Final Report on the Project

Finite Element Simulation of Heat Transport With Applications to Metallic Powder Bed Adhesive Manufacturing Processes.

Introduction

Laser powder bed fusion (LPBF) is a widely used additive manufacturing process in which a high-powered laser selectively melts a powder bed to create a 3D object layer by layer. The LPBF process involves the precise control of several parameters, including the laser power, scanning speed, layer thickness, and powder bed composition, among others. The resulting properties of the final product, such as the porosity, surface roughness, and mechanical strength, are influenced by these parameters.

To optimize the LPBF process and produce high-quality parts, it is necessary to understand the complex physical phenomena that occur during the process. Modelling has emerged as an important tool for this purpose, allowing researchers and engineers to simulate the LPBF process and predict the properties of the final product.

Modelling of the LPBF process involves the use of mathematical and computational models to simulate the behavior of the laser, the powder bed, and the molten metal during the process. These models consider a range of physical phenomena, such as heat transfer, fluid flow, and phase change, to accurately predict the behavior of the system.

In this report, a finite element model for simulating single track experiment with a moving Gaussian laser beam. The model solves the well-known heat equation as presented in the next section. The model is also modified with the "Adhesion Model". This model calculates the effective material properties as a function based on the temperature at each node. A C++ code is written to solve the heat equation in a 2D domain during the laser melt process of a single-track experiment. In the next section, a brief introduction about the theory is discussed and the different main parts that were added to the code such as the heat input and the adhesion model. Finally, the last section describes the code in detail.

Theory

The program implements the heat equation:

$$\rho c_p \frac{\partial}{\partial t} u - \nabla.(k \nabla u) = Q_s(t) \qquad \forall x \in \Omega, t \in (0, T)$$

$$u(x, 0) = u_0(x) = 298 \qquad \forall x \in \Omega$$

$$u(x, t) = g_0(x, t) = 298 \qquad \forall x \in \partial\Omega, t \in (0, T)$$

To discretize the equations above using the theta scheme that discretize the equation in time using the following approach:

$$\rho c_p \frac{u^n(\mathbf{x}) - u^{n-1}(\mathbf{x})}{k_n} - k[(1-\theta)\Delta u^{n-1}(\mathbf{x})] = [(1-\theta)Q_n(\mathbf{x}, t_{n-1}) + \theta Q_n(\mathbf{x}, t_n)]$$

Where $k_n = t_n - t_{n-1}$ is the step size, ρ is the density at liquidus temperature, c_p is the specific heat capacity at liquidus temperature, k is the thermal conductivity, and θ represents the time discretization scheme, for example, when $\theta = 0$ it is explicit Euler, $\theta = 1$ it is implicit Euler, while $\theta = \frac{1}{2}$ it is explicit Crank-Nicolson time discretization. In this project, $\theta = \frac{1}{2}$ will be chosen since it has the highest convergence order.

The weak form of the PDE can be obtained multiplying by a test function then integrating by parts. The final weak form has the form:

$$MU^n - MU^{n-1} + k_n[(1-\theta)AU^{n-1} + \theta AU^n] = k_n[(1-\theta)Q_s^{n-1} + \theta F^n]$$

Where M is the mass matrix and A is the stiffness matrix that results from discretizing the Laplacian.

By rearranging the equation, so that all known quantities on the right side of the equation and all the unknowns are on right hand side of the equation we get:

$$(M + k_n\theta A)U^n = MU^{n-1} - k_n(1-\theta)AU^{n-1} + k_n[(1-\theta)Q_s^{n-1} + \theta F^n]$$

The linear system on the left-hand side is symmetric and positive definite so it can be solved with Conjugate Gradient Method.

Laser Heat Input

To calculate the heat supplied to the powder bed, a laser beam with known diameter is used. This beam is moving along the x-direction and has a Gaussian beam profile. The heat input can be calculated from the following equation.

$$Q_s = \frac{\alpha P}{\pi r^2}\exp\left(-\frac{x^2 + y^2}{2r^2}\right)$$

Where α is the absorptivity of the material to the laser energy, P is the laser power, r is the beam diameters, and x and y are the distance from the beam center to each point on the mesh. The position o the laser beam at each time step is calculated using the following equation:

$$x_0 = x_i + vt$$

Where x_i is the initial position of the laser beam, v is the scanning speed and t is the time.

