Linear equation solvers
Iterative methods

Ivo Roghair, Martin van Sint Annaland

Chemical Process Intensification,
Eindhoven University of Technology

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?
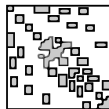- Sparse matrix formats: Yale, CRS, CCS
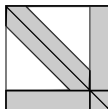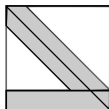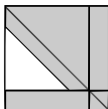
Sparse matrix storage format

- 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
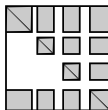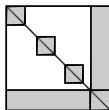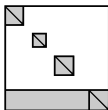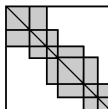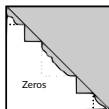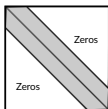
Today's outline

1 Introduction

2 Sparse matrices

3 Laplace's equation

4 Creating a sparse system

5 Iterative methods

6 Summary

Laplace's equation

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

$$T = \text{Temperature}$$

$$\alpha = \text{Thermal diffusivity}$$

In steady state:

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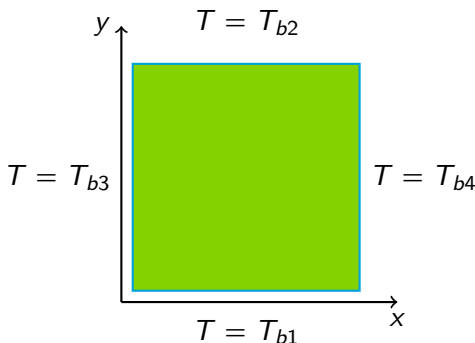
$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

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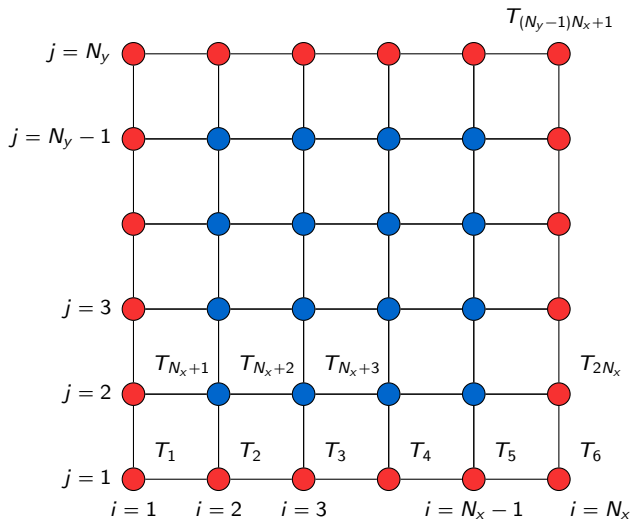
- Define a grid of points in x and y
- Index of the grid points using 2D coordinates i and j
- Set up the equations using a 1D index system:
$$T_{i,j} = T_{i+N_x(j-1)}$$

Discretization of Laplace's equation (II)

Estimate the second-order differentials: assume a piece-wise linear profile in the temperature:
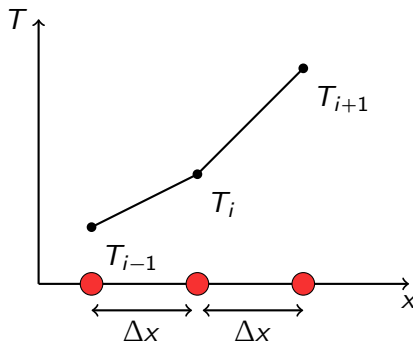
$$\frac{\partial^2 T}{\partial x^2} \approx \frac{\left.\frac{\partial T}{\partial x}\right|_{i+\frac{1}{2}} - \left.\frac{\partial T}{\partial x}\right|_{i-\frac{1}{2}}}{\Delta x}$$

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

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

Discretization of Laplace's equation (III)

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

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

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

1 Introduction

2 Sparse matrices

3 Laplace's equation

4 Creating a sparse system

5 Iterative methods

6 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. $T_1 = 1T_b$).

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

Creating the linear system

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

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

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

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

The function `spdiags` uses the following arguments:

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

Matrix sparsity

- Let's check the matrix layout:

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

- This command shows the non-zero values of a matrix

- Apart from the main diagonal, there are offset bands!

nz = 113
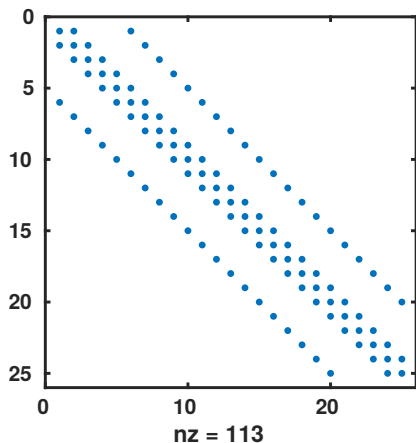
About boundary conditions

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

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

- However, we have set all nodes to be a function of their neighbors...

- Find the boundary node indices using $k = i + Nx(j - 1)$
  - i = 1, j = 1:Ny
  - i = Nx, j = 1:Ny
  - j = 1, i = 1:Nx
  - j = Ny, i = 1:Nx

- Reset the row in A to zeros, set $A_{kk} = 1$

- Set value in rhs: $b_k = T_{k,\text{boundary}}$

- Boundary conditions are often more elaborate to implement! See setBoundaryConditions.m.

Partial implementation of the boundary conditions

See `setBoundaryConditions.m`.

