

# 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 (6BER03), 2024-2025

# 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:

```
1 import numpy as np
2 from scipy.sparse import csr_matrix
3
4 A = np.eye(100000)
5 print(A.nbytes)
6
7 S = csr_matrix(A)
8 print(S.data.nbytes)
```

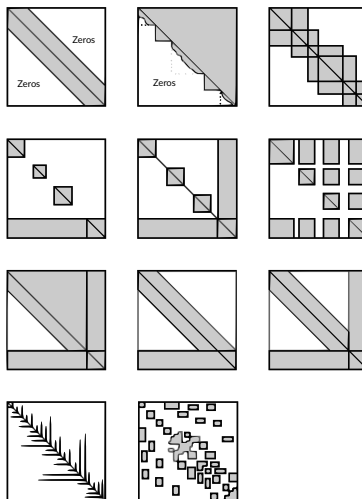
- Can you think of a way to achieve this?
- Sparse matrix formats: Yale, CRS, CCS

# 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

- 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

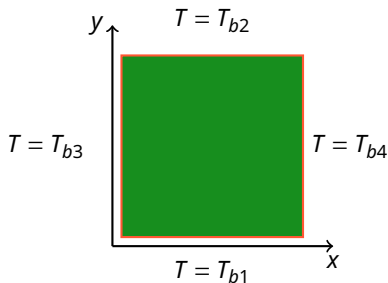


# Laplace's equation

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

$T$  = Temperature

$\alpha$  = Thermal diffusivity



# Laplace's equation

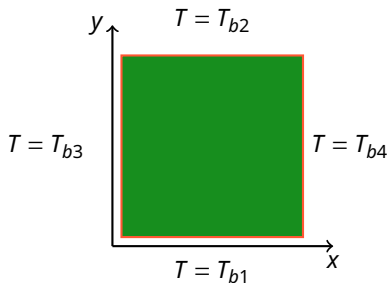
$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T$$

$T$  = Temperature

$\alpha$  = Thermal diffusivity

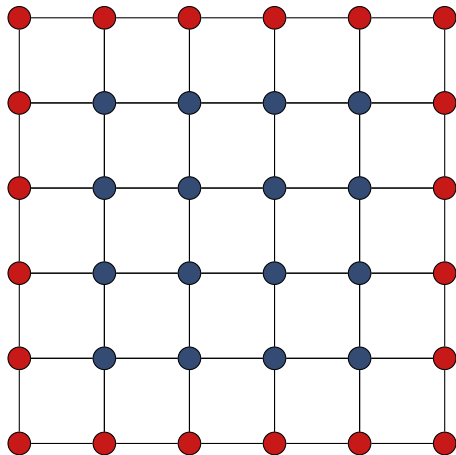
In steady state:

$$\nabla^2 T = 0$$



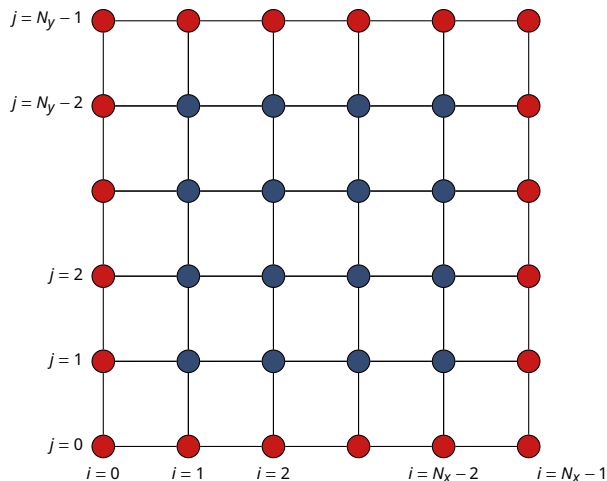
$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

