APSC Project
Report

Finite Element Modeling of Metal Additive Manufacturing

Submitted by

Claire BRUNA-ROSSO Mat. 10498604

Professors :
Luca FORMAGGIA & Carlo De FALCO

Politecnico di Milano
Milano, Italia

September 2015

Abstract

The additive manufacturing for metals is a fast growing technology. It allows to build geometrically complex parts from a powder bed by selectively melting it using a laser beam. It currently requires further knowledge to be fully managed, which can be brought by simulations. One significant variable to characterize the process quality is the temperature. To predict the thermal field, the heat conduction equation is to be solved. The numerical method chosen was the Finite Element one. To implement it the open source C++ library Deal.ii was used.
The process brings several modeling challenges, among which a moving localized heat source, a time-dependent geometry and sharp thermal gradients. Another issue currently faced by additive manufacturing is that the process complexity leads to computationally heavy models. To tackle the dynamic geometry issue, an element death and birth technique was implemented. Then an adaptative mesh refinement method was used to obtain a good precision where needed without generating excessive computational loads.
The model was then tested both for validation purposes, in configurations where an analytical solution is available, and in more realistic situations.
The model showed good agreement with the theory when considering the temperature computed and the convergence rates. The adaptative mesh refinement procedure also showed a promising influence on the model efficiency. The realistic simulations sensibles results and a behavior closer to the one which is expected. However the model is overly simplified when compared to the real process. Several features are to be implemented to make sharper thermal field predictions, such as temperature dependent material properties or a gaussian distribution of the heat input power.

Contents

List of Figures

Chapter 1

Introduction

For the past ten years, new metal manufacturing techniques have appeared, opening to a new range of manufacturable geometries. Those are the additive manufacturing (AM) methods such as the selective laser melting (SLM). It consists in building a part layer by layer by selectively melting the metal powder with a laser source. However some limits appears in term of the quality of the part produced. Moreover a lot a physical phenomena are involved in the process making it hard to understand and manage (figure 1).

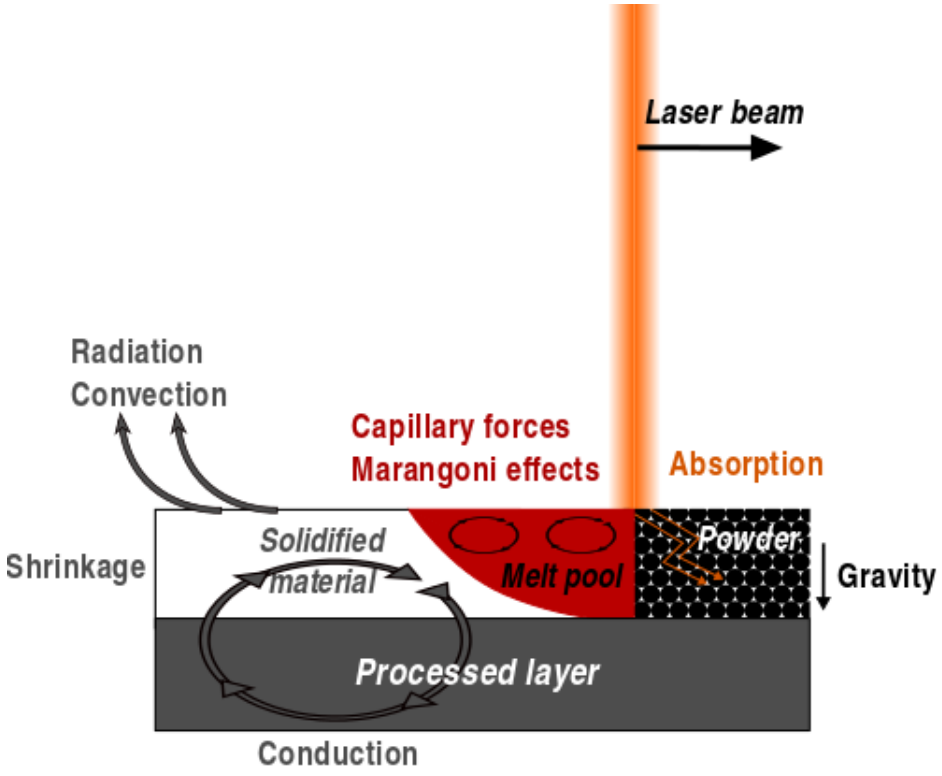


Figure 1.1: SLM process and physical phenomena

In order to improve the SLM knowledge and management, the simulation appears as a relevant tool thanks to its versatility, flexibility, and time and money savings. A relevant variable to predict in term of quality specifications is the temperature. Indeed it can be linked to the mechanical and material properties (microstructure, density, yield stress etc). In the case of SLM, the major heat transfer mode is conduction, convection and radiation inside the part can be neglected. The mathematical problem to be solved is thus a parabolic one, for which the

Finite Element (FE) method is adapted. Some attempts in modeling the SLM using FE have already been performed and are quite successful [2]. However some issues appeared, among which the too large computing time because of a large number of degree of freedom. Indeed, large thermal gradients induced by the laser source require a fine mesh, high laser velocity requires small time steps, the manufacturing process leads to time-dependent geometries, and finally the part produced have complex and possibly large geometries. In order to address those issues, two specific features are to be implemented in the present project :

- Adaptative mesh refinement : finer mesh where high gradients occurs, coarser where the solution has small variations
- Element birth technique : As the part building progress, the elements corresponding to the newly added material are activated. This technique prevents from updating the geometry at every time step.

First, an overview of the code is made. Then the mathematical problem and its resolution using the finite element method is presented. The chapter 4 to 6 introduce the specific features which have been implemented to meet the projects objectives, namely the adaptative mesh refinement, the finite element domain evolution and the handling of the solution vector through those steps as well as through time evolution. Finally the results which contain the final code, its validation, and an example of the results it can produce are presented in chapter 7.

Chapter 2

Overview

This section will first briefly introduce the deal.II library and then presents some specific features which were extensively used and/or of special importance for the current project. Finally the program organization and a brief description of each method implemented are provided.

2.1 Deal.II library

A short presentation of deal.II is provided, as well as an insight on the library features that are of special importance in this project.

2.1.1 Introduction

Deal.II is an open source project aiming at providing high quality tools to support the development of Finite Element solvers. For more details, please refer to the project website `www.dealii.org`. Among the various classes offered by the library, the ones related to the following functionalities have been used :

- General FEM classes :
  - Triangulation
  - Quadrature
  - Mapping
  - Finite Element
  - Solver
  - Postprocessing
- More specific classes :
  - FE collection : allow to use different finite elements on the same triangulation
  - Mesh refinement : several classes and methods are implemented to allow automatic mesh refinement and the transfer of the solution through different meshes.

2.1.2 Specific features

Template

One other feature provided by deal.II which has been extensively throughout the present project in the use of templates. Indeed, every method depends on the template parameter `dim`. It

corresponds to the dimension of the problem to be dealt with. It allows to make generic programming and have the same programs running in any dimension. Nevertheless, care has to be taken to make sure that all the methods are correctly implemented to work for any value of `dim`. Within the scope of this project, only the 2D case was considered for simplicity and rapidity considerations, but the program is equally suited for 3-dimensional problems.

The `DoFHandler` class

As this object is going to be used a lot throughout this report, further explanations are provided. This class is the one taking care of the degrees of freedom of the finite element model. That is to say it makes the link between the triangulation (mesh) and the finite element (the test functions on each cell). Indeed, the triangulation describes the geometry of the domain, i.e how many cells it contains, their type (hexaedra, square, lines...) and their location. The finite element object describes how many degrees of freedom are located on each cells. A mesh and a finite element object put together define a finite element space V_h. Distributing the degrees of freedom through a triangulation using the right finite element object using a `DoFHandler` object ensures that V_h has a basis and that the vectors (=shape functions) of this basis are enumerated and indexed.
Finally the `DoFHandler` object associate each degree of freedom to its value. More specifically, in the finite element space, the temperature field can be written : $T = \sum_0^N T_i v_i$, were v_i is the set of test functions and N the number of degree of freedom. The `DoFHandler` object match each T_i to the right v_i.

The `hp::` namespace

This namespace contains classes which have been implemented in order to be used to perform hp-refinement. It allows to use the same features (quadrature, mapping, dof_handler etc.) as in the classic case (i.e on type of F.E element used everywhere on the domain). Within this project, only h-refinement (i.e mesh-refinement) has been performed (see chapter **??**), but this namespace has been used to be able to use two types of element which are manually distributed on every cell (see chapter 4).
For example the `hp::FE_collection` object allows to store all the FE types (i.e. the kind of shape functions) needed, and `fe_index` provided by the `hp::dof_handler` allows to choose the right one.

2.2 Program organization

The program is divided into three classes :

1. The `RightHandSide` class : allows to compute the values of a RHS function
2. The `InitialCondition` class : allows to compute the values of an initial condition function
3. The `HeatEquation` class : contains all the necessary methods to solve the problem at hand (see section 2.3)

The code is finally completed with a `main` function as usual.

2.3 The `HeatEquation` class

This class is the most important one of the code. It contains the following methods :

The `HeatEquation` method

Constructor of the class. It initialize all the parameters required for the simulations.

The `create_coarse_grid` method

Create the initial mesh of the domain.

The `cell_is_in_metal_domain` and `cell_is_in_void_domain` methods

Methods taking as input a cell of the mesh, and returning a boolean: true if the cell is in the metal (resp. void) domain, false otherwise.

