Linear equation solvers

Direct methods (elimination methods)

Ivo Roghair, Martin van Sint Annaland

Chemical Process Intensification,
Eindhoven University of Technology

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:

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

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

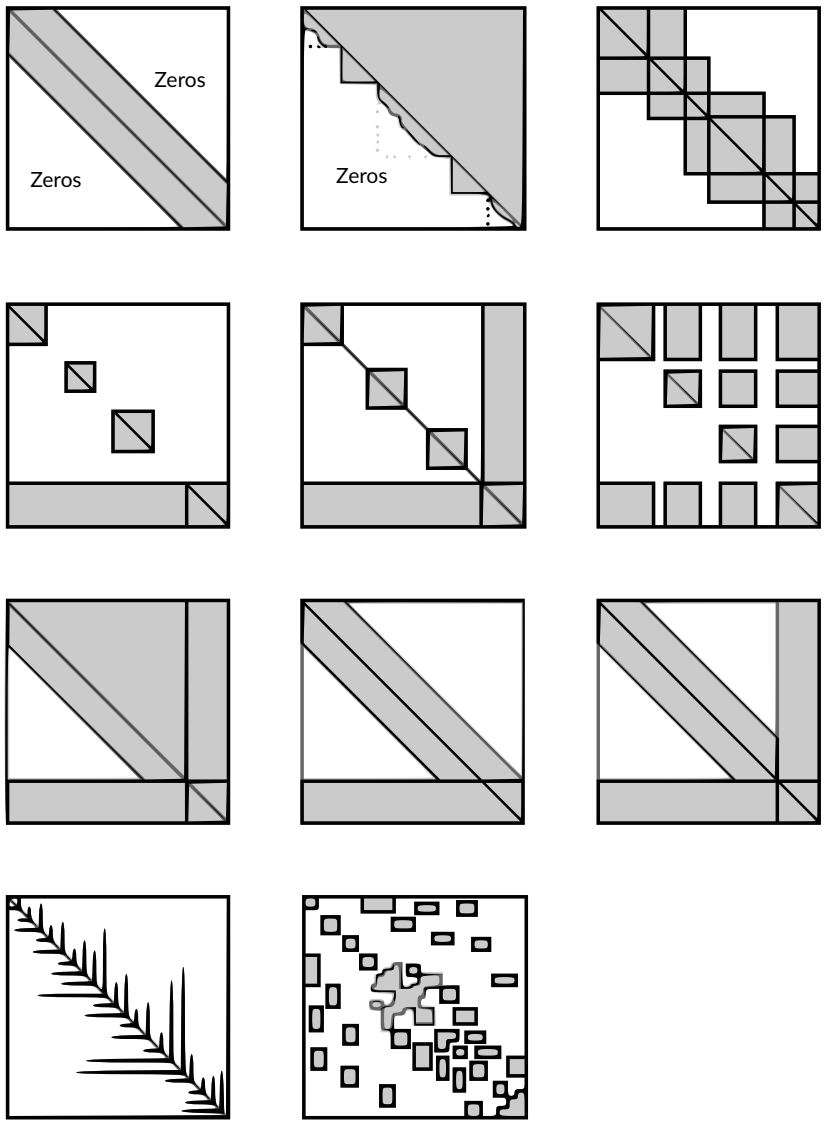
Sparse matrix storage format

- Example: Yale storage format, storing 3 vectors:
  - `A = [5 8 3 6]`
  - `IA = [0 0 2 3 4]`
  - `JA = [0 1 2 1]`

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

- `A` stores the non-zero values
- `IA` stores the index in A of the first non-zero in row i
- `JA` stores the column index
- Note: zero-based indices are used here!