Adhesion Model

In LPBF, the powder particles start to melt as the temperature increases above the solidus temperature, when the temperature reaches the liquidus temperature, full melting occurs. This melting is accompanied by a decrease in the void volume ratio (porosity) between the particles. The value of the porosity changes from the initial value to zero when full melting occurred. The value of the porosity as a function of temperature can be modelled using the following equation:

$$\varphi = \begin{cases} \varphi_i & T_0 < T < T_s \\ \dfrac{\varphi_i}{T_s - T_l}(T - T_l) & T_s < T < T_l \\ 0 & T > Tm \end{cases}$$

Where φ_i is the initial value of porosity, T_l is the liquidus temperature, and T_s is the solidus temperature of the alloyed powder. The effective material properties are calculated using the following equations:

$$\rho_{eff} = \rho(1 - \varphi)$$

$$k_{eff} = k(1 - \varphi)^n$$

The effective properties are calculated at each node by knowing the temperature at that node at each time step.

Commented Code

```
/* ---------------------------------------------------------------------
 *
 * Copyright (C) 2013 - 2021 by the deal.II authors
 *
 * This file is part of the deal.II library.
 *
 * The deal.II library is free software; you can use it, redistribute
 * it, and/or modify it under the terms of the GNU Lesser General
 * Public License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 * The full text of the license can be found in the file LICENSE.md at
 * the top level directory of deal.II.
 *
 * ---------------------------------------------------------------------

 *
 * Author: Wolfgang Bangerth, Texas A&M University, 2013
 */


// The program starts with the usual include files, all of which you should
// have seen before by now:
#include <deal.II/base/utilities.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/function.h>
#include <deal.II/base/logstream.h>
#include <deal.II/lac/vector.h>
#include <deal.II/lac/full_matrix.h>
#include <deal.II/lac/dynamic_sparsity_pattern.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/lac/solver_cg.h>
```

```
#include <deal.II/lac/precondition.h>
#include <deal.II/lac/affine_constraints.h>
#include <deal.II/grid/tria.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/grid_refinement.h>
#include <deal.II/grid/grid_out.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/fe/fe_q.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/vector_tools.h>
#include <deal.II/numerics/error_estimator.h>
#include <deal.II/numerics/solution_transfer.h>
#include <deal.II/numerics/matrix_tools.h>

#include <fstream>
#include <iostream>


// Then the usual placing of all content of this program into a namespace and
// the importation of the deal.II namespace into the one we will work in:
namespace Step26
{
  using namespace dealii;


  // @sect3{The <code>HeatEquation</code> class}
  //
  // The next piece is the declaration of the main class of this program. It
  // follows the well trodden path of previous examples. If you have looked at
  // step-6, for example, the only thing worth noting here is that we need to
  // build two matrices (the mass and Laplace matrix) and keep the current and
  // previous time step's solution. We then also need to store the current
  // time, the size of the time step, and the number of the current time
  // step. The last of the member variables denotes the theta parameter
  // discussed in the introduction that allows us to treat the explicit and
  // implicit Euler methods as well as the Crank-Nicolson method and other
  // generalizations all in one program.
  //
  // As far as member functions are concerned, the only possible surprise is
  // that the <code>refine_mesh</code> function takes arguments for the
  // minimal and maximal mesh refinement level. The purpose of this is
  // discussed in the introduction.
  template <int dim>
```

```
class HeatEquation
{
public:
  HeatEquation();
  void run();

private:
  void setup_system();
  void solve_time_step();
  void output_results() const;
  void refine_mesh(const unsigned int min_grid_level,
                   const unsigned int max_grid_level);
  void initialize_adhesion_model();
  void update_adhesion_model();

  FullMatrix<double> cell_mass();
  Vector<double> cell_rhs(double heat_input);

  Triangulation<dim> triangulation;
  FE_Q<dim>          fe;
  DoFHandler<dim>    dof_handler;

  AffineConstraints<double> constraints;

  SparsityPattern      sparsity_pattern;
  SparseMatrix<double> mass_matrix;
  SparseMatrix<double> laplace_matrix;
  SparseMatrix<double> system_matrix;

  Vector<double> solution;
  Vector<double> old_solution;
  Vector<double> system_rhs;
  Vector<double> Theta_model;
  Vector<double> k_modified;



  double       time;
  double       time_step;
  unsigned int timestep_number;



  const double theta;
```

```
  };



  // @sect3{Equation data}

  // In the following classes and functions, we implement the various pieces
  // of data that define this problem (right hand side and boundary values)
  // that are used in this program and for which we need function objects. The
  // right hand side represnet the heat input for a moving heat source with
Gaussian
  // distrution. This is calculated using the formula in the Laser Heat Input
section.
  // The position of the laser beam is updated at each time step, then the
eculedian
  // distance between the center of the beam at each time step and each node on
the
  // mesh is calculated and used in the equation.
  template <int dim>
  class RightHandSide : public Function<dim>
  {
  public:
    RightHandSide()
      : Function<dim>()
      , period(0.2)
    {}

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

  private:
    const double period;
  };



  template <int dim>
  double RightHandSide<dim>::value(const Point<dim> & point,
                                   const unsigned int component) const
  {
    (void)component;
    AssertIndexRange(component, 1);
    Assert(dim == 2, ExcNotImplemented());
    const Point<dim> beam_initial_position(0.2e-3, 0.5e-3);
    const Tensor<1, dim> beam_velocity_vector{{2.0, 0.0}};
```

```
  /* Compute current position: */
  const double time = this->get_time();
  const Point<dim> beam_position =
      beam_initial_position + time * beam_velocity_vector;

  const double distance = point.distance(beam_position);
  const double beam_radius = 50e-7;
  const double laser_power = 250.0;
  const double absorptivity = 0.5;

  /* Compute Gaussian: */
  const double laser_beam_radius = beam_radius * std::sqrt(laser_power);
  const double gaussian =
      /*std::exp((1, 2));*/
     std::exp(-std::pow(distance / laser_beam_radius, 2)/2);

  const double heat_input =  absorptivity * laser_power * gaussian/
                            (M_PI *  laser_beam_radius * laser_beam_radius);

  return 20000. * heat_input;
}


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



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

