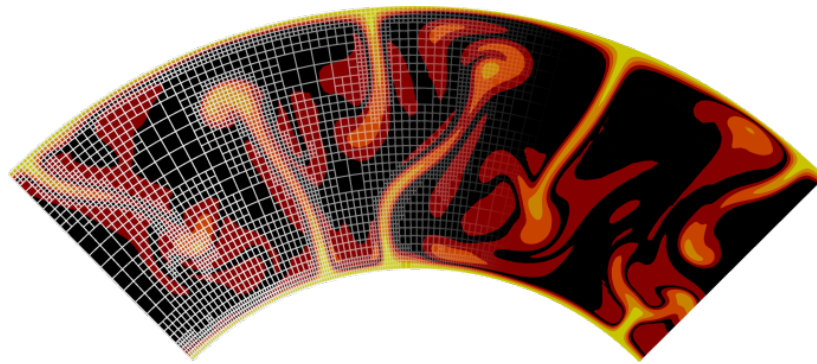
Specialized Numerical Methods for Transport Phenomena

The deal.II library

Introduction to triangulations



Bruno Blais and Laura Prieto Saavedra

Associate Professor
Department of Chemical Engineering
Polytechnique Montréal

October 22, 2025

Outline

`deal.II` library

Triangulations: Why and what?

Talk is cheap, where's the code? Generating a Triangulation

Talk is cheap, where's the code? Iterating over cells

Talk is cheap, where's the code? Outputting a Triangulation

Conclusion

Outline

deal.II

What it is

A C++ software library supporting the creation of finite element codes and an open community of users and developers.

Mission

To provide well-documented tools to build finite element codes for a broad variety of PDEs, from laptops to supercomputers.

Vision

To create an open, inclusive, participatory community providing users and developers with a state-of-the-art, comprehensive software library that constitutes the go-to solution for all finite element problems.

deal.II: Concretely?

What it is

Provides everything required to solve transport phenomena using FEM:

- Mesh management (Triangulation)
- Tools to assemble FEM equations
- HPC-ready sparse linear algebra
- Mesh adaptation
- Post-processing support (vtu outputs)

What it is not

deal.II is not a solver. It is a toolbox to build your own highly efficient FEM solver for the physics of your choice.

Challenges or opportunity?

This is challenging because it requires you to know sufficient FEM to program your solver. This is filled with opportunities because it gives you absolute control over it.

Some numbers

Community

- ≈ 16 principal developers
- $>$1000 contributors
- Developed by research groups in the USA, Germany, Italy, Sweden and Canada

Some numbers

- 1.5M lines of C++
- $>$1000 page of documentation
- 90 tutorial programs (our group has made step-68 and step-70)
- 12000 unit tests
- Demonstrated scalability up to 300 000 cores

A PhD project

Created in the late 1990s

`deal.II` was created by Wolfgang Bangerth and Guido Kanschat in the late 90s. It was Wolfgang's PhD thesis. It has been in active development since then (5-10 Pull requests per day, 365 days a year). Won the prestigious Jon A Wilkinson prize in 2007 for best scientific library.

Industrial usages

`deal.II` is used in over 10 large-scale open-source projects to solve problems related to geodynamics (Aspect), fluid mechanics in chemical processes (Lethe), biomechanics (LifeX), etc. This is just the tip of the iceberg (Canadian research lab use it for wood drying simulations, etc.).

Open science

`deal.II` is openscience. Use this to your advantage during the semester.

- ALL classes are documented here `https://www.dealii.org/current/doxygen/deal.II/index.html`
- There are over 80 examples. The first 6 are helpful for this class.
- The community is **mega chill** and **inclusive**.

Outline

A domain and its boundaries

The 2D heat transfer equation:

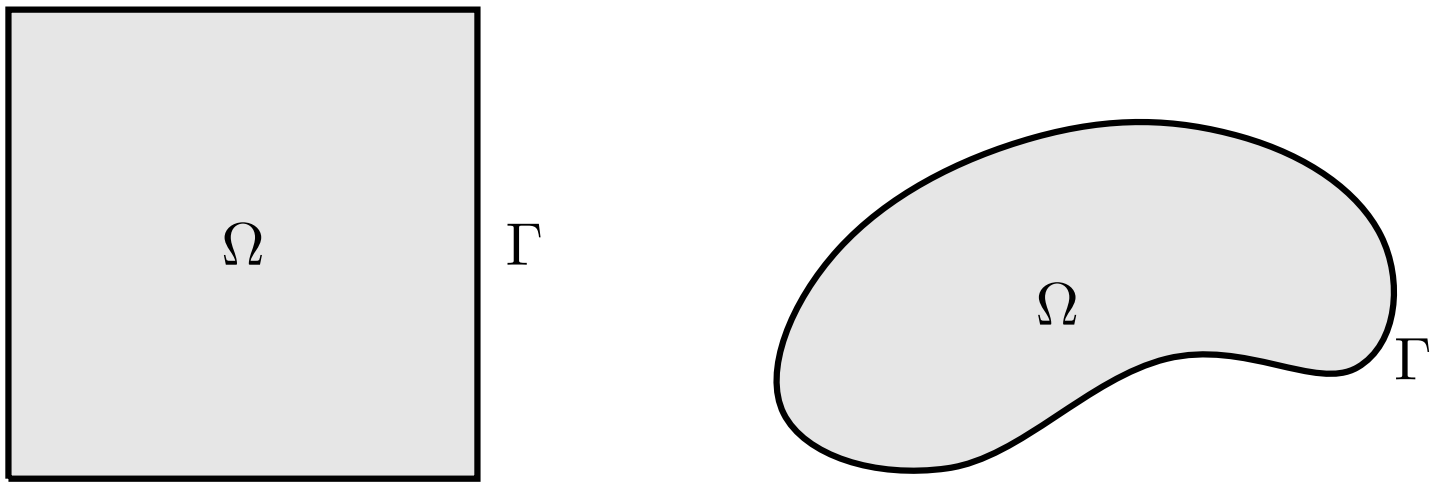
$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{1}$$

describes temperature in space $T(x, y)$. To solve this equation we need:

- To define a domain
- To define its boundaries

The domain

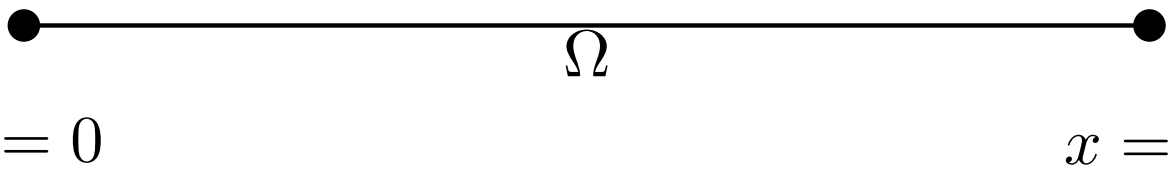
The domain is the space in which we wish to describe temperature (or fluid velocity, etc.). It may be simple or highly complex. It defines the limits of the independent coordinates of the PDE (e.g. x, y). In this class, we note the domain Ω, with $\Omega \in \mathbb{R}^d$, where d is the number of spatial dimension (1, 2, 3). The contour of this domain is noted Γ with $\Gamma \in \mathbb{R}^{d-1}$.



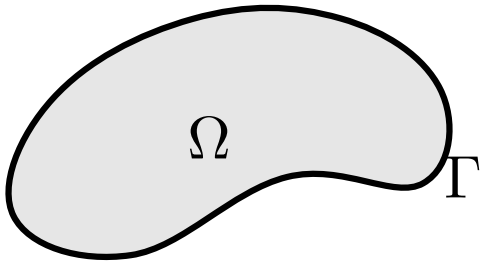
Simple geometry and a more complex one in 2D

Dimensions and domain

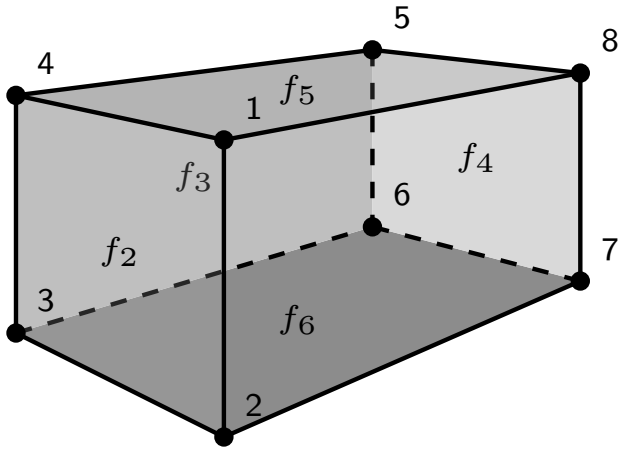
- In 1D, $\Omega \in \mathbb{R}^1$ et $\Gamma \in \mathbb{R}^0$. Here $\Gamma = \{0, L\}$.