# Discretization of Laplace's equation (I)



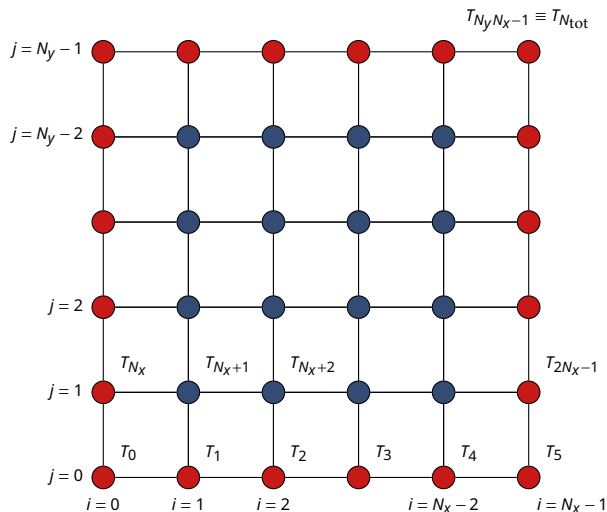
- Define a grid of points in  $x$  and  $y$

# 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  $j$

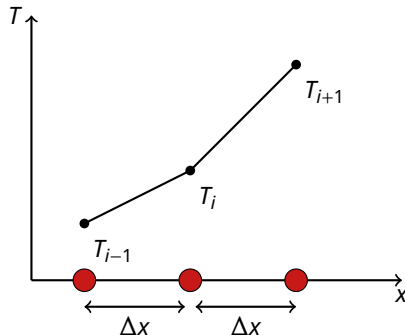
# Discretization of Laplace's equation (I)



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

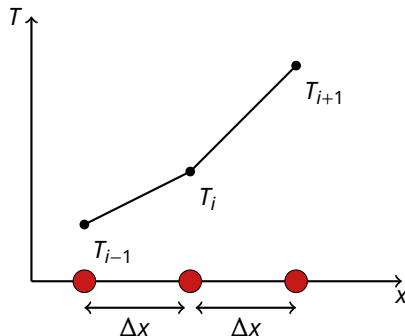
## Discretization of Laplace's equation (II)

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.  $T_1 = 1T_b$ ).

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

# 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:

```

1 import numpy as np
2 from scipy.sparse import diags
3
4 Nx,Ny = 50,50 # Number of grid points along x,y direction
5 Nc = Nx*Ny # Total number of points
6
7 e = np.ones(Nc)
8 A = diags([e, e, -4*e, e, e], [-Nx, -1, 0, 1, Nx], shape=(Nc,Nc))
9 b = np.zeros(Nc)

```

The function `diags` uses the following arguments:

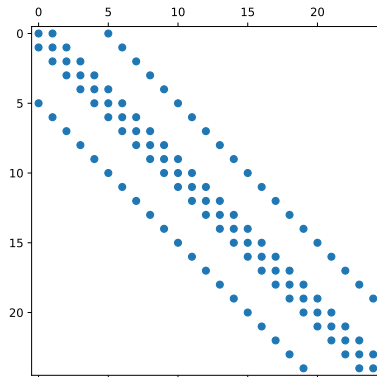
- The coefficients that have to be put on the diagonals arranged as columns in a matrix
- The position of the bands with respect to the main diagonal
- Size of the resulting matrix (in our case square  $N_x N_y \times N_x N_y$ )

# Matrix sparsity

- Let's check the matrix layout by adding:

```
1 print(A)  
2 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