```
// @sect3{The <code>HeatEquation</code> implementation}
//
// It is time now for the implementation of the main class. Let's
// start with the constructor which selects a linear element, a time
// step constant at 1/500 (remember that one period of the source
// on the right hand side was set to 0.2 above, so we resolve each
// period with 100 time steps) and chooses the Crank Nicolson method
// by setting $\theta=1/2$.
template <int dim>
HeatEquation<dim>::HeatEquation()
  : fe(1)
  , dof_handler(triangulation)
  , time_step(0.001 / 100)
  , theta(0.5)
{}



// @sect4{<code>HeatEquation::setup_system</code>}
//
// The next function is the one that sets up the DoFHandler object,
// computes the constraints, and sets the linear algebra objects
// to their correct sizes. We also compute the mass and Laplace
// matrix here by simply calling two functions in the library.
//
// Note that we do not take the hanging node constraints into account when
// assembling the matrices (both functions have an AffineConstraints argument
// that defaults to an empty object). This is because we are going to
// condense the constraints in run() after combining the matrices for the
// current time-step.
//
// In the last section of this function, the mass matrix and the laplacian
// matrix are scaled with the values of the material properties. This current
// version of the code assums a constant material properties.
template <int dim>
void HeatEquation<dim>::setup_system()
{
  dof_handler.distribute_dofs(fe);

  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;

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

    DynamicSparsityPattern dsp(dof_handler.n_dofs());
    DoFTools::make_sparsity_pattern(dof_handler,
                                    dsp,
                                    constraints,
                                    /*keep_constrained_dofs = */ true);
    sparsity_pattern.copy_from(dsp);

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

    MatrixCreator::create_mass_matrix(dof_handler,
                                      QGauss<dim>(fe.degree + 1),
                                      mass_matrix);
    MatrixCreator::create_laplace_matrix(dof_handler,
                                         QGauss<dim>(fe.degree + 1),
                                         laplace_matrix);

    solution.reinit(dof_handler.n_dofs());
    old_solution.reinit(dof_handler.n_dofs());
    system_rhs.reinit(dof_handler.n_dofs());
    Theta_model.reinit(dof_handler.n_dofs());

    const double rho = 19300;
    const double thermal_conductivity = 170;
    const double heat_capacity = 133;

    mass_matrix *= rho * heat_capacity;
    laplace_matrix *= thermal_conductivity;