Sparse matrix layout examples



Today's outline

Laplace's equation

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

T = Temperature

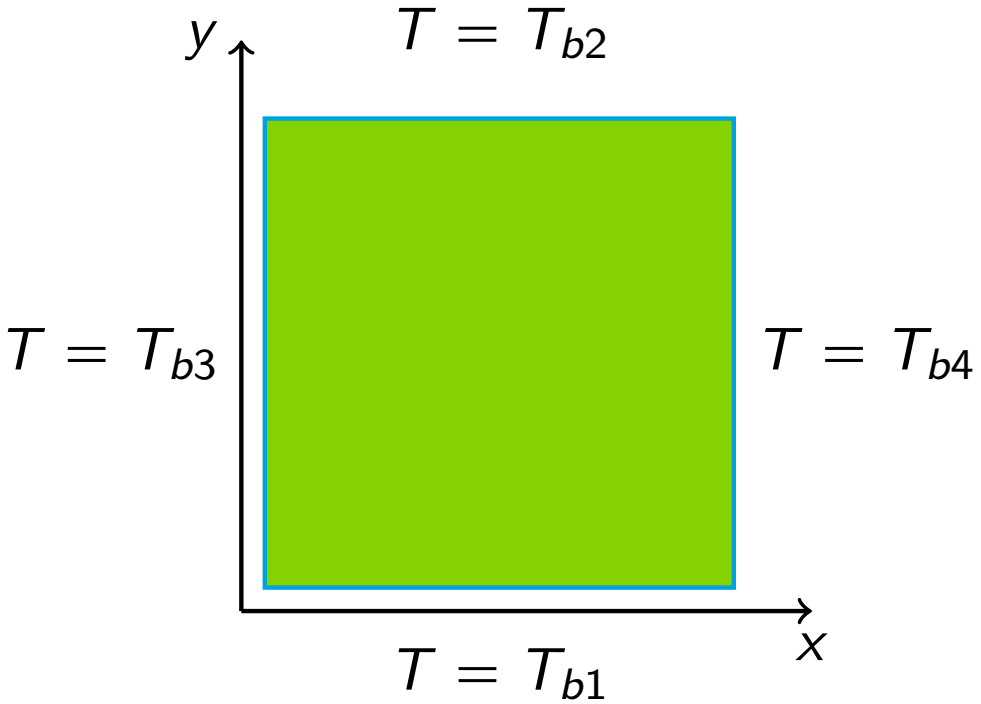
α = Thermal diffusivity

Laplace's equation

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

T = Temperature

α = Thermal diffusivity



Laplace's equation

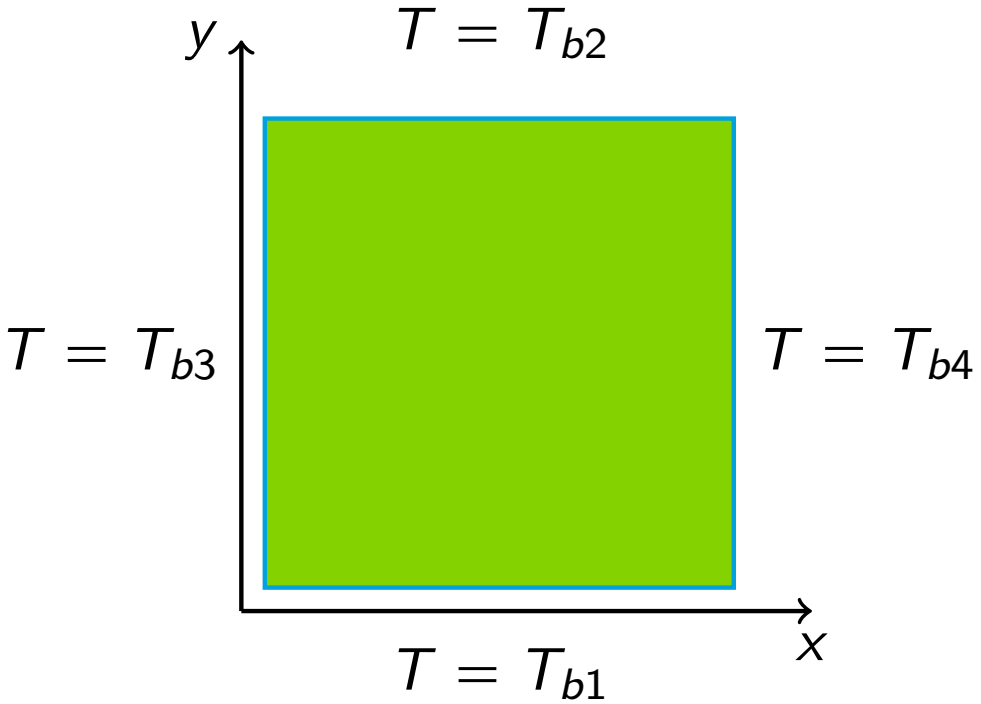
$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T$$

T = Temperature

α = Thermal diffusivity

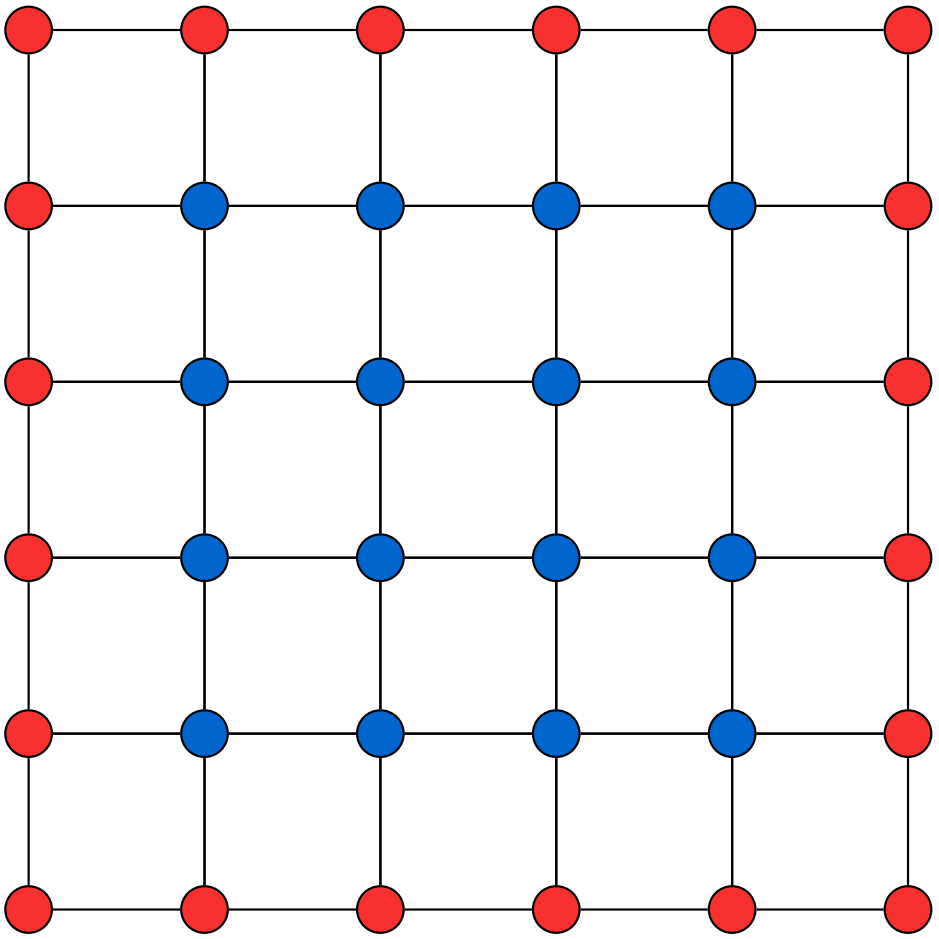
In steady state:

$$\nabla^2 T = 0$$



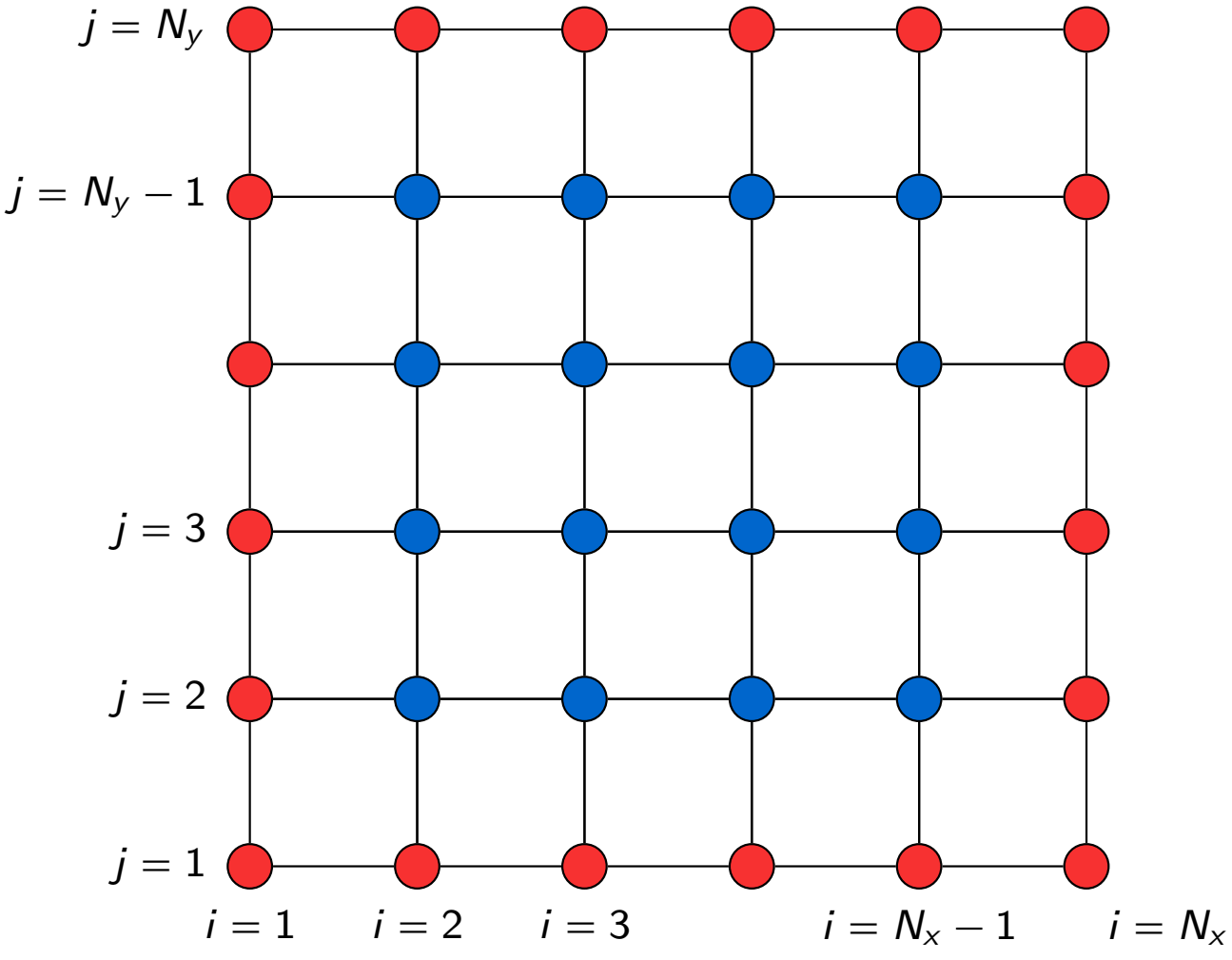
$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

