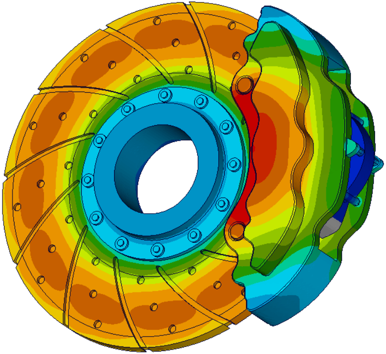
Specialized Numerical Methods for Transport Phenomena

The finite element method: Poisson problem in 2D and 3D

Bruno Blais and Laura Prieto Saavedra

Associate Professor
Department of Chemical Engineering
Polytechnique Montréal

October 22, 2025



Outline

Recapitulation

FEM: Weak form in 2D and 3D

Sparse linear algebra

Let's talk code

Outline

Recapitulation

FEM: Weak form in 2D and 3D

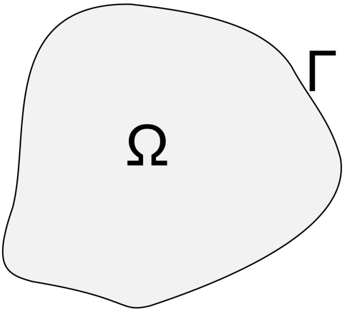
Sparse linear algebra

Let's talk code

The heat equation, a prototype PDE

We are interested in solving equations such as the heat equation on a Ω domain whose contour is Γ:

$$\nabla^2 T = 0$$



1D version of the problem

We began by solving the 1D heat equation, with Dirichlet boundary conditions:

$$\frac{\mathrm{d}^2T}{\mathrm{d}x^2} = 0$$

We multiplied by an a priori unknown test function $u(x)$ and obtained:

$$\frac{\mathrm{d}^2T}{\mathrm{d}x^2}u(x) = 0$$

This equation is integrated over the entire domain of interest and obtain the form **strong integral**:

$$\int_\Omega \frac{\mathrm{d}^2T}{\mathrm{d}x^2}u(x)\mathrm{d}x = 0$$

Continued

Integrating by parts, and imposing zero Dirichlet boundary conditions to the test functions, we obtained:

$$\int_\Omega \frac{\mathrm{d}T}{\mathrm{d}x}\frac{\mathrm{d}u(x)}{\mathrm{d}x}\mathrm{d}x = 0$$

Using interpolation to express temperature:

$$\int_\Omega \sum_{j=0}^{n} T_j \frac{\mathrm{d}\varphi_j}{\mathrm{d}x}\frac{\mathrm{d}u(x)}{\mathrm{d}x}\mathrm{d}x = 0$$

Finally, we chose a Galerkin approach for $u(x)$.

$$\sum_{j=0}^{n} T_j \int_\Omega \frac{\mathrm{d}\varphi_j}{\mathrm{d}x}\frac{\mathrm{d}\varphi_i}{\mathrm{d}x} = 0$$

Last course

In the last course, we have seen how the Finite Element Method works and we have used it to solve a 1D problem. Steps of the resolution:

- Define the triangulation and the elements (Ω_h)
- Define the interpolation function (φ_i) et and their gradient ($\frac{\mathrm{d}\varphi_i}{\mathrm{d}x}$)
- Define the structure of the matrix
- Calculate the integral to calculate the matrix (ex. $\int_{\Omega_1} \frac{\mathrm{d}\varphi_0}{\mathrm{d}x}\frac{\mathrm{d}\varphi_1}{\mathrm{d}x}$)
- Solve the linear system of equations to find the coefficients T_j

The temperature is now known everywhere because of the interpolation support!

What's left?

Generalizing FEM to higher spatial dimension is doable:

- Interpolation over segments becomes interpolation over cells (2D or 3D)
- Integrals over segments become integrals over cells (2D or 3D)
- Our 1D weak form has to be a 2D or 3D weak form

Luckily for us, `deal.II` will abstract all these concepts. Programming 1D, 2D or 3D will be identical. Understanding it, however, will be more subtle.