- In 2D, $\Omega \in \mathbb{R}^2$ and $\Gamma \in \mathbb{R}^1$



- In 3D, $\Omega \in \mathbb{R}^3$ et $\Gamma \in \mathbb{R}^2$. Here $\Gamma = \cup_{i=1}^{6} f_i$



Triangulation of a domain

In the majority of numerical methods, to solve a PDE in a domain you need to subdivide the domain into smaller entities which are called cells. This set of cells will be what we call a **triangulation** or **mesh**.
In FEM, the cells will be the elements on which we will assemble simpler equations. Intuitively, the more cells you will have, the more accurate a solution will be.

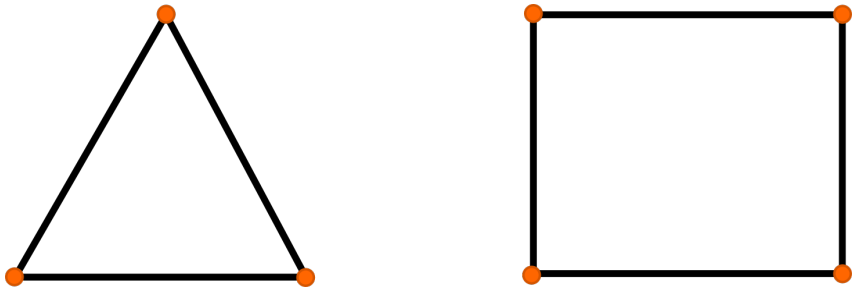
Triangulation of a domain

In the majority of numerical methods, to solve a PDE in a domain you need to subdivide the domain into smaller entities which are called cells. This set of cells will be what we call a **triangulation** or **mesh**.
In FEM, the cells will be the elements on which we will assemble simpler equations. Intuitively, the more cells you will have, the more accurate a solution will be.
What are the different cells we may encounter?

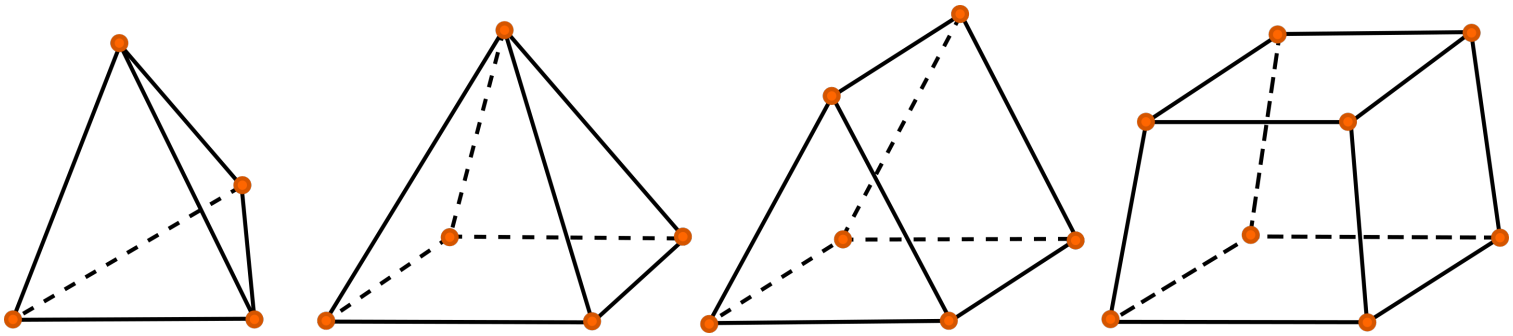
Cells in 1D to 3D

In 1D, a domain Ω will be a line. It is discretized using smaller segments:

In 2D, we can choose between triangles and quadrilaterals:

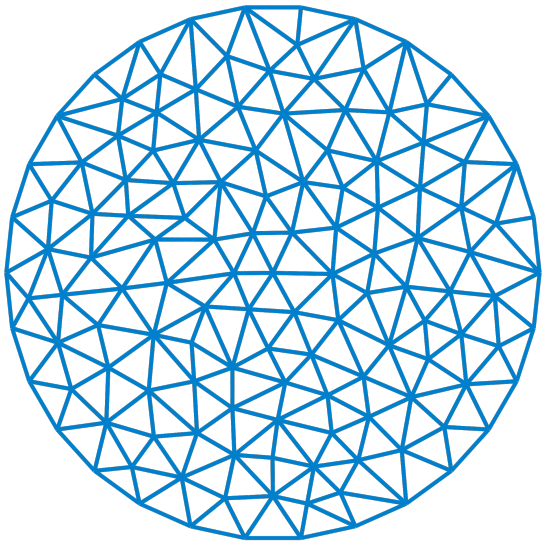


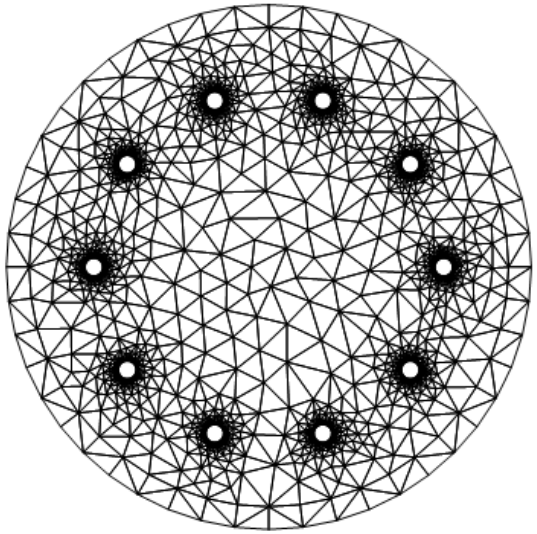
In 3D, the spectrum is larger. Ranges from tetrahedron to hexahedron:

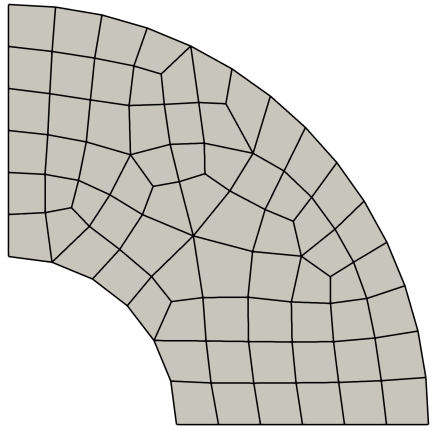


Triangulations in 2D

In 2D we can generate Triangulations for geometries of arbitrary complexity using triangulation algorithms (e.g. Delaunay, Frontal). This generates an assembly of Triangles which covers the domain. We can also use quadrilaterals, which are extremely useful in certain fields (boundary layers in fluid mechanics, solid mechanics).