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

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

%% Repeat for y-direction
```

How applying boundary conditions affects the linear system

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

- Make sure that matrix `A` and right hand side vector `b` are in your workspace, as well as `Nx` and `N_y`

- Create a vector that holds the temperature at each boundary:

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

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

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

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

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

A full program, including solver

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

```matlab
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
```
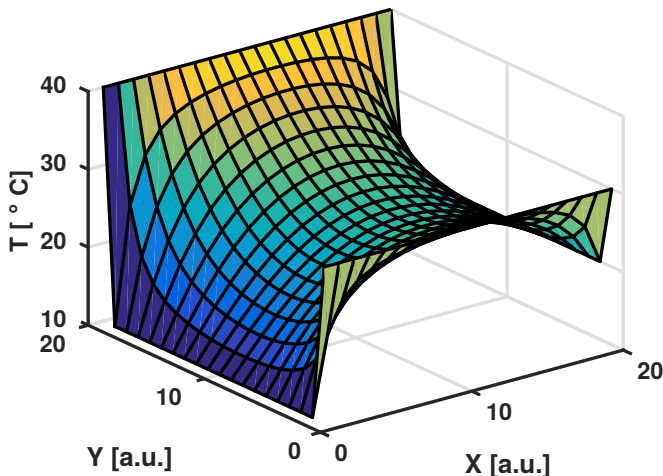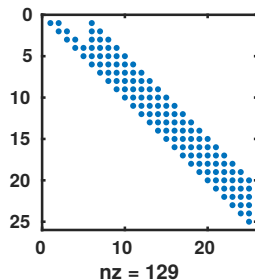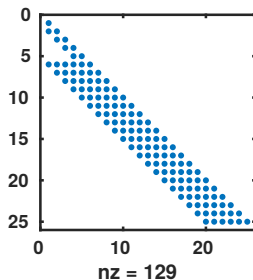
Sample results

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

LU decomposition of a sparse matrix

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

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


nz = 129

nz = 129

LU decomposition

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

Alternatives for elimination methods

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

Today's outline

❶ Introduction

❷ Sparse matrices

❸ Laplace's equation

❹ Creating a sparse system

❺ Iterative methods

❻ Summary

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:

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

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

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

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

Jacobi method for Laplace's equation

See `laplace_jacobi.m` (from Canvas)

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

About the straightforward implementation

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

> 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:

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

Jacobi method: solving a system

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

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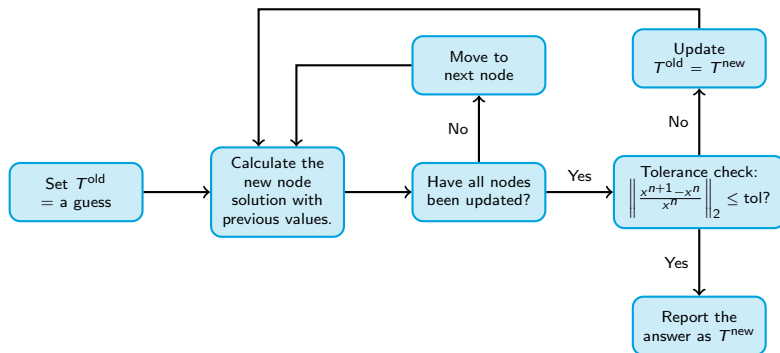$$Dx^{\text{new}} = b - Rx^{\text{old}}$$
$$x^{\text{new}} = D^{-1}(b - Rx^{\text{old}})$$

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

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

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

Diagram of the Jacobi method

The core of the solver

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

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

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

A few details on this algorithm

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

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

The solver using array indices

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

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

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

Iterations 1, 2, 3 and 10
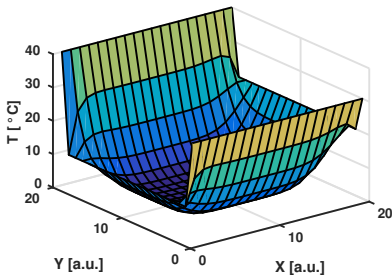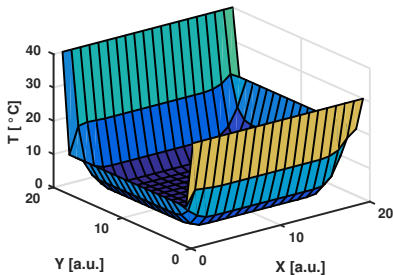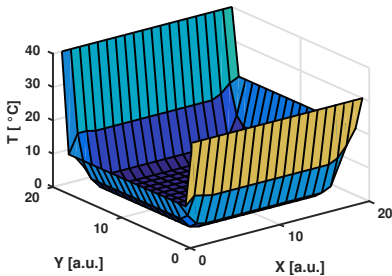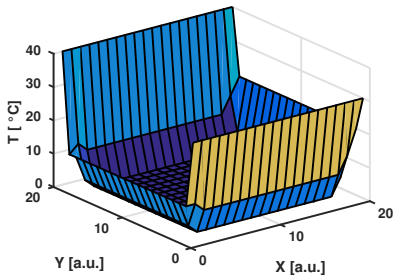
Gauss-Seidel method

The Gauss-Seidel method is quite similar to Jacobi method

- The only difference is that the new estimate x^{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 `laplace_gaussseidel.m` for the algorithm.
- The straightforward script works well for the current Laplace equation, but we define the generic Gauss-Seidel algorithm on the following slides.

Gauss-Seidel method

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

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

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

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

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

Today's outline

1 Introduction

2 Sparse matrices

3 Laplace's equation

4 Creating a sparse system

5 Iterative methods

6 Summary

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)