Outline

Recapitulation

FEM: Weak form in 2D and 3D

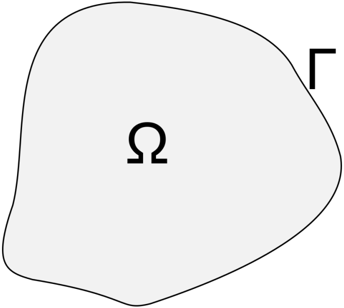
Sparse linear algebra

Let's talk code

The heat equation, a prototype PDE

Let's go back to our heat equation on a Ω domain whose contour is Γ:

$$\nabla^2 T = 0$$



Heat equation in 2D and 3D

$$\nabla^2 T = 0$$

In 2D is:

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

and in 3D:

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

To be general, we will keep using tensor notation. This will make our representation agnostic of dimension.

Multiple dimensions

Let's start from our equation in tensor form:

$$\nabla^2 T = 0$$

which is equivalent to:

$$\nabla \cdot (\nabla T) = 0$$

or, in Einstein notation:

$$\partial_i \partial_i T = 0$$

We will note $\boldsymbol{x}$ the position vector. Remember that ∇T is a vector in $\mathbb{R}^d$ where d is the number of dimension in space.

Strong form

$$\nabla \cdot (\nabla T) = 0$$

We multiply by a test function $u(\boldsymbol{x})$

$$u\nabla \cdot (\nabla T) = 0$$

We integrate over Ω

$$\int\limits_{\Omega} u\nabla \cdot (\nabla T)\,\mathrm{d}\Omega = 0$$

We have to be careful, integrating over Ω has a different meaning now since Ω is either a line, a surface or a volume.

Green’s first identity

What is Green’s first identity?

$$\iiint\limits_{\Omega} u\nabla\cdot\left(\nabla T\right)\mathrm{d}\Omega = \iint\limits_{\Gamma} u\left(\nabla T\right)\cdot\boldsymbol{n}\mathrm{d}\Gamma - \iiint\limits_{\Omega}\nabla u\cdot\nabla T\mathrm{d}\Omega$$

with $\boldsymbol{n}$ the outward pointing unit normal vector.
Where does this come from? Let’s try to develop an understanding of it.

Applying Green's identity

$$\iiint\limits_{\Omega} u\nabla\cdot\left(\nabla T\right)\mathrm{d}\Omega = \iint\limits_{\Gamma} u\left(\nabla T\right)\cdot\boldsymbol{n}\mathrm{d}\Gamma - \iiint\limits_{\Omega} \nabla u\cdot\nabla T\mathrm{d}\Omega$$

Thus the problem we have to solve is:

$$\iiint\limits_{\Omega} \nabla u\cdot\nabla T\mathrm{d}\Omega - \iint\limits_{\Gamma} u\left(\nabla T\right)\cdot\boldsymbol{n}\mathrm{d}\Gamma = 0$$

Now what do we do with this?

Boundary term

$$-\iint\limits_{\Gamma} u\left(\nabla T\right)\cdot \boldsymbol{n}\mathrm{d}\Gamma$$

This is our boundary term. It becomes zero for Dirichlet Boundary conditions. If we have Neumann or Robin boundary conditions, this term is replaced by the value of the flux.
For example, if $-\nabla T \cdot \mathbf{n} = q \forall (x, y) \in \Gamma$, then :

$$-\iint\limits_{\Gamma} u\left(\nabla T\right)\cdot \boldsymbol{n}\mathrm{d}\Gamma = \iint\limits_{\Gamma} uq\mathrm{d}\Gamma$$

Volumetric term

For now let's assume we have Dirichlet Boundary conditions. The PDE we are solving becomes:

$$\iiint\limits_{\Omega} \nabla u \cdot \nabla T \mathrm{d}\Omega = 0$$

We will use the same approach. First, we replace T with its interpolation.

$$\iiint\limits_{\Omega} \nabla u \cdot \sum_{j} T_j \left(\nabla \phi_j\right) \mathrm{d}\Omega = 0$$