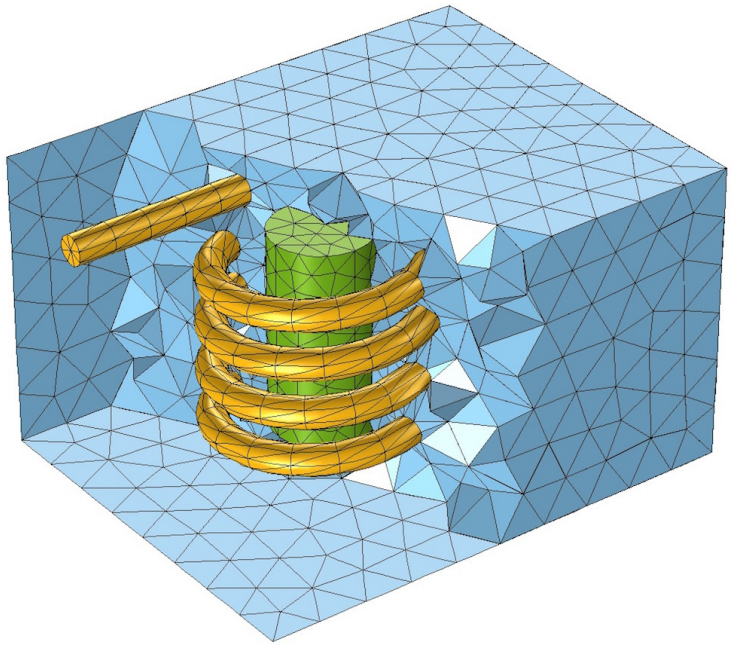


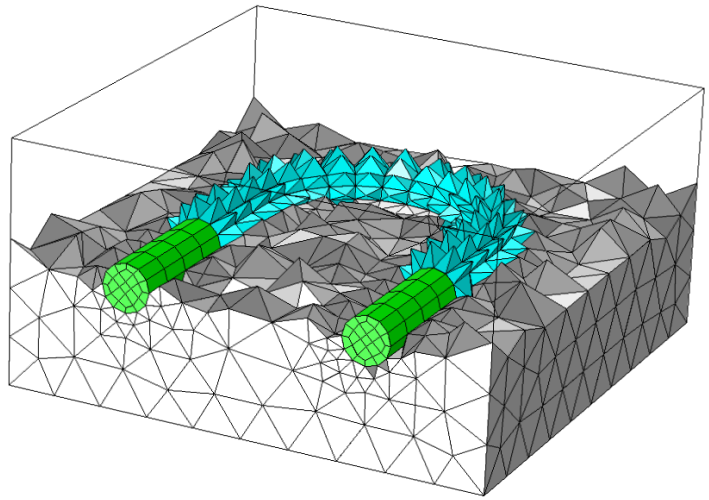




Triangulations in 3D - Tetrahedron

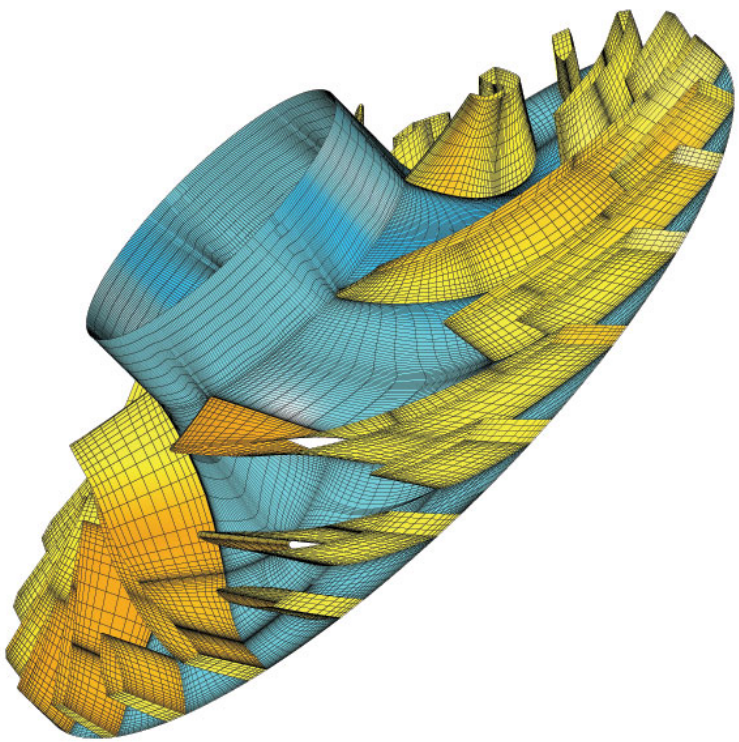
We often use tetrahedron to mesh an arbitrary volume. Tetrahedron are the cell type for which it is always possible to **automatically** triangulate a geometry.

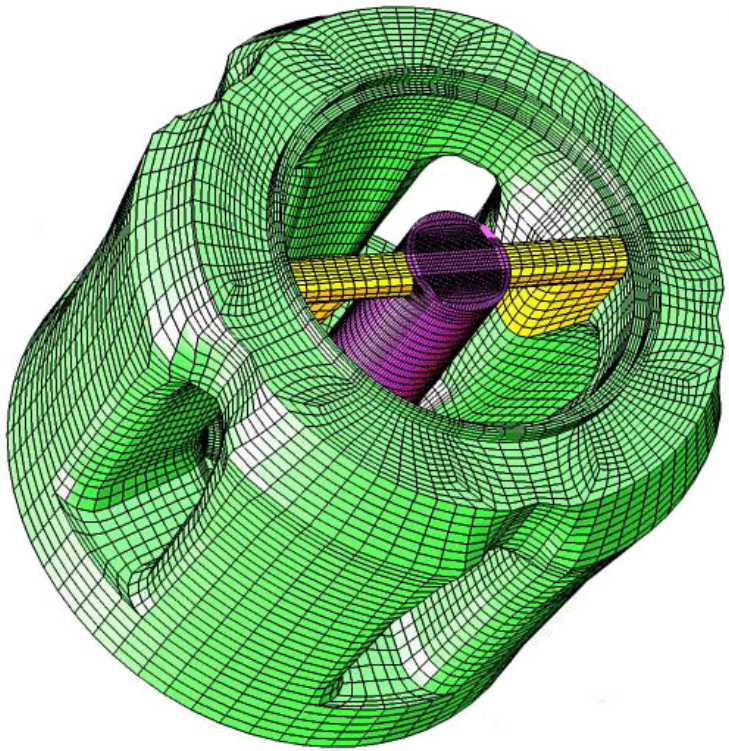




Triangulations in 3D - Hexahedron

In FEM, hexahedral meshes always lead to better results. However, there are no ways right now to automatically triangulate a 3D geometry using hexahedra. This is an active research field.





