C++ parallel implementation of incompressible viscous time-dependent NSE simulation using projection method in Dealii

Muhammad Mohebujjaman *

MTHSC9830 Spring 2015 Project

Abstract

In this project, we implement the projection method algorithm in dealii [1] with multithreading for finding solution of incompressible viscous time-dependent Navier-Stokes equations. This work is an extension of dealii example step-35. Calculating right hand side function for a true solution from Navier-Stokes equation, we implement the right side function and known boundary value function into our code. We checked the convergence rate with respect to different norms for velocity and pressure for the algorithm. Finally we plot some graphs for velocity and pressure at different times with some specified values of the parameter.

1 Introduction

We know that in numerical simulation of time-dependent viscous incompressible flows the velocity and pressure are coupled by the incompressibility constraint. This is one of the major difficulty in solving time-dependent Navier-Stokes equations [3]. To overcome this difficulty in time-dependent problem, the interest of using projection methods started in the late 1960s. This method was originally introduced by Alexandre Chorin [2] as an efficient means of solving time-dependent Navier-Stokes equations. The key advantage feature of projection methods is that, at each time step, one only needs to solve a sequence of decoupled elliptic equations for the velocity and the pressure, making it very efficient for large scale numerical simulations. The equations describe the flow of an unsteady viscous incompressible fluid are given by

$$\begin{aligned} u_t + u\cdot\nabla u - \nu\Delta u + \nabla p &= f \text{ in } \Omega\times[0,T] && (1.1)\\ \nabla\cdot u &= 0 \text{ in } \Omega\times[0,T] && (1.2)\\ u\big|_{t=0} &= u_0 \text{ in } \Omega, && (1.3)\\ u\big|_{\partial\Omega} &= u_b \text{ in } [0,T] && (1.4) \end{aligned}$$

Where u represents the velocity of the flow, p is the pressure, f is a smooth forcing term, ν is the kinematic viscosity, Ω is the domain, u_0 is an initial velocity field. For the time-dependent case, after the time discretization, we would arrive a system like

$$\begin{aligned} \frac{1}{\Delta t}u^k - \nu\Delta u^k + \nabla p^k &= F^k\\ \nabla\cdot u &= 0 \end{aligned}$$

*Department of Mathematical Sciences, Clemson University, Clemson, SC 29634; mmohebu@clemson.edu

where Δt is the time-step. This could be solved using a Schur complement approach as the structure of this system is pretty similar to the Stokes system. But the condition number of the Schur complement is proportional to $(\Delta t)^{-2}$. Therefore, for small Δt, the condition number becomes very high and this makes the system very difficult to solve and so for the time-dependent Navier-Stokes equations, Schur complement approach is not a useful avenue to the solution.

2 Notations and Preliminaries

We consider the time-dependent Navier-Stokes equations on a finite time interval $[0, T]$ and in an open, connected, bounded Lipschitz domain $\Omega \subset \mathbb{R}^d, d \in 2, 3$. We denote the usual $L^2(\Omega)$ norm and its inner product by $\|.\|$ and $(\cdot, \cdot)$ respectively. The $L^p(\Omega)$ norms and the Sobolev $W_p^k(\Omega)$ norms are denoted by $\|.\|_{L^p}$ and $\|.\|_{W_p^k(\Omega)}$ respectively for $k \in \mathbb{N}$, $1 \leq p \leq \infty$. In particular, $H^k(\Omega)$ is used to represent the Sobolev space $W_2^k(\Omega)$. $\|.\|_k$ and $|.|_k$ denote the norm and the seminorm in $H^k(\Omega)$. For X being a normed function space in Ω, $L^p(0, T; X)$ is the space of all functions defined on $[0, T] \times \Omega$ for which the norm

$$\|u\|_{L^p(0,T;X)} = \left(\int_0^T \|u\|_X^p dx \right)^{1/p}, p \in [1, \infty)$$

is finite. For $p = \infty$, the usual modification is used in the definition of this space.

3 Projection Methods

As we have already discussed that the difficulty in the solution of the time-dependent Navier-Stokes equations comes from the fact that the velocity and the pressure are coupled through the incompressibility constraint

$$\nabla \cdot u = 0,$$

for which the pressure is the Lagrange multiplier. Projection methods aim at decoupling the constraint from the diffusion operator. The algorithm for the projection methods is as follows:

Step 1: Compute an intermediate velocity u^* by ignoring pressure term and incompressibility. Solve the time discretized equation.

Step 2: Perform a step which involves a Poisson equation for pressure, and is equivalent to a projection of u^* onto the div free space.

Note that we change the advection term $u \cdot \nabla u$ by its skew symmetric form $u \cdot \nabla u + \frac{1}{2}(\nabla \cdot u)u$ which is consistent with the continuous case as $\nabla \cdot u = 0$ but this may not be true for pointwise for the discrete solution. But this skew symmetric form is essential to have an unconditional stability of the time-stepping scheme. To avoid non-linearity in the advection term, we use the second order extrapolation u^* of u^{k+1}.

To obtain a fully discrete setting of the projection method, we multiply both sides of (1.1) from right by the test vector v and integrate by parts. That is, we have some steps as follows:

$$-\nu \int_\Omega \Delta u \cdot v \, d\Omega$$

For the Dirichlet boundary conditions on the whole boundary, we have

$$-\nu \int_\Omega \Delta u \cdot v \, d\Omega = -\nu \int_{\partial\Omega} \nabla u \cdot n \, v dS + \nu \int_\Omega \nabla u \cdot \nabla v d\Omega = \nu \int_\Omega \nabla u \cdot \nabla v d\Omega$$

In this formulation the velocity components are fully decouples.But for non-standard boundary conditions, we can use the following identity

$$\Delta u = \nabla\nabla \cdot u - \nabla \times \nabla \times u$$

Integrating by parts and taking into account the boundary conditions we have

$$-\nu \int_\Omega \Delta u \cdot v \; d\Omega = \nu \int_\Omega (\nabla \cdot u \nabla \cdot v + \nabla \times u \nabla \times v) \; d\Omega$$

This couples the components of the velocity vector. For pressure boundary condition, we need to write the following

$$\int_\Omega \nabla p \cdot v \; d\Omega = \int_{\partial\Omega} p \; v \cdot n \; dS - \int_\Omega p \nabla \cdot v \; d\Omega = - \int_\Omega p \nabla \cdot v \; d\Omega$$

Where the boundary integral equals zero due to the pressure boundary condition.

Now we will describe how the projection methods look like in a semi-discrete setting. Our aim is to obtain a sequence of velocities u^k and pressures p^k. We will also obtain a sequence ϕ^k of auxiliary variables. Suppose that from the initial conditions we have u^0, p^0 and setting $\phi^0 = 0$. Using a first order method we can find u^1, p^1 and setting $\phi^1 = p^1 - p^0$. Then the projection method for the governing equations (1.1)-(1.4) consists of the following steps:

Stpe 0: Extrapolation. Define

$$u^* = 2u^k - u^{k-1}, \quad p^\# = p^k + \frac{4}{3}\phi^k - \frac{1}{3}\phi^{k-1}$$

Stpe 1: Diffusion step. Find u^{k+1} solving the single linear equation

$$\frac{1}{2\Delta t}(3u^{k+1} - 4u^k + u^{k-1}) + u^* \cdot \nabla u^{k+1} + \frac{1}{2}(\nabla \cdot u^*)u^{k+1} - \nu\Delta u^{k+1} + \nabla p^\# = f^{k+1}$$

$$u^{k+1}\big|_{\partial\Omega} = u_b$$

Stpe 2: Projection. Find ϕ^{k+1} that solves

$$\Delta\phi^{k+1} = \frac{3}{2\tau}\nabla \cdot u^{k+1}, \quad \partial_n \phi^{k+1}\big|_{\partial\Omega} = 0$$

Stpe 3: Pressure correction. Here we have two options:

- Incremental Method in Standard Form

$$p^{k+1} = p^k + \phi^{k+1}$$

- Incremental Method in Rotational Form

$$p^{k+1} = p^k + \phi^{k+1} - \nu\nabla \cdot u^{k+1}$$

4 Numerical Experiments

In this section, we run numerical experiment in dealii [1] with multi-threading to test the proposed scheme. We will verify predicted convergence rates for velocity and pressure with respect to different norms. For all our simulations, we choose (Q_2, Q_1) elements.

4.1 Convergence rate verification

To exhibit the effectiveness of the projection method with BDF2 time stepping scheme, we compute convergence rates on a series of refined meshes and timesteps. We choose the test problem with solution

$$v = \begin{pmatrix} \cos y + (1+e^t)\sin y \\ \sin x + (1+e^t)\cos x \end{pmatrix}, \quad p = \sin(x+y)(1+e^t)$$

on a rectangular domain $\{(x,y) : 0 \le x \le 1, \ \ 0 \le y \le 1\}$ and the right hand side function f is calculated accordingly. To test the temporal convergence rate for velocity, we use a mesh width of $h = 1/64$, end time $T = 1$, and compute with varying timestep sizes. Errors and rates are shown in Table(1) with respect to $\|\phi\|_{2,2} := \|\phi\|_{L^2(0,T;L^2(\Omega)^d)}$, $\|\phi\|_{2,1} := \|\phi\|_{L^2(0,T;H^1(\Omega)^d)}$ and $\|\phi\|_{\infty,2} := \|\phi\|_{L^\infty(0,T;L^2(\Omega)^d)}$ norms. To test the spatial convergence rate for velocity solution, we select a small end time $T = 0.001$, and timestep $\Delta t = T/8$, and compute errors and their rates on successively refined meshes with respect to the same norms as before and shown in Table(2). To get pressure uniqueness, we used $p^{'} = p - p_{mean}$ and $p_h^{'} = p_h - p_{hmean}$ and to compute the temporal and spatial convergence rate for pressure solution, we choose end time $T = 1$ and compute the errors and theirs rates with varying timestep and mesh size together. The errors and rates are shown in Table(3).

Δt	$\|u-u_h\|_{2,2}$	Rate	$\|u-u_h\|_{2,1}$	Rate	$\|u-u_h\|_{\infty,2}$	Rate
$\frac{T}{2}$	0.122286		0.821992		0.172938	
$\frac{T}{4}$	0.0424948	1.5249	0.241278	1.7684	0.0679916	1.3468
$\frac{T}{8}$	0.0140192	1.5999	0.0784814	1.6203	0.0198712	1.7747
$\frac{T}{16}$	0.00391612	1.8399	0.0261534	1.5854	0.00597801	1.7329
$\frac{T}{32}$	0.00103406	1.9211	0.00778855	1.7476	0.00157841	1.9212
$\frac{T}{64}$	0.000266434	1.9565	0.00222171	1.8097	0.000409103	1.9479
$\frac{T}{128}$	6.77244e-05	1.9760	0.000625176	1.8293	0.000104284	1.9719
$\frac{T}{256}$	1.70838e-05	1.9870	0.000176967	1.8208	2.63378e-05	1.9853

Table 1: Convergence rates for $\nu = 0.1$, $T = 1.0$, $h = 1/64$ with (Q_2, Q_1) elements.