The `set_active_fe_indice` method

Method affecting the right finite element to the cell depending on which domain (metal or void) it is. It uses the boolean computed by the cell_is_in_metal_domain and `cell_is_in_void_domain` methods.

The `refine_mesh` method

Method which refine and coarsen the cells, depending on a a posteriori error estimator.

The `setup_system` method

Memory allocation of all the matrices and vectors required for the problem finite element resolution.

The `store_old_vector` method

Method called at the end of each time step which store the solution in a map whose keys are the node positions.

The `transfer_old_vector` method

Considering that the degrees of freedom (dof) changes between two times steps because of mesh refinement and changes in the finite element used on each cell, the "old" solution (from the previous time step) has to be transfered to the new dofs. It uses the map filled by the `store_old_vector` method).

The `assemble_system` method

Actually compute the matrices (mass matrix and rigidity matrix) and the right hand side (RHS) vector of the system to be solved.

The `solve_time_step` method

Actually compute the solution using a direct solver.

The `output_results` method

Generates .vtk files containing the results of the computations to be used for future postprocessing.

The `run` method

Only public method of the class other than the constructor. This is the backbone of the program, it organizes all the previous method and contains the time loop to solve the problem at hand.

Chapter 3

Finite Element Solving

This chapter first shortly presents the problem at hand and the mathematical methods used to solve it (for more details on the methods, see Quarteroni (2014) [4]). Then it introduces the methods implemented related to the finite element resolution.

3.1 Problem Definition

Within the scope of this project, a simplified version of the additive manufacturing was modeled. Its main features which have to be taken into account are the following ones :

- A moving heat source (provided by the laser beam)
- A constant temperature in the building chamber
- A base plate (on which the part is built) at a constant temperature
- A cooling by convection between the part free surfaces and the building chamber atmosphere.

Those characteristics translate into the following problem which was dealt with within this project :

$$\begin{cases} c\frac{\partial T(\mathbf{x},t)}{\partial t} - k\Delta T(\mathbf{x},t) &= f(\mathbf{x},t) & \text{on } \Omega, \text{ t } >0 \\ T(\mathbf{x},t) &= T_{amb} & \text{on } \Omega, \text{ t } = 0 \\ T(\mathbf{x},t) &= T_s & \text{on } \Gamma_D, \text{ t } >0 \\ \frac{\partial T(\mathbf{x},t)}{\partial n} - h(T(\mathbf{x},t) - T_{amb}) &= 0 & \text{on } \Gamma_R, \text{ t } >0 \end{cases}$$

Where T is the temperature, k is the thermal conductivity, c is the thermal capacity, T_s is the (fixed) temperature of the substrate on which the part is built, T_{amb} is the ambient temperature in the building chamber, and h is the convection coefficient.
The domain Γ_D where is applied a Dirichlet boundary condition represents the area in contact with the substrate.
The domain Γ_R where is applied a Robin (convection) boundary condition represents the area in contact with the building chamber atmosphere.
The whole domain is represented on the figure 3.1.

3.2 Finite element formulation of the problem

This section will first present a brief recap of the discretization method used. Then the space and time discretization processes of the actual problem are described.

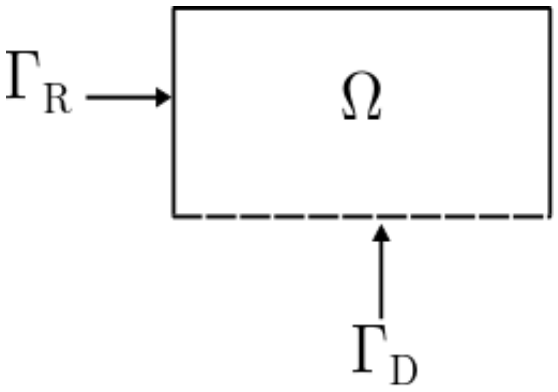


Figure 3.1: Domain Ω

3.2.1 Galerkin Discretization Method Summary

Physical phenomena provide problems which can be modeled by systems of Partial Differential Equation (PDE) associated with Boundary conditions (BC). In case an analytical solution cannot be derived, one needs to turn the continuum problem with an infinite number of degrees of freedom to a discrete and solvable one with a finite number of unknowns.
Here the Galerkin method was applied to convert the continuous operator (i.e. the PDE) in a discrete one. It consists in two major steps :

1. **Weak formulation of the problem :**
   Choosing a Hilbert space V having the adequate regularity properties imposed by the problem, it can be written in its weak form :

   $$\text{Find } u \in V,\ \forall v \in V\ a(u,v) = F(v) \tag{3.1}$$

   where a is a coercive and continuous form, and F is a bounded linear functional on V.

2. **Dimension reduction :**
   Choosing a functional space V_h of finite dimension approximating V, the discrete finite dimensional problem is obtained :

   $$\text{Find } u_h \in V_h \subset V,\ \forall v_h \in V_h\ a(u_h,v_h) = F(v_h) \tag{3.2}$$

   This steps allows to numerically computes the vector u_h as a linear combination of the V_h basis vectors.

3.2.2 Weak formulation

In order to obtain the weak formulation necessary to derive the Finite Element one to be implemented, two discretizations have to be performed : in time and in space. There are two methods to do so :

- Method of lines : first discretize in space and then in time. It leads to a large system of ordinary differential equations
- Rothe method : first discretize in time and then in space. It leads to a steady partial differential equation at each time step

Here the Rothe method has been implemented. There are two main reasons for that :

- A low-order time-integration method is used (see section 3.2.2)
- The space discretization is evolving at each time-step because of both the adaptative mesh refinement and the simulation of material addition

Performing first the time discretization prevents to make it at each time-step, which thus improves the efficiency.

Time Discretization

A θ-scheme (trapezoidal) time discretization is used. The semi-discrete version of the problem then obtained is the following one :

$$c\frac{T^n(\mathbf{x}) - T^{n-1}(\mathbf{x})}{\tau_n} - k[(1-\theta)\Delta T^{n-1}(\mathbf{x}) + \theta\Delta T^n(\mathbf{x})] = (1-\theta)f(\mathbf{x}, t_{n-1}) + \theta f(\mathbf{x}, t_n) \quad (3.3)$$

Where $\tau_n = t^n - t^{n-1}$.
Using this formulation allows to easily implement the three most common time integration formulas, namely :

- Forward Euler : $\theta = 0$
- Backward Euler : $\theta = 1$
- Crank - Nicolson : $\theta = 0.5$

Space Discretization

The space discretization of the equation 3.3, the usual method was used : first multiply by a test function v belonging to a correct functional space V, then integrate by part the term including the Laplacian, and finally apply the boundary conditions. The fully discrete problem thus reads :

Find $T \in V$ such that :

$\forall v \in V$:

$$c\int_\Omega T^n v d\Omega \; - \; k\tau_n\theta\int_\Omega \nabla T^n \nabla v \, d\Omega \; - \; h\tau_n\theta\int_{\Gamma_R} T^n v d\gamma \; =$$
$$\tau_n\int_\Omega [(1-\theta)f^{n-1} + \theta f^n] v \, d\Omega \; + \; c\int_\Omega T^{n-1} v d\Omega \; + \; k\tau_n(1-\theta)\int_\Omega \nabla T^{n-1}\nabla v \, d\Omega \; -$$
$$h\tau_n(1-\theta)\int_{\Gamma_R} T^{n-1} v \, d\gamma \; - \; h\tau_n T_{amb}\int_{\Gamma_R} v \, d\gamma \quad (3.4)$$

Considering that the highest derivation order is 1, the space V has to be chosen in order to ensure T and ∇T integrability so that every integral makes sense. Moreover we have Dirichlet BC on Γ_D. The correct space is thus $V = H^1_{\Gamma_D}(\Omega) = \{v \in H^1(\Omega), v|_{\Gamma_D} = 0\}$.

3.2.3 Finite element formulation

As briefly introduced in the chapter 3.2.1, the Galerkin method is used to obtain a discrete problem with a finite number of unknowns. The space V is approximated with a subspace of finite dimension V_h. Then the temperature T can be written as a linear combination of the V_h

basis functions : $T = \sum_h T_h v_h$.
Its algebraic form is :

$$(M + \tau_n \theta K + \theta \tau_n M^{\Gamma_R}) T^n = \\ M T^{n-1} - \tau_n (1-\theta) K T^{n-1} + \tau_n [(1-\theta) F^{n-1} + \theta F^n] \\ - (1-\theta) \tau_n V^{\Gamma_R} T^{n-1} - \tau_n T^{amb} V^{\Gamma_R} \quad (3.5)$$

Where:

- M is the mass matrix : $M_{ij} = c \int_\Omega v_i(\mathbf{x}) v_j(\mathbf{x}) d\mathbf{x}$
- K is the stiffness matrix : $K_{ij} = k \int_\Omega \nabla v_i(\mathbf{x}) \nabla v_j(\mathbf{x}) d\mathbf{x}$
- F is the RHS vector : $F_i = \int_\Omega f(\mathbf{x}) v_i(\mathbf{x}) d\mathbf{x}$
- $M_{ij}^{\Gamma_R} = h \int_{\Gamma_R} v_i(\mathbf{x}) v_j(\mathbf{x}) d\mathbf{x}$
- $V_i^{\Gamma_R} = h \int_{\Gamma_R} v_i(\mathbf{x}) d\mathbf{x}$

Those elements are the ones to be assembled in order to compute the vector of nodal temperatures T at each time step.

