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

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:

```python
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)
```

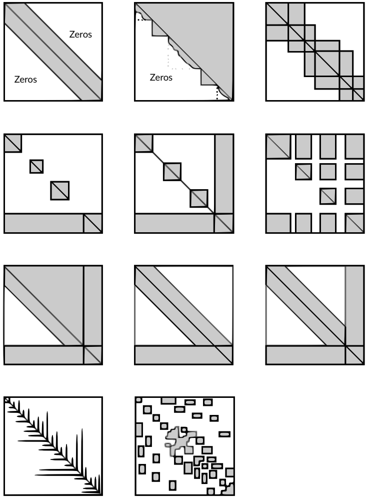
- Can you think of a way to achieve this?
- Sparse matrix formats: Yale, CRS, CCS

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

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Laplace's equation

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

$T =$ Temperature

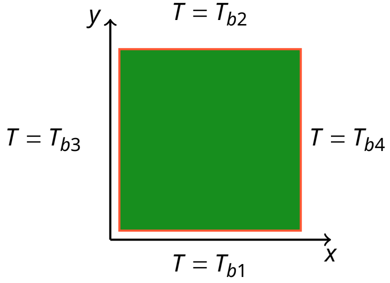
$\alpha =$ Thermal diffusivity

Laplace's equation

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

T = Temperature

α = Thermal diffusivity



Laplace's equation

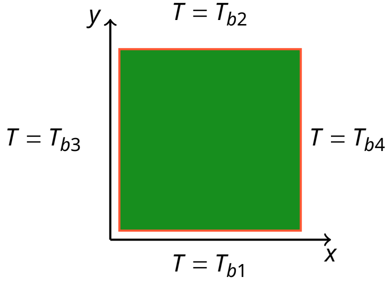
$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T$$

$T =$ Temperature

$\alpha =$ Thermal diffusivity

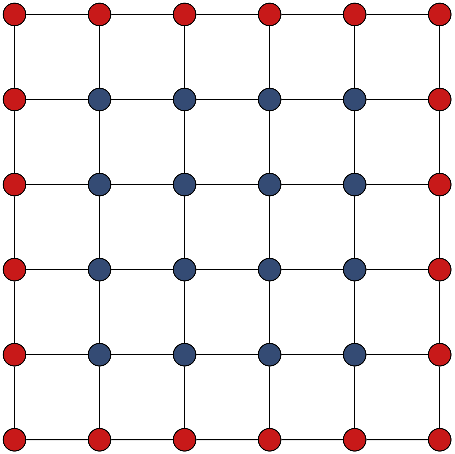
In steady state:

$$\nabla^2 T = 0$$



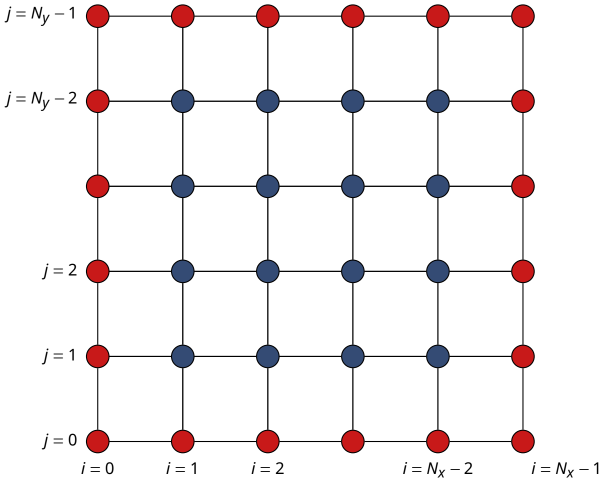
$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

Discretization of Laplace’s equation (I)