# Ref.	$\|u-u_h\|_{2,2}$	Rate	$\|u-u_h\|_{2,1}$	Rate	$\|u-u_h\|_{\infty,2}$	Rate
1	3.04545e-05		0.000395489		0.00103236	
2	3.99087e-06	2.9319	0.000101963	1.9556	0.000138586	2.8971
3	4.95461e-07	3.0099	2.59035e-05	1.9768	1.70667e-05	3.0215
4	6.23184e-08	2.9910	6.47765e-06	1.9996	2.14725e-06	2.9906
5	7.76263e-09	3.0050	1.61257e-06	2.0061	2.63869e-07	3.0246
6	1.00147e-09	2.9544	3.98647e-07	2.0162	3.53456e-08	2.9002

Table 2: Convergence rate for $\nu = 0.1$, $T = 0.001$, $\Delta t = T/8$, with (Q_2, Q_1) elements.

Δt	$\|p^{'} - p_h^{'}\|_{2,2}$	Rate	# of Ref.	$\|p^{'} - p_h^{'}\|_{2,2}$	Rate
$\frac{T}{2}$	0.113634		1	0.00200424	
$\frac{T}{4}$	0.0877048	0.3737	2	0.000521994	1.9410
$\frac{T}{8}$	0.0456035	0.9435	3	0.000125518	2.0561
$\frac{T}{16}$	0.0135359	1.7524	4	3.13784e-05	2.0001
$\frac{T}{32}$	0.00365159	1.8902	5	7.84443e-06	2.0000
$\frac{T}{64}$	0.000952086	1.9394	6	1.9611e-06	2.0000
$\frac{T}{128}$	0.000258669	1.8800	7	4.96725e-07	1.9811

Table 3: Convergence rates for for $\nu = 0.1$, $T = 1.0$ with (Q_2, Q_1) elements.

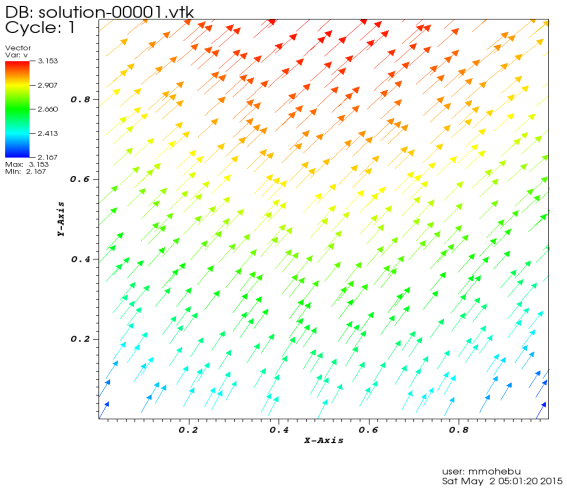

Figure 1: Velocity solution at $t = 0.0$, $\nu = 0.1$, $T = 0.001$, $\Delta t = 0.000125$, and $h = 1/64$.

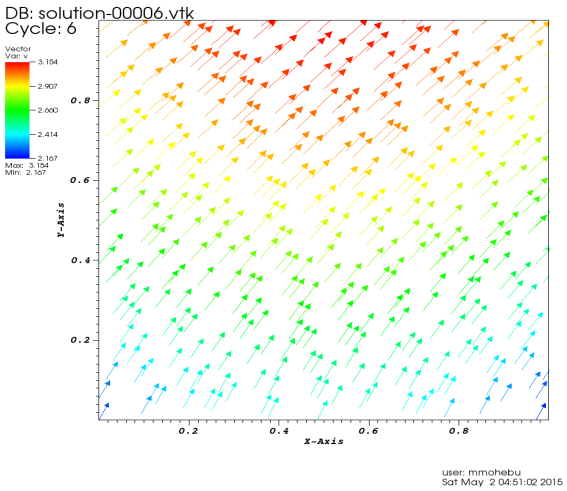

Figure 2: Velocity solution at $t = 0.00075$, $\nu = 0.1$, $T = 0.001$, $\Delta t = 0.000125$, and $h = 1/64$.

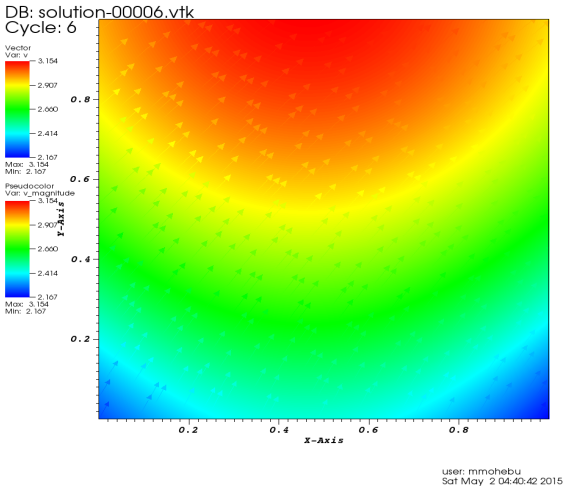

Figure 3: Velocity solution at $t = 0.00075$, $\nu = 0.1$, $T = 0.001$, $\Delta t = 0.000125$, and $h = 1/64$.

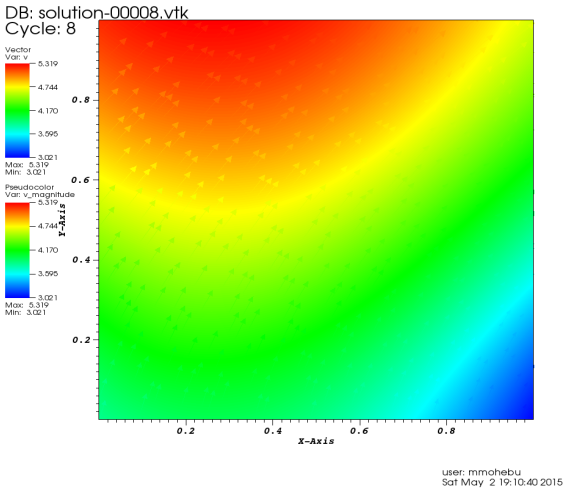

Figure 4: Speed contour plot at $t = 1.0$, $\nu = 0.1$, $T = 1$, $\Delta t = 0.125$, and $h = 1/64$.

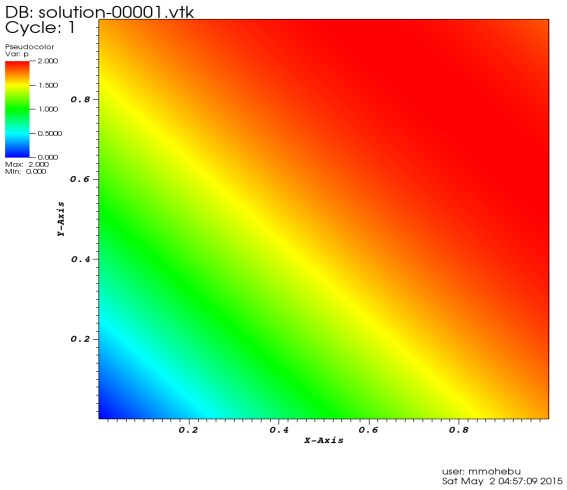

Figure 5: pressure solution at $t = 0.0$, $\nu = 0.1$, $T = 0.001$, $\Delta t = 0.000125$, and $h = 1/64$.

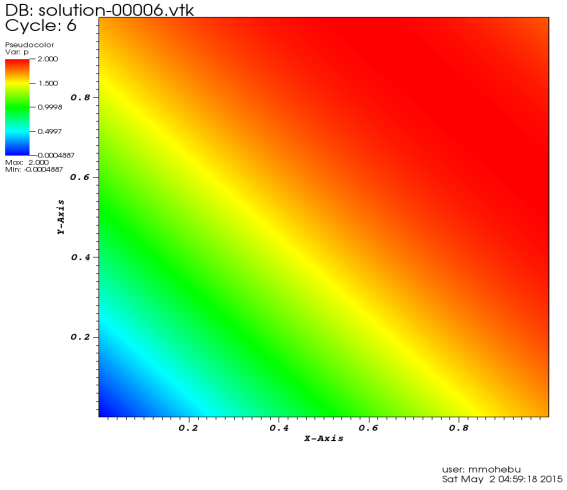

Figure 6: pressure solution at $t = 0.00075$, $\nu = 0.1$, $T = 0.001$, $\Delta t = 0.000125$, and $h = 1/64$.

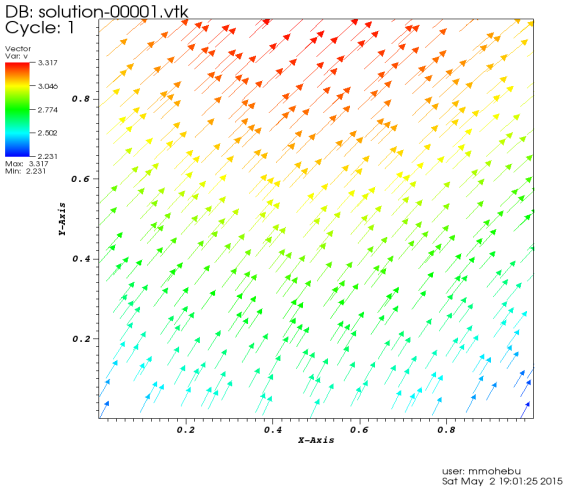

Figure 7: Velocity solution at $t = 0.0$, $\nu = 0.1$, $T = 1.0$, $\Delta t = 0.125$, and $h = 1/64$.

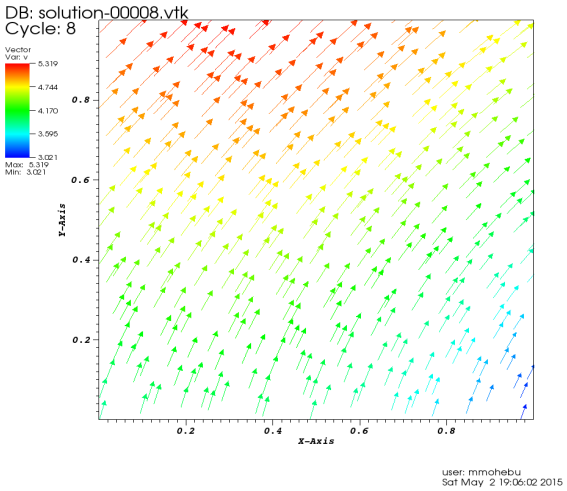

Figure 8: Velocity solution at $t = 1.0$, $\nu = 0.1$, $T = 1.0$, $\Delta t = 0.125$, and $h = 1/64$.

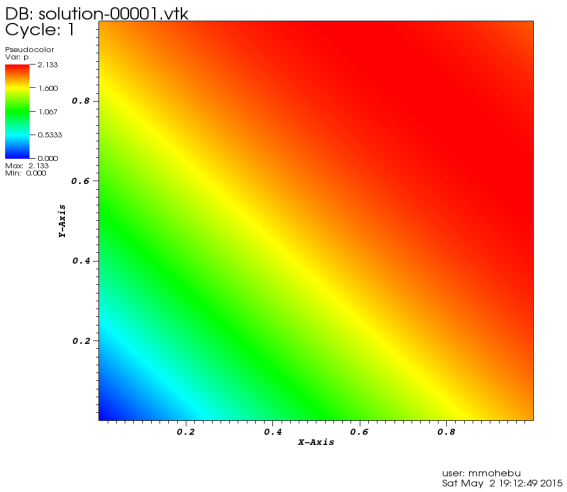

Figure 9: pressure solution at $t = 0.0$, $\nu = 0.1$, $T = 1$, $\Delta t = 0.125$, and $h = 1/64$.