3.3 Implementation

This section presents the different methods related to the implementation of the finite element resolution of the problem.

3.3.1 Mesh creation

Performed by the method `create_coarse_mesh`. Here it has been decided to solve the equation on a 2D rectangle domain (figure 3.1). This is easily implemented using deal.II functions.

```
template <int dim>
    void HeatEquation<dim>::create_coarse_grid ()
    {
    // Creation of two points
      Point<dim> p1;
      Point<dim> p2;

   // Affecting coordinates to p1 and p2
      for (unsigned int n=0; n<dim ; n++)
      {
         p1[n] = 0;
         p2[n] = edge_length;
      }

      p2[dim-1] = layerThickness*number_layer;

   // Generate a parallelepipedium (rectangle in 2D) having a [p1 p2] diagonal
      GridGenerator::hyper_rectangle(triangulation, p1, p2);
    }
```

3.3.2 System data memory allocation

The memory allocation for the system data, namely the mass matrix, the stiffness matrix, and the right hand side vector are made through the `setup_system` method. It also compute the non-zero components of the two matrices.
This method also distributes the degrees of freedom on the grid, according to the finite element type (here linear Lagrange element or `FE_Nothing` finite element) of each cell. Practically, since we compute only one scalar field (the temperature) at each time step, we have one degree of freedom at each node belonging to a cell having Lagrange element, and zero if the node belongs only to cells having a `FE_Nothing` finite element.

```
template <int dim>
 void HeatEquation<dim>::setup_system()
{
  // DOF distribution using proper finite element
  dof_handler.distribute_dofs(fe_collection);

  // Creation of the constraints to handle hanging nodes
  constraints.clear();
  DoFTools::make_hanging_node_constraints(dof_handler,
                constraints);
  constraints.close();6

  // Creation of the sparsity pattern for the system matrices
  DynamicSparsityPattern dsp(dof_handler.n_dofs());
  DoFTools::make_sparsity_pattern(dof_handler,
                        dsp,
                        constraints,
                        true);
  sparsity_pattern.copy_from(dsp);

  mass_matrix.reinit(sparsity_pattern);
  laplace_matrix.reinit(sparsity_pattern);
  system_matrix.reinit(sparsity_pattern);

  // Mass matrix computation
  MatrixCreator::create_mass_matrix(dof_handler,
                          quadrature_collection,
                          mass_matrix,
                          (const Function<dim> *)0,
                          constraints);

  // Stiffness matrix computation
  MatrixCreator::create_laplace_matrix(dof_handler,
                            quadrature_collection,
                            laplace_matrix,
                            (const Function<dim> *)0,
                            constraints);

  // Memory allocation for the RHS vector
  system_rhs.reinit(dof_handler.n_dofs());
}
```

3.3.3 Finite element system computation

The computations needed to solve the finite element problem (equation 3.5) are done by the method `assemble_system`. It allows to obtain the system written in the section 3.2. It first computes the matrices and right hand side vector without taking into account the Robin boundary conditions.

```
template <int dim>
    void HeatEquation<dim>::assemble_system()
      {
      Vector<double> tmp;
      Vector<double> tmp2;
      Vector<double> forcing_terms;

      tmp.reinit (solution.size());
      tmp2.reinit (solution.size());
      forcing_terms.reinit (solution.size());

      tmp2.add(heat_capacity, old_solution);
        mass_matrix.vmult(system_rhs, tmp2); // rhs = c*M*T^(n-1)

        laplace_matrix.vmult(tmp, old_solution); // tmp = K*T^(n-1)
        system_rhs.add(-(1 - theta) * time_step * heat_conductivity, tmp); //rhs
            = (c*M-(1-theta)*tau*k)*K*T^(n-1)

        // Computation of the forcing terms (= laser heat input)
        RightHandSide<dim> rhs_function;

        rhs_function.set_time(time); // t=tn
        VectorTools::create_right_hand_side(dof_handler,
                                  quadrature_collection,
                                         rhs_function,
                                         tmp); //tmp = F^n
        forcing_terms = tmp; // forcing_terms = F^n
        forcing_terms *= time_step * theta; // forcing_terms = tau*theta*F^n

        rhs_function.set_time(time - time_step); // t=t(n-1)
        VectorTools::create_right_hand_side(dof_handler,
                                quadrature_collection,
                                         rhs_function,
                                         tmp); //tmp = F^(n-1)
        forcing_terms.add(time_step * (1 - theta), tmp); // forcing_terms =
            tau*theta*F^n+tau*(1-theta)*F^(n-1)

        system_rhs += forcing_terms; // rhs = tau*theta*F^n+tau*(1-theta)*F^(n-1)
            + (c*M-(1-theta)*tau*k*K)*T^(n-1)

        system_matrix.add(heat_capacity, mass_matrix); // A = cM
        system_matrix.add(theta * time_step * heat_conductivity, laplace_matrix);
            // A = c*M+ theta*K*tau*K

      }
```

Then the missing terms deriving form the convective BC are computed and added to the system matrix and RHS.

```
  // Applying Robin boundary conditions

  const unsigned int n_face_q_points =
      face_quadrature_collection[0].size(); // quadrature points on faces
  const unsigned int n_q_points = quadrature_collection[0].size(); //
      quadrature points on elements

  // Finite element evaluated in quadrature points of a cell.
  hp::FEValues<dim> hp_fe_values (fe_collection,
                      quadrature_collection,
                      update_values | update_quadrature_points |
                          update_JxW_values);

  // Finite element evaluated in quadrature points of the faces of a cell.
hp::FEFaceValues<dim> hp_fe_face_values(fe_collection,
                          face_quadrature_collection,
                          update_values | update_quadrature_points|
                              update_JxW_values );

// Iteration over all the cells
typename hp::DoFHandler<dim>::active_cell_iterator
  cell = dof_handler.begin_active(),
  endc = dof_handler.end();
  for (; cell!=endc; ++cell)
    {
   const unsigned int dofs_per_cell = cell->get_fe().dofs_per_cell;
   FullMatrix<double> cell_matrix;
   Vector<double>     cell_rhs;
   std::vector<types::global_dof_index> local_dof_indices (dofs_per_cell);

   if (dofs_per_cell != 0 ) // Skip the cells which are in the void domain
   {
      cell_matrix.reinit(dofs_per_cell, dofs_per_cell);
      cell_matrix = 0;

      cell_rhs.reinit(dofs_per_cell);
      cell_rhs = 0;

      for (unsigned int face_number=0; face_number <
          GeometryInfo<dim>::faces_per_cell; ++face_number)
      {
      // Tests to select the cell faces which belong to the Robin boundary
        if (cell->face(face_number)->at_boundary()
           && (cell->face(face_number)->boundary_id() == 0))
          {
           // Term to be added in the RHS
           hp_fe_face_values.reinit (cell, face_number);
           for (unsigned int q_point=0; q_point<n_face_q_points; ++q_point)
             {
              // Computation and storage of the value of the shape functions
                  on boundary face integration points
               const FEFaceValues<dim> &fe_face_values =
                   hp_fe_face_values.get_present_fe_values();
                for (unsigned int i=0; i<dofs_per_cell; ++i)
                cell_rhs(i) += (fe_face_values.shape_value(i,q_point) *
```

```
                                fe_face_values.JxW(q_point));

              // Computation and addition of the terms in the RHS
               cell->get_dof_indices (local_dof_indices);
               for (unsigned int i=0; i<dofs_per_cell; ++i)
                  system_rhs(local_dof_indices[i]) -=
                   time_step*convection_coeff*cell_rhs(i)*
                   ((1-theta)*old_solution(i)-Tamb);
             }

            // Term to be added in the system matrix
             hp_fe_values.reinit(cell);
             const FEValues<dim> &fe_values =
                 hp_fe_values.get_present_fe_values();
             // Computation and storage of the value of the shape functions
                 on boundary cell integration points
                for (unsigned int q_point=0; q_point<n_q_points; ++q_point)
                  for (unsigned int i=0; i<dofs_per_cell; ++i)
                    for (unsigned int j=0; j<dofs_per_cell; ++j)
                     cell_matrix(i,j) += (fe_values.shape_value(i,q_point) *
                              fe_values.shape_value(j,q_point) *
                              fe_values.JxW(q_point));

                cell->get_dof_indices (local_dof_indices);
                   for (unsigned int i=0; i<dofs_per_cell; ++i)
                     for (unsigned int j=0; j<dofs_per_cell; ++j)
                      boundary_matrix.add (local_dof_indices[i],
                               local_dof_indices[j],
                               cell_matrix(i,j));
           }
         }
       }
   // Computation and addition of the terms in the system matrix
      system_matrix.add (theta * time_step * convection_coeff,
          boundary_matrix);

    }

    constraints.condense (system_matrix, system_rhs);

  }
```

3.3.4 System resolution

The system solution computation is performed using a direct resolution, since the system is linear and can be written as $Ax = b$. The system to be solved is also not too big (within this project only the 2D case was considered) and thus a direct method was deemed preferable to an iterative one. The deal.II library has the UMFPack (`http://faculty.cse.tamu.edu/davis/suitesparse.html`) suite directly incorporated, so it was used in order to practically solve the problem. This suite provides routines implemented for linear sparse systems resolutions, and is thus adapted to the problem at hand.

```
template<int dim>
  void HeatEquation<dim>::solve_time_step()
```

```
{
 SparseDirectUMFPACK A_direct;

 // Initialization and LU factorization of A_direct
  A_direct.initialize(system_matrix);
  // Direct resolution
  A_direct.vmult (solution, system_rhs);

  // Application of hanging nodes constraints
  constraints.clear();
  DoFTools::make_hanging_node_constraints(dof_handler,
                                constraints);
  constraints.close();

  constraints.distribute (solution);

    std::cout << "***Time step " << timestep_number << " at t=" << time
              << std::endl;
    std::cout << std::endl
                      << "=========================================="
                      << std::endl
                      << "Number of active cells: " <<
                          triangulation.n_active_cells()
                      << std::endl
                      << "Number of degrees of freedom: " <<
                          dof_handler.n_dofs()
                      << std::endl
                << std::endl;

}
```

Chapter 4

Domain Evolution

One of the main challenge brought by manufacturing simulations is the dynamic shape of the domain. Indeed the building of the part induces a time-dependent geometry. Updating the geometry of the domain at each time-step would be very expensive in terms of computational cost. Indeed, a new mesh would have to be recomputed at each time step. Moreover the techniques used to handle the solution through different meshes, domains, and time steps would be complexified a lot. That is why a different approach has been chosen, namely the death and birth technique.