  }

// This function intializes the variable Theta_model which is used in the
// adhesion model present in the report. the initial value of Theta_model
// has a value of 0.64 which is used for the unimodel particle model.
  template <int dim>
  void HeatEquation<dim>::initialize_adhesion_model() {
```

```
    const double Theta_i = 0.64;
    for (unsigned int i = 0; i < old_solution.size(); ++i) {
      Theta_model[i] = Theta_i;
    }
  }

// This function updates the "Theta_model" variable, which represents the
// adhesion of the material. It uses a hysteresis model to calculate the
// new value of "Theta_model" at each node based on the current temperature
// solution vector "solution". If the temperature is above a threshold "Tl",
// "Theta_model" is set to 0. If the temperature is below another threshold
// "Ts", "Theta_model" is set to a constant value "Theta_i". Otherwise,
// "Theta_model" is set to a linearly decreasing function of the temperature
// between "Tl" and "Ts". The new value of "Theta_model" is then calculated
// using a hysteresis model, where it is set to the minimum of the previous
// value and the new value. Finally, the constraints are applied to the updated
// "Theta_model" variable.

  template <int dim>
  void HeatEquation<dim>::update_adhesion_model() {
    const double Theta_i = 0.640;
    const double Ts = 1000;
    const double Tl = 2000;

    // Loop over the solution vector and calculate theta at each node
    for (unsigned int i = 0; i < solution.size(); ++i) {
      double new_theta = 0.;
      if (solution[i] >= Tl) {
        new_theta = 0.;
      } else if (solution[i] <= Ts) {
        new_theta = Theta_i;
      } else {
        double Theta_T = (Theta_i / (Ts - Tl)) * (solution[i] - Tl);
        new_theta = Theta_T;
      }

      // Hysteresis model:
      const double old_theta = Theta_model[i];
      Theta_model[i] = std::min(old_theta, new_theta);
    }

    constraints.distribute(Theta_model);
  }
```

```
// @sect4{<code>HeatEquation::solve_time_step</code>}
//
// The next function is the one that solves the actual linear system
// for a single time step. There is nothing surprising here:
template <int dim>
void HeatEquation<dim>::solve_time_step()
{
  SolverControl            solver_control(1000, 1e-8 * system_rhs.l2_norm());
  SolverCG<Vector<double>> cg(solver_control);

  PreconditionSSOR<SparseMatrix<double>> preconditioner;
  preconditioner.initialize(system_matrix, 1.0);

  cg.solve(system_matrix, solution, system_rhs, preconditioner);

  constraints.distribute(solution);

  std::cout << "     " << solver_control.last_step() << " CG iterations."
            << std::endl;
}



// @sect4{<code>HeatEquation::output_results</code>}
//
// Neither is there anything new in generating graphical output other than the
// fact that we tell the DataOut object what the current time and time step
// number is, so that this can be written into the output file:
template <int dim>
void HeatEquation<dim>::output_results() const
{
  DataOut<dim> data_out;

  data_out.attach_dof_handler(dof_handler);
  data_out.add_data_vector(solution, "U");
  data_out.add_data_vector(system_rhs, "heat_input");
  data_out.add_data_vector(Theta_model, "Theta");
  data_out.build_patches();

  data_out.set_flags(DataOutBase::VtkFlags(time, timestep_number));

  const std::string filename =
    "solution-" + Utilities::int_to_string(timestep_number, 3) + ".vtk";
  std::ofstream output(filename);
  data_out.write_vtk(output);
```