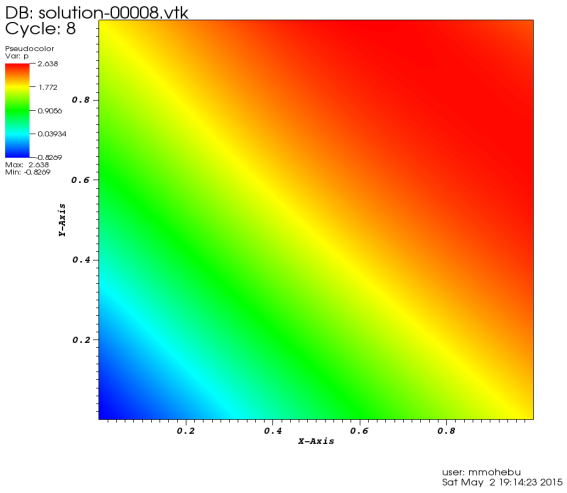

Figure 10: pressure solution at $t = 1.0$, $\nu = 0.1$, $T = 1$, $\Delta t = 0.125$, and $h = 1/64$.

5 Conclusion

The project was basically writing code for the projection method. We did not prove the convergence rate for the algorithm theoretically. The velocity convergence rates as varying time steps what we get from running our code are 2, 1.82 and 2 with respect to $\|\cdot\|_{L^2(0,T;L^2)}$, $\|\cdot\|_{L^2(0,T;H^1)}$ and $\|\cdot\|_{L^2(0,T;L^\infty)}$ norms respectively. Varying mesh widths, the found velocity convergence rates are 3, 2 and 3 with respect to $\|\cdot\|_{L^2(0,T;L^2)}$, $\|\cdot\|_{L^2(0,T;H^1)}$ and $\|\cdot\|_{L^2(0,T;L^\infty)}$ norms respectively. The pressure convergence rates 1.9 and 2 for varying time steps and mesh widths respectively. The obtained convergence rates for velocity and pressure seem reasonable with respect to the available theory in literature for the projection methods.

References

[1] W. Bangerth, D. Davydov, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, B. Turcksin, and D. Wells. The `deal.II` library, version 8.4. *Journal of Numerical Mathematics*, 24, 2016.

[2] A.J. Chorin. Numerical solution of the navier-stokes equations. *Mathematics of Computation*, 22:745–762, 1968.

[3] J.L. Guermond, P. Minev, and J. Shen. An overview of projection methods for incompressible flows. *Computer Methods in Applied Mechanics and Engineering*, 195:6011–6045, 2006.