Laplace's equation



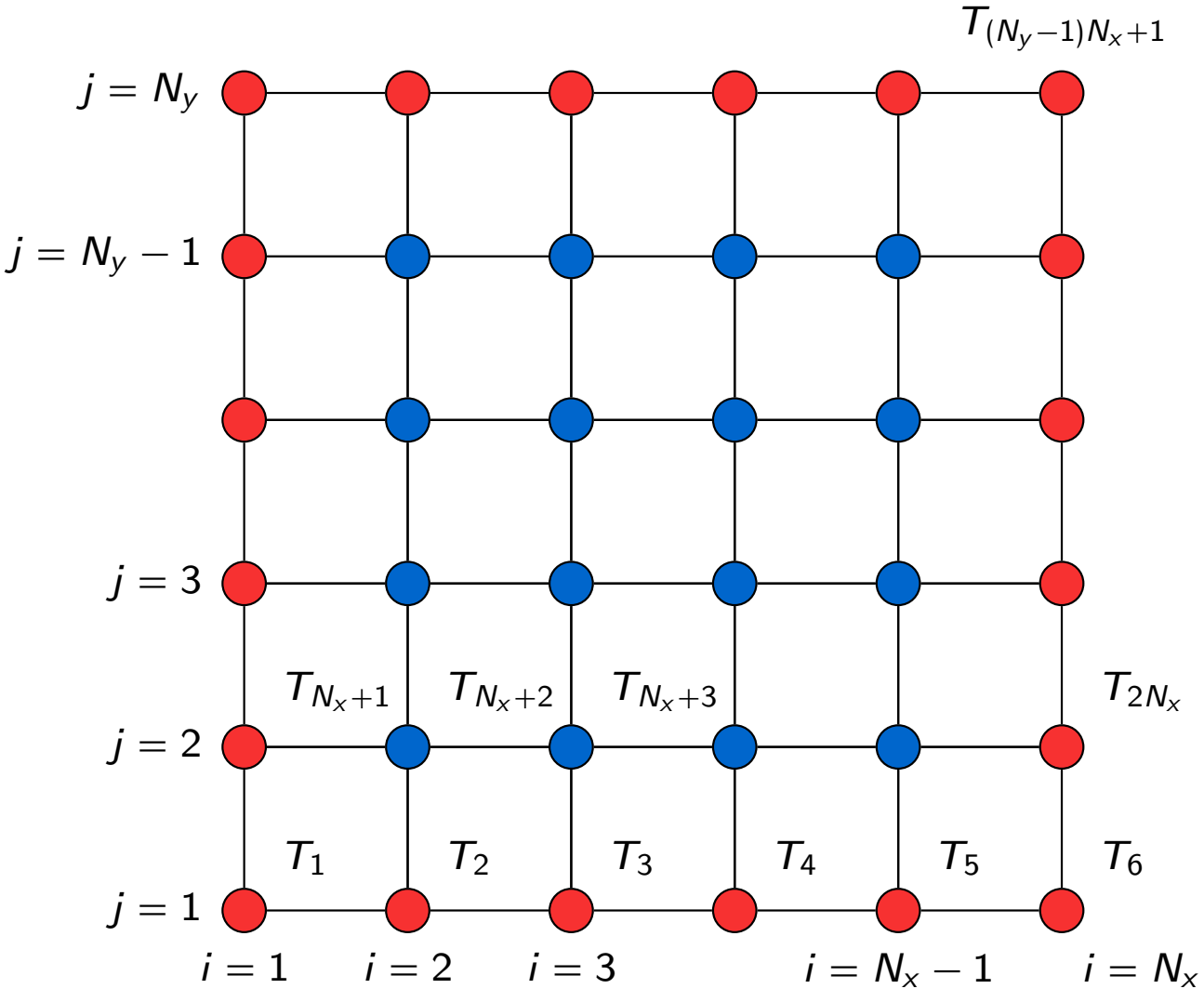
- Define a grid of points in x and y

Laplace's equation



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

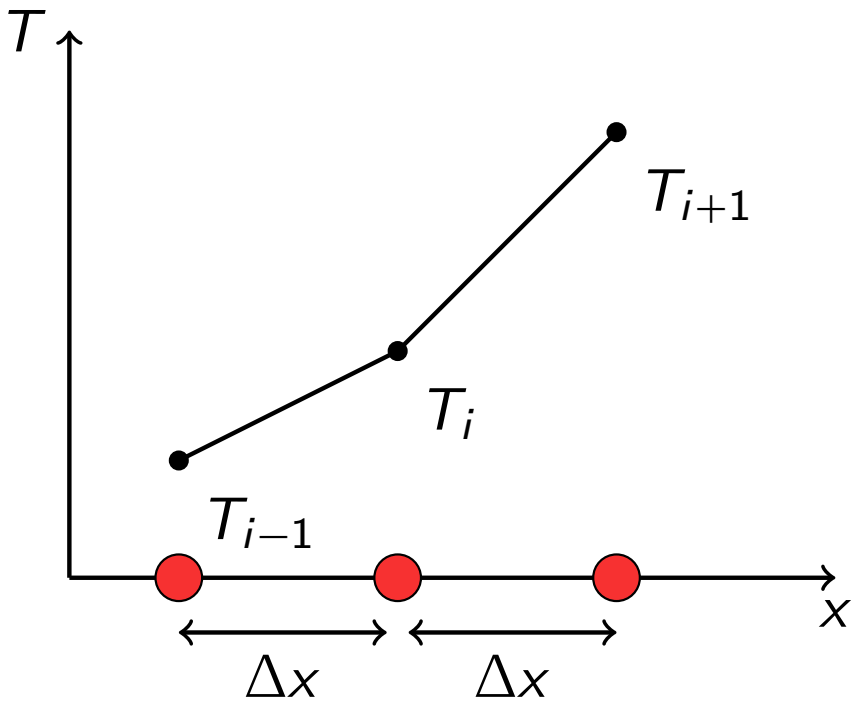
Laplace's equation



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

Laplace's equation

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



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

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

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

Laplace's equation

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

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

Laplace's equation

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

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

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

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

Laplace's equation

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

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

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

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

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. $T_1 = 1T_b$).