```
}


// @sect4{<code>HeatEquation::refine_mesh</code>}
//
// This function is the interesting part of the program. It takes care of
// the adaptive mesh refinement. The three tasks
// this function performs is to first find out which cells to
// refine/coarsen, then to actually do the refinement and eventually
// transfer the solution vectors between the two different grids. The first
// task is simply achieved by using the well-established Kelly error
// estimator on the solution. The second task is to actually do the
// remeshing. That involves only basic functions as well, such as the
// <code>refine_and_coarsen_fixed_fraction</code> that refines those cells
// with the largest estimated error that together make up 60 per cent of the
// error, and coarsens those cells with the smallest error that make up for
// a combined 40 per cent of the error. Note that for problems such as the
// current one where the areas where something is going on are shifting
// around, we want to aggressively coarsen so that we can move cells
// around to where it is necessary.
//
// As already discussed in the introduction, too small a mesh leads to
// too small a time step, whereas too large a mesh leads to too little
// resolution. Consequently, after the first two steps, we have two
// loops that limit refinement and coarsening to an allowable range of
// cells:
template <int dim>
void HeatEquation<dim>::refine_mesh(const unsigned int min_grid_level,
                                    const unsigned int max_grid_level)
{
  Vector<float> estimated_error_per_cell(triangulation.n_active_cells());

  KellyErrorEstimator<dim>::estimate(
    dof_handler,
    QGauss<dim - 1>(fe.degree + 1),
    std::map<types::boundary_id, const Function<dim> *>(),
    solution,
    estimated_error_per_cell);

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

  if (triangulation.n_levels() > max_grid_level)
```

```
  for (const auto &cell :
       triangulation.active_cell_iterators_on_level(max_grid_level))
    cell->clear_refine_flag();
for (const auto &cell :
     triangulation.active_cell_iterators_on_level(min_grid_level))
  cell->clear_coarsen_flag();
// These two loops above are slightly different but this is easily
// explained. In the first loop, instead of calling
// <code>triangulation.end()</code> we may as well have called
// <code>triangulation.end_active(max_grid_level)</code>. The two
// calls should yield the same iterator since iterators are sorted
// by level and there should not be any cells on levels higher than
// on level <code>max_grid_level</code>. In fact, this very piece
// of code makes sure that this is the case.

// As part of mesh refinement we need to transfer the solution vectors
// from the old mesh to the new one. To this end we use the
// SolutionTransfer class and we have to prepare the solution vectors that
// should be transferred to the new grid (we will lose the old grid once
// we have done the refinement so the transfer has to happen concurrently
// with refinement). At the point where we call this function, we will
// have just computed the solution, so we no longer need the old_solution
// variable (it will be overwritten by the solution just after the mesh
// may have been refined, i.e., at the end of the time step; see below).
// In other words, we only need the one solution vector, and we copy it
// to a temporary object where it is safe from being reset when we further
// down below call <code>setup_system()</code>.
//
// Consequently, we initialize a SolutionTransfer object by attaching
// it to the old DoF handler. We then prepare the triangulation and the
// data vector for refinement (in this order).
SolutionTransfer<dim> solution_trans(dof_handler);
SolutionTransfer<dim> solution_trans_theta(dof_handler);

Vector<double> previous_solution;
Vector<double> previous_theta;
previous_solution = solution;
previous_theta = Theta_model;

triangulation.prepare_coarsening_and_refinement();
solution_trans.prepare_for_coarsening_and_refinement(previous_solution);
solution_trans_theta.prepare_for_coarsening_and_refinement(previous_theta);

// Now everything is ready, so do the refinement and recreate the DoF
// structure on the new grid, and finally initialize the matrix structures
```

```
    // and the new vectors in the <code>setup_system</code> function. Next, we
    // actually perform the interpolation of the solution from old to new
    // grid. The final step is to apply the hanging node constraints to the
    // solution vector, i.e., to make sure that the values of degrees of
    // freedom located on hanging nodes are so that the solution is
    // continuous. This is necessary since SolutionTransfer only operates on
    // cells locally, without regard to the neighborhood.
    triangulation.execute_coarsening_and_refinement();
    setup_system();

    solution_trans.interpolate(previous_solution, solution);
    solution_trans_theta.interpolate(previous_theta, Theta_model);
    constraints.distribute(solution);
    constraints.distribute(Theta_model);
  }