6 Appendix

```
/* ---------------------------------------------------------------------
 *
 * Copyright (C) 2009 - 2013 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 at
 * the top level of the deal.II distribution.
 *
 * ---------------------------------------------------------------------

 *
 * Author: Abner Salgado, Texas A&M University 2009
 * Extended by Muhammad Mohebujjaman
 */


// @sect3{Include files}

// We start by including all the necessary deal.II header files and some C++
// related ones. Each one of them has been discussed in previous tutorial
```

```
// programs, so we will not get into details here.
#include <deal.II/base/parameter_handler.h>
#include <deal.II/base/point.h>
#include <deal.II/base/function.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/multithread_info.h>
#include <deal.II/base/thread_management.h>
#include <deal.II/base/work_stream.h>
#include <deal.II/base/parallel.h>
#include <deal.II/base/utilities.h>
#include <deal.II/base/conditional_ostream.h>

#include <deal.II/lac/vector.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/solver_gmres.h>
#include <deal.II/lac/sparse_ilu.h>
#include <deal.II/lac/sparse_direct.h>
#include <deal.II/lac/constraint_matrix.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/tria_accessor.h>
#include <deal.II/grid/tria_iterator.h>
#include <deal.II/grid/tria_boundary_lib.h>
#include <deal.II/grid/grid_in.h>

#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_accessor.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/dofs/dof_renumbering.h>

#include <deal.II/fe/fe_q.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/fe/fe_tools.h>
#include <deal.II/fe/fe_system.h>

#include <deal.II/numerics/matrix_tools.h>
#include <deal.II/numerics/vector_tools.h>
#include <deal.II/numerics/data_out.h>

#include <fstream>
#include <cmath>
#include <iostream>
#include <deal.II/base/function_time.h>
// Finally this is as in all previous programs:
namespace Step35
{
  using namespace dealii;
```

```
using namespace numbers;


// @sect3{Run time parameters}
//
// Since our method has several parameters that can be fine-tuned we put
// them into an external file, so that they can be determined at run-time.
//
// This includes, in particular, the formulation of the equation for the
// auxiliary variable $\phi$, for which we declare an <code>enum</code>.
// Next, we declare a class that is going to read and store all the
// parameters that our program needs to run.
namespace RunTimeParameters
{
  enum MethodFormulation
  {
    METHOD_STANDARD,
    METHOD_ROTATIONAL
  };

  class Data_Storage
  {
  public:
    Data_Storage();
    ~Data_Storage();
    void read_data (const char *filename);
    MethodFormulation form;
    double initial_time,
           final_time,
           Reynolds;
    double dt;
    unsigned int n_global_refines,
             pressure_degree;
    unsigned int vel_max_iterations,
             vel_Krylov_size,
             vel_off_diagonals,
             vel_update_prec;
    double vel_eps,
           vel_diag_strength;
    bool verbose;
    unsigned int output_interval;
  protected:
    ParameterHandler prm;
  };

  // In the constructor of this class we declare all the parameters. The
  // details of how this works have been discussed elsewhere, for example in
  // step-19 and step-29.
  Data_Storage::Data_Storage()
  {
    prm.declare_entry ("Method_Form", "rotational",
```

```
                    Patterns::Selection ("rotational|standard"),
                    " Used to select the type of method that we are going "
                    "to use. ");
prm.enter_subsection ("Physical data");
{
  prm.declare_entry ("initial_time", "0.",
                     Patterns::Double (0.),
                     " The initial time of the simulation. ");
  prm.declare_entry ("final_time", "1.",
                     Patterns::Double (0.),
                     " The final time of the simulation. ");
  prm.declare_entry ("Reynolds", "1.",
                     Patterns::Double (0.),
                     " The Reynolds number. ");
}
prm.leave_subsection();

prm.enter_subsection ("Time step data");
{
  prm.declare_entry ("dt", "5e-4",
                     Patterns::Double (0.),
                     " The time step size. ");
}
prm.leave_subsection();

prm.enter_subsection ("Space discretization");
{
  prm.declare_entry ("n_of_refines", "0",
                     Patterns::Integer (0, 15),
                     " The number of global refines we do on the mesh. ");
  prm.declare_entry ("pressure_fe_degree", "1",
                     Patterns::Integer (1, 5),
                     " The polynomial degree for the pressure space. ");
}
prm.leave_subsection();

prm.enter_subsection ("Data solve velocity");
{
  prm.declare_entry ("max_iterations", "1000",
                     Patterns::Integer (1, 1000),
                     " The maximal number of iterations GMRES must make. ");
  prm.declare_entry ("eps", "1e-12",
                     Patterns::Double (0.),
                     " The stopping criterion. ");
  prm.declare_entry ("Krylov_size", "30",
                     Patterns::Integer(1),
                     " The size of the Krylov subspace to be used. ");
  prm.declare_entry ("off_diagonals", "60",
                     Patterns::Integer(0),
                     " The number of off-diagonal elements ILU must "
                     "compute. ");
```

```
    prm.declare_entry ("diag_strength", "0.01",
                       Patterns::Double (0.),
                       " Diagonal strengthening coefficient. ");
    prm.declare_entry ("update_prec", "15",
                       Patterns::Integer(1),
                       " This number indicates how often we need to "
                       "update the preconditioner");
  }
  prm.leave_subsection();

  prm.declare_entry ("verbose", "true",
                     Patterns::Bool(),
                     " This indicates whether the output of the solution "
                     "process should be verbose. ");

  prm.declare_entry ("output_interval", "1",
                     Patterns::Integer(1),
                     " This indicates between how many time steps we print "
                     "the solution. ");
}



Data_Storage::~Data_Storage()
{}



void Data_Storage::read_data (const char *filename)
{
  std::ifstream file (filename);
  AssertThrow (file, ExcFileNotOpen (filename));

  prm.read_input (file);

  if (prm.get ("Method_Form") == std::string ("rotational"))
    form = METHOD_ROTATIONAL;
  else
    form = METHOD_STANDARD;

  prm.enter_subsection ("Physical data");
  {
    initial_time = prm.get_double ("initial_time");
    final_time   = prm.get_double ("final_time");
    Reynolds     = prm.get_double ("Reynolds");

  }
  prm.leave_subsection();

  prm.enter_subsection ("Time step data");
  {
```

```
      dt = prm.get_double ("dt");
    }
    prm.leave_subsection();

    prm.enter_subsection ("Space discretization");
    {
      n_global_refines = prm.get_integer ("n_of_refines");
      pressure_degree     = prm.get_integer ("pressure_fe_degree");
    }
    prm.leave_subsection();

    prm.enter_subsection ("Data solve velocity");
    {
      vel_max_iterations = prm.get_integer ("max_iterations");
      vel_eps            = prm.get_double ("eps");
      vel_Krylov_size    = prm.get_integer ("Krylov_size");
      vel_off_diagonals  = prm.get_integer ("off_diagonals");
      vel_diag_strength  = prm.get_double ("diag_strength");
      vel_update_prec    = prm.get_integer ("update_prec");
    }
    prm.leave_subsection();

    verbose = prm.get_bool ("verbose");

    output_interval = prm.get_integer ("output_interval");
  }
}



// @sect3{Equation data}

// In the next namespace, we declare the initial and boundary conditions:
namespace EquationData
{
  // As we have chosen a completely decoupled formulation, we will not take
  // advantage of deal.II's capabilities to handle vector valued
  // problems. We do, however, want to use an interface for the equation
  // data that is somehow dimension independent. To be able to do that, our
  // functions should be able to know on which spatial component we are
  // currently working, and we should be able to have a common interface to
  // do that. The following class is an attempt in that direction.
  template <int dim>
  class MultiComponentFunction: public Function<dim>
  {
  public:
    MultiComponentFunction (const double initial_time = 0.);
    void set_component (const unsigned int d);
  protected:
    unsigned int comp;
```

```
    };

    template <int dim>
    MultiComponentFunction<dim>:: MultiComponentFunction (const double initial_time)
      :
      Function<dim> (1, initial_time), comp(0)
    {}


    template <int dim>
    void MultiComponentFunction<dim>::set_component(const unsigned int d)
    {
      Assert (d<dim, ExcIndexRange (d, 0, dim));
      comp = d;
    }

/* ---------------------------------------------------------------------
 * Here we will define the right hand side vector as a multi-component
 * function which is computed according to the true solution( velocity
 * vector and pressure function. An alternative right hand side funct-
 * ion is also given according to its true solution.
 *
 *---------------------------------------------------------------------*/

    template <int dim>
    class RightHandSide : public MultiComponentFunction<dim>
    {
    public:
      virtual double value (const Point<dim> &p,
                            const unsigned int component = 0)const;
      virtual void value_list (const std::vector< Point<dim> > &points,
                               std::vector<double> &values,
                               const unsigned int component = 0) const;

    };

    template <int dim>
    double RightHandSide<dim>::value (const Point<dim> &p,
                                      const unsigned int) const
    {
      const double nu = 0.1;
      if (this->comp == 0)
        {
          //return(-sin(p(0))*sin(p(1))+nu*cos(p(1)));

         //return (0*nu);
         return (std::exp(this->get_time())*sin(p(1))
                  -sin(p(0))*sin(p(1))+(1.0+std::exp(this->get_time()))*sin(p(0)-p(1))
                  +std::pow((1.0+std::exp(this->get_time())),2.0)*cos(p(0))*cos(p(1))
                  +nu*cos(p(1))+nu*(1.0+std::exp(this->get_time()))*sin(p(1))
                  +cos(p(0)+p(1))*(1.0+std::exp(this->get_time())));
```

```
    }

  else
    {//return (0);
       //return (cos(p(0))*cos(p(1))+nu*sin(p(0)));
     /*  return (std::pow(E,this->get_time())*(cos(p(0))
           -sin(p(0)-p(1))+nu*cos(p(0))+cos(p(0)+p(1)))
           +cos(p(0))*cos(p(1))-sin(p(0)-p(1))
           -std::pow((1.0+std::pow(E,this->get_time())),2.0)*sin(p(0))*cos(p(1))
           +nu*sin(p(0))+nu*cos(p(0))+cos(p(0)+p(1))); */
   return( std::exp(this->get_time())*cos(p(0))+cos(p(0))*cos(p(1))
         -(1.0+std::exp(this->get_time()))*sin(p(0)-p(1))
         -std::pow((1.0+std::exp(this->get_time())),2)*sin(p(0))*sin(p(1))
         +nu*sin(p(0))+nu*(1.0+std::exp(this->get_time()))*cos(p(0))
         +cos(p(0)+p(1))*(1.0+std::exp(this->get_time())));
     }
}
//

template <int dim>
void RightHandSide<dim>::value_list (const std::vector<Point<dim> > &points,
                                     std::vector<double> &values,
                                     const unsigned int) const
{
  const unsigned int n_points = points.size();
  Assert (values.size() == n_points,
          ExcDimensionMismatch (values.size(), n_points));
  for (unsigned int i=0; i<n_points; ++i)
    values[i] = RightHandSide<dim>::value (points[i]);
}

// With this class defined, we declare classes that describe the boundary
// conditions for velocity and pressure:
template <int dim>
class Velocity : public MultiComponentFunction<dim>
{
public:
  Velocity (const double initial_time = 0.0);

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

  virtual void value_list (const std::vector< Point<dim> > &points,
                           std::vector<double> &values,
                           const unsigned int component = 0) const;
  //virtual Tensor<1,dim> gradient (const Point<dim>   &p,
    //                              const unsigned int  component = 0) const;

};
```

```
    template <int dim>
    Velocity<dim>::Velocity (const double initial_time)
      :
      MultiComponentFunction<dim> (initial_time)
    {}


    template <int dim>
    void Velocity<dim>::value_list (const std::vector<Point<dim> > &points,
                                    std::vector<double> &values,
                                    const unsigned int) const
    {
      const unsigned int n_points = points.size();
      Assert (values.size() == n_points,
              ExcDimensionMismatch (values.size(), n_points));
      for (unsigned int i=0; i<n_points; ++i)
        values[i] = Velocity<dim>::value (points[i]);
    }
/* ---------------------------------------------------------------------
 * To compute H1 norm for velocity vector, we define here the gradient
 * of the velocity computing by hand from the true solution.
 *
 *---------------------------------------------------------------------*/

  /*  template <int dim>
    Tensor<1,dim> Velocity<dim>::gradient (const Point<dim>   &p,
                                   const unsigned int  component) const
     {  Tensor<1,dim> return_value;
       if (this->comp == 0)
        {   //return_value = 0.0;
           return_value[0] = 0;
           return_value[1] = -sin(p(1))+(1.0+std::exp(this->get_time()))*cos(p(1));
           return return_value;
         }
        else