```
1 bnd_bottom = np.arange(Nx)
2 bnd_left = np.arange(Ny) * Nx
3 bnd_right = bnd_left + Nx - 1
4 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:

```

1 def set_boundary_conditions(A, b, Tb, Nx, Ny):
2
3     A = lil_matrix(A) # Required for efficient modification of the sparsity
4
5     # Select nodes that lie at the boundaries
6     bnd_bottom = np.arange(Nx)
7     bnd_left = np.arange(Ny) * Nx
8     bnd_right = bnd_left + Nx - 1
9     bnd_top = bnd_bottom + Nx*(Ny-1)
10
11     bnd_all = np.unique(np.concatenate((bnd_bottom, bnd_left, bnd_right, bnd_top)))
12
13     # Reset the coefficient row to zero, add a 1 only on the main diagonal
14     A[bnd_all,:] = 0
15     A[bnd_all,bnd_all] = 1
16
17     b[bnd_bottom] = Tb['bottom']
18     b[bnd_left] = Tb['left']
19     b[bnd_right] = Tb['right']
20     b[bnd_top] = Tb['top']
21
22     return A.tocsr(), b

```



# How applying boundary conditions affects the linear system

Using the functions provided in `laplace_demo.py`:

```
1 Nx = Ny = 5 # number of internal grid cells over x/y-direction
2
3 T_boundary = {'bottom': 300, 'left': 1000, 'right': 1000, 'top': 500}
4
5 A,b = create_laplace_coefficient_matrix(Nx,Ny)
6 A,b = set_boundary_conditions(A, b, T_boundary, Nx, Ny)
```

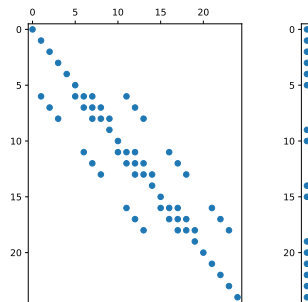
# How applying boundary conditions affects the linear system

Using the functions provided in `laplace_demo.py`:

```
1 Nx = Ny = 5 # number of internal grid cells over x/y-direction
2
3 T_boundary = {'bottom': 300, 'left': 1000, 'right': 1000, 'top': 500}
4
5 A, b = create_laplace_coefficient_matrix(Nx, Ny)
6 A, b = set_boundary_conditions(A, b, T_boundary, Nx, Ny)
```

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

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



# A full program, including solver

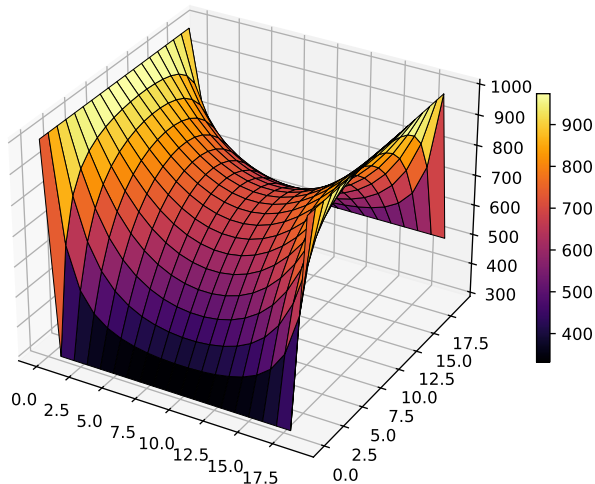
The program and auxiliary functions are on Canvas (laplace\_demo.py)

```

1 import numpy as np
2 from scipy.sparse.linalg import spsolve
3 from matplotlib import cm
4 import matplotlib.pyplot as plt
5
6 Nx = Ny = 20
7
8 T_boundary = {'bottom': 300, 'left': 1000, 'right': 1000, 'top': 500}
9
10 A,b = create_laplace_coefficient_matrix(Nx,Ny)
11 A,b = set_boundary_conditions(A, b, T_boundary, Nx, Ny)
12
13 T = spsolve(A,b).reshape((Nx,Ny))
14
15 fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
16 x,y = np.meshgrid(np.arange(Nx),np.arange(Ny))
17 surf = ax.plot_surface(x,y,T,cmap=cm.inferno)
18 fig.colorbar(surf, shrink=0.5)
19 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  $T|_{x=0} = T|_{x=1} = T|_{y=0} = 0$  and  $T|_{y=1} = 1$ :

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

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`

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import matplotlib.cm as cm
4
5 Nx = Ny = 20
6
7 xf,yf = np.meshgrid(np.linspace(0,1,Nx),np.linspace(0,1,Ny))
8 term = np.zeros_like(xf)
9 N = 100
10
11 for m in range(1,N,2):
12     term = term + (np.sin(m*np.pi*xf)*np.sinh(m*np.pi*yf)) / (m*np.sinh(m*np.pi))
13
14 sol = term * 4 / np.pi
15 fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
16 surf = ax.plot_surface(xf,yf,sol,cmap=cm.inferno)
17 fig.colorbar(surf, shrink=0.5)
18 plt.show()
```