In this class

Dimensions in this class

In this course, we will solve problems in 1D, 2D and 3D. We believe it is important to understand some of the nuances associated with dimensionality. In `deal.II` this will be easy because the dimension is templated.

The art of triangulating

In this course, we will not program the generation of triangulation. We will use either meshes created using GMSH, or we will use the `deal.II` `GridGenerator` which already contains a ton of useful grids.

GridGenerator

Let's take a look at the GridGenerator documentation together:
`https://www.dealii.org/developer/doxygen/deal.II/namespaceGridGenerator.html`

Cells types used in this class

Supported

The `deal.II` framework was made around quadrilateral and hexahedral cells (so-called tensor cells), but it now supports all aforementioned cell types.

In this class

We will focus on line, quadrilateral and hexahedral unstructured meshes. The main reason for this is that it is really easy to explain and understand interpolation and quadrature in these type of cells. They are also extremely accurate element types.

Outline

The triangulation class

A triangulation is a collection of cells of dimension `dim`. If the cells are 2D, then they are made of lines, which are made of vertices, and so on and so forth. To manage this information, `deal.II` provides the `Triangulation` class.

```
^^ITriangulation<dim> tria;
^^ITriangulation<1> tria_1d;
^^ITriangulation<2> tria_2d;
^^ITriangulation<3> tria_3d;
```

Constructing a triangulation

By default, a simulation code will not know the geometry it will simulate before starting. Consequently, this class is created empty and must be filled with a triangulation. There are multiple ways to achieve this.

- Generating a triangulation *in situ* using `GridGenerator`
- Loading a mesh generated by another software using `GridIn`
- Making a copy of another existing Triangulation

Let's look at the documentation together...