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

Creating the linear system

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

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

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

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

The function `spdiags` uses the following arguments:

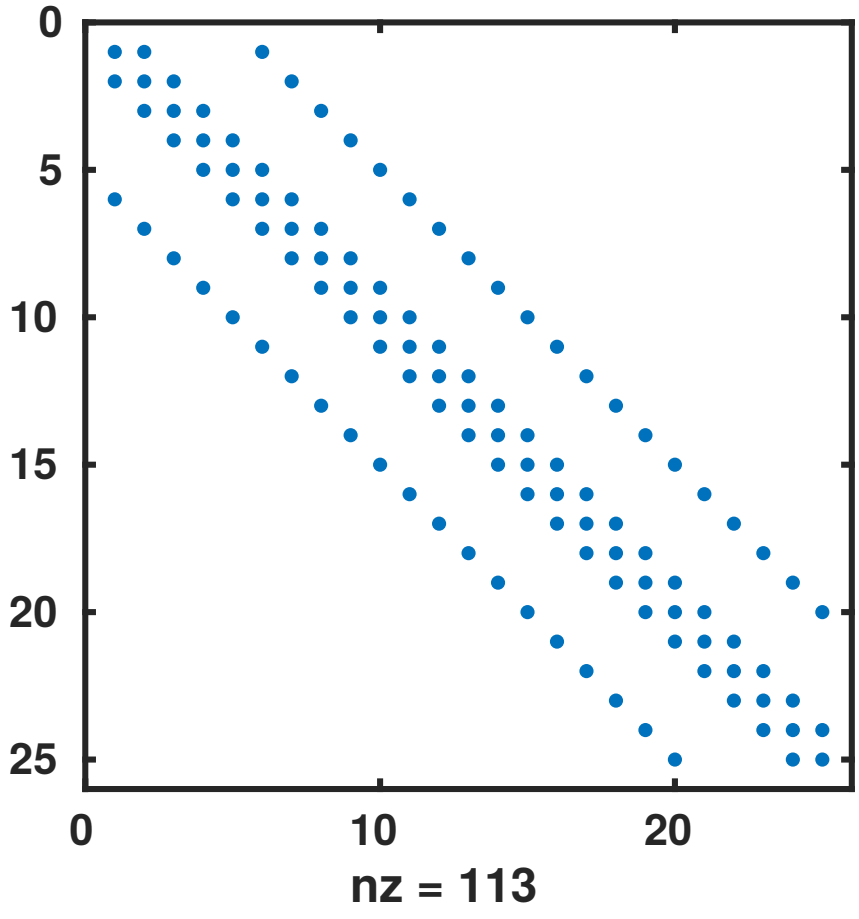
- The coefficients that have to be put on the diagonals arranged as columns in a matrix
- The position of the bands with respect to the main diagonal
- Size of the resulting matrix (in our case square $N_xN_y \times N_xN_y$)

Matrix sparsity

- Let's check the matrix layout:

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

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



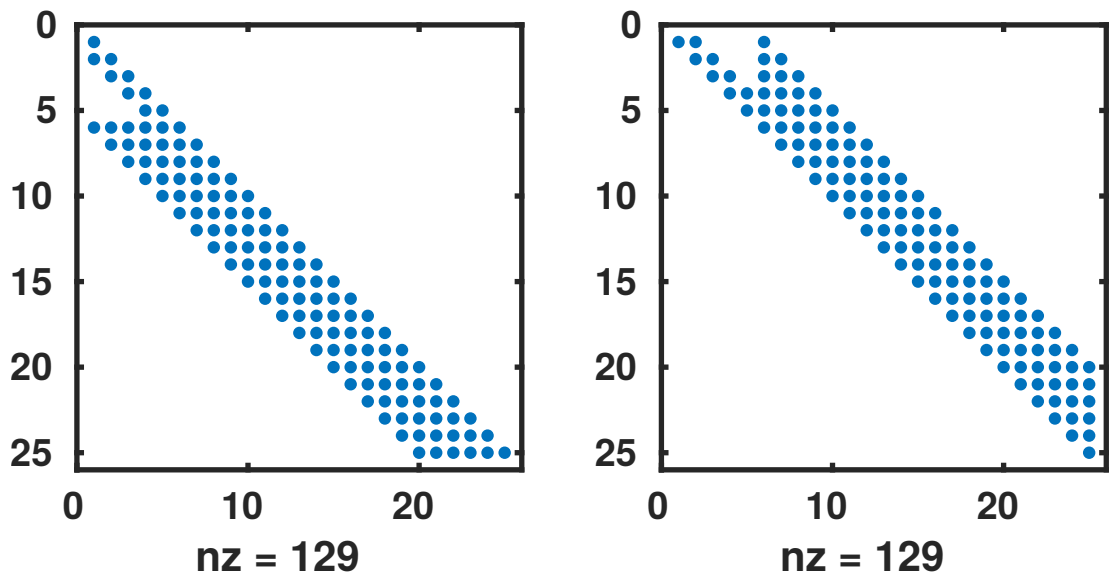
LU decomposition of a sparse matrix

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

LU decomposition of a sparse matrix

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

- 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!