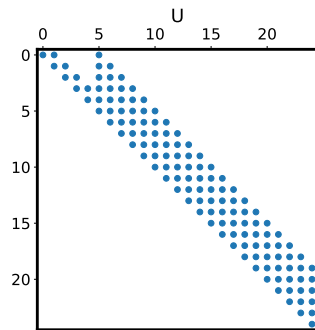
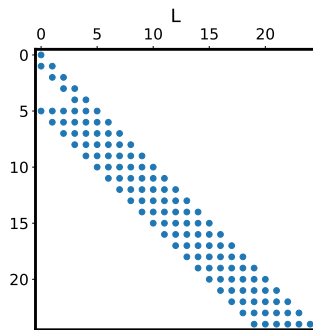
# LU decomposition of a sparse matrix

```
1 import numpy as np
2 from scipy.linalg import lu
3 import matplotlib.pyplot as plt
4 from laplace_demo import
   create_laplace_coefficient_matrix
5
6 A,b = create_laplace_coefficient_matrix(5,5)
7
8 # Perform LU decomposition
9 # Note: lu does not work on sparse arrays,
10 # so we map to a full array
11 P,L,U = lu(A.toarray())
12
13 # Plot the sparsity patterns of L and U
14 plt.subplot(121)
15 plt.spy(L)
16 plt.title('L')
17 plt.subplot(122)
18 plt.spy(U)
19 plt.title('U')
20 plt.tight_layout()
```

# 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!

```
1 import numpy as np
2 from scipy.linalg import lu
3 import matplotlib.pyplot as plt
4 from laplace_demo import
5     create_laplace_coefficient_matrix
6
7 A,b = create_laplace_coefficient_matrix(5,5)
8
9 # Perform LU decomposition
10 # Note: lu does not work on sparse arrays,
11 # so we map to a full array
12 P,L,U = lu(A.toarray())
13
14 # Plot the sparsity patterns of L and U
15 plt.subplot(121)
16 plt.spy(L)
17 plt.title('L')
18 plt.subplot(122)
19 plt.spy(U)
20 plt.title('U')
21 plt.tight_layout()
```





# 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

- 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.py` for animation included (from Canvas)

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Set grid resolution
5 nx = 40
6 ny = 40
7
8 # Set old solution array
9 T = np.zeros((nx,ny))
10
11 # Set boundary conditions
12 T[0,:] = 40 # Left
13 T[nx-1,:] = 60 # Right
14 T[:,0] = 20 # Bottom
15 T[:,ny-1] = 30 # Top
16
17 # Set new solution array (inc bnd
18   conditions)
19 Tnew = T.copy()
```

