JICS

CSURE 2014 Summer Program

Modeling the Effects of Increased Glucose Concentration on Intraocular Pressure

Author:
Caroline Su
(University of California, Berkeley)
Alexander Cope
(Centre College)
Contributor:
Ben Ramsey
(University of Tennessee, Knoxville)

Supervisor:
Dr. Kwai Wong
(University of Tennessee, Knoxville)

August 8th, 2014

Abstract

Glaucoma is one of the leading causes of blindness in the world. Two of the known risk factors for glaucoma development are high fluid pressure within the eye and diabetes. We have developed a model which relates aqueous humor flow in the anterior chamber of the eye to increased glucose concentration. Our ultimate goal is to develop this model and perform simulations in 3D using COMSOL and Deal.II and obtain consistent results. Currently, standard Navier-Stokes flow has been modeled successfully and consistently using COMSOL and Deal.II in 2D. The simulations are in the process of being expanded to run in 3D. From there, the modified equations can be incorporated into the simulations.

1 Introduction

Despite its small size, the human eye is a complex organ which has demanded the attention of researchers for centuries. The eye is made up of three main chambers: the anterior chamber, the posterior chamber, and the vitreous chamber. The anterior chamber (located between the cornea and the iris) contains the aqueous humor, a clear, water-like fluid which will be the focus of this study. The aqueous humor distributes nutrients to the anterior part of the eye and regulates the intraocular pressure (IOP). The aqueous humor is continuously produced in the ciliary body and is released into the anterior chamber. From there, the fluid travels through the trabecular meshwork, a matrix like structure, and then into the Schlemms canal, where the aqueous humor mixes with venous blood. This pathway accounts for the majority of aqueous outflow. The trabecular meshwork and Schlemms canal are considered to be the main contributors to resistance to aqueous outflow.

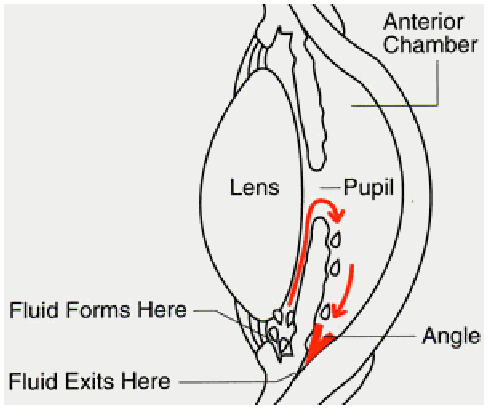

Figure 1: http://www.theeyecenter.com/educational/005.htm

Increased IOP has been well-documented as a risk-factor for the development of glaucoma, the 2nd leading cause of blindness in the United States. IOP is thought to normally increase due to one of two main causes: an increased resistance to aqueous flow within the trabecular meshwork and the Schlemms canal, which would result in open-angle glaucoma;

Porosity (ϵ)	Permeability (m^2) of TM	Pressure in AC (Pa)
0.4	7.59×10^{-14}	1271
0.3	2.35×10^{-14}	1429
0.25	1.19×10^{-14}	1655
0.225	8.09×10^{-15}	1867
0.2	5.33×10^{-15}	2211
0.175	3.36×10^{-15}	2805
0.15	1.99×10^{-15}	3905
0.125	1.09×10^{-15}	6154
0.1	5.27×10^{-16}	11437

Table 1: Effects of decreased porosity on IOP[3]

κ	AvgPreAC	AvgPreTM	$\triangle Pre$
2×10^{-05}	-2.1230×10^{-1}	-2.9939×10^{0}	2.7816×10^{0}
2×10^{-06}	-5.7427×10^{-2}	-2.4961×10^{1}	2.4904×10^{1}
2×10^{-07}	1.4873×10^{0}	-2.4463×10^{2}	2.4314×10^{2}
2×10^{-08}	1.6932×10^{1}	-2.4410×10^{3}	2.4241×10^{3}
2×10^{-09}	1.7138×10^{2}	-2.4404×10^{4}	2.4233×10^{4}

Table 2: Permeability of trabecular meshwork and relation to IOP. κ represents permeability and $\triangle P$ represents the overall change in IOP [2]

or the sudden moving forward of the iris, closing the gap through which the aqueous humor exits the anterior chamber, which is known as closed-angle glaucoma.