         {  //return_value = 0.0;
           return_value[0] = cos(p(0))-(1.0+std::exp(this->get_time()))*sin(p(0));
           return_value[1] = 0;
           return return_value;
         }
      } */
/* ---------------------------------------------------------------------
 * Here we define the known exact velocity vector.
 *
 *---------------------------------------------------------------------*/

    template <int dim>
    double Velocity<dim>::value (const Point<dim> &p,
                                 const unsigned int) const
```

```
    {
      if (this->comp == 0)
          //return (cos(p(1)));
        return (cos(p(1))+(1.0+std::exp(this->get_time()))*sin(p(1)));
      else
         return (sin(p(0))+(1.0+std::exp(this->get_time()))*cos(p(0)));
          //return (sin(p(0)));
    }

/* ---------------------------------------------------------------------
 * For the solution of the Navier-Stokes equations we need the boundary
 * conditions. As we know the true solution for velocity, we are assign-
 * ing velocity boundary values exactly same as the value coming frome
 * the true solution.
 *
 *---------------------------------------------------------------------*/

    template <int dim>
    class BoundaryValues : public MultiComponentFunction<dim>
    {
      public:
      BoundaryValues () : MultiComponentFunction<dim>() {}

      virtual double value (const Point<dim>   &p,
                            const unsigned int  component = 0) const;
      virtual void value_list (const std::vector< Point<dim> > &points,
                               std::vector<double> &values,
                               const unsigned int component = 0) const;
    };

    template <int dim>
    void BoundaryValues<dim>::value_list (const std::vector<Point<dim> > &points,
                                  std::vector<double> &values,
                                  const unsigned int) const
    {
      const unsigned int n_points = points.size();
      Assert (values.size() == n_points,
              ExcDimensionMismatch (values.size(), n_points));
      for (unsigned int i=0; i<n_points; ++i)
        values[i] = BoundaryValues<dim>::value (points[i]);
    }

    template <int dim>
    double BoundaryValues<dim>::value (const Point<dim> &p,
                                   const unsigned int /*component*/) const
    {
      if (this->comp == 0)
            //return (cos(p(1)));
          return (cos(p(1))+(1.0+std::exp(this->get_time()))*sin(p(1)));
      else
          return (sin(p(0))+(1.0+std::exp(this->get_time()))*cos(p(0)));
```

```
          //return (sin(p(0)));
    }


    template <int dim>
    class Pressure: public Function<dim>
    {
    public:
      Pressure (const double initial_time = 0.0);

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

      virtual void value_list (const std::vector< Point<dim> > &points,
                               std::vector<double> &values,
                               const unsigned int component = 0) const;
    };

    template <int dim>
    Pressure<dim>::Pressure (const double initial_time)
      :
      Function<dim> (1, initial_time)
    {}
/* ---------------------------------------------------------------------
 * Here we are defining the true pressure function, which is a scaler
 * function of space and time. This function is necessary to get init-
 * ial pressure values at time t=0 and at times t=dt. This function
 * will also be called while computing pressure error.
 *
 *---------------------------------------------------------------------*/


    template <int dim>
    double Pressure<dim>::value (const Point<dim> &p,
                                 const unsigned int) const
    { //std::cout<<" Pressure Time= "<<this->get_time()<<std::endl;
      return sin(p(0)+p(1))*(1.0+std::exp(this->get_time()));
       // return (0);
    }

    template <int dim>
    void Pressure<dim>::value_list (const std::vector<Point<dim> > &points,
                                    std::vector<double> &values,
                                    const unsigned int) const
    {
      const unsigned int n_points = points.size();
      Assert (values.size() == n_points, ExcDimensionMismatch (values.size(), n_points));
      for (unsigned int i=0; i<n_points; ++i)
        values[i] = Pressure<dim>::value (points[i]);
    }
  }
```

```
// @sect3{The <code>NavierStokesProjection</code> class}

// Now for the main class of the program. It implements the various versions
// of the projection method for Navier-Stokes equations.  The names for all
// the methods and member variables should be self-explanatory, taking into
// account the implementation details given in the introduction.
template <int dim>
class NavierStokesProjection
{
public:
  NavierStokesProjection (const RunTimeParameters::Data_Storage &data);

  void run (const bool         verbose    = false,
            const unsigned int n_plots = 10);
protected:
  RunTimeParameters::MethodFormulation type;

  const unsigned int deg;
  const double       dt;
  const double       t_0, T, Re;

  EquationData::Velocity<dim>       vel_exact;
  EquationData::Pressure<dim> pressure_function;
  EquationData::BoundaryValues<dim> BV;
  EquationData::RightHandSide<dim> right_hand_side;



  std::map<types::global_dof_index, double>    boundary_values;
  std::vector<types::boundary_id> boundary_indicators;

  Triangulation<dim> triangulation;

  FE_Q<dim>          fe_velocity;
  FE_Q<dim>          fe_pressure;

  DoFHandler<dim>    dof_handler_velocity;
  DoFHandler<dim>    dof_handler_pressure;

  QGauss<dim>        quadrature_pressure;
  QGauss<dim>        quadrature_velocity;

  SparsityPattern    sparsity_pattern_velocity;
  SparsityPattern    sparsity_pattern_pressure;
  SparsityPattern    sparsity_pattern_pres_vel;

  SparseMatrix<double> vel_Laplace_plus_Mass;
  SparseMatrix<double> vel_it_matrix[dim];
```

```
  SparseMatrix<double> vel_Mass;
  SparseMatrix<double> vel_Laplace;
  SparseMatrix<double> vel_Advection;
  SparseMatrix<double> pres_Laplace;
  SparseMatrix<double> pres_Mass;
  SparseMatrix<double> pres_Diff[dim];
  SparseMatrix<double> pres_iterative;

  Vector<double> pres_n;
  Vector<double> pres_n_minus_1;
  Vector<double> interpolate_pressure; //to store interpolating pressure values.
  Vector<double> pressure_deviation;
  Vector<double> phi_n;
  Vector<double> phi_n_minus_1;
  Vector<double> u_n[dim];
  Vector<double> u_n_minus_1[dim];
  Vector<double> u_star[dim];
  Vector<double> force[dim];
  Vector<double> v_tmp;
  Vector<double> pres_tmp;
  Vector<double> rot_u;

  SparseILU<double> prec_velocity[dim];
  SparseILU<double> prec_pres_Laplace;
  SparseDirectUMFPACK prec_mass;
  SparseDirectUMFPACK prec_vel_mass;

  DeclException2 (ExcInvalidTimeStep,
                  double, double,
                  << " The time step " << arg1 << " is out of range."
                  << std::endl
                  << " The permitted range is (0," << arg2 << "]");

  //void create_triangulation_and_dofs (const unsigned int n_refines);

  void make_grid (const unsigned int n_refines);

  void initialize();

  void interpolate_velocity ();

  void diffusion_step (const bool reinit_prec);

  void projection_step (const bool reinit_prec);

  void update_pressure (const bool reinit_prec);

private:
  unsigned int vel_max_its;
  unsigned int vel_Krylov_size;
  unsigned int vel_off_diagonals;
```

```
unsigned int vel_update_prec;
double       vel_eps;
double       vel_diag_strength;

void initialize_velocity_matrices();

void initialize_pressure_matrices();


// The next few structures and functions are for doing various things in
// parallel. They follow the scheme laid out in @ref threads, using the
// WorkStream class. As explained there, this requires us to declare two
// structures for each of the assemblers, a per-task data and a scratch
// data structure. These are then handed over to functions that assemble
// local contributions and that copy these local contributions to the
// global objects.
//
// One of the things that are specific to this program is that we don't
// just have a single DoFHandler object that represents both the
// velocities and the pressure, but we use individual DoFHandler objects
// for these two kinds of variables. We pay for this optimization when we
// want to assemble terms that involve both variables, such as the
// divergence of the velocity and the gradient of the pressure, times the
// respective test functions. When doing so, we can't just anymore use a
// single FEValues object, but rather we need two, and they need to be
// initialized with cell iterators that point to the same cell in the
// triangulation but different DoFHandlers.
//
// To do this in practice, we declare a "synchronous" iterator -- an
// object that internally consists of several (in our case two) iterators,
// and each time the synchronous iteration is moved up one step, each of
// the iterators stored internally is moved up one step as well, thereby
// always staying in sync. As it so happens, there is a deal.II class that
// facilitates this sort of thing.
typedef std_cxx11::tuple< typename DoFHandler<dim>::active_cell_iterator,
        typename DoFHandler<dim>::active_cell_iterator
        > IteratorTuple;

typedef SynchronousIterators<IteratorTuple> IteratorPair;

void initialize_gradient_operator();

struct InitGradPerTaskData
{
  unsigned int              d;
  unsigned int              vel_dpc;
  unsigned int              pres_dpc;
  FullMatrix<double>        local_grad;
  std::vector<types::global_dof_index> vel_local_dof_indices;
  std::vector<types::global_dof_index> pres_local_dof_indices;
```

```
      InitGradPerTaskData (const unsigned int dd,
                           const unsigned int vdpc,
                           const unsigned int pdpc)
        :
        d(dd),
        vel_dpc (vdpc),
        pres_dpc (pdpc),
        local_grad (vdpc, pdpc),
        vel_local_dof_indices (vdpc),
        pres_local_dof_indices (pdpc)
      {}
    };

    struct InitGradScratchData
    {
      unsigned int  nqp;
      FEValues<dim> fe_val_vel;
      FEValues<dim> fe_val_pres;
      InitGradScratchData (const FE_Q<dim> &fe_v,
                           const FE_Q<dim> &fe_p,
                           const QGauss<dim> &quad,
                           const UpdateFlags flags_v,
                           const UpdateFlags flags_p)
        :
        nqp (quad.size()),
        fe_val_vel (fe_v, quad, flags_v),
        fe_val_pres (fe_p, quad, flags_p)
      {}
      InitGradScratchData (const InitGradScratchData &data)
        :
        nqp (data.nqp),
        fe_val_vel (data.fe_val_vel.get_fe(),
                    data.fe_val_vel.get_quadrature(),
                    data.fe_val_vel.get_update_flags()),
        fe_val_pres (data.fe_val_pres.get_fe(),
                     data.fe_val_pres.get_quadrature(),
                     data.fe_val_pres.get_update_flags())
      {}
    };

    void assemble_right_hand_side_term(); //Right hand side assemble function

/* ---------------------------------------------------------------------
 * Per task date for right hand side function
 *
 *---------------------------------------------------------------------*/

    struct RightSidePerTaskData
    {
      unsigned int              d;
      unsigned int              vel_dpc;
```

```
      Vector<double>            local_rhs;
      std::vector<types::global_dof_index> vel_local_dof_indices;
      RightSidePerTaskData (const unsigned int dd,
                            const unsigned int vdpc)
        :
        d(dd),
        vel_dpc (vdpc),
        local_rhs (vdpc),
        vel_local_dof_indices (vdpc)
      {}
    };

/* ---------------------------------------------------------------------
 * Scratch data for right hand side function
 *
 *---------------------------------------------------------------------*/

    struct RightSideScratchData
    {
      unsigned int  nqp;
      FEValues<dim> fe_val_vel;
      RightSideScratchData (const FE_Q<dim> &fe_v,
                            const QGauss<dim> &quad,
                            const UpdateFlags flags_v)
        :
        nqp (quad.size()),
        fe_val_vel (fe_v, quad, flags_v)
      {}
      RightSideScratchData (const RightSideScratchData &data)
        :
        nqp (data.nqp),
        fe_val_vel (data.fe_val_vel.get_fe(), data.fe_val_vel.get_quadrature(), data.fe_val_vel.ge

      {}
    };


    void assemble_one_cell_of_gradient (const IteratorPair  &SI,
                                        InitGradScratchData &scratch,
                                        InitGradPerTaskData &data);

    void copy_gradient_local_to_global (const InitGradPerTaskData &data);

    void assemble_one_cell_of_right_hand_side (const typename DoFHandler<dim>::active_cell_iterato
                                RightSideScratchData &scratch,
                                RightSidePerTaskData &data);

    void copy_right_hand_side_local_to_global(const RightSidePerTaskData &data);

    // The same general layout also applies to the following classes and
    // functions implementing the assembly of the advection term:
```

```
void assemble_advection_term();

struct AdvectionPerTaskData
{
  FullMatrix<double>        local_advection;
  std::vector<types::global_dof_index> local_dof_indices;
  AdvectionPerTaskData (const unsigned int dpc)
    :
    local_advection (dpc, dpc),
    local_dof_indices (dpc)
  {}
};

struct AdvectionScratchData
{
  unsigned int                 nqp;
  unsigned int                 dpc;
  std::vector< Point<dim> >    u_star_local;