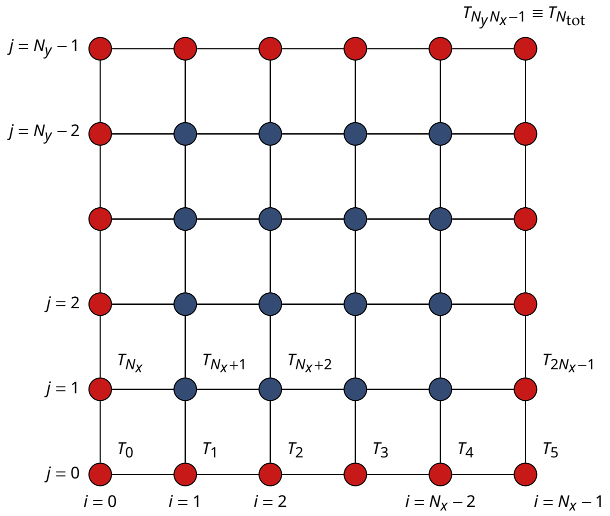
- Define a grid of points in x and y

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

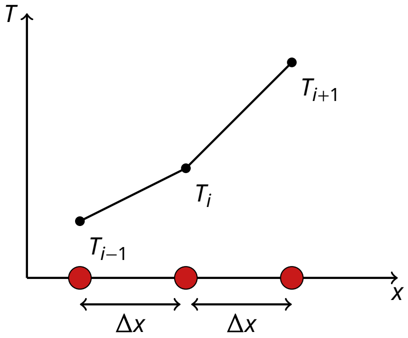
Discretization of Laplace's equation (I)



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

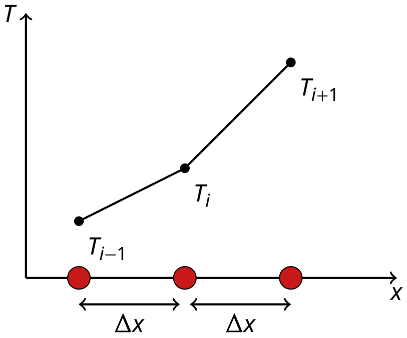
Discretization of Laplace's equation (II)

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:



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

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

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

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:

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

EINDHOVEN UNIVERSITY OF TECHNOLOGY

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. $T_1 = 1T_b$).

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

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:

```python
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:

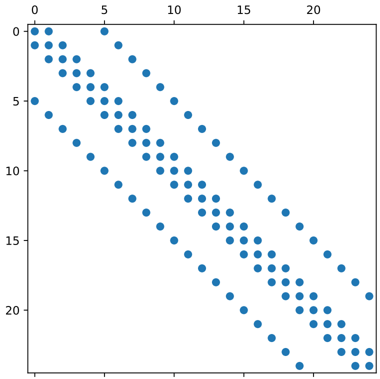
- The coefficients that have to be put on the diagonals arranged as columns in a matrix
- The position of the bands with respect to the main diagonal
- Size of the resulting matrix (in our case square $N_xN_y \times N_xN_y$)

Matrix sparsity

- Let's check the matrix layout by adding:

```python
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!

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Implementation of the boundary conditions

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

```python
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`:

```python
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)
```

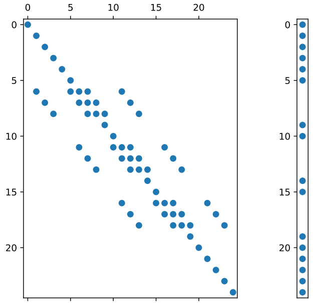
How applying boundary conditions affects the linear system

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]);
```