4.1 Principle of the method

Instead of updating at each time step the shape and the mesh of the domain, the final shape is meshed from the beginning. Then all the cells are given a `FE_Nothing` finite element. In the deal.II library, it corresponds to an element with zero degree of freedom. It can be used to represent inactive cells which is the case of the ones belonging to the area of the part that has not been built yet. It allows to implicitly remove those cells from the computation. Then at every time-step, the cells corresponding to the newly added material are given first order Lagrange elements. However, this method leads to a non-uniform finite element space on all the domain which implies the use of specific tools provided by the deal.II `hp::` namespace.

4.2 Implementation

4.2.1 Measurement of the part height

In order to know which cell have to be given `FE_Nothing` and Lagrange elements, the first step is to separate the cells belonging to the already built region of the part from the "void". This is performed by the method `part_heigth_measure`.

```
template<int dim>
  void HeatEquation<dim>::part_height_measure()
  {
      double max_height = part_height; // Maximal height of a vertice belonging
          to the metal domain in the previous function call

     // iteration over all the cells and storage of the maximal height of a
         vertice belonging to the metal domain
          typename hp::DoFHandler<dim>::active_cell_iterator
           cell = dof_handler.begin_active(),
```

```
          endc = dof_handler.end();
        for (; cell!=endc; ++cell)
          {
            if (cell->active_fe_index() == 0)
            {
            double height_temp = 0;
            for (unsigned int v=0;
               v < GeometryInfo<dim>::vertices_per_cell; ++v)
               {
                  height_temp = cell->vertex(v)[dim-1];
                  if (height_temp > max_height)
                     max_height = height_temp;
               }
            }
          }
        part_height = max_height;
  }
```

The implementation of this method also ensure that a refinement cannot move a cell from the metal zone to the void zone.

4.2.2 Activation of the right FE

After having measured the part height, which constitutes the limit between the metal and the void, i.e. the limit between Lagrangian and `FE_Nothing` elements, the right type of finite element has to be given to every cell. For this purpose, three methods have been implemented :

- `cell_is_in_metal` : return true if the cell is in the metal domain
- `cell_is_in_void` : return true if the cell is in the void domain
- `set_active_fe_indices` : using the two methods above, affect the right type of finite element, i.e. `FE_Nothing` for the void, and Lagrangian element for the metal

The `cell_is_in_metal` and `cell_is_in_void` methods

This two methods work in a similar way : it makes an iteration on all the cells, looks at the coordinates of all the vertices and determine wether the cell belongs to the metal (resp. void) domain with respect to a time-dependent height. This height is the maximum between the limit corresponding to the number of layer already or being processed and the coordinate of the highest vertex that have been measured as belonging to the metal domain.

```
template <int dim>
   bool
  HeatEquation<dim>::
   cell_is_in_metal_domain (const typename hp::DoFHandler<dim>::cell_iterator
       &cell)
   {
    bool in_metal=false;
    unsigned int n = 0;
    for (unsigned int v=0; v < GeometryInfo<dim>::vertices_per_cell; ++v)
        {
            double limit = (1+floor(5*time))*layerThickness ;
          in_metal = (cell->vertex(n)[dim-1]) < std::max(part_height, limit);
          if (in_metal==false)
```

```
            n++;
        }
     return in_metal;
   }
```

```
   template <int dim>
   bool
  HeatEquation<dim>::
   cell_is_in_void_domain (const typename hp::DoFHandler<dim>::cell_iterator
       &cell)
   {
     bool in_void=false;
     unsigned int n = 0;
     for (unsigned int v=0; v < GeometryInfo<dim>::vertices_per_cell; ++v)
        {
           double limit = (1+floor(5*time))*layerThickness ;
           in_void = cell->vertex(n)[dim-1] > std::max(part_height, limit);
           if (in_void==false)
              n++;
        }
      return in_void;
   }
```

The `active_fe_indices` method

This method uses the booleans indicating wether a cell belongs to the void or the metal zone to set the right kind of finite element. An assertion also verify that every cell has been provided with a finite element, and throw an error if it is not the case.

```
template <int dim>
   void
   HeatEquation<dim>::set_active_fe_indices ()
   {

    // iteration over all the cells of the mesh
    for (typename hp::DoFHandler<dim>::active_cell_iterator
          cell = dof_handler.begin_active();
          cell != dof_handler.end(); ++cell)
       {
       // Lagrange element if the cell is in the metal domain
         if (cell_is_in_metal_domain(cell))
           cell->set_active_fe_index (0);
         // Zero element if the cell is in the void domain
         else if (cell_is_in_void_domain(cell))
           cell->set_active_fe_index (1);
         // Throw an error if none of the two cases above is encountered
         else
           Assert (false, ExcNotImplemented());
        }
   }
```

After this method is being called, the Finite Element space V_h needs to change to adapt to the current step of the part building. Indeed the shape function basis is not the same anymore

since new degrees of freedom have been brought to the problem. The system matrices and the right hand side thus have to be recomputed. That is why `setup_system` is being called right after inside the `run` method.

Important notice : The death and birth method has a major drawback. The upper free surface of the part (the one which is being processed) is not a boundary anymore. Indeed, it has inactive elements above. It is thus not possible to apply the convection BC while it would be necessary to do so. In the future a feature will be added to the code to compensate the temperature overestimation that is brought by the absence of convective cooling on the top surface.

Chapter 5

Adaptative mesh refinement

5.1 Motivations

As briefly presented in the introduction, adaptative mesh refinement allows to locally adapt the mesh size to the local solution variations. It provides a finer mesh where the error is significant while coarsening it where the error is lower. Considering that one of the limitations for the use of simulation in additive manufacturing is the fact that the models are or too heavy, or too simplified, this feature seemed pertinent to be include in this program. Indeed, the mesh is refined near the area of interest, namely close to the laser input, and coarsen away from it, i.e. where the metal is solidified and its temperature virtually stabilized. That allows to reduce the number of degree of freedom (i.e. the solution vector to be computed) while keeping a good accuracy, and thus improve the code efficiency.

5.2 Implementation

Nota bene : The present program only perform refinement without coarsening. The reason comes from a bug that has been brought to the knowledge of deal.II developers through the mailing list. A test case provoking the runtime error encountered has been sent in order to help them fixing this issue.

As briefly introduced earlier, the mesh coarsening and refinement has been included in order to reduce the computational load. It has been implemented in the first part of the `refine_mesh` method.

```
template<int dim>
void HeatEquation<dim>::refine_mesh(const unsigned int min_grid_level,
                          const unsigned int max_grid_level)
 {
   // Error estimation to set refine flags

    Vector<float> estimated_error_per_cell (triangulation.n_active_cells());

      // Computation of the vector of the Kelly error estimator on each cell

    KellyErrorEstimator<dim>::estimate (dof_handler,
                                        face_quadrature_collection,
                                        typename FunctionMap<dim>::type(),
                                        solution,
                                        estimated_error_per_cell);
```

```
  GridRefinement::refine_and_coarsen_fixed_fraction (triangulation,
                                                     estimated_error_per_cell,
                                                     0.6, 0.4);

// Clear the refine flag of the cell which are already at the maximal level
    of refinement

    if (triangulation.n_levels() > max_grid_level)
      for (typename Triangulation<dim>::active_cell_iterator
           cell = triangulation.begin_active(max_grid_level);
           cell != triangulation.end(); ++cell)
        cell->clear_refine_flag ();

    // Considering the bug in deal.II library all coarsen flags are removed
    // Otherwise only the coarsening flag of the cells which are at the
        minimal level of refinement would be cleared

    for (typename Triangulation<dim>::active_cell_iterator
         cell = triangulation.begin_active();
         cell != triangulation.end(); ++cell)
      cell->clear_coarsen_flag ();
```

The Kelly error estimator intends to give an approximation of the error on each cell using the integral of the jump of the solution gradient along the faces (or edges in 2D) of each cell. See Kelly et al. (1983) for more details [3]. It is computed using the following formula :

$$\eta_K^2 = \sum_{F \in \partial K} c_F \int_{\partial K_F} \left[a \frac{\partial T}{\partial n} \right] dx \tag{5.1}$$

K being the current cell, F the cell faces, c_F a coefficient practically determined, and $[.]$ the jump of the argument on the face.
The function return a vector used as input for the refine and coarsen methods.
After having flagged the cells which required coarsening and those who require refinement, the procedure actually compute the new triangulation and the new `DoFHandler`.

```
    // Computation of the new triangulation and DoFHandler
  triangulation.prepare_coarsening_and_refinement();
  SolutionTransfer<dim, Vector<double>, hp::DoFHandler<dim> >
      solution_trans(dof_handler);
  solution_trans.prepare_for_coarsening_and_refinement(solution);
  triangulation.execute_coarsening_and_refinement();
  dof_handler.distribute_dofs(fe_collection);
```

Chapter 6

Solution Vector Handling

The main challenge brought by this project is the management of the solution vector. Indeed, it has to be transfer through :

- Different time steps
- Different meshes
- Different finite elements

Within all this features, care has to be taken to ensure a proper handling of the vector, and thus ensure the correctness of the code.