  std::vector< Tensor<1,dim> > grad_u_star;
  std::vector<double>          u_star_tmp;
  FEValues<dim>                fe_val;
  AdvectionScratchData (const FE_Q<dim> &fe,
                        const QGauss<dim> &quad,
                        const UpdateFlags flags)
    :
    nqp (quad.size()),  // nqp = quad.size();
    dpc (fe.dofs_per_cell),
    u_star_local (nqp),
    grad_u_star (nqp),
    u_star_tmp (nqp),   // std::vector<double> u_star_tmp  (nqp);  resize to that length
    fe_val (fe, quad, flags)  // FEValues fe_val (fe, quad, flags);
  {}

  AdvectionScratchData (const AdvectionScratchData &data)
    :
    nqp (data.nqp),
    dpc (data.dpc),
    u_star_local (nqp),
    grad_u_star (nqp),
    u_star_tmp (nqp),
    fe_val (data.fe_val.get_fe(),
            data.fe_val.get_quadrature(),
            data.fe_val.get_update_flags())
  {}
};

void assemble_one_cell_of_advection (const typename DoFHandler<dim>::active_cell_iterator &cel
                                     AdvectionScratchData &scratch,
                                     AdvectionPerTaskData &data);

void copy_advection_local_to_global (const AdvectionPerTaskData &data);
```

```
    // The final few functions implement the diffusion solve as well as
    // postprocessing the output, including computing the curl of the
    // velocity:
    void diffusion_component_solve (const unsigned int d);

    double Error (); // velocity error function.

    void output_results (const unsigned int step);

    double pressure_error (); // pressure error function.

    void assemble_vorticity (const bool reinit_prec);
  };


/* -----------------------------------------------------------------
 * Here we want to compute pressure error squre. To do so, we compute
 * finite dimensional pressure with mean zero and interpolating true
 * pressure function and computing mean, we compute interpolating pre-
 * ssure with zero. Then we compute L2 error square by subtracting
 * finite dimensional pressure from interpolating pressure and integr-
 * ating.
 *
 *-----------------------------------------------------------------*/

  template <int dim>
  double NavierStokesProjection<dim>::pressure_error ()
   {

    double L2_pressure_error = 0;
    double finite_pres_mean_value = 0.0;
    double true_pres_mean_value = 0.0;

    //Compute mean of the finite dimensional pressure
    finite_pres_mean_value = VectorTools::compute_mean_value (dof_handler_pressure,
                                            QGauss<2>(4),
                                            pres_n,
                                            0);
    //Interpolate pressure from true pressure function
    VectorTools::interpolate (dof_handler_pressure, pressure_function, interpolate_pressure);

    //Compute the mean of the interpolating pressure
    true_pres_mean_value = VectorTools::compute_mean_value (dof_handler_pressure,
                                            QGauss<2>(4),
                                            interpolate_pressure,
                                            0);
    //Copy of pressure vector at time t^n
    pressure_deviation = pres_n;

/* -----------------------------------------------------------------
```

```
 * As we want to compute |p-p_{mean}-(p_h-p_{hmean}| which is equal to
 * |p-(p_h-p_{hmean}+p_{mean}|
 * Therefore, we are subtracting p_{hmean} and adding p_{mean} from the
 * finite dimensional pressure.
 * Note that: p_{hmean} is for mean of finite dimensional pressure and
 * p_{mean} is for mean of interpolating pressure.
 *
 *---------------------------------------------------------------------*/

    pressure_deviation.add(-finite_pres_mean_value + true_pres_mean_value);


    Vector<double> difference_per_cell_3 (triangulation.n_active_cells());

  // Integrating the defference per cell over the domain.
    VectorTools::integrate_difference (dof_handler_pressure,
                                       pressure_deviation,
                                       pressure_function,
                                       difference_per_cell_3,
                                       QGauss<dim>(4),
                                       VectorTools::L2_norm);

    //Compute L2 pressure error
    L2_pressure_error = difference_per_cell_3.l2_norm();
    std::cout<<"pressure deviation "<<L2_pressure_error<<std::endl;

    // returning L2 pressure error square
    return (L2_pressure_error*L2_pressure_error);
     }

/* ---------------------------------------------------------------------
 * Here we will compute L2/H1 velocity error squares.
 *
 *---------------------------------------------------------------------*/

  template <int dim>
  double NavierStokesProjection<dim>::Error ()
   {
    //defining necessary variables storing zero values in these.

    double L2error[2];
    double H1semi_error[2];
    double summ = 0.0;

    for (int d=0;d<2;++d)
     {
       L2error[d] = 0.0;
       H1semi_error[d] = 0.0;

     // For L2 and H1 we are declaring two vectors difference_per_cell_1 and
     //difference_per_cell_2 respectively and necessary information in these
```

```
      Vector<double> difference_per_cell_1 (triangulation.n_active_cells());
      Vector<double> difference_per_cell_2 (triangulation.n_active_cells());

    // setting the proper velocity component
      vel_exact.set_component(d);

    //integrate the differnece in L2 sense

      VectorTools::integrate_difference (dof_handler_velocity,
                                         u_n[d],
                                         vel_exact,
                                         difference_per_cell_1,
                                         QGauss<dim>(4),
                                         VectorTools::L2_norm);

//    The following code segments are exactly similar things for H1
      /* VectorTools::integrate_difference (dof_handler_velocity,
                                         u_n[d],
                                         vel_exact,
                                         difference_per_cell_2,
                                         QGauss<dim>(4),
                                         VectorTools::H1_seminorm); */

      //Compute the L2 norm

      L2error[d] = difference_per_cell_1.l2_norm();
      //Compute H1 norm
      // H1semi_error[d] = difference_per_cell_2.l2_norm();

    // summing up L2 error/H1 error squares

      summ = summ +L2error[d]*L2error[d]+H1semi_error[d]*H1semi_error[d];
    }

   //returning the appropriate error squares
   return summ;
  }


 // @sect4{ <code>NavierStokesProjection::NavierStokesProjection</code> }

 // In the constructor, we just read all the data from the
 // <code>Data_Storage</code> object that is passed as an argument, verify
 // that the data we read is reasonable and, finally, create the
 // triangulation and load the initial data.
 template <int dim>
 NavierStokesProjection<dim>::NavierStokesProjection(const RunTimeParameters::Data_Storage &data)
   :
   type (data.form),
   deg (data.pressure_degree),
```

```
  dt (data.dt),
  t_0 (data.initial_time),
  T (data.final_time),
  Re (data.Reynolds),
  //vel_exact (data.initial_time),
  //BV (data.initial_time),
  //right_hand_side (data.initial_time),
  fe_velocity (deg+1),
  fe_pressure (deg),
  dof_handler_velocity (triangulation),
  dof_handler_pressure (triangulation),
  quadrature_pressure (deg+1),
  quadrature_velocity (deg+2),
  vel_max_its (data.vel_max_iterations),
  vel_Krylov_size (data.vel_Krylov_size),
  vel_off_diagonals (data.vel_off_diagonals),
  vel_update_prec (data.vel_update_prec),
  vel_eps (data.vel_eps),
  vel_diag_strength (data.vel_diag_strength)
{
  if (deg < 1)
    std::cout << " WARNING: The chosen pair of finite element spaces is not stable."
              << std::endl
              << " The obtained results will be nonsense"
              << std::endl;

  AssertThrow (!  ( (dt <= 0.) || (dt > .5*T)), ExcInvalidTimeStep (dt, .5*T));

  make_grid (data.n_global_refines);
  initialize();
}


// @sect4{ <code>NavierStokesProjection::create_triangulation_and_dofs</code> }

// The method that creates the triangulation and refines it the needed
// number of times.  After creating the triangulation, it creates the mesh
// dependent data, i.e. it distributes degrees of freedom and renumbers
// them, and initializes the matrices and vectors that we will use.

template <int dim>
void NavierStokesProjection<dim>::make_grid (const unsigned int n_refines)
{ std::cout<<" my mesh "<<n_refines<<std::endl;
  GridGenerator::hyper_cube (triangulation, 0, 1);
  std::cout << "Number of refines = " << n_refines
            << std::endl;
  triangulation.refine_global (n_refines);
  std::cout << "Number of active cells: " << triangulation.n_active_cells()
            << std::endl;

  boundary_indicators = triangulation.get_boundary_indicators();
```

```
  dof_handler_velocity.distribute_dofs (fe_velocity);
  DoFRenumbering::boost::Cuthill_McKee (dof_handler_velocity);
  dof_handler_pressure.distribute_dofs (fe_pressure);
  DoFRenumbering::boost::Cuthill_McKee (dof_handler_pressure);

      for (unsigned int d=0; d<dim; ++d)
    {
      u_n[d].reinit (dof_handler_velocity.n_dofs());
      u_n_minus_1[d].reinit (dof_handler_velocity.n_dofs());
      u_star[d].reinit (dof_handler_velocity.n_dofs());
      force[d].reinit (dof_handler_velocity.n_dofs());
    }

  initialize_velocity_matrices();
  initialize_pressure_matrices();
  initialize_gradient_operator();


  pres_n.reinit (dof_handler_pressure.n_dofs());
  pres_n_minus_1.reinit (dof_handler_pressure.n_dofs());
  interpolate_pressure.reinit (dof_handler_pressure.n_dofs());
  phi_n.reinit (dof_handler_pressure.n_dofs());
  phi_n_minus_1.reinit (dof_handler_pressure.n_dofs());
  pres_tmp.reinit (dof_handler_pressure.n_dofs());


  v_tmp.reinit (dof_handler_velocity.n_dofs());
  rot_u.reinit (dof_handler_velocity.n_dofs());

  std::cout << "dim (X_h) = " << (dof_handler_velocity.n_dofs()*dim)
            << std::endl
            << "dim (M_h) = " << dof_handler_pressure.n_dofs()
            << std::endl
            << "Re        = " << Re
            << std::endl
            << std::endl;
}


// @sect4{ <code>NavierStokesProjection::initialize</code> }

// This method creates the constant matrices and loads the initial data
template <int dim>
void
NavierStokesProjection<dim>::initialize()
{
  vel_Laplace_plus_Mass = 0.;
  vel_Laplace_plus_Mass.add (1./Re, vel_Laplace);
  vel_Laplace_plus_Mass.add (1.5/dt, vel_Mass);
```

```
  EquationData::Pressure<dim> pres (t_0);

  VectorTools::interpolate (dof_handler_pressure, pres, pres_n_minus_1);

  pres.advance_time (dt);
  VectorTools::interpolate (dof_handler_pressure, pres, pres_n);
  phi_n = 0.;
  phi_n_minus_1 = 0.;
  for (unsigned int d=0; d<dim; ++d)
    {
      vel_exact.set_time (t_0);
      vel_exact.set_component(d);
      VectorTools::interpolate (dof_handler_velocity, vel_exact, u_n_minus_1[d]);
      vel_exact.advance_time (dt);

      VectorTools::interpolate (dof_handler_velocity, vel_exact, u_n[d]);
    }
}


// @sect4{ The <code>NavierStokesProjection::initialize_*_matrices</code> methods }

// In this set of methods we initialize the sparsity patterns, the
// constraints (if any) and assemble the matrices that do not depend on the
// timestep <code>dt</code>. Note that for the Laplace and mass matrices, we
// can use functions in the library that do this. Because the expensive
// operations of this function -- creating the two matrices -- are entirely
// independent, we could in principle mark them as tasks that can be worked
// on in %parallel using the Threads::new_task functions. We won't do that
// here since these functions internally already are parallelized, and in
// particular because the current function is only called once per program
// run and so does not incur a cost in each time step. The necessary
// modifications would be quite straightforward, however.
template <int dim>
void
NavierStokesProjection<dim>::initialize_velocity_matrices()
{
  sparsity_pattern_velocity.reinit (dof_handler_velocity.n_dofs(),
                                    dof_handler_velocity.n_dofs(),
                                    dof_handler_velocity.max_couplings_between_dofs());
  DoFTools::make_sparsity_pattern (dof_handler_velocity,
                                   sparsity_pattern_velocity);
  sparsity_pattern_velocity.compress();

  vel_Laplace_plus_Mass.reinit (sparsity_pattern_velocity);
  for (unsigned int d=0; d<dim; ++d)
    vel_it_matrix[d].reinit (sparsity_pattern_velocity);
  vel_Mass.reinit (sparsity_pattern_velocity);
  vel_Laplace.reinit (sparsity_pattern_velocity);
  vel_Advection.reinit (sparsity_pattern_velocity);
```

```
  MatrixCreator::create_mass_matrix (dof_handler_velocity,
                                     quadrature_velocity,
                                     vel_Mass);
  MatrixCreator::create_laplace_matrix (dof_handler_velocity,
                                        quadrature_velocity,
                                        vel_Laplace);
}



// The initialization of the matrices that act on the pressure space is
// similar to the ones that act on the velocity space.
template <int dim>
void
NavierStokesProjection<dim>::initialize_pressure_matrices()
{
  sparsity_pattern_pressure.reinit (dof_handler_pressure.n_dofs(), dof_handler_pressure.n_dofs()
                                    dof_handler_pressure.max_couplings_between_dofs());
  DoFTools::make_sparsity_pattern (dof_handler_pressure, sparsity_pattern_pressure);

  sparsity_pattern_pressure.compress();

  pres_Laplace.reinit (sparsity_pattern_pressure);