About boundary conditions

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

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

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

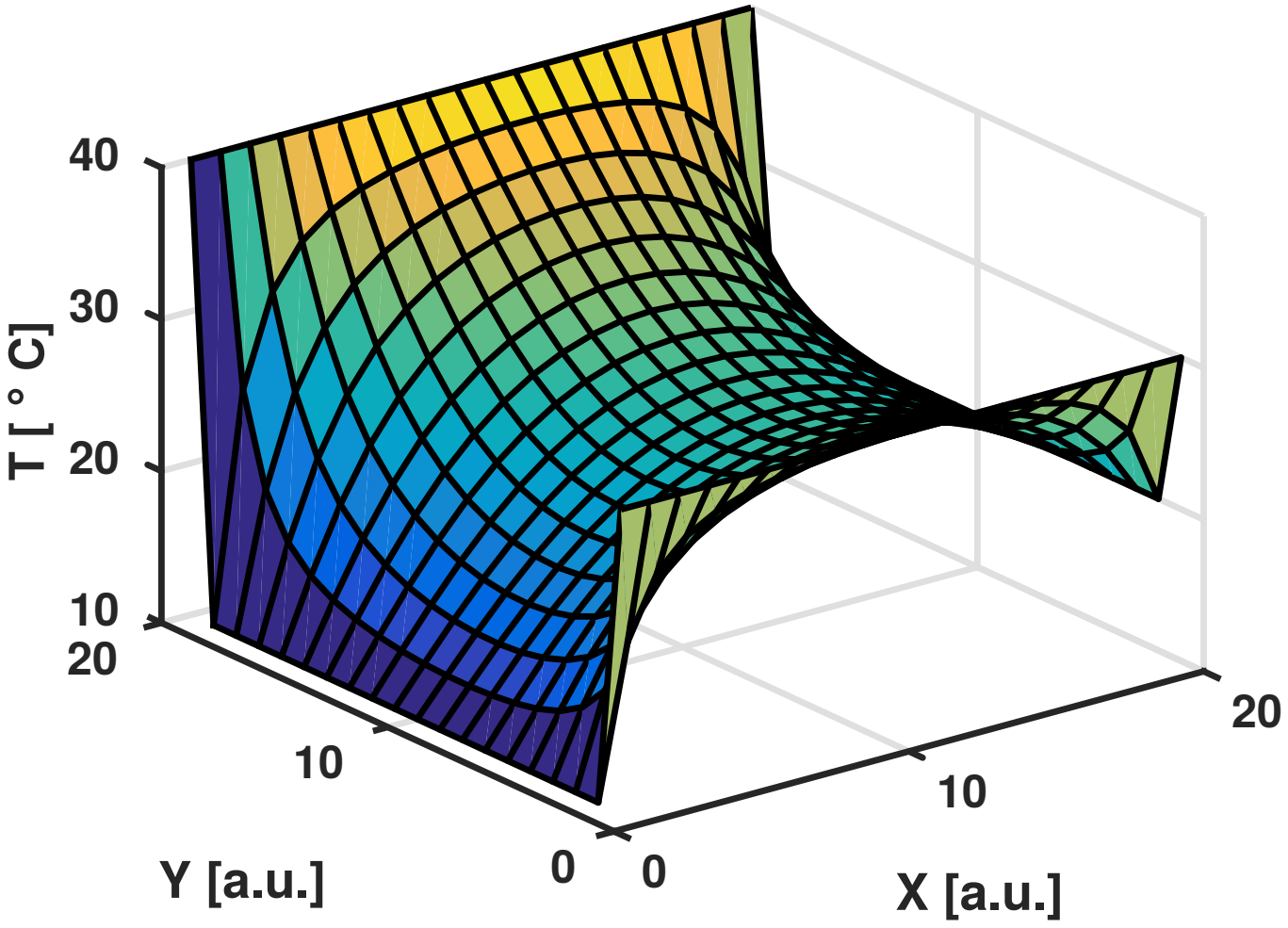
A full program, including solver

The program and auxiliary functions are on Canvas:

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

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

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

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

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

Sample results

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



LU decomposition

- LU decomposition and Gaussian elimination on a matrix like A requires more memory (with 3D problems, the offset in the diagonal would even be bigger!)
- In general extra memory allocation will not be a problem for MATLAB
- MATLAB is clever, in that sense that it attempts to reorder equations, to move elements closer to the diagonal)
- Alternatives for Gaussian elimination
- Use iterative methods when systems are large and sparse.
- Often such systems are encountered when we want to solve PDEs 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: straightforward implementation

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

Jacobi method: straightforward implementation

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

Jacobi method: straightforward implementation

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

Jacobi method: straightforward implementation

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

Jacobi method: straightforward implementation

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

Jacobi method: straightforward implementation

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

About the straightforward implementation

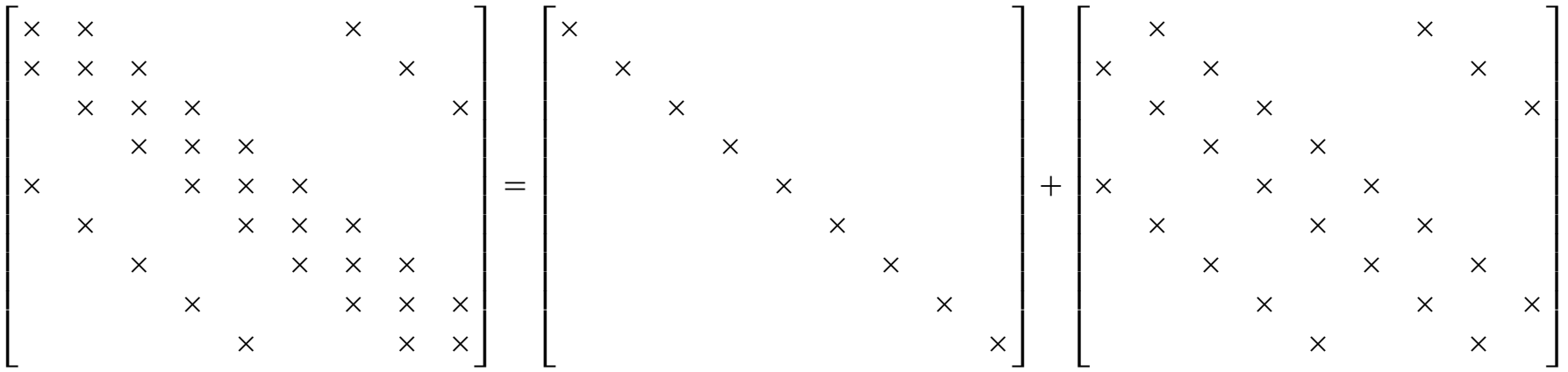
- The method as implemented works fine for a simple Laplace equation
- We did not take into account the thermal diffusivity (set to unity)
- We did not take into account the grid spacing (set to unity)
- For generic systems of linear equations, the implementation cannot be used.