```
1 # Create grid for plotting
2 x,y = np.meshgrid(np.arange(1,nx+1), np.arange(1,ny+1))
3
4 # Perform iterations
5 for iter in range(1,1001):
6     for i in range(1,nx-1):
7         for j in range(1,ny-1):
8             # Calculate new solution
9             Tnew[i,j] = \
10                 (T[i-1,j]+T[i+1,j]+T[i,j-1]+T[i,j+1])/4.0
11 T = Tnew.copy()
```

# Jacobi method for Laplace's equation

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

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Set grid resolution
5 nx = 40
6 ny = 40
7
8 # Set old solution array
9 T = np.zeros((nx,ny))
10
11 # Set boundary conditions
12 T[0,:] = 40 # Left
13 T[nx-1,:] = 60 # Right
14 T[:,0] = 20 # Bottom
15 T[:,ny-1] = 30 # Top
16
17 # Set new solution array (inc bnd
18   conditions)
19 Tnew = T.copy()
```

```
1 # Create grid for plotting
2 x,y = np.meshgrid(np.arange(1,nx+1), np.arange(1,ny+1))
3
4 # Perform iterations
5 for iter in range(1,1001):
6     for i in range(1,nx-1):
7         for j in range(1,ny-1):
8             # Calculate new solution
9             Tnew[i,j] = \
10                 (T[i-1,j]+T[i+1,j]+T[i,j-1]+T[i,j+1])/4.0
11 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  $R$ :

$$A = D + R$$

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

# 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:

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

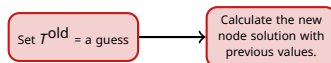
$$x_i^{n+1} = \frac{1}{A_{ii}} \left( b_i - \sum_{j \neq i} A_{ij} x_j^n \right)$$

# Diagram of the Jacobi method

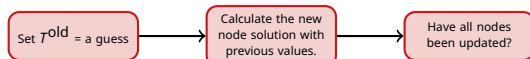
Set  $\tau^{\text{old}}$  = a guess



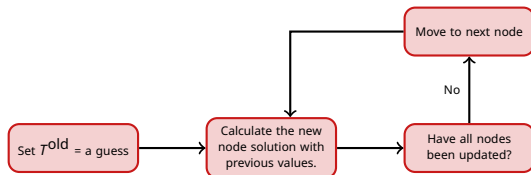
# Diagram of the Jacobi method



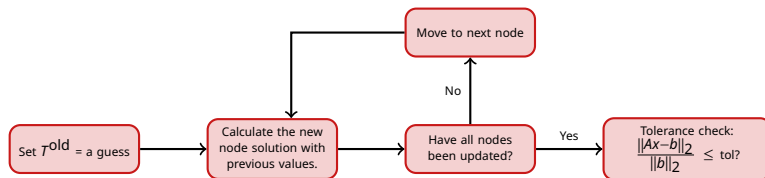
# Diagram of the Jacobi method



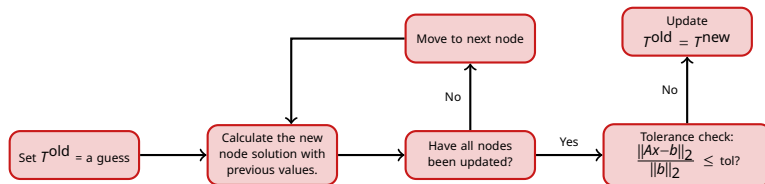
# Diagram of the Jacobi method



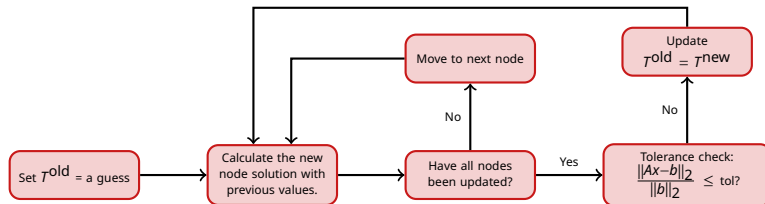
# Diagram of the Jacobi method



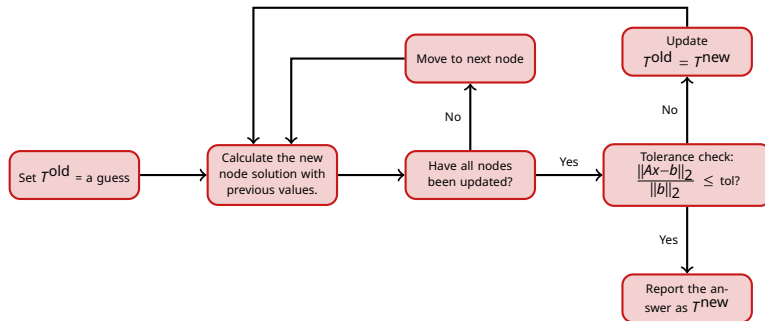
# Diagram of the Jacobi method



# Diagram of the Jacobi method



# Diagram of the Jacobi method



# The core of the solver

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