  pres_iterative.reinit (sparsity_pattern_pressure);
  pres_Mass.reinit (sparsity_pattern_pressure);

  MatrixCreator::create_laplace_matrix (dof_handler_pressure,
                                        quadrature_pressure,
                                        pres_Laplace);
  MatrixCreator::create_mass_matrix (dof_handler_pressure,
                                     quadrature_pressure,
                                     pres_Mass);
}



// For the gradient operator, we start by initializing the sparsity pattern
// and compressing it.  It is important to notice here that the gradient
// operator acts from the pressure space into the velocity space, so we have
// to deal with two different finite element spaces. To keep the loops
// synchronized, we use the <code>typedef</code>'s that we have defined
// before, namely <code>PairedIterators</code> and
// <code>IteratorPair</code>.
template <int dim>
void
NavierStokesProjection<dim>::initialize_gradient_operator()
{
  sparsity_pattern_pres_vel.reinit (dof_handler_velocity.n_dofs(),
                                    dof_handler_pressure.n_dofs(),
                                    dof_handler_velocity.max_couplings_between_dofs());
  DoFTools::make_sparsity_pattern (dof_handler_velocity,
                                   dof_handler_pressure,
```

```
                                    sparsity_pattern_pres_vel);
  sparsity_pattern_pres_vel.compress();

  InitGradPerTaskData per_task_data (0, fe_velocity.dofs_per_cell,
                                     fe_pressure.dofs_per_cell);
  InitGradScratchData scratch_data (fe_velocity,
                                    fe_pressure,
                                    quadrature_velocity,
                                    update_gradients | update_JxW_values | update_values|update_
                                    update_values);

  for (unsigned int d=0; d<dim; ++d)
    {
      pres_Diff[d].reinit (sparsity_pattern_pres_vel);
      per_task_data.d = d;
      WorkStream::run (IteratorPair (IteratorTuple (dof_handler_velocity.begin_active(),
                                                    dof_handler_pressure.begin_active()
                                                   )
                                    ),
                       IteratorPair (IteratorTuple (dof_handler_velocity.end(),
                                                    dof_handler_pressure.end()
                                                   )
                                    ),
                       *this,
                       &NavierStokesProjection<dim>::assemble_one_cell_of_gradient,
                       &NavierStokesProjection<dim>::copy_gradient_local_to_global,
                       scratch_data,
                       per_task_data
                      );
    }
}



template <int dim>
void
NavierStokesProjection<dim>::
assemble_one_cell_of_gradient (const IteratorPair  &SI,
                               InitGradScratchData &scratch,
                               InitGradPerTaskData &data)
{
  scratch.fe_val_vel.reinit (std_cxx11::get<0> (SI.iterators));
  scratch.fe_val_pres.reinit (std_cxx11::get<1> (SI.iterators));

  std_cxx11::get<0> (SI.iterators)->get_dof_indices (data.vel_local_dof_indices);
  std_cxx11::get<1> (SI.iterators)->get_dof_indices (data.pres_local_dof_indices);


  data.local_grad = 0.;

  Vector<double> rhs_vector (dim);
```

```
    for (unsigned int q=0; q<scratch.nqp; ++q)
      {

        for (unsigned int i=0; i<data.vel_dpc; ++i)
          for (unsigned int j=0; j<data.pres_dpc; ++j)
            data.local_grad (i, j) += -scratch.fe_val_vel.JxW(q) *
                                      scratch.fe_val_vel.shape_grad (i, q)[data.d] *
                                      scratch.fe_val_pres.shape_value (j, q);
      }

  }


  template <int dim>
  void  NavierStokesProjection<dim>::
  copy_gradient_local_to_global(const InitGradPerTaskData &data)
  {
    for (unsigned int i=0; i<data.vel_dpc; ++i)
      for (unsigned int j=0; j<data.pres_dpc; ++j)
        pres_Diff[data.d].add (data.vel_local_dof_indices[i], data.pres_local_dof_indices[j],
                               data.local_grad (i, j) );
  }
// Right hand side functions are assembling here which is similar to the
// advection term assembling and gradient vector assembling
  template <int dim>
  void
  NavierStokesProjection<dim>::assemble_right_hand_side_term()
  {

    RightSidePerTaskData data (0, fe_velocity.dofs_per_cell);

    RightSideScratchData scratch (fe_velocity,
                                  quadrature_velocity,
                                  update_JxW_values | update_values|update_quadrature_points);

    for(unsigned int d=0; d<dim; ++d){
      force[d]=0;
      data.d = d;
      right_hand_side.set_component(d);
      WorkStream::run (dof_handler_velocity.begin_active(),
                       dof_handler_velocity.end(), *this,
                       &NavierStokesProjection<dim>::assemble_one_cell_of_right_hand_side,
                       &NavierStokesProjection<dim>::copy_right_hand_side_local_to_global,
                       scratch,
                       data);
    }
  }

  template <int dim>
  void
  NavierStokesProjection<dim>::
```

```
assemble_one_cell_of_right_hand_side (const typename DoFHandler<dim>::active_cell_iterator &cell
                                RightSideScratchData &scratch,
                                RightSidePerTaskData &data)
{
  scratch.fe_val_vel.reinit(cell);
  cell->get_dof_indices (data.vel_local_dof_indices);


  data.local_rhs = 0.;

  for (unsigned int q=0; q<scratch.nqp; ++q)
      for (unsigned int i=0; i<data.vel_dpc; ++i)
          data.local_rhs (i) +=  scratch.fe_val_vel.JxW(q) *
                                 ( scratch.fe_val_vel.shape_value (i, q) *
                                  right_hand_side.value(scratch.fe_val_vel.quadrature_point(q)) )

}


template <int dim>
void  NavierStokesProjection<dim>::
copy_right_hand_side_local_to_global(const RightSidePerTaskData &data)
{
  for (unsigned int i=0; i<data.vel_dpc; ++i)
    force[data.d](data.vel_local_dof_indices[i]) += data.local_rhs (i);
}


// @sect4{ <code>NavierStokesProjection::run</code> }

// This is the time marching function, which starting at <code>t_0</code>
// advances in time using the projection method with time step
// <code>dt</code> until <code>T</code>.
//
// Its second parameter, <code>verbose</code> indicates whether the function
// should output information what it is doing at any given moment: for
// example, it will say whether we are working on the diffusion, projection
// substep; updating preconditioners etc. Rather than implementing this
// output using code like
// @code
//   if (verbose) std::cout << "something";
// @endcode
// we use the ConditionalOStream class to do that for us. That
// class takes an output stream and a condition that indicates whether the
// things you pass to it should be passed through to the given output
// stream, or should just be ignored. This way, above code simply becomes
// @code
//   verbose_cout << "something";
// @endcode
// and does the right thing in either case.
template <int dim>
```

```
void
NavierStokesProjection<dim>::run (const bool verbose,
                                  const unsigned int output_interval)
{
  ConditionalOStream verbose_cout (std::cout, verbose);

  const unsigned int n_steps =  static_cast<unsigned int>((T - t_0)/dt);


  double L2H1Error = 0.0;
  double L2ErrorSquare = 0.0;

  double L2_pressure_error = 0.0;
  double L2L2_pressure_error = 0.0;

  //printing outputs for initial conditions
  output_results(1);

  //settting time in exact velocity function
  vel_exact.set_time (dt);

  //setting time in exact pressure function
  pressure_function.set_time (dt);

  //calculating interpolating errors
  std::cout<<"Error  "<<std::sqrt(Error ())<<std::endl;

  //settting time in exact velocity function
  vel_exact.set_time (2.*dt);

  //setting time in exact pressure function
  pressure_function.set_time (2.*dt);

  //setting time in right hand side function
  right_hand_side.set_time(2.*dt);

  // setting time in boundary value function
  BV.set_time (2.*dt);

  for (unsigned int n = 2; n<=n_steps; ++n)
    {

      if (n % output_interval == 0)
        {
          verbose_cout << "Plotting Solution" << std::endl;
          output_results(n);
        }
      std::cout << "Step = " << n << " Time = " << (n*dt) << std::endl;

      verbose_cout << "  Interpolating the velocity " << std::endl;
```

```
      interpolate_velocity();

      verbose_cout << "  Diffusion Step" << std::endl;
      if (n % vel_update_prec == 0)
        verbose_cout << "    With reinitialization of the preconditioner"
                     << std::endl;
      diffusion_step ((n%vel_update_prec == 0) || (n == 2));
      std::cout<<"Error  in this step "<< std::sqrt(Error ()) <<std::endl;

      // Adding velocity error squares, it may be L2 in space or H1 in space
      // depending on how we are computing errors in Error() function. As both
      // options are already included in that function but H1 error code segments
      // are in comments now.

      L2ErrorSquare = L2ErrorSquare + Error ();

      verbose_cout << "  Projection Step" << std::endl;
      projection_step ( (n == 2));
      verbose_cout << "  Updating the Pressure" << std::endl;
      update_pressure ( (n == 2));

      //Adding pressure error squares

      L2_pressure_error = L2_pressure_error + pressure_error ();

      //advancing time in true pressure function
      pressure_function.advance_time(dt);

      //advancing time in true velocity function
      vel_exact.advance_time(dt);

      //advancing time in right hand side function
      right_hand_side.advance_time(dt);

      //advancing time in the boundary value function
      BV.advance_time(dt);
    }
// computing L2 in time and L2 in space pressure error norm

  L2L2_pressure_error = sqrt(L2_pressure_error*dt);
  std::cout<<"L2L2_pressure_error "<<L2L2_pressure_error<<std::endl;

// computing L2 in time and L2 in space for velocity error norm
  L2H1Error = sqrt(L2ErrorSquare*dt);
  std::cout<<"L2H1Error ="<<L2H1Error<<std::endl;

  //printing errors
  output_results (n_steps);

}
```

```
template <int dim>
void
NavierStokesProjection<dim>::interpolate_velocity()
{
  for (unsigned int d=0; d<dim; ++d)
    u_star[d].equ (2., u_n[d], -1, u_n_minus_1[d]);
}


// @sect4{<code>NavierStokesProjection::diffusion_step</code>}

// The implementation of a diffusion step. Note that the expensive operation
// is the diffusion solve at the end of the function, which we have to do
// once for each velocity component. To accelerate things a bit, we allow
// to do this in %parallel, using the Threads::new_task function which makes
// sure that the <code>dim</code> solves are all taken care of and are
// scheduled to available processors: if your machine has more than one
// processor core and no other parts of this program are using resources
// currently, then the diffusion solves will run in %parallel. On the other
// hand, if your system has only one processor core then running things in
// %parallel would be inefficient (since it leads, for example, to cache
// congestion) and things will be executed sequentially.
template <int dim>
void
NavierStokesProjection<dim>::diffusion_step (const bool reinit_prec)
{

  pres_tmp.equ (-1., pres_n, -4./3., phi_n, 1./3., phi_n_minus_1);

  assemble_advection_term();
  assemble_right_hand_side_term();

  for (unsigned int d=0; d<dim; ++d)
    {



      v_tmp.equ (2./dt,u_n[d],-.5/dt,u_n_minus_1[d]);
      vel_Mass.vmult_add (force[d], v_tmp);

      pres_Diff[d].vmult_add (force[d], pres_tmp);
      u_n_minus_1[d] = u_n[d];

      vel_it_matrix[d].copy_from (vel_Laplace_plus_Mass);
      vel_it_matrix[d].add (1., vel_Advection);

      //setting component for the boundary value function
      BV.set_component(d);
      boundary_values.clear();
```

```
      //interpolating boundary values from the true function

      VectorTools::interpolate_boundary_values (dof_handler_velocity,
                                                0,
                                                BV,
                                                boundary_values);

      MatrixTools::apply_boundary_values (boundary_values,
                                          vel_it_matrix[d],
                                          u_n[d],
                                          force[d]);


    }


  Threads::TaskGroup<void> tasks;
  for (unsigned int d=0; d<dim; ++d)
    {
      if (reinit_prec)
        prec_velocity[d].initialize (vel_it_matrix[d],
                                     SparseILU<double>::
                                     AdditionalData (vel_diag_strength,
                                                     vel_off_diagonals));
      tasks += Threads::new_task (&NavierStokesProjection<dim>::
                                  diffusion_component_solve,
                                  *this, d);
    }
  tasks.join_all();
}



template <int dim>
void
NavierStokesProjection<dim>::diffusion_component_solve (const unsigned int d)
{
  SolverControl solver_control (vel_max_its, vel_eps*force[d].l2_norm());
  SolverGMRES<> gmres (solver_control,
                       SolverGMRES<>::AdditionalData (vel_Krylov_size));

  gmres.solve (vel_it_matrix[d], u_n[d], force[d], prec_velocity[d]);
}


// @sect4{ The <code>NavierStokesProjection::assemble_advection_term</code> method and related}

// The following few functions deal with assembling the advection terms,