Test function

$$\iiint\limits_{\Omega} \nabla u \cdot \sum_{j} T_j \left(\nabla \phi_j\right) \mathrm{d}\Omega = 0$$

We decide to use a Galerkin method, so we choose u to be ϕ_i. We will have as many equations as we have unknowns.

$$\iiint\limits_{\Omega} \nabla \phi_i \cdot \sum_{j} T_j \left(\nabla \phi_j\right) \mathrm{d}\Omega = 0$$

We can rearrange this!

$$\sum_{j} T_j \iiint\limits_{\Omega} \nabla \phi_i \cdot \nabla \phi_j \mathrm{d}\Omega = 0$$

Integration and interpolation

The same rules we have seen apply for integrating and interpolating over surface and volumes.

Integrals

Using tensor product, we can generalize our 1D integral into a 2D and 3D integral respectively by carrying out a tensor product on each dimension.

Interpolation

Using tensor product, we can generalize our 1D Lagrange polynomials into a 2D and 3D polynomials respectively by carrying out a tensor product on each dimension.

Some comments are necessary

Gradients are vectors

In 2D:

$$\nabla \phi_i = \begin{bmatrix} \frac{\partial \phi_i}{\partial x} \\ \frac{\partial \phi_i}{\partial y} \end{bmatrix}$$

In 3D:

$$\nabla \phi_i = \begin{bmatrix} \frac{\partial \phi_i}{\partial x} \\ \frac{\partial \phi_i}{\partial y} \\ \frac{\partial \phi_i}{\partial z} \end{bmatrix}$$

`deal.II` will make our life much easier for this...

Integrals are now surface or volume integrals

We are still integrating over a triangulation, but now all cells are 2D or 3D objects. Thus we need to use higher dimensionality quadratures.

Gradients: how does it work exactly?

In our equations, we will need the gradients with respect to the physical space, e.g. in 2D:

$$\nabla \phi_i = \begin{bmatrix} \frac{\partial \phi_i}{\partial x} \\ \frac{\partial \phi_i}{\partial y} \end{bmatrix}$$

But our shape function are only defined in the reference space. How do we go from one to the other?

More comments!

$$\sum_j T_j \iiint\limits_{\Omega} \nabla\phi_i \cdot \nabla\phi_j \mathrm{d}\Omega = 0 \tag{1}$$

This can be decomposed furthermore:

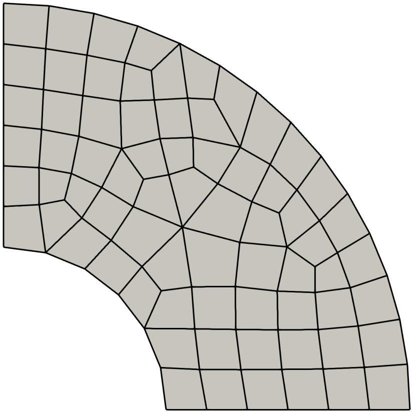
$$\sum_j T_j \sum_e \iiint\limits_{\Omega_e} \nabla\phi_i \cdot \nabla\phi_j \mathrm{d}\Omega = 0 \tag{2}$$

in which Ω_e are the elements (cells).

Which ϕ_i and ϕ_j are non-zero on Ω_e?

There is an explicit answer to this question.

What happens for a mesh?



Let us talk about the degrees of freedom and how they change the matrix...

More comments!

Which ϕ_i and ϕ_j are non-zero?

Only those for which their colocation point (support point) lie within or at the edge of the cell.

- We know *a priori* which ϕ_i and ϕ_j interact with one another.
- Few of them actually interact...
- For an an unstructured mesh, nothing allows us to infer the numbering. We need a structure to store this information. This is called a connectivity table.
- The bigger the mesh, the more zeros we have... This is an issue we will address in the following section.

Outline

Recapitulation

FEM: Weak form in 2D and 3D

Sparse linear algebra

Let's talk code

Memory requirement

It will not be computationally tractable to store all of these zeros.