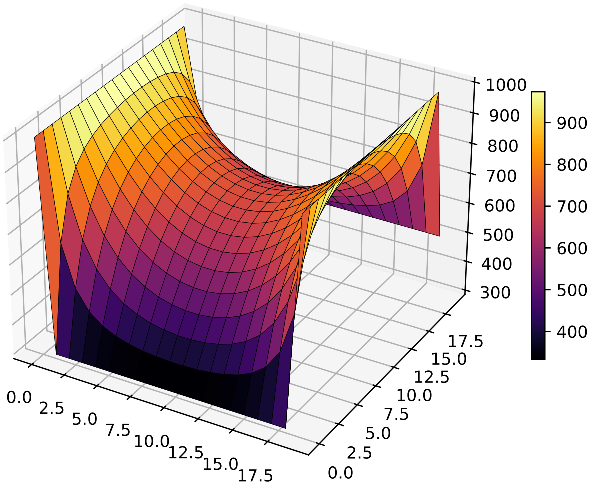


A full program, including solver

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

```python
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



EINDHOVEN UNIVERSITY OF TECHNOLOGY

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()
```

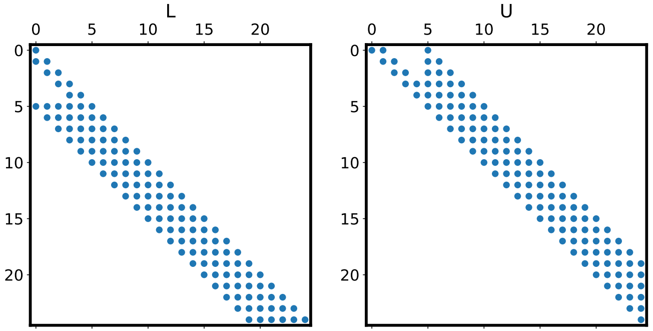
LU decomposition of a sparse matrix

```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()
```

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

EINDHOVEN UNIVERSITY OF TECHNOLOGY

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:

$$A \quad = \quad D \quad + \quad R$$

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

Jacobi method: solving a system

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

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

- 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

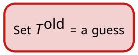


Diagram of the Jacobi method

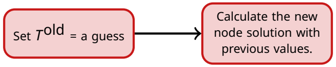

Set T^{old} = a guess
Calculate the new node solution with previous values.

Diagram of the Jacobi method

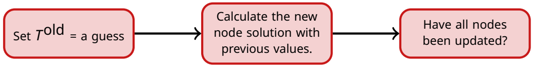

Set T^{old} = a guess
Calculate the new node solution with previous values.
Have all nodes been updated?

Diagram of the Jacobi method

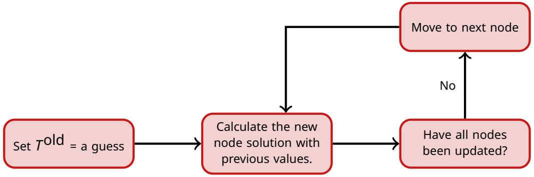

Move to next node
No
Set T^{old} = a guess
Calculate the new node solution with previous values.
Have all nodes been updated?

Diagram of the Jacobi method

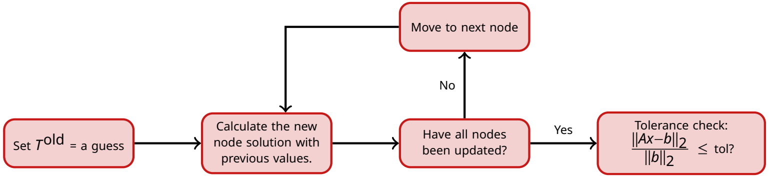

Set T^{old} = a guess
Calculate the new node solution with previous values.
Have all nodes been updated?
No
Move to next node
Yes
Tolerance check: $\frac{\|Ax-b\|_2}{\|b\|_2} \leq$ tol?