//new

  // @sect4{<code>HeatEquation::run</code>}
  //
  // This is the main driver of the program, where we loop over all
  // time steps. At the top of the function, we set the number of
  // initial global mesh refinements and the number of initial cycles of
  // adaptive mesh refinement by repeating the first time step a few
  // times. Then we create a mesh, initialize the various objects we will
  // work with, set a label for where we should start when re-running
  // the first time step, and interpolate the initial solution onto
  // out mesh (we choose the zero function here, which of course we could
  // do in a simpler way by just setting the solution vector to zero). We
  // also output the initial time step once.
  //
  // @note If you're an experienced programmer, you may be surprised
  // that we use a <code>goto</code> statement in this piece of code!
  // <code>goto</code> statements are not particularly well liked any
  // more since Edsgar Dijkstra, one of the greats of computer science,
  // wrote a letter in 1968 called "Go To Statement considered harmful"
  // (see <a href="http://en.wikipedia.org/wiki/Considered_harmful">here</a>).
  // The author of this code subscribes to this notion whole-heartedly:
  // <code>goto</code> is hard to understand. In fact, deal.II contains
  // virtually no occurrences: excluding code that was essentially
  // transcribed from books and not counting duplicated code pieces,
  // there are 3 locations in about 600,000 lines of code at the time
  // this note is written; we also use it in 4 tutorial programs, in
  // exactly the same context as here. Instead of trying to justify
  // the occurrence here, let's first look at the code and we'll come
```

```
  // back to the issue at the end of function.
  template <int dim>
  void HeatEquation<dim>::run()
  {
    const unsigned int initial_global_refinement       = 4;
    const unsigned int n_adaptive_pre_refinement_steps = 4;

    GridGenerator::hyper_rectangle(triangulation, Point<2>(0.0,0.0),
Point<2>(2.5e-3, 1.0e-3));
    triangulation.refine_global(initial_global_refinement);

    setup_system();

    unsigned int pre_refinement_step = 0;

    Vector<double> tmp;
    Vector<double> forcing_terms;

  start_time_iteration:

    time            = 0.0;
    timestep_number = 0;

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


    VectorTools::interpolate(dof_handler,
                             Functions::ZeroFunction<dim>(),
                             old_solution);
    solution = old_solution;
    initialize_adhesion_model();

    output_results();

    // Then we start the main loop until the computed time exceeds our
    // end time of 0.001. The first task is to build the right hand
    // side of the linear system we need to solve in each time step.
    // Recall that it contains the term $MU^{n-1}-(1-\theta)k_n AU^{n-1}$.
    // We put these terms into the variable system_rhs, with the
    // help of a temporary vector:
    while (time <= 0.001)
      {
        time += time_step;
        ++timestep_number;
```

```
      std::cout << "Time step " << timestep_number << " at t=" << time
                << std::endl;

      mass_matrix.vmult(system_rhs, old_solution);

      laplace_matrix.vmult(tmp, old_solution);
      system_rhs.add(-(1 - theta) * time_step, tmp);

      // The second piece is to compute the contributions of the source
      // terms. This corresponds to the term $k_n
      // \left[ (1-\theta)F^{n-1} + \theta F^n \right]$. The following
      // code calls VectorTools::create_right_hand_side to compute the
      // vectors $F$, where we set the time of the right hand side
      // (source) function before we evaluate it. The result of this
      // all ends up in the forcing_terms variable:
      RightHandSide<dim> rhs_function;
      rhs_function.set_time(time);
      VectorTools::create_right_hand_side(dof_handler,
                                          QGauss<dim>(fe.degree + 1),
                                          rhs_function,
                                          tmp);
      forcing_terms = tmp;
      forcing_terms *= time_step * theta;

      rhs_function.set_time(time - time_step);
      VectorTools::create_right_hand_side(dof_handler,
                                          QGauss<dim>(fe.degree + 1),
                                          rhs_function,
                                          tmp);

      forcing_terms.add(time_step * (1 - theta), tmp);

      // Next, we add the forcing terms to the ones that
      // come from the time stepping, and also build the matrix
      // $M+k_n\theta A$ that we have to invert in each time step.
      // The final piece of these operations is to eliminate
      // hanging node constrained degrees of freedom from the
      // linear system:
      system_rhs += forcing_terms;

      system_matrix.copy_from(mass_matrix);
      system_matrix.add(theta * time_step, laplace_matrix);

      constraints.condense(system_matrix, system_rhs);
```

```
// There is one more operation we need to do before we
// can solve it: boundary values. To this end, we create
// a boundary value object, set the proper time to the one
// of the current time step, and evaluate it as we have
// done many times before. The result is used to also
// set the correct boundary values in the linear system:
{
  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,
                                           0,
                                           boundary_values_function,
                                           boundary_values);