```
1 # While not converged or max_it not reached
2 while (x_diff > tol and it_jac < 1000):
3     x_old = x.copy()
4     for i in range(N):
5         s = 0
6         for j in range(N):
7             if j != i:
8                 # Sum off-diagonal*x_old
9                 s += A[i,j] * x_old[j]
10            # Compute new x value
11            x[i] = (b[i] - s) / A[i,i]
12
13 # Increase number of iterations
14 it_jac += 1
15 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:

```
1 # While not converged or max_it not reached
2 while (x_diff > tol and it_jac < 1000):
3     x_old = x.copy()
4     for i in range(N):
5         s = 0
6         for j in range(N):
7             if j != i:
8                 # Sum off-diagonal*x_old
9                 s += A[i,j] * x_old[j]
10            # Compute new x value
11            x[i] = (b[i] - s) / A[i,i]
12
13 # Increase number of iterations
14 it_jac += 1
15 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 ( $\text{norm}(A@x - b)/\text{norm}(b) > \text{tol}$ ). 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 ( $\text{norm}(A@x - b) / \text{norm}(b) > \text{tol}$ ). 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

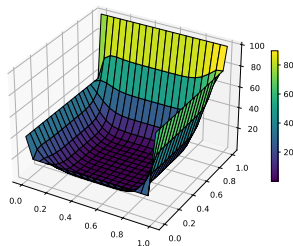
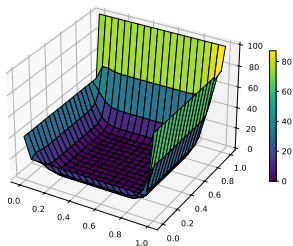
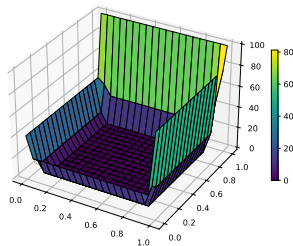
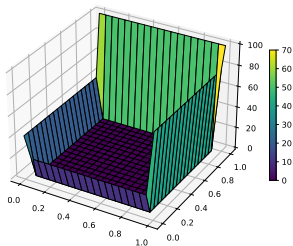
- The while loop holds two aspects
  - A convergence criterion ( $\text{norm}(A@x - b)/\text{norm}(b) > \text{tol}$ ). Some considerations are:
    - $L_1$ -norm (sum)
    - $L_2$ -norm (Euclidian distance)
    - $L_\infty$ -norm (max)
  - Protection against infinite loops (no convergence)
- Reset the 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 on  $j$  by a vector-operation in a new function `jacobi_vec(A, b, tol=1e-2)`:

```
1 # While not converged or max_it not reached
2 while (x_diff > tol and it_jac < 1000):
3     x_old = x.copy()
4     for i in range(N):
5         j = np.r_[np.arange(i), np.arange(i+1, N)]
6         # Sum off-diagonal * x_old
7         s = A[i, j] @ x_old[j]
8         # Compute new x value
9         x[i] = (b[i] - s) / A[i, i]
10
11 # Increase number of iterations
12 it_jac += 1
13 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  $x^{\text{new}}$  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 `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  $x^{\text{new}}$  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 `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:

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

$$LT = b - UT$$

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

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

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

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

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

# 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.