Diagram of the Jacobi method

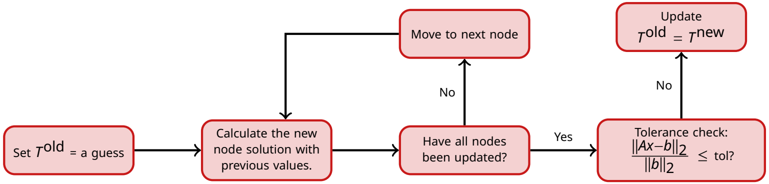

Set T^{old} = a guess
Calculate the new node solution with previous values.
Have all nodes been updated?
No
Move to next node
Yes
Tolerance check: $\frac{\|Ax-b\|_2}{\|b\|_2} \leq$ tol?
No
Update $T^{old} = T^{new}$

Diagram of the Jacobi method

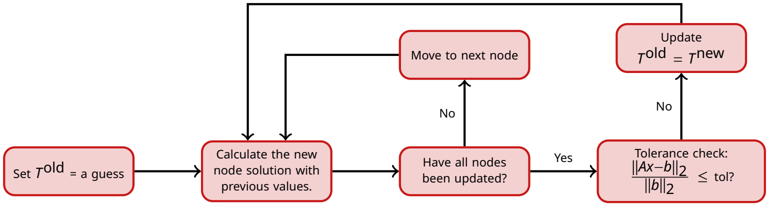
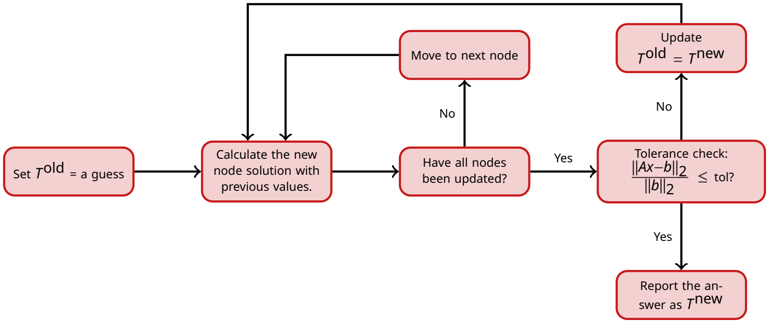

Set T^{old} = a guess
Calculate the new node solution with previous values.
Have all nodes been updated?
No
Move to next node
Yes
Tolerance check: $\frac{\|Ax-b\|_2}{\|b\|_2} \leq$ tol?
No
Update $T^{\text{old}} = T^{\text{new}}$

Diagram of the Jacobi method


Set T^{old} = a guess
Calculate the new node solution with previous values.
Have all nodes been updated?
No
Move to next node
Yes
Tolerance check: $\frac{\|Ax-b\|_2}{\|b\|_2} \leq$ tol?
No
Update $T^{\text{old}} = T^{\text{new}}$
Yes
Report the answer as T^{new}

The core of the solver

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

```python
# 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:

```python
# 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

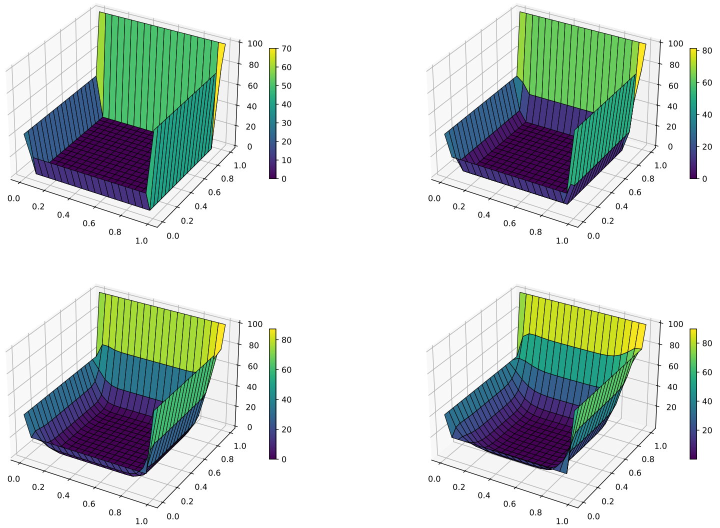
- Start vector x is not shown in the example, but should be there!
- It can have huge impact on performance!
- The for-loops also have a large performance penalty!

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)`:

```python
# 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

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Today's outline

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.