6.1 Transfer through time steps

The transfer through time steps is the simplest one of the three. As usual, the method consisting in storing the solution vector of the previous time step in an "old solution" vector was used. It is important to store the previous solution since it is needed to compute the right hand side of the current time step.

6.2 Transfer through mesh refinement

During the process of mesh refinement, the triangulation evolves. Throughout the procedure, the solution vector changes size, so memory has to be properly allocated, and the values of the solution have to be interpolated to the new `DoFHandler`. Those actions are taken care of by the `SolutionTransfer` class used in the second part of the `refine_mesh` method.

```
      // Solution interpolation on the new Dof handler
    Vector<double> new_solution(dof_handler.n_dofs());
    solution_trans.interpolate(solution, new_solution);
    solution.reinit(dof_handler.n_dofs());
    solution=new_solution;

    old_solution = solution;

    constraints.clear();
    DoFTools::make_hanging_node_constraints(dof_handler,
                                            constraints);
    constraints.close();
```

```
    // Computation of the hanging nodes constraints
    constraints.distribute (solution);
  }
```

As shown in the code above, the transfer of the solution is completely transparent to the user. The idea below the `SolutionTransfer::interpolate` method is to interpolate the value at a point added in the triangulation from the value of adjacent ones. The final step is to ensure the continuity of the hanging nodes points by computing and applying node constraints to the solution vector.

6.3 Transfer through different finite elements

Just as the management of the solution through mesh refinement, its transfer through a different finite element domain involves a size change of the vector. This time other degrees of freedom are brought from the cells which are given Lagrangian elements instead of `FE_Nothing` elements. Contrary to the mesh refinement procedure, no interpolation of the solution is to be performed, because the dof added does not correspond to the division of an already active cell. That is why the `SolutionTransfer` class was not relevant is that case. Specific methods had thus to be implemented : one for the solution storage on the old `DoFHandler`, and an other to transfer to the updated one.

6.3.1 Solution storage

Principle and implementation

This method objective is to store the solution vector right after it is computed. The principle is to associate the degrees of freedom coordinates in space (key) and the solution value at this point (data) within a map.

```
template <int dim>
     void HeatEquation<dim>::store_old_vectors()
  {
      map_old_solution.clear();
      const MappingQ1<dim,dim> mapping;

      typename hp::DoFHandler<dim>::active_cell_iterator
         cell1 = dof_handler.begin_active(),
         endc1 = dof_handler.end();
          for ( ;cell1!=endc1; cell1++)
          {
          // Temporary variable to get the number of dof for the currently
              visited cell

              const unsigned int dofs_per_cell = cell1 -> get_fe().dofs_per_cell;

              std::vector<Point<dim>> support_points(dofs_per_cell);

              if (dofs_per_cell != 0) // To skip the cell with FE = FE_Nothing
                  because there is no support point there
              {
               // Get the coordinates of the support points on the unit cell
```

```
                support_points = fe_collection[0].get_unit_support_points();

                //Get the coordinates of the support points on the real cell
                for (int i=0;i<dofs_per_cell;i++)
                  {
                     support_points[i] =
                         mapping.transform_unit_to_real_cell(cell1,
                         support_points[i]);
                     map_old_solution[support_points[i]] =
                         VectorTools::point_value(dof_handler, old_solution,
                         support_points[i]);
                  }
               }
            }
  }
```

Point comparison operators

In order to use a certain type of data as a key for a map, it has to have comparison operators : $<$, $>$, $=$.
Here, the type `Point` is used, but does not have those operators implemented inside the deal.II library. It was thus necessary to code them. The (arbitrary) law which has been chosen in order to rank two `Point` is first in ascending order of x, then in ascending order of y, and finally (in case of a 3D problem) in ascending order of z. The implementation was made using the template parameter `dim`, and care has been taken to make a dimension independent code.

```
#ifndef POINT_COMPARISON_OPERATOR_H_
#define POINT_COMPARISON_OPERATOR_H_

#include <deal.II/base/config.h>
#include <deal.II/base/exceptions.h>
#include <deal.II/base/tensor_base.h>
#include <deal.II/base/point.h>
#include <cmath>

DEAL_II_NAMESPACE_OPEN

/**
 * Comparison operator overloading. @relates Point
 */
template <int dim, typename Number>
inline
bool operator < (const Point<dim,Number> &p1, const Point<dim,Number> &p2)
{
   unsigned int j=0;
   for(unsigned int i=0;i<dim;i++)
   {
      if (p1[i] != p2[i])
      {
      return p1[j]<p2[j];
      }

      else
      {
```

```
      j++;
      }
   }
   if(j==dim)
   {
      return false;
   }
}

template <int dim, typename Number>
inline
bool operator > (const Point<dim,Number> &p1, const Point<dim,Number> &p2)
{
   unsigned int j=0;
   for(unsigned int i=0;i<dim;i++)
   {
      if (p1[i] != p2[i])
      {
      return p1[j]>p2[j];
      }
      else
      {
      j++;
      }
   }
   if(j==dim)
      {
         return false;
      }
}

template <int dim, typename Number>
inline
bool operator == (const Point<dim,Number> &p1, const Point<dim,Number> &p2)
{
   unsigned int j=0;
   for(unsigned int i=0;i<dim; i++)
   {
      if (p1[i] == p2[i])
      {
         j++;
      }
   }
   if (j == dim)
   {
      return true;
   }
   else
   {
      return false;
   }
}

DEAL_II_NAMESPACE_CLOSE

#endif /* POINT_COMPARISON_OPERATOR_H_ */
```

6.3.2 Solution transfer

This method aims at matching the degrees of freedom of the new `DoFHandler` with the appropriate solution value.
Since the `DoFHandler` has evolved, the degrees of freedom are not numbered in the same way anymore, so the old solution vector is of no use. In fact its values would not be ordered in a proper way. Otherwise a simple use of a `push_back` would have been sufficient, adding the appropriate number of $T = T_{amb}$ at the end of `solution`, corresponding to the temperature of the newly added material (= the newly activated degrees of freedom). However, the coordinates remain identical, so the association of each point to its temperature will be done using them.

```
template <int dim>
  void HeatEquation<dim>::transfer_old_vectors()
  {
   // Creation of a solution of the same size of the number of dof of the new
       FE space
        Vector<double> long_solution;
        long_solution.reinit(dof_handler.n_dofs(), false);

        const MappingQ1<dim,dim> mapping;

        std::vector<types::global_dof_index> local_dof_indices;
      // Iteration over all the cells of the updated dof_handler to fill the
          solution vector from the one from of the former one
          unsigned int k=0;
          typename hp::DoFHandler<dim>::active_cell_iterator
           cell = dof_handler.begin_active(),
           endc = dof_handler.end();
             for ( ;cell!=endc; cell++)
                {
                 // Vector of the points already stored
                  std::vector<Point<dim>> points_stored;

                 // Number of dof of the currently visited cell
                  const unsigned int dofs_per_cell = cell ->
                      get_fe().dofs_per_cell;

                  // Vector of the support points of one cell
                  std::vector<Point<dim>> support_points(dofs_per_cell);

                  if (dofs_per_cell != 0) // To skip the cell with FE =
                      FE_Nothing because they have not any support point
                    {
                       // Get the coordinates of the support points on the unit
                           cell
                      support_points =
                          fe_collection[0].get_unit_support_points();

                        // Vector of the degree of freedom indices of one cell
                      local_dof_indices.resize(dofs_per_cell);
                      cell->get_dof_indices(local_dof_indices);
```

```
                    //Get the coordinates of the support points on the real
                        cell
                   for (int i=0;i<dofs_per_cell;i++)
                     {
                      support_points[i] =
                          mapping.transform_unit_to_real_cell(cell,
                          support_points[i]);
                      typename std::map< Point<dim>, double>::iterator
                          iter_old = map_old_solution.begin();

                      double solution_temp = 0;

                      // Variable to check if a point has already been stored
                      unsigned int is_point_stored = 0;

                      // Iteration in the old solution map
                       for ( ; iter_old!= map_old_solution.end(); iter_old++)
                          {
                            // Test if the point visited corresponds to a
                                point in the "old" dof_handler
                            if (support_points[i] == iter_old->first)
                            {
                               // store the Point currently visited
                               points_stored.push_back(support_points[i]);

                            }
                       typename std::vector<Point<dim>>::iterator
                           it_points=points_stored.begin() ;

                            // Test if the point has not been visited and
                                stored yet

                            for ( ; it_points != points_stored.end();
                                it_points++)
                            {
                               if (*it_points == iter_old->first)
                               {
                                  is_point_stored ++ ;
                                  // =1 if it is the first time the point it
                                      visited, >1 if it is not
                               }
                            }
                          }
                       // In case it has not been visited yet --> we store
                           the solution
                          if (is_point_stored == 1)
                          {
                           solution_temp =
                               map_old_solution.find(support_points[i])->second;
                           // Write the solution at the right place inside
                               vector
                           long_solution[local_dof_indices[i]] =
                               solution_temp;
                           k++;
                          }
```

```
                        }
                    }
                }

            solution.reinit(dof_handler.n_dofs());
            old_solution = long_solution;
            constraints.clear();
            DoFTools::make_hanging_node_constraints(dof_handler,
                                    constraints);
            constraints.close();
            constraints.distribute (old_solution);
}
```

Chapter 7

Results

7.1 Final code

This section presents more in details the class used to compute the initial and boundary conditions and the RHS values. Since most of the `HeatEquation` class methods have been explained earlier, only the `run` method will be dealt with here.