As a result of the relatively high frequency of glaucoma diagnoses, many researchers have sought to model changes in IOP as a result of aqueous humor flow. Recently, a model developed by J.A. Ferreira et. al. [3] simulated 2D flow of the the aqueous humor and its relation to both obstruction of the trabecular meshwork and as a form of drug distribution. They found that, as expected, increased obstruction of the trabecular meshwork (modeled by decreasing the porosity parameter in Darcys Law) resulted in increased IOP.

Similarly, Crowder and Ervin [2] developed an axisymmetric model using both Navier-Stokes and Darcys Law. As expected, they found that permeability and IOP were inversely proportional; with each order of magnitude that permeability decreased, IOP increased by the same order of magnitude (see Table 2).

Experimental studies indicate the role of temperature changes in aqueous humor flow [4]. When an eye is open, a temperature gradient is established between the cornea (approximately outside temperature) and the iris/lens (approximately normal body temperature). This temperature gradient establishes buoyancy driven flow, which was modeled by both Canning et. al[1] and Fitt and Gonzalez[4]. These models revealed the dominant effect of buoyancy driven flow over other forces, with flow speeds being of order 0.1 mm/s.

When temperature is uniform within the anterior chamber, buoyancy-driven flow is minimal, causing the main driving force to be flow of aqueous humor through the pupil aperture. In simulations run by Fitt and Gonzalez[4], maximum flow velocity was found to be near $7.5 \times 10^{-}6 {}^{m}/_{s}$.

Previous studies have also found a strong correlation between those affected by diabetes and the development of glaucoma. While the exact mechanism resulting in the development of glaucoma due to diabetes in unknown, research completed by Tsuyoshi Sato and Sayon Roy[6] found that fibronectin production in the trabecular meshwork does increase in high-glucose concentration environments. It is believed this increase in fibronectin results in an increased resistance to flow of the aqueous humor out of the anterior chamber of the eye.

2 Objective

The ultimate goal of our model is to simulate IOP under both normal glucose concentration and high glucose concentrations. We will model human eyes under different situations: with all other factors remaining constant, let glucose concentration be the experimental factor, starting with that of an average adult ($5.5 {}^{mmol}/_{L}$) and increase to that of an adult with type 2 diabetes ($8 {}^{mmol}/_{L}$). Then we will compare the results of calculated IOP and discuss the correlation between diabetes and glaucoma. It is expected that as glucose concentration increases, IOP will also increase. If successful, this model could provide insight as to why diabetics are at a higher risk of developing open-angle glaucoma.

We will also attempt to develop our own software which will solve our systems of equations. The software will be designed to run in parallel using MPI and Trilinos software packages. The code will initially be designed to solve the Laplace equation,

$$\triangle u = 0 \tag{1}$$

Once this code has been made to work for 1D and 2D, it will then be expanded to fit the equations of our model. By developing a program that can solve these equations in parallel across many processes, simulations can be performed on more refined meshes in a faster amount of time.

3 Methods

The models we have developed borrows significantly from previous 2D and 3D models of aqueous humor flow. Many of the equation used were borrowed and modified from the 3D model used by Adan Villamarin et. al.[7]. The cornea will be modeled as a rigid structure, allowing the no-slip boundary condition to be used. The iris will also be considered as a rigid structure, although the model may later be expanded to treat the iris as a linearly elastic structure[5]. The pupil aperture will be treated as the inlet with a radius of 8 mm through which the aqueous humor enters the anterior chamber, but its size and shape will remain constant throughout the simulation. The eye will be positioned such that to model flow when a person is standing and looking forward (see Figure 1). The aqueous humor will flow into the anterior chamber with an initial velocity of 1.2 mm/s (J.A Ferreira et al)[3].

The fluid flow in the anterior chamber (which will be 3mm wide between the cornea and the pupil aperture) will be modeled using a modified form of the Navier-Stokes equations for incompressible flow, which includes an additional term for thermal change:

$$\rho\bar{v} \cdot \bigtriangledown\bar{v} = -\bigtriangledown p + \mu \bigtriangledown^2 \bar{v} + rho_0\bar{g}\beta(T - T_{ref}) \tag{2}$$

The equations below describes the continuity that aqueous humor for steady and incompressible flow has to satisfy in the anterior chamber:

$$\bigtriangledown \cdot \bar{v} = 0 \tag{3}$$

And the equation below describes the convective and diffusive transport of energy by aqueous humor where viscous dissipation effect has been neglected:

$$\rho C_p\bar{v} \cdot \bigtriangledown T = k \bigtriangledown^2 T \tag{4}$$

where k is the thermal conductivity and C_p is the heat capacity. This modified set of equations reflects the buoyancy-driven nature of aqueous humor flow.