- ≈ 100 Q1 elements in 2D : $(10^2 \times 10^2)$ matrix, 10^4 doubles, 0.08MB
- ≈ 1000 Q1 elements in 2D : $(10^3 \times 10^3)$ matrix, 10^6 doubles, 8MB
- $\approx 10^4$ Q1 elements in 2D : $(10^4 \times 10^4)$ matrix, 10^8 doubles, 800MB
- $\approx 10^6$ Q1 elements in 2D : $(10^6 \times 10^6)$ matrix, 10^{12} doubles, 8000GB

Memory requirement

It will not be computationally tractable to store all of these zeros.

- ≈ 100 Q1 elements in 2D : $(10^2 \times 10^2)$ matrix, 10^4 doubles, 0.08MB
- ≈ 1000 Q1 elements in 2D : $(10^3 \times 10^3)$ matrix, 10^6 doubles, 8MB
- $\approx 10^4$ Q1 elements in 2D : $(10^4 \times 10^4)$ matrix, 10^8 doubles, 800MB
- $\approx 10^6$ Q1 elements in 2D : $(10^6 \times 10^6)$ matrix, 10^{12} doubles, 8000GB

How can we solve large problems then?

Avoid storing all the zeros! Use sparse matrices.

Sparse matrices

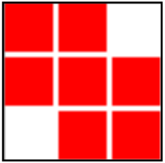
We need to use matrices that only store the non-zero elements. This requires two things:

- Knowing which elements of the matrix will be non-zero (a priori) to allocate the necessary memory.
- A storage technique which is adequate for this.

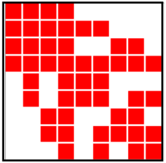
What do we store?

- We know *a priori* which row-columns will be non-zero.
- We can pre-emptively allocate just the right amount of memory required for our sparse matrix. This *pattern* is called a *sparsity pattern*.
- Establishing it is more of a technical issue than a scientific one. Luckily for us, `deal.II` takes care of that for us.

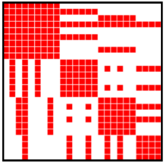
1D - Q1 - 2 cells



2D -Q1 - 4 cells



2D -Q2 - 4 cells



Storage techniques

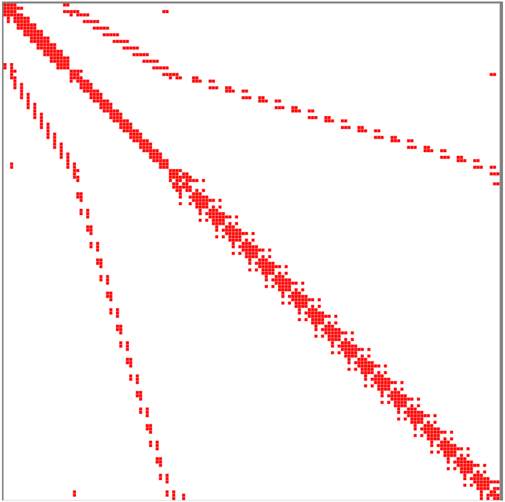
There are multiple formats to store sparse matrices. Some are more adequate to generate sparsity patterns, while other are better for solving linear systems.

- Dictionary Of Keys (DOK). Similar to a python dictionary
- List of list
- Coordinate list
- Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC)

Let us see an example of CSR storage...

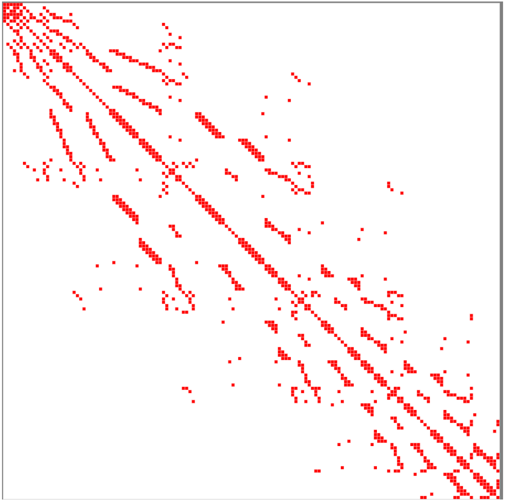
Why do you think that this storage technique is suitable for us?