7.1.1 The `InitialCondition` class

The class aims at producing the object and computing the function $T_0(\mathbf{x})$ which will be applied as an initial condition on the entire domain Ω. The one implemented here is a constant temperature T_{amb} representing the ambient temperature inside the building chamber.
Nota Bene : In order to impose potentially location dependent initial condition, the method `value` has a `Point` as input. It is unused in the current implementation (and so generates a warning) but might be in the future.

```
template<int dim>
 class InitialCondition : public Function<dim>
 {
 public:
   InitialCondition ()
     :
     Function<dim>()
       {}

   virtual double value (const Point<dim> &p,
                 const unsigned int component = 0) const;
 };



 template<int dim>
 double InitialCondition<dim>::value (const Point<dim> &p,
                                      const unsigned int component) const
 {
   Assert (component == 0, ExcInternalError());
   return 1; // Initial Temperature value
 }
```

7.1.2 The `BoundaryCondition` class

This class aims at producing the object and computing the function to be applied as Dirichlet boundary condition. Here the boundary condition is a fixed temperature on the lowest side of the domain.
Nota Bene : In order to impose potentially location dependent Dirichlet BC, the method `value` has a `Point` as input. It is unused in the current implementation (and so generates a warning) but might be in the future.

```
template<int dim>
  class BoundaryValues : public Function<dim>
  {
  public:
    virtual double value (const Point<dim> &p,
                          const unsigned int component = 0) const;
  };

  template<int dim>
  double BoundaryValues<dim>::value (const Point<dim> &/*p*/,
                                     const unsigned int component) const
  {
    Assert(component == 0, ExcInternalError());
    return 1.; // Temperature to impose
  }
```

7.1.3 The `RightHandSide` Class

As expected, the `RightHandSide` method produce the object and compute the values of the function to be used as right hand side. However considering that this class has to represent the laser heat input, its implementation is a bit more complex. In this example, the value of the imposed heat is constant in time (since the laser has a constant power) but its position is time-dependent. The way it has been implemented consists in imposing $f_i = heat\ input$ to all nodes situated on a short moving segment on the surface of the part, while setting $f_i = 0$ elsewhere.

```
template<int dim>
  class RightHandSide : public Function<dim>
  {
  public:
    RightHandSide ()
      :
      Function<dim>(),
      period (0.2) // Time needed to complete one layer
    {}

    virtual double value (const Point<dim> &p,
                          const unsigned int component = 0) const;

    private:
    const double period;
  };

  template<int dim>
  double RightHandSide<dim>::value (const Point<dim> &p,
```

```
                                const unsigned int component) const
{
   {
       Assert (component == 0, ExcInternalError());

       const double time = this->get_time(); // get the time value
       double point_within_layer = (time/period - std::floor(time/period)) //
           return the x coordinate of point on which the laser has to be
           centered;
       double limit = (1+floor(time*5))*0.1 // returns the y coordinate of the
           part surface;
       double dist = point_within_layer;
       const double tol_dist = 5e-2;

           if ((p[0] > dist-tol_dist) && (p[0] < dist+tol_dist) && (p[1]>
               limit-tol_dist) && (p[1]<limit+tol_dist))
          {    return 1000; }
           else
             return 0;

    }
}
```

7.1.4 The `run` Method

This method is the backbone of the program. It organizes the time loop and the calls of all the methods introduced earlier to actually solve the problem at hands.

```
 template<int dim>
 void HeatEquation<dim>::run()
 {
   // Initial mesh creation

   const unsigned int initial_global_refinement = 5; // minimal level of
       refinement
   const unsigned int n_adaptative_pre_refinement_steps = 7; // maximal level
       of refinement
initial
   create_coarse_grid ();
   triangulation.refine_global (initial_global_refinement);

   // Set the right FE type for each cell
   set_active_fe_indices();

   // Initialize the matrices and RHS
   setup_system();
   old_solution.reinit(dof_handler.n_dofs());

   // INITIAL CONDITION

   //Instantiation
        InitialCondition<dim> initial_condition;

  //Projection of the function defined in the initial condition class on the
```

```
    limit of the domain
VectorTools::project(dof_handler,
                     constraints,
                     quadrature_collection,
                     initial_condition,
                     old_solution,
                     false,
                     face_quadrature_collection);

// DIRICHLET BOUNDARY CONDITIONS
{
   // Instantiation
         BoundaryValues<dim> boundary_values_function;
         boundary_values_function.set_time(time);

         std::map<types::global_dof_index, double> boundary_values;
         VectorTools::interpolate_boundary_values(dof_handler,
                                                  1,
                                                  boundary_values_function,
                                                  boundary_values);

     // Modifies old_solution vector, rhs vector and system matrix according
        to Dirichlet BCs

         MatrixTools::apply_boundary_values(boundary_values,
                                            system_matrix,
                                            old_solution,
                                            system_rhs);
}

// Initialisation of the solution vector taking into account the initial
    condition

solution = old_solution;

// Solution map filling

store_old_vectors();

 std::cout << "***Time step " << timestep_number << " at t=" << time
           << std::endl;
 std::cout << std::endl
       << "============================================="
       << std::endl
       << "Number of active cells: " << triangulation.n_active_cells()
       << std::endl
       << "Number of degrees of freedom: " << dof_handler.n_dofs()
       << std::endl
       << std::endl;

 output_results();

 // Beginning of the time loop

 while (time <= 1.)
   {
```

```
      time += time_step;
      ++timestep_number;

   // Part height measurement considering the possible new layer
  part_height_measure();

  // Give the right FE type to each cell
  set_active_fe_indices();

  // Compute the mass and stiffness matrices on the new FE domain
   setup_system();

   // Transfer the solution (Temperature field) to the new dofhandler
  transfer_old_vectors();

  // Compute the system matrix and RHS vector
  assemble_system();

  // Dirichlet boundary conditions
      {
        BoundaryValues<dim> boundary_values_function;
        boundary_values_function.set_time(time);

        std::map<types::global_dof_index, double> boundary_values;
        VectorTools::interpolate_boundary_values(dof_handler,
                                                 1,
                                                 boundary_values_function,
                                                 boundary_values);

        MatrixTools::apply_boundary_values(boundary_values,
                                           system_matrix,
                                           solution,
                                           system_rhs);
      }

      // Compute the vector of nodal temperatures with direct solver
      solve_time_step();

      // Adaptative mesh refinement
      refine_mesh(initial_global_refinement,
             n_adaptative_pre_refinement_steps);

      part_height_measure();

      // Produce the output files
      output_results();

      // Fill the map matching each point to its temperature
      store_old_vectors();
     }
  }
}
```

7.1.5 Code Utilization

Please refer to the README file provided with this report.

7.2 Code Validation

In order to validate the code, two examples for which an analytical solution exists have been simulated (the codes have been provided as test_case1.cc and test_case2.cc). Both of them have been run in order to ensure the program robustness for either the time-integration, the initial and boundary conditions, and also the pertinence of the Adaptative Mesh Refinement (AMR) in terms of computational efficiency. All the cases tested are summed up in the table 7.2.

Nota bene : In order to have an analytical solution of the problem, the time-dependence of the finite element has been removed (all cells were given linear Lagrangian elements). However, the data management method (namely `store_old_vector` and `transfer_old_vector`) are used so that we can test that they are well-programmed and do not bring any error.

Simulation	Test case	Time Step	Mesh refinement
1	1	0.01	Yes
2	1	0.005	Yes
3	1	0.01	No
4	2	0.01	Yes
5	2	0.005	Yes
6	2	0.01	No

Table 7.1: Validation simulations

7.2.1 Test case 1

This test case aims at solving the following problem :

$$\Omega = [0,1]^2$$

$$\begin{cases} c\frac{\partial T(x,y,t)}{\partial t} - k\Delta T(x,y,t) &= 0 & \text{on } \Omega,\ t>0 \\ T(x,y,t) &= 10\times\sin(\pi x)\sin(\pi y) & \text{on } \Omega,\ t=0 \\ T(x,y,t) &= 0 & \text{on } \partial\Omega,\ t>0 \end{cases}$$

The exact solution can be written as :

$$T(x,y,t) = a(t)\sin(\pi x)\sin(\pi y)$$

with :

$$a(t) = 10\times\exp(-2\pi^2 t)$$

The system was solved for two different time increments, 0.01 and 0.005, and with and without mesh refinement. The mesh size in case the refinement was not used was fixed to 32×32 elements, while in case of mesh refinement the mesh size could vary between 8×8 elements and 32×32. The results are displayed on figure 7.1. Then the L2-error of the difference between the analytical and computed temperature was calculated. (figure 7.4).