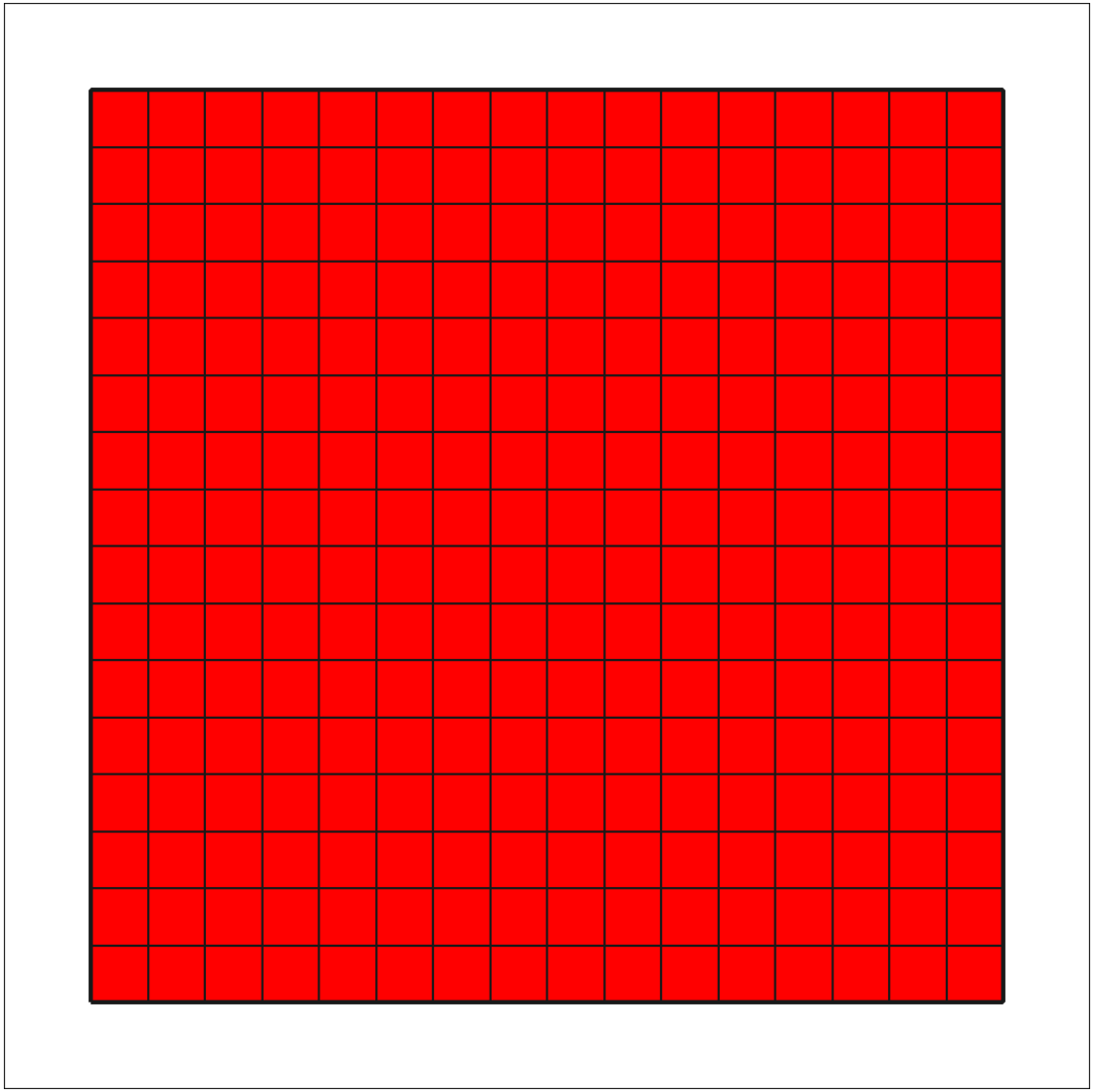
Generating with GridGenerator

Using the GridGenerator class, a triangulation can be filled with a simple mesh of a geometry.

```
Triangulation<2> triangulation;
GridGenerator::hyper_cube(triangulation,0,1);
triangulation.refine_global(4);
```

This creates a square mesh in 2D. The bottom left corner is (0,0) and the top right corner is (1,1). We then refine the mesh 4 times. Each refinement subdivides the cell by two in each direction. We thus go from $1 \rightarrow 4 \rightarrow 16 \rightarrow 64 \rightarrow 256$ cells when we do four refinements.

What does it look like?



Loading with GridIn

The GridIn functions enable you to read meshes from various format. Over 10 different formats are currently supported. Here is an example for a GMSH file in 3D

```
Triangulation<3> triangulation; // the triangulation object
GridIn<3> gridin; // the GridIn object which will read
// We connect the reader to the triangulation
gridin.attach_triangulation(triangulation);
// We create a file stream to open the file
std::ifstream f("example.msh");
gridin.read_msh(f); // We read the file
```

Outline

Iterating over cells

Now that we have a triangulation loaded, we will often want to `iterate` over the content of the triangulations (the cells). Generally, you would expect that we would traverse the structure using a simple for loop using a structure like this:

```
^^I^^Ifor(int i = 0; i < n_cells; ++i)
^^I^^I{
^^I^^I^^ICell cell = tria.get_cell(i);
^^I^^I}
```

This is not what we do in practice.

Iterating over cells

There is no reason for cells to be stored sequentially. This is even more important when you have mesh adaptation. What we do it use C++ range-based iterators.

```
// auto is a nice keyword that asks the compiler to
// automatically detect the type of variable at declaration
 for (auto &cell : triangulation.active_cell_iterators())
{
^^I// cell is now a reference to a cell. We can now call the
    functions associated with
^^I// cells using cell->function()
}
```

For more information on C++ iterators, please consult the following page: https://learn.microsoft.com/en-us/cpp/standard-library/iterators?view=msvc-170