In order to simplify the model, the trabecular meshwork and Schlemms canal will be treated as a uniform porous region over. The initial outflow pressure will be treated as 1200 Pa[3] This will allow the permeability of the trabecular meshwork and Schlemms canal to be modeled using a modified form of Darcys law which accounts for fibronectin production:

$$\alpha = {}^{\mu}/_{\triangle p} \triangle e\bar{v} - f(g_c) \tag{5}$$

where $\triangle e$ is the thickness of the porous domain and $f(g_c)$ represents the fibronectin function, and g_c is the given glucose concentration (in mg/dL). As the rate of fibronectin production within the trabecular meshwork is unknown, we assume this value can be modeled by the ODE:

$$ {}^{df}/_{dg_c} = rg_c \tag{6}$$

where $r > 0$. This ODE can be solved using separation of variables to yield:

$$f(g_c) = {}^{1}/_{2}r(g_c)^2 + C \tag{7}$$

where C is some constant.

The PDEs will be solved using Finite Element Method software. Initial tests will be performed using a 2D model created in Cubit and imported into the FEATool software (see Figure 2). Once the model has been fine-tuned, the model will be expanded into a 3D model displaying the relevant portions of the eye. This mesh will also be created using the Cubit software and will be solved using the deall.II programs. Finally, the simulations will be run in a parallel environment on the Darter Cray XC30 system found at the Oak Ridge National Laboratory. This supercomputer has a peak performance of 250 TeraFLOPS (10^{12} floating point operations per second).

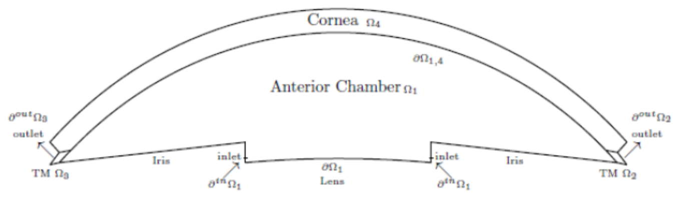

Figure 2: 2D model of the human eye[3]

Parameter	Value
Initial Velocity (v_0)	1.2 $^{mm}/_{s}$
Outlet Pressure (p_0)	1200 Pa
Reference Temperature (T_{ref})	22 °C
Aqueous Humor Density (ρ)	1000$^{kg}/_{m^3}$ (water property)
Aqueous Humor Viscosity (μ)	0.001$^{kg}/_{m \cdot s}$ (water property)
Aqueous Humor Specific Heat (C_p)	4182$^{J}/_{kg \cdot K}$ (water property)
Aqueous Humor Thermal Conductivity (k)	0.6$^{W}/_{m \cdot K}$
Glucose Concentration (g_c)	99.1001$^{mg}/_{dL}$ (healthy eye)
	144.1456$^{mg}/_{dL}$ (type 2 diabetic eye)

4 Results

A 2D Cubit-generated mesh representing the anterior chamber portion of the eye was imported into the FEM solver. The simulation was run using a low Reynolds number (Re=1.2), 0.12 as the inlet velocity, and 9.4 as the outlet pressure. The results were output in a .vtk format in order to view them using VisIt software. It is important to note that these initial results do not include flow out of the trabecular meshwork. Instead, small regions near the edges of the anterior chamber were selected as the outflow area.

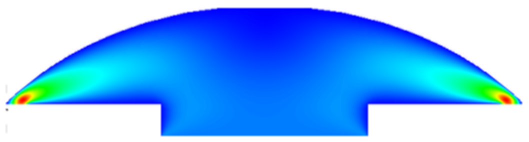

Figure 3: normal eye: 2D velocity

The effects of a higher Reynolds number were also assessed. Figure 4 shows the results of flow using the same values for velocity and pressure, but with Re = 2.5

Both a low resolution and a high resolution 3D mesh was generated in Cubit. Boundary conditions were set up to be consistent with those established in the 2D simulations. The same initial conditions and Reynolds number were also used for the simulations. Initial 3D simulations were run using the low resolution mesh due to computational complexity of

