should be there!
- It can have huge impact on performance!
- The for-loops also have a large performance penalty!

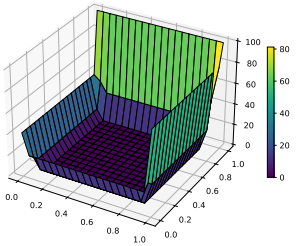
The solver using array indices

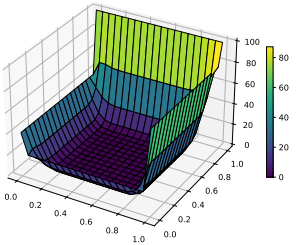
Make a copy of the Jacobian solver, and replace the for-loop on `j` by a vector-operation in a new function `jacobi_vec(A, b, tol=1e-2)`:

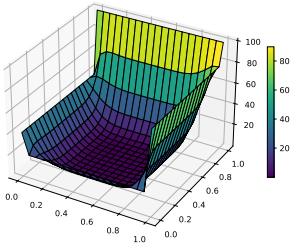
```
# While not converged or max_it not reached
while (x_diff > tol and it_jac < 1000):
    x_old = x.copy()
    for i in range(N):
        j = np.r_[np.arange(i),np.arange(i+1,N)]
        # Sum off-diagonal*x_old
        s = A[i,j] @ x_old[j]
        # Compute new x value
        x[i] = (b[i] - s) / A[i,i]

    # Increase number of iterations
    it_jac += 1
    x_diff = norm(A@x - b)/norm(b)
```

Iterations 1, 2, 5 and 10









Gauss-Seidel method

The Gauss-Seidel method is quite similar to Jacobi method

- The only difference is that the new estimate x^{new} is returned to the solution x^{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^{new} is returned to the solution x^{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 `gaussseidel(A, b, tol=1e-2)` 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^{new} is returned to the solution x^{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 `gaussseidel(A, b, tol=1e-2)` 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:

$$\begin{aligned}(L+U)T &= b \\ LT &= b - UT \\ LT^{\text{new}} &= b - UT^{\text{old}} \\ T^{\text{new}} &= L^{-1}(b - UT^{\text{old}})\end{aligned}$$

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

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

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

Create yourself: Gauss-Seidel method

- Create a copy of the `jacobi` method and rename it to `gaussseidel`
- Rework the inner algorithm to reflect the changes for the Gauss-Seidel method
- Test! Perform a timing check and check if the solution is correct.
- Next, create a new copy of the just created method and vectorize it, analogous to our vectorized Jacobi method

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.