Refinement

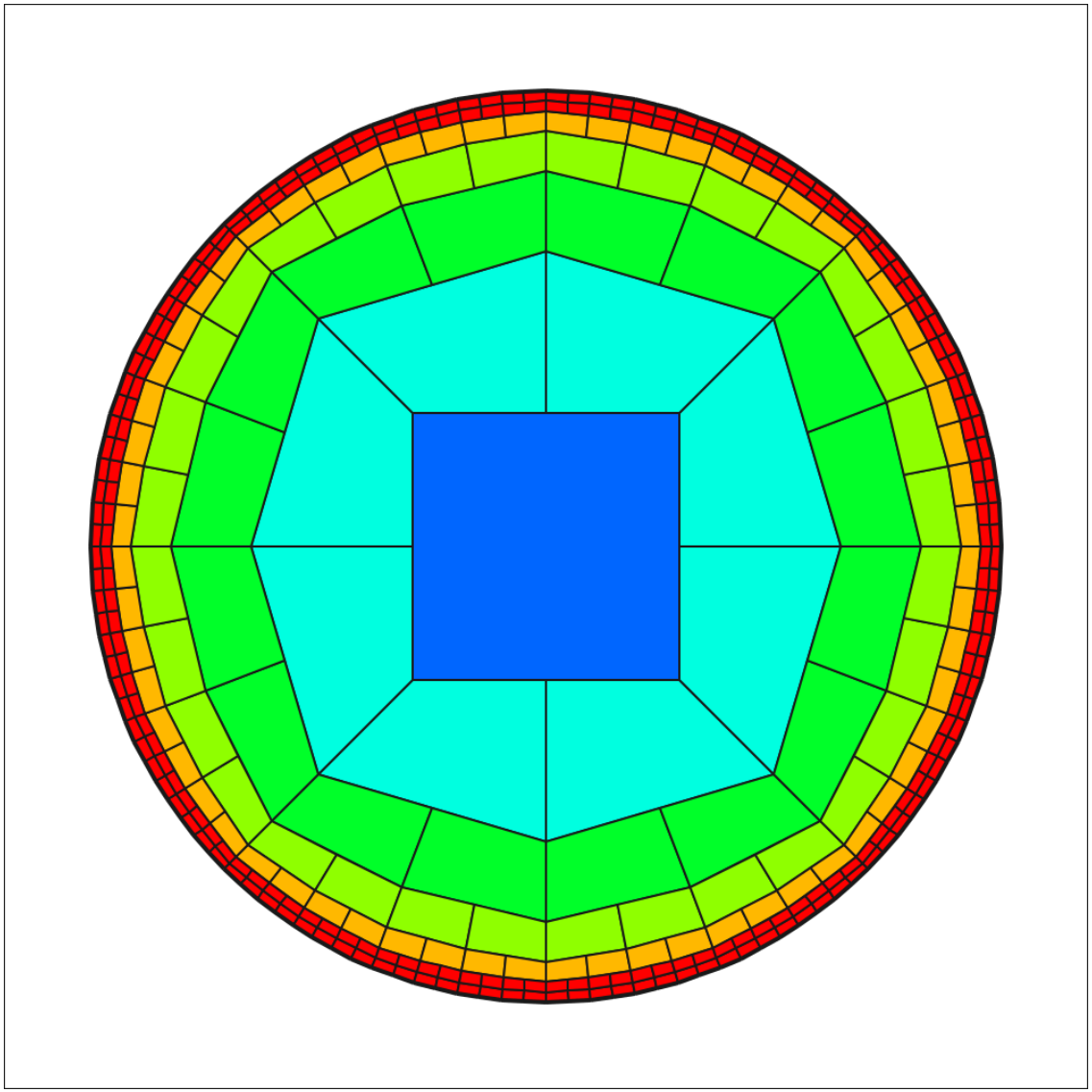
Cells in `deal.II` have many capabilities. You can get their vertices, their barycenter, their status, etc.
One attractive feature is that you can refine and coarse cells as you wish. This enables you to dynamically refine a mesh. We will explain in the following weeks what is the root of this capability.

Example

This example generates the mesh of a circle and refines it if the cells are at the boundary.

```
Triangulation<2> triangulation;
const Point<2> center(1, 0);
const double radius = 0.5;
GridGenerator::hyper_ball(triangulation, center, radius);
for (unsigned int step = 0; step < 5; ++step)
{
^^Ifor (auto &cell : triangulation.active_cell_iterators())
^^I{
^^I^^Iif (cell->at_boundary())
^^I^^Icell->set_refine_flag();
^^I}
^^Itriangulation.execute_coarsening_and_refinement();
}
std::ofstream out("circle-grid.svg");
GridOut     grid_out;
grid_out.write_svg(triangulation, out);^^I
```

Example: Result



Outline

Outputting the triangulation

Writing a `Triangulation` to a raw text file would not be a complicated endeavour, but software such as Paraview do not support raw text file. `deal.II` provides the GridOut interface to output triangulation to a large array of mesh formats.

```
// Open a stream to write a file
std::ofstream out("grid-2.vtk");
// Create the grid output object
GridOut      grid_out;
// Write the output grid
grid_out.write_vtu(triangulation, out);
```

Outline

Conclusion

- Finite elements (and FVM) simulations require a Triangulation of a domain.
- This notion is well represented using OO programming.
- We have seen how to load a triangulation, iterate over its cell, manipulate the cells and output the resulting `Triangulation`.
- The first part of HW2 will enable you to learn more about this.
- Looping over cells and interacting with them will be dimension independent...