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

// With this out of the way, all we have to do is solve the
// system, generate graphical data, and...
solve_time_step();
update_adhesion_model();
output_results();

// ...take care of mesh refinement. Here, what we want to do is
// (i) refine the requested number of times at the very beginning
// of the solution procedure, after which we jump to the top to
// restart the time iteration, (ii) refine every fifth time
// step after that.
//
// The time loop and, indeed, the main part of the program ends
// with starting into the next time step by setting old_solution
// to the solution we have just computed.
if ((timestep_number == 1) &&
    (pre_refinement_step < n_adaptive_pre_refinement_steps))
  {
    refine_mesh(initial_global_refinement,
                initial_global_refinement +
                  n_adaptive_pre_refinement_steps);
    ++pre_refinement_step;
```

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

            std::cout << std::endl;

            goto start_time_iteration;
          }
        else if ((timestep_number > 0) && (timestep_number % 5 == 0))
          {
            refine_mesh(initial_global_refinement,
                        initial_global_refinement +
                          n_adaptive_pre_refinement_steps);
            tmp.reinit(solution.size());
            forcing_terms.reinit(solution.size());
          }

        old_solution = solution;
      }
  }
} // namespace Step26
// Now that you have seen what the function does, let us come back to the issue
// of the <code>goto</code>. In essence, what the code does is
// something like this:
// @code
//   void run ()
//   {
//     initialize;
//   start_time_iteration:
//     for (timestep=1...)
//     {
//        solve timestep;
//        if (timestep==1 && not happy with the result)
//        {
//          adjust some data structures;
//          goto start_time_iteration; // simply try again
//        }
//        postprocess;
//     }
//   }
// @endcode
// Here, the condition "happy with the result" is whether we'd like to keep
// the current mesh or would rather refine the mesh and start over on the
// new mesh. We could of course replace the use of the <code>goto</code>
// by the following:
```

```
// @code
//   void run ()
//   {
//     initialize;
//     while (true)
//     {
//        solve timestep;
//        if (not happy with the result)
//           adjust some data structures;
//        else
//           break;
//     }
//     postprocess;
//
//     for (timestep=2...)
//     {
//        solve timestep;
//        postprocess;
//     }
//   }
// @endcode
// This has the advantage of getting rid of the <code>goto</code>
// but the disadvantage of having to duplicate the code that implements
// the "solve timestep" and "postprocess" operations in two different
// places. This could be countered by putting these parts of the code
// (sizable chunks in the actual implementation above) into their
// own functions, but a <code>while(true)</code> loop with a
// <code>break</code> statement is not really all that much easier
// to read or understand than a <code>goto</code>.
//
// In the end, one might simply agree that <i>in general</i>
// <code>goto</code> statements are a bad idea but be pragmatic and
// state that there may be occasions where they can help avoid code
// duplication and awkward control flow. This may be one of these
// places, and it matches the position Steve McConnell takes in his
// excellent book "Code Complete" @cite CodeComplete about good
// programming practices (see the mention of this book in the
// introduction of step-1) that spends a surprising ten pages on the
// question of <code>goto</code> in general.


// @sect3{The <code>main</code> function}
//
// Having made it this far,  there is, again, nothing
// much to discuss for the main function of this
```

```
// program: it looks like all such functions since step-6.
int main()
{
  try
    {
      using namespace Step26;

      HeatEquation<2> heat_equation_solver;
      heat_equation_solver.run();
    }
  catch (std::exception &exc)
    {
      std::cerr << std::endl
                << std::endl
                << "----------------------------------------------------"
                << std::endl;
      std::cerr << "Exception on processing: " << std::endl
                << exc.what() << std::endl
                << "Aborting!" << std::endl
                << "----------------------------------------------------"
                << std::endl;

      return 1;
    }
  catch (...)
    {
      std::cerr << std::endl
                << std::endl
                << "----------------------------------------------------"
                << std::endl;
      std::cerr << "Unknown exception!" << std::endl
                << "Aborting!" << std::endl
                << "----------------------------------------------------"
                << std::endl;
      return 1;
    }

  return 0;
}
```