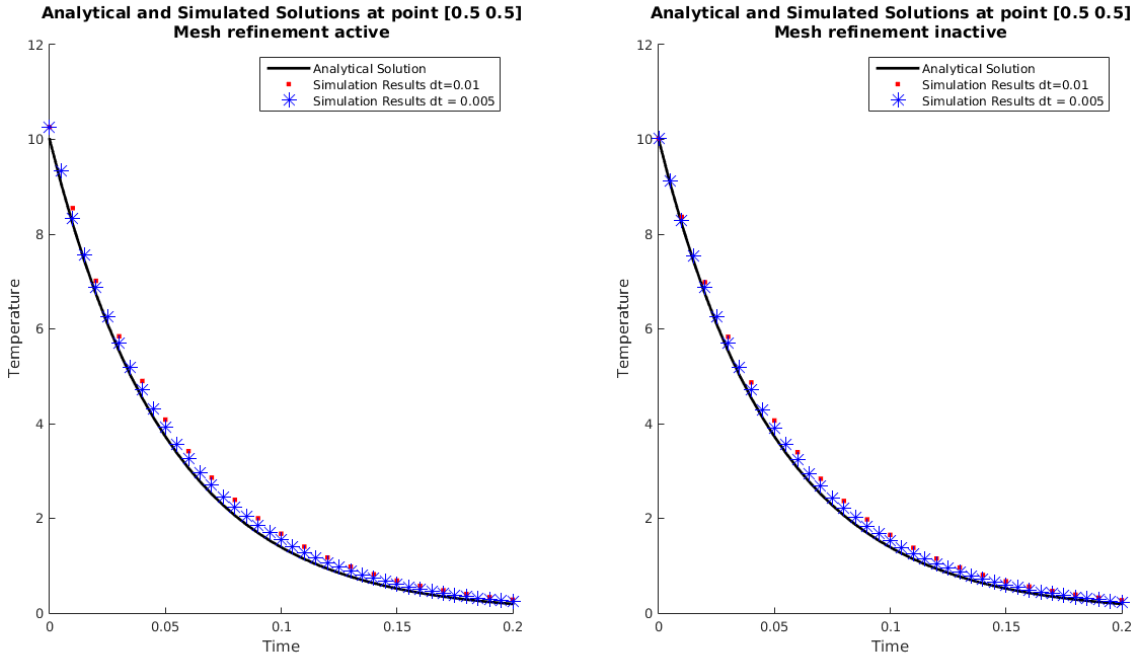


Figure 7.1: Test case 1 results comparison with analytical solution

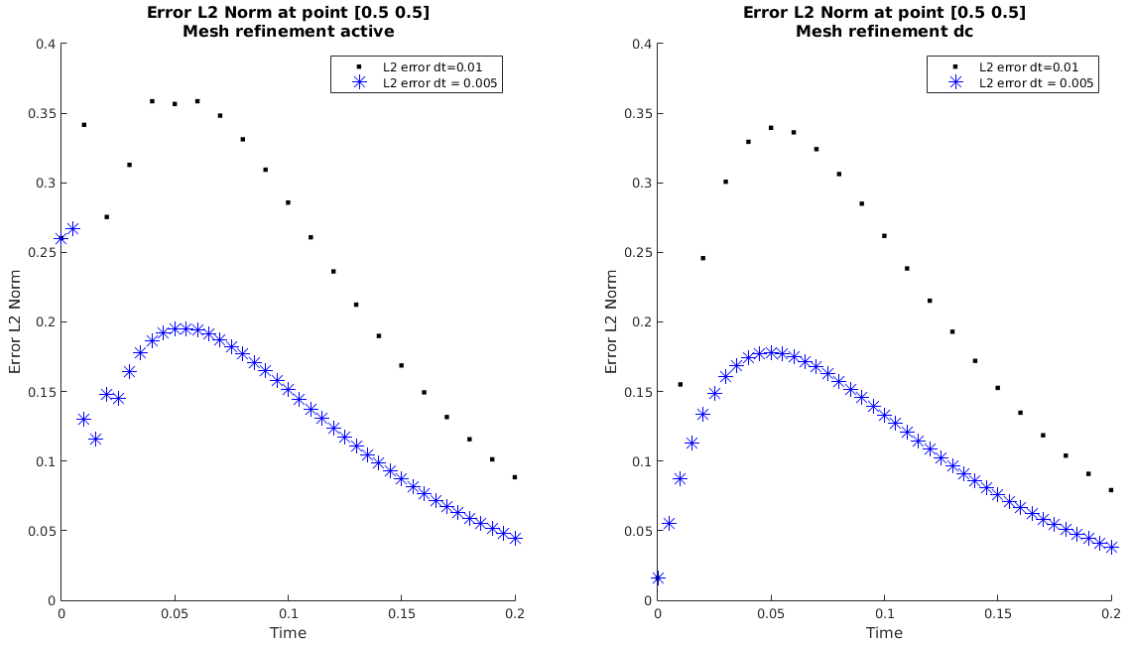


Figure 7.2: Test case 1 L2-errors

7.2.2 Test case 2

This test case aims at solving the following problem :

$$\Omega = [0, 1]^2$$

$$\begin{cases} c\frac{\partial T(x,y,t)}{\partial t} - k\Delta T(x,y,t) &= \sin(\pi x)\sin(\pi y) & \text{on } \Omega, \text{ t } >0 \\ T(x,y,t) &= 0 & \text{on } \Omega, \text{ t } = 0 \\ T(x,y,t) &= 0 & \text{on } \partial\Omega, \text{ t } >0 \end{cases}$$

The exact solution can still be written as :

$$T(x,y,t) = a(t)\sin(\pi x)\sin(\pi y)$$

with :

$$a(t) = \frac{1}{2\pi^2}(1 - \exp(-2\pi^2 t))$$

Just as the test case 1, the system was solved for two different time increments, 0.01 and 0.005, and with and without mesh refinement. The results are displayed on figure 7.3. Then the L2-error of the difference between the analytical and computed temperature was calculated (figure 7.4).

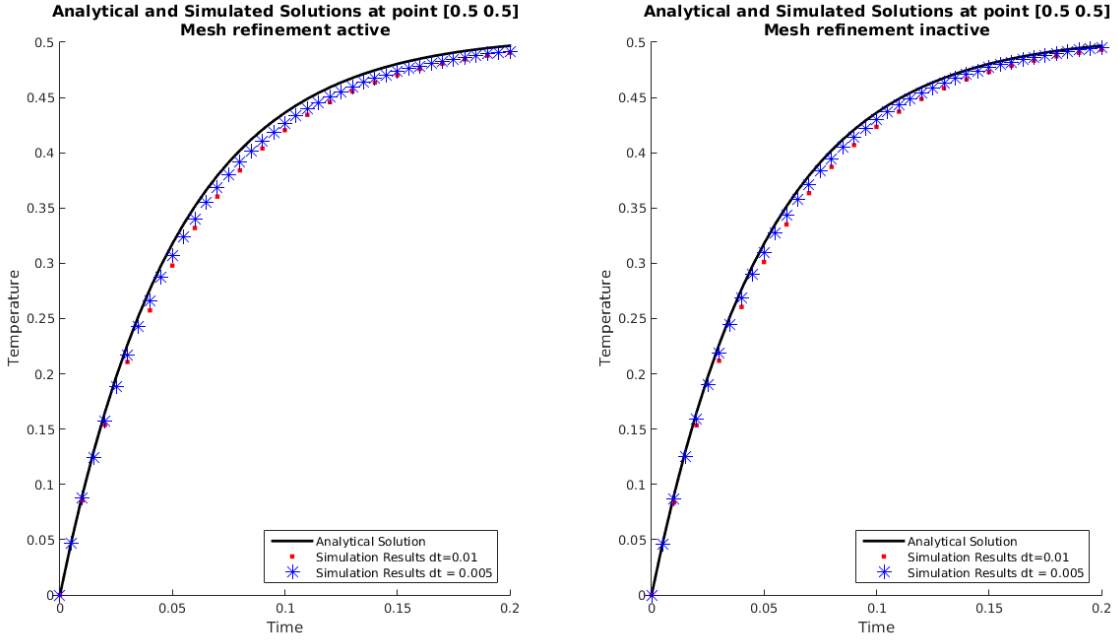


Figure 7.3: Test case 2 results comparison with analytical solution

7.2.3 Test case results discussion

What could be seen when comparing the results obtained numerically and analytically is that the heat equation resolution was correctly implemented. Indeed, the simulation outputs of both test cases are really close to the analytical one. Moreover, the convergence rate is in accordance with the expecting one considering that a first order time scheme (Backward Euler) is being used. Indeed when the time is divided by two (from 0.01 to 0.005), the L2 error is also divided by two. It can be concluded that the matrices assembling, the right hand side and the initial condition implementation are correct.
The adaptative mesh refinement procedure also worked as expected by reducing the CPU time (cf. table 7.2.3) almost three times without inducing a significant augmentation of the L2 error (table7.2.3).