// which is the part of the system matrix for the diffusion step that
// changes at every time step. As mentioned above, we will run the assembly
```

```
// loop over all cells in %parallel, using the WorkStream class and other
// facilities as described in the documentation module on @ref threads.
template <int dim>
void
NavierStokesProjection<dim>::assemble_advection_term()
{
  vel_Advection = 0.;
  AdvectionPerTaskData data (fe_velocity.dofs_per_cell);
  AdvectionScratchData scratch (fe_velocity, quadrature_velocity,
                                update_values |
                                update_JxW_values |
                                update_gradients);
  WorkStream::run (dof_handler_velocity.begin_active(),
                   dof_handler_velocity.end(), *this,
                   &NavierStokesProjection<dim>::assemble_one_cell_of_advection,
                   &NavierStokesProjection<dim>::copy_advection_local_to_global,
                   scratch,
                   data);
}



template <int dim>
void
NavierStokesProjection<dim>::
assemble_one_cell_of_advection(const typename DoFHandler<dim>::active_cell_iterator &cell,
                               AdvectionScratchData &scratch,
                               AdvectionPerTaskData &data)
{
  scratch.fe_val.reinit(cell);
  cell->get_dof_indices (data.local_dof_indices);
  for (unsigned int d=0; d<dim; ++d)
    {
      scratch.fe_val.get_function_values (u_star[d], scratch.u_star_tmp);
      for (unsigned int q=0; q<scratch.nqp; ++q)
        scratch.u_star_local[q](d) = scratch.u_star_tmp[q];
    }

  for (unsigned int d=0; d<dim; ++d)
    {
      scratch.fe_val.get_function_gradients (u_star[d], scratch.grad_u_star);
      for (unsigned int q=0; q<scratch.nqp; ++q)
        {
          if (d==0)
            scratch.u_star_tmp[q] = 0.;
          scratch.u_star_tmp[q] += scratch.grad_u_star[q][d];
        }
    }

  data.local_advection = 0.;
  for (unsigned int q=0; q<scratch.nqp; ++q)
```

```
    for (unsigned int i=0; i<scratch.dpc; ++i)
      for (unsigned int j=0; j<scratch.dpc; ++j)
        data.local_advection(i,j) += (scratch.u_star_local[q] *
                                      scratch.fe_val.shape_grad (j, q) *
                                      scratch.fe_val.shape_value (i, q)
                                      +
                                      0.5 *
                                      scratch.u_star_tmp[q] *
                                      scratch.fe_val.shape_value (i, q) *
                                      scratch.fe_val.shape_value (j, q))
                                     *
                                     scratch.fe_val.JxW(q) ;
}



template <int dim>
void
NavierStokesProjection<dim>::
copy_advection_local_to_global(const AdvectionPerTaskData &data)
{
  for (unsigned int i=0; i<fe_velocity.dofs_per_cell; ++i)
    for (unsigned int j=0; j<fe_velocity.dofs_per_cell; ++j)
      vel_Advection.add (data.local_dof_indices[i],
                         data.local_dof_indices[j],
                         data.local_advection(i,j));
}



// @sect4{<code>NavierStokesProjection::projection_step</code>}

// This implements the projection step:
template <int dim>
void
NavierStokesProjection<dim>::projection_step (const bool reinit_prec)
{
  pres_iterative.copy_from (pres_Laplace);

  pres_tmp = 0.;
  for (unsigned d=0; d<dim; ++d)
    pres_Diff[d].Tvmult_add (pres_tmp, u_n[d]);

  phi_n_minus_1 = phi_n;

  static std::map<types::global_dof_index, double> bval;

  //to get unique pressure we are doing the following two lines.
  if (reinit_prec)
     bval[0]=0.0;
    //VectorTools::interpolate_boundary_values (dof_handler_pressure, 0,
```

```
    //                                          ZeroFunction<dim>(), bval);


  MatrixTools::apply_boundary_values (bval, pres_iterative, phi_n, pres_tmp);

  if (reinit_prec)
    prec_pres_Laplace.initialize(pres_iterative,
                                 SparseILU<double>::AdditionalData (vel_diag_strength,
                                     vel_off_diagonals) );

  SolverControl solvercontrol (vel_max_its, vel_eps*pres_tmp.l2_norm());
  SolverCG<> cg (solvercontrol);
  cg.solve (pres_iterative, phi_n, pres_tmp, prec_pres_Laplace);

  phi_n *= 1.5/dt;
}


// @sect4{ <code>NavierStokesProjection::update_pressure</code> }

// This is the pressure update step of the projection method. It implements
// the standard formulation of the method, that is @f[ p^{n+1} = p^n +
// \phi^{n+1}, @f] or the rotational form, which is @f[ p^{n+1} = p^n +
// \phi^{n+1} - \frac{1}{Re} \nabla\cdot u^{n+1}.  @f]
template <int dim>
void
NavierStokesProjection<dim>::update_pressure (const bool reinit_prec)
{
  pres_n_minus_1 = pres_n;
  switch (type)
    {
    case RunTimeParameters::METHOD_STANDARD:
      pres_n += phi_n;
      break;
    case RunTimeParameters::METHOD_ROTATIONAL:
      if (reinit_prec)
        prec_mass.initialize (pres_Mass);
      pres_n = pres_tmp;
      prec_mass.solve (pres_n);
      pres_n.sadd(1./Re, 1., pres_n_minus_1, 1., phi_n);
      break;
    default:
      Assert (false, ExcNotImplemented());
    };
}


// @sect4{ <code>NavierStokesProjection::output_results</code> }

// This method plots the current solution. The main difficulty is that we
// want to create a single output file that contains the data for all
```

```
// velocity components, the pressure, and also the vorticity of the flow. On
// the other hand, velocities and the pressure live on separate DoFHandler
// objects, and so can't be written to the same file using a single DataOut
// object. As a consequence, we have to work a bit harder to get the various
// pieces of data into a single DoFHandler object, and then use that to
// drive graphical output.
//
// We will not elaborate on this process here, but rather refer to step-32,
// where a similar procedure is used (and is documented) to
// create a joint DoFHandler object for all variables.
//
// Let us also note that we here compute the vorticity as a scalar quantity
// in a separate function, using the $L^2$ projection of the quantity
// $\text{curl} u$ onto the finite element space used for the components of
// the velocity. In principle, however, we could also have computed as a
// pointwise quantity from the velocity, and do so through the
// DataPostprocessor mechanism discussed in step-29 and step-33.
template <int dim>
void NavierStokesProjection<dim>::output_results (const unsigned int step)
{
  assemble_vorticity ( (step == 1));

  const FESystem<dim> joint_fe (fe_velocity, dim,
                                fe_pressure, 1,
                                fe_velocity, 1);

  DoFHandler<dim> joint_dof_handler (triangulation);
  joint_dof_handler.distribute_dofs (joint_fe);
  Assert (joint_dof_handler.n_dofs() ==
          ((dim + 1)*dof_handler_velocity.n_dofs() +
           dof_handler_pressure.n_dofs()),
          ExcInternalError());
  static Vector<double> joint_solution (joint_dof_handler.n_dofs());
  std::vector<types::global_dof_index> loc_joint_dof_indices (joint_fe.dofs_per_cell),
      loc_vel_dof_indices (fe_velocity.dofs_per_cell),
      loc_pres_dof_indices (fe_pressure.dofs_per_cell);
  typename DoFHandler<dim>::active_cell_iterator
  joint_cell = joint_dof_handler.begin_active(),
  joint_endc = joint_dof_handler.end(),
  vel_cell   = dof_handler_velocity.begin_active(),
  pres_cell  = dof_handler_pressure.begin_active();

  for (; joint_cell != joint_endc; ++joint_cell, ++vel_cell, ++pres_cell)
    {
      joint_cell->get_dof_indices (loc_joint_dof_indices);
      vel_cell->get_dof_indices (loc_vel_dof_indices),
               pres_cell->get_dof_indices (loc_pres_dof_indices);
      for (unsigned int i=0; i<joint_fe.dofs_per_cell; ++i)
        switch (joint_fe.system_to_base_index(i).first.first)
          {
          case 0:
```

```
            Assert (joint_fe.system_to_base_index(i).first.second < dim,
                    ExcInternalError());
            joint_solution (loc_joint_dof_indices[i]) =
              u_n[ joint_fe.system_to_base_index(i).first.second ]
              (loc_vel_dof_indices[ joint_fe.system_to_base_index(i).second ]);
            break;
          case 1:
            Assert (joint_fe.system_to_base_index(i).first.second == 0,
                    ExcInternalError());
            joint_solution (loc_joint_dof_indices[i]) =
              pres_n (loc_pres_dof_indices[ joint_fe.system_to_base_index(i).second ]);
            break;
          case 2:
            Assert (joint_fe.system_to_base_index(i).first.second == 0,
                    ExcInternalError());
            joint_solution (loc_joint_dof_indices[i]) =
              rot_u (loc_vel_dof_indices[ joint_fe.system_to_base_index(i).second ]);
            break;
          default:
            Assert (false, ExcInternalError());
          }
      }

    std::vector<std::string> joint_solution_names (dim, "v");
    joint_solution_names.push_back ("p");
    joint_solution_names.push_back ("rot_u");
    DataOut<dim> data_out;
    data_out.attach_dof_handler (joint_dof_handler);
    std::vector< DataComponentInterpretation::DataComponentInterpretation >
    component_interpretation (dim+2,
                              DataComponentInterpretation::component_is_part_of_vector);
    component_interpretation[dim]
      = DataComponentInterpretation::component_is_scalar;
    component_interpretation[dim+1]
      = DataComponentInterpretation::component_is_scalar;
    data_out.add_data_vector (joint_solution,
                              joint_solution_names,
                              DataOut<dim>::type_dof_data,
                              component_interpretation);
  /*    Vector<double> difference_per_cell[2];
 difference_per_cell[0].reinit(triangulation.n_active_cells());
 difference_per_cell[1].reinit(triangulation.n_active_cells());
 {
for (unsigned int d=0;d<dim;++d)
{
       //Vector<double> difference_per_cell_2 (triangulation.n_active_cells());
       vel_exact.set_component(d);
       VectorTools::integrate_difference (dof_handler_velocity,
                                    u_n[d],
                                    vel_exact,
                                    difference_per_cell[d],
```

```
                                     QGauss<dim>(4),
                                     VectorTools::L2_norm);
std::string name="e"+Utilities::int_to_string(d);
data_out.add_data_vector(difference_per_cell[d],name);
}

    }*/
    data_out.build_patches (deg + 1);
    std::ofstream output (("solution-" +
                           Utilities::int_to_string (step, 5) +
                           ".vtk").c_str());
    data_out.write_vtk (output);
  }



  // Following is the helper function that computes the vorticity by
  // projecting the term $\text{curl} u$ onto the finite element space used
  // for the components of the velocity. The function is only called whenever
  // we generate graphical output, so not very often, and as a consequence we
  // didn't bother parallelizing it using the WorkStream concept as we do for
  // the other assembly functions. That should not be overly complicated,
  // however, if needed. Moreover, the implementation that we have here only
  // works for 2d, so we bail if that is not the case.
  template <int dim>
  void NavierStokesProjection<dim>::assemble_vorticity (const bool reinit_prec)
  {

    Assert (dim == 2, ExcNotImplemented());
    if (reinit_prec)
      prec_vel_mass.initialize (vel_Mass);

    FEValues<dim> fe_val_vel (fe_velocity, quadrature_velocity,
                              update_gradients |
                              update_JxW_values |
                              update_values);
    const unsigned int dpc = fe_velocity.dofs_per_cell,
                       nqp = quadrature_velocity.size();
    std::vector<types::global_dof_index> ldi (dpc);
    Vector<double> loc_rot (dpc);

    std::vector< Tensor<1,dim> > grad_u1 (nqp), grad_u2 (nqp);
    rot_u = 0.;

    typename DoFHandler<dim>::active_cell_iterator
    cell = dof_handler_velocity.begin_active(),
    end  = dof_handler_velocity.end();
    for (; cell != end; ++cell)
      {
        fe_val_vel.reinit (cell);
        cell->get_dof_indices (ldi);
```

```
        fe_val_vel.get_function_gradients (u_n[0], grad_u1);
        fe_val_vel.get_function_gradients (u_n[1], grad_u2);
        loc_rot = 0.;
        for (unsigned int q=0; q<nqp; ++q)
          for (unsigned int i=0; i<dpc; ++i)
            loc_rot(i) += (grad_u2[q][0] - grad_u1[q][1]) *
                          fe_val_vel.shape_value (i, q) *
                          fe_val_vel.JxW(q);

        for (unsigned int i=0; i<dpc; ++i)
          rot_u (ldi[i]) += loc_rot(i);
      }

    prec_vel_mass.solve (rot_u);

  }
}


// @sect3{ The main function }

// The main function looks very much like in all the other tutorial programs,
// so there is little to comment on here:
int main()
{
  try
    {
      using namespace dealii;
      using namespace Step35;

      RunTimeParameters::Data_Storage data;
      data.read_data ("parameter-file.prm");

      deallog.depth_console (data.verbose ? 2 : 0);

      NavierStokesProjection<2> test (data);

      test.run (data.verbose, data.output_interval);

    }
  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;
    }
  std::cout << "-----------------------------------------------------"
            << std::endl
            << "Apparently everything went fine!"
            << std::endl
            << "Don't forget to brush your teeth :-)"
            << std::endl << std::endl;

  return 0;
}
```