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

- Now we can solve AT=b by:

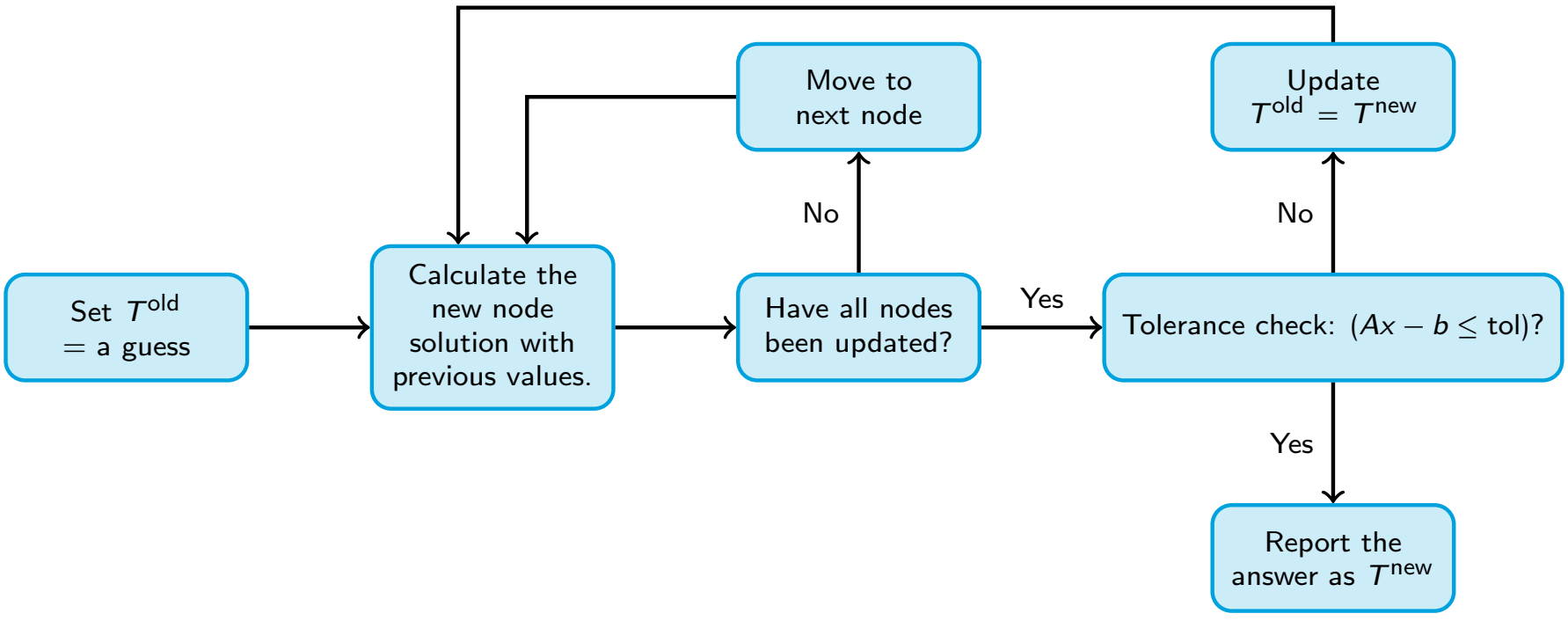
$$
\begin{aligned}
AT &= b \\
(D+R)T &= b \\
DT &= b - RT \\
DT^{\text{new}} &= b - RT^{\text{old}} \\
T^{\text{new}} &= D^{-1}(b - RT^{\text{old}})
\end{aligned}
$$

- 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



The core of the solver

```
1  while ( norm(A*x-b, 2) > err && it_jac < 1000 )
2    x_old = x;
3    for i=1:N
4      s = 0;
5      for j = 1:N
6        if (j ~= i)
7          % Sum off-diagonal*x_old
8          s = s+A(i,j)*x_old(j);
9        end
10     end
11     % Compute new x value
12     x(i) = (b(i)-s)/A(i,i);
13   end
14   % Increate number of iterations
15   it_jac = it_jac+1;
16 end
```

Some lines explained

1. The while loop holds two aspects
   - A convergence criterion (`norm(A*x-b, 2)>err`). Some considerations are:
     - L_1-norm (sum)
     - L_2-norm (Euclidian distance)
     - L_∞-norm (max)

Some lines explained

1. The while loop holds two aspects
   - A convergence criterion (`norm(A*x-b, 2)>err`). Some considerations are:
     - L_1-norm (sum)
     - L_2-norm (Euclidian distance)
     - L_∞-norm (max)
   - Protection against infinite loops (no convergence)

Some lines explained

1. The while loop holds two aspects
   - A convergence criterion (`norm(A*x-b, 2)>err`). Some considerations are:
     - L_1-norm (sum)
     - L_2-norm (Euclidian distance)
     - L_∞-norm (max)
   - Protection against infinite loops (no convergence)
2. Store the previous solution

Some lines explained

1. The while loop holds two aspects
   - A convergence criterion (`norm(A*x-b, 2)>err`). Some considerations are:
     - L_1-norm (sum)
     - L_2-norm (Euclidian distance)
     - L_∞-norm (max)
   - Protection against infinite loops (no convergence)
2. Store the previous solution
4. Reset the sum for each row, before summing for the new unknown node