Naive sparsity pattern



Optimized sparsity pattern

Sparsity patterns can be renumbered to have a decreased bandwidth.



Code: Sparsity pattern

```
// 1. Create matrix
SparseMatrix<double> system_matrix;
^^I
// 2. Create a dynamic pattern (cheap to alter but not efficient
    computationally)
DynamicSparsityPattern dsp(dof_handler.n_dofs());

// Generate the sparsity pattern
DoFTools::make_sparsity_pattern(dof_handler, dsp);

// 3. Create a static sparsity pattern
SparsityPattern sparsity_pattern;
// Copy the dynamic one in the static one
sparsity_pattern.copy_from(dsp);

// 4. Use it to make our sparse matrix!
system_matrix.reinit(sparsity_pattern);
```

Solving the linear system

The matrix and the vector we have built will allow us to obtain the solution. However, we still need to solve a linear system of equations. There are multiple ways to achieve this.

Direct solver

Solves the system *exactly*. Consumes a lot of memory and generally does not scale well with the number of unknown or in parallel.

Iterative solvers

Solves the system iteratively. They are very sensitive to the type of preconditioning, however, we need them if we want to solve very large systems.

Iterative solvers

Iterative solvers require two components:

The solver itself

Many types of solver that may exploit the structure of the matrix (e.g., Conjugate Gradient (CG)) or that may be general (e.g., GMRES).

Preconditioner

Alters the structure of the matrix to allow the iterative solver to reach a solution faster. Again, there are many types of preconditioners (ILU, AMG, GMG, etc.)

Outline

Recapitulation

FEM: Weak form in 2D and 3D

Sparse linear algebra

Let's talk code

Looping over cells

```
QGauss<dim> quadrature_formula(fe.degree + 1);

FEValues<dim> fe_values(fe,^^Iquadrature_formula,
update_values | update_gradients |
update_quadrature_points | update_JxW_values);

const unsigned int dofs_per_cell = fe.n_dofs_per_cell();
FullMatrix<double> cell_matrix(dofs_per_cell, dofs_per_cell);
Vector<double>  cell_rhs(dofs_per_cell);

std::vector<types::global_dof_index> local_dof_indices(dofs_per_cell);
for (const auto &cell : dof_handler.active_cell_iterators())
{
^^Ife_values.reinit(cell);
^^Icell_matrix = 0;
^^Icell_rhs  = 0;
^^I// The integration part
}
```

Inside the loop

```
^^Ifor (const unsigned int q_index :
    fe_values.quadrature_point_indices())
^^I{
^^I^^Ifor (const unsigned int i : fe_values.dof_indices())
^^I^^I{
^^I^^I^^Ifor (const unsigned int j : fe_values.dof_indices())
^^I^^I^^I{
^^I^^I^^I^^Icell_matrix(i, j) +=
^^I^^I^^I^^I(fe_values.shape_grad(i, q_index) * // grad phi_i(x_q)
^^I^^I^^I^^Ife_values.shape_grad(j, q_index) * // grad phi_j(x_q)
^^I^^I^^I^^Ife_values.JxW(q_index));    // dx
^^I^^I^^I}
^^I^^I^^I
^^I^^I^^Icell_rhs(i) += (fe_values.shape_value(i, q_index) * //
    phi_i(x_q)
^^I^^I^^Iright_hand_side.value(x_q) * // f(x_q)
^^I^^I^^Ife_values.JxW(q_index));     // dx
^^I^^I}
^^I}
^^I
```

Assembling inside the cell loop

```
^^Icell->get_dof_indices(local_dof_indices);
^^Ifor (const unsigned int i : fe_values.dof_indices())
^^I{
^^I^^Ifor (const unsigned int j : fe_values.dof_indices())
^^I^^Isystem_matrix.add(local_dof_indices[i],
^^I^^Ilocal_dof_indices[j],
^^I^^Icell_matrix(i, j));
^^I^^I
^^I^^Isystem_rhs(local_dof_indices[i]) += cell_rhs(i);
^^I}
^^I
```