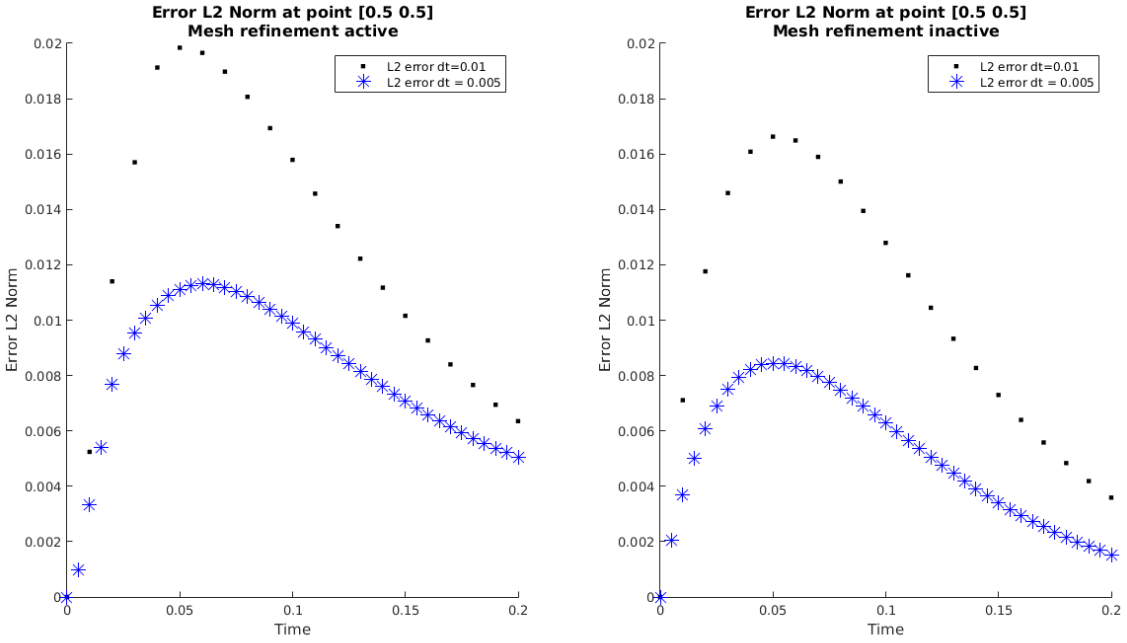


Figure 7.4: Test case 2 L2-errors

Test case number	CPU time without AMR	CPU time with AMR	% Reduction
1	28.1s	10.3s	273%
2	27.8s	9.7s	289%

Table 7.2: CPU time values and reductions

Test case number	L2 error without AMR	L2 error with AMR
1	0.34	0.36
2	0.0198	0.0166

Table 7.3: L2 Errors

7.3 Results Examples

After the code has been validated in simplified situation, it has been run in cases using all the features provided by the code.

7.3.1 Parameters

The values used for the parameters are close to the one of an aluminium. They are summed up in the table 7.3.1.

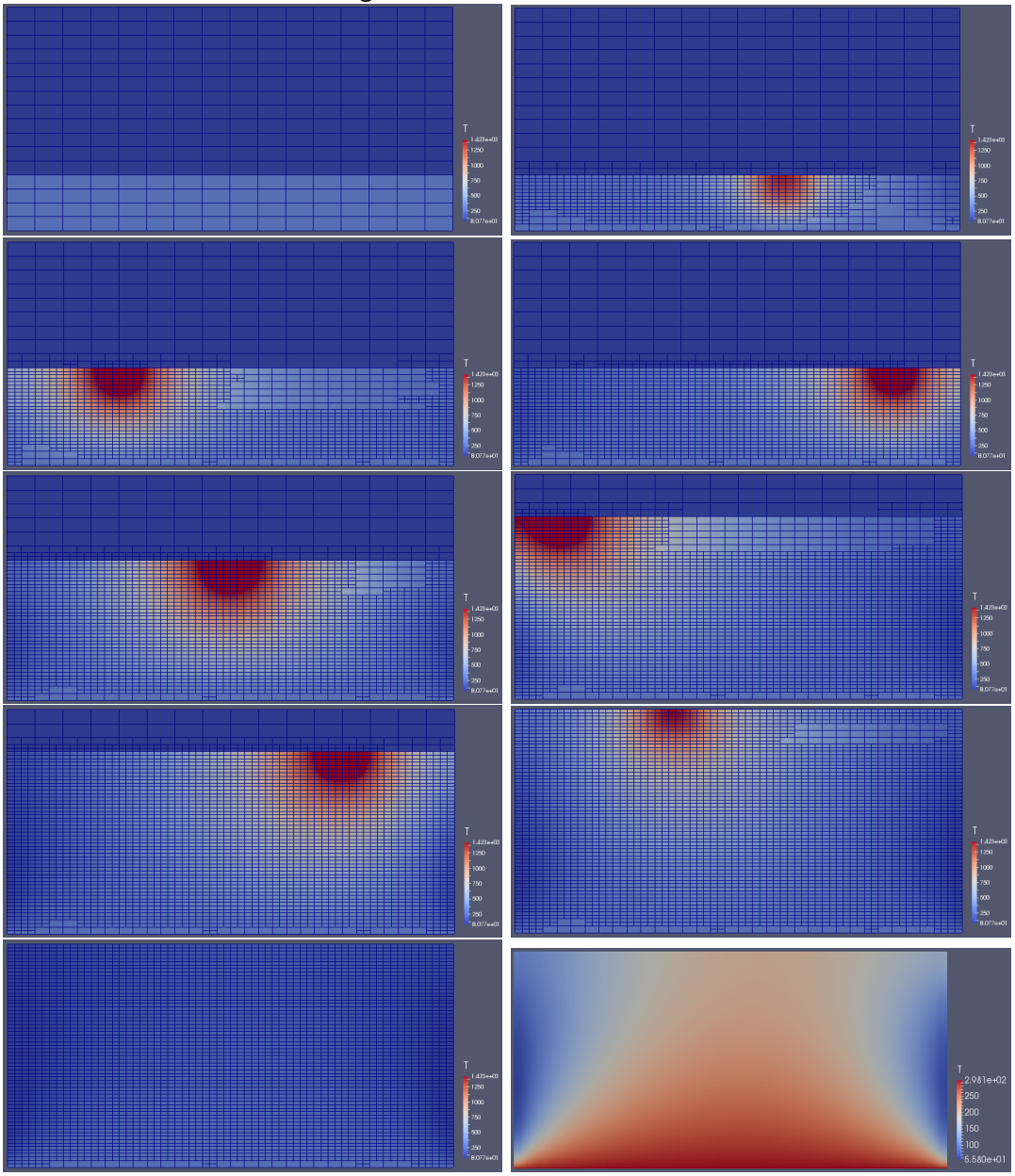
7.3.2 Results visualization

The code outputs have a .vtk extension. The software Paraview was used to visualize them. The temperature field obtained from the simulation are illustrated in the figure 7.3.2. The first nine images corresponds to regularly spaced screenshots of the simulation, the first being t=0 and the ninth t=1. The last image is also at t=1 but has a different temperature scale so that the boundary condition implementation is clearer.

Parameter	Value
Time step	0.01
Final time	1
Edge length	1
Number of layer	5
Laser speed	5
Heat conductivity	204
Heat capacity	2.43
Convection coefficient	100

Table 7.4: Simulation parameters

Figure 7.5: Simulation results



The temperature field has a shape in accordance with the expected behavior : a near circular distribution of the heat around the laser input center. The displacement of the heat input, and the domain evolution also correspond to the expectations. The last image illustrates quite well the fixed temperature at t=0 and the convection boundary conditions on the free surfaces. It can also be noticed that the lack of convection on the top surface of the part (inherent to the element death and birth method) except on the last one overestimate the temperature. Indeed it is lower on the eighth image that on the previous layer, which has no physical significance. This issue should be addressed in further development of the code. Finally the range of temperature simulated is in accordance with published data of similar models [2]. More precise comparison cannot be performed since the present model is way less complex and accurate than the one cited. It is nevertheless encouraging as preliminary results before future improvements of the current code.

Chapter 8

Conclusion and Future Work

The objectives of the present project were :

- Solve the heat equation using the Finite Element method with a moving heat source
- Implement adaptative mesh refinement
- Implement a time-dependent domain

In order to finally model the additive manufacturing of metal parts. For this purpose, a code was developed using the open source c++ library deal.II. Using simplified cases for which an analytical solution is available, the code was tested and validated. Then some preliminary simulations were performed aiming at showing the type of results the program can deliver. The validation procedure showed good agreement between the simulated solution and the analytical one. Moreover, the convergence rates were in accordance with the theory. Thusly the model implementation was deemed correct. The simulation of the building of several layer using a moving heat source provided sensible results and a behavior in accordance with the expectations : the temperatures were within the range of validated model results [2] and the melt pool shape is correct considering the assumptions that were made within the scope of this project.
However the current model neglect several physical phenomena of major importance in the additive manufacturing process and are to be included in future developments of the codes. Among them are the material properties dependence with the temperature and the phase change influence on the thermal field, which will be developed in the next version of the code.

References

[1] W. Bangerth et al. deal.ii library website, 2015.

[2] N. E. Hodge, R. M. Ferencz, and J. M. Solberg. Implementation of a thermomechanical model for the simulation of selective laser melting. *Computational Mechanics*, 54(1):33–51, 2014. Times Cited:1 Cited References Count:35.

[3] D. W. Kelly, J. P. De S. R. Gago, O. C. Zienkiewicz, and I. Babuška. A posteriori error analysis and adaptive processes in the finite element method: Part I–error analysis. *Int. J. Num. Meth. Engrg.*, 19:1593–1619, 1983.

[4] A. Quarteroni. *Numerical Models for Differential Problems*, volume 8 of *MS&A - Modeling, Simulation and Applications*. Springer Milan, 2014.