Some lines explained

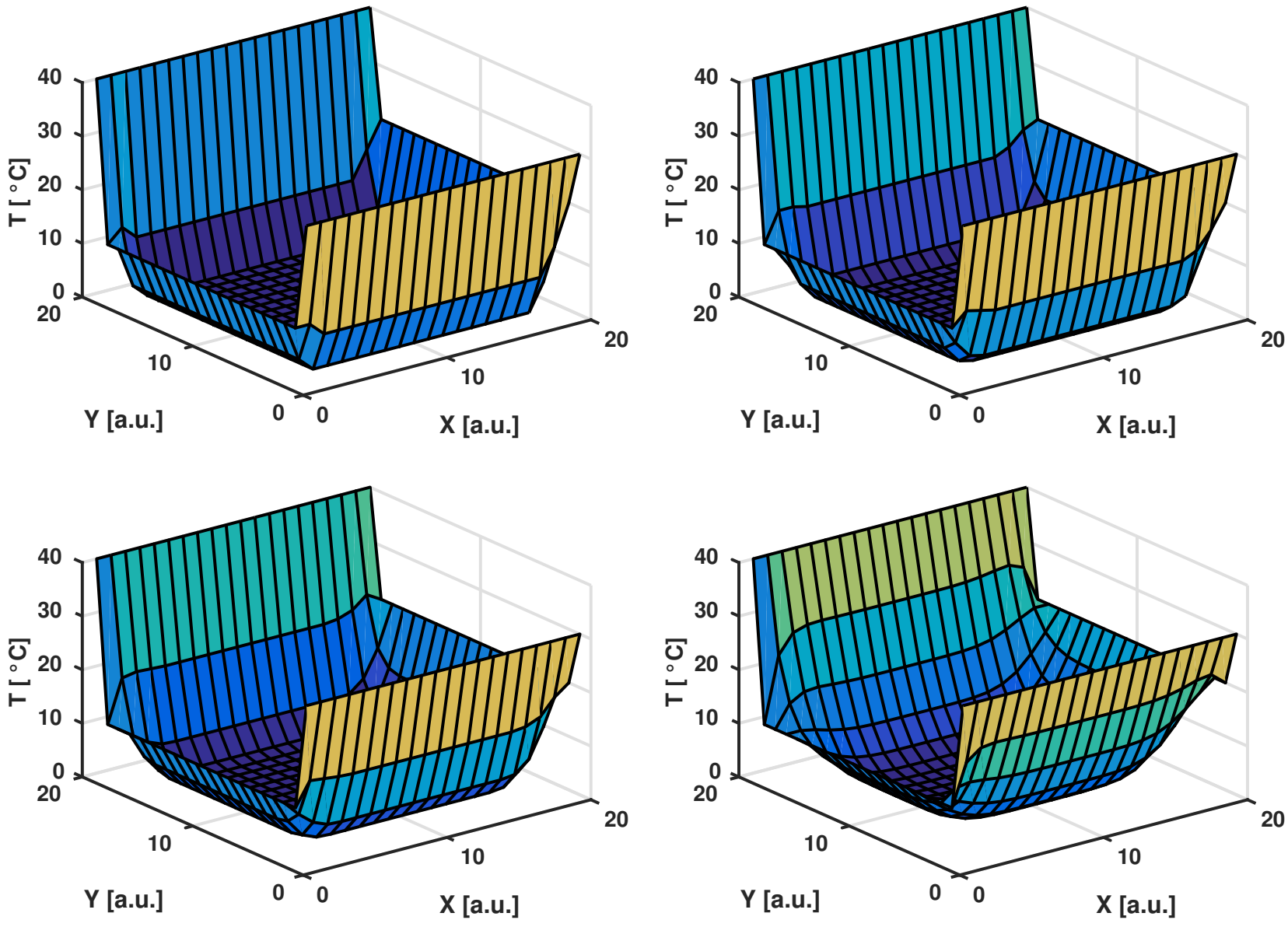
1. The while loop holds two aspects
   - A convergence criterion (`norm(A*x-b, 2)>err`). Some considerations are:
     - L_1-norm (sum)
     - L_2-norm (Euclidian distance)
     - L_∞-norm (max)
   - Protection against infinite loops (no convergence)
2. Store the previous solution
4. Reset the sum for 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

```
function [x0,it_jac] = solveJacobi(A,b,tol)
% Set default error
if nargin < 3
  tol = 1e-6;
end

x0 = 25*ones(size(b)); % Initial guess
x = zeros(size(b));    % Pre-allocate vector x
N = length(A);         % Number of equations
it_jac = 1;            % Init number of iterations

% --- Initial iteration to get x here ---

while ( norm(x-x0, 1) > tol > tol && it_jac < 1000 )
  x0 = x;
  for i = 1:N
    x(i) = (1/A(i,i))*((b(i) - A(i,[1:i-1,i+1:N]) * x0([1:i
        -1,i+1:N])));
  end
  it_jac = it_jac + 1;
end
it_jac
```

Iterations 1, 2, 3 and 10



Gauss-Seidel method

The Gauss-Seidel method is quite similar to Jacobi method

- The only difference is that the new estimate x^{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`
- The straightforward script works well for the Laplace equation, but we define the generic Gauss-Seidel algorithm as:

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:

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

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

Today's outline

Summary

- Partial differential equations can be written as sparse systems of linear equations
- Sparse systems can be handled with a direct method like Gaussian elimination
- 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 may be attractive.
- The Jacobi method was introduced. Many other methods are based on the Jacobi method (SOR method, for example)