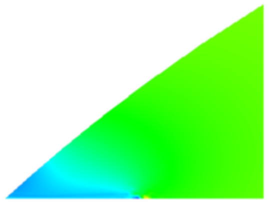
Figure 4: normal eye: 2D pressure

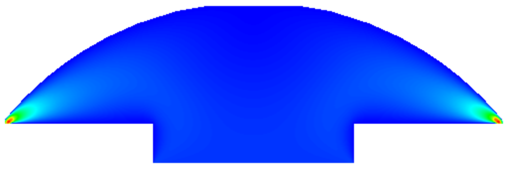
Figure 5: normal eye: 2D velocity (Re=2.5)

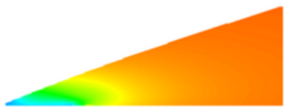
Figure 6: normal eye: 2D pressure (Re=2.5)

simulating 3D fluid flow. The results can be seen in Figure 10. In order to obtain more accurate results, a simulation was run using the more refined mesh. The results can be seen in Figure 7, 8 and 9.

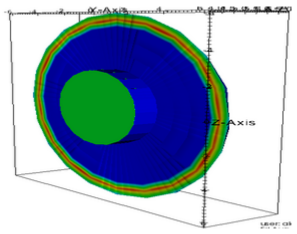

Figure 7: normal eye: 3D velocity

The 1D Laplace solver was run using 5 global elements with the values of u at boundary nodes 0 and 4 being 0 and 100, respectively. The expected values of the inner 3 nodes are 25, 50, and 75, which were the results obtained by our 1D Laplace solver. The output of the code can be seen below.

```
Lapalce 1D Solver output with 5 global elements and 3 processors:

Epetra::CrsMatrix
Number of Global Rows     = 5
```

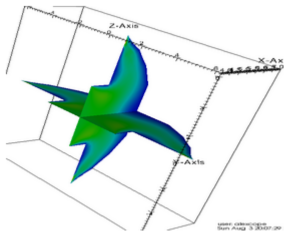

Figure 8: normal eye: 3D velocity (sliced)

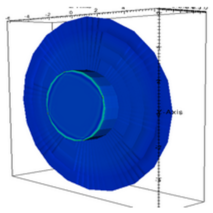

Figure 9: normal eye: 3D relative pressure

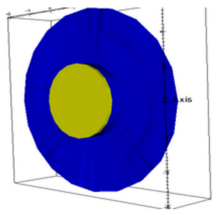

Figure 10: normal eye: 3D velocity (low resolution)

```
Number of Global Cols     = 5
Number of Global Diagonals = 5
Number of Global Nonzeros = 13
Global Maximum Num Entries = 3

Number of My Rows      = 2
Number of My Cols      = 3
Number of My Diagonals = 2
Number of My Nonzeros  = 5
My Maximum Num Entries = 3

Epetra::CrsMatrix
Number of My Rows      = 2
Number of My Cols      = 4
Number of My Diagonals = 2
Number of My Nonzeros  = 6
My Maximum Num Entries = 3
```

```
Epetra::CrsMatrix
Number of My Rows      = 1
Number of My Cols      = 2
Number of My Diagonals = 1
Number of My Nonzeros  = 2
My Maximum Num Entries = 2

       Processor     Row Index     Col Index     Value
       0                    0                    0                    1
       0                    0                    1                    0
       0                    1                    0                    -1
       0                    1                    1                    2
       0                    1                    2                    -1
       1                    2                    2                    2
       1                    2                    3                    -1
       1                    2                    1                    -1
       1                    3                    2                    -1
       1                    3                    3                    2
       1                    3                    4                    -1
       2                    4                    4                    1
       2                    4                    3                    0

              *******************************************************
              ***** Problem: Epetra::CrsMatrix
              ***** Preconditioned GMRES solution
              ***** Order 1 Neumann series polynomial
              ***** No scaling
              *******************************************************

              iter:    0                   residual = 1.000000e+00
              iter:    1                   residual = 4.515050e-01
              iter:    2                   residual = 1.814261e-01
              iter:    3                   residual = 3.162662e-04
              iter:    4                   residual = 1.048770e-35


              Solution time: 0.000562 (sec.)
              total iterations: 4
Solved x: Epetra::Vector     MyPID          GID            Value
                             0              0              0
                             0              1              25
Solved x: Epetra::Vector     1              2              50
                             1              3              75
Solved x: Epetra::Vector     2              4              100
```

A 2D geometry of the anterior chamber area of human eye is created within COMSOL software, then boundary conditions are added on different regions/sections. Inflow is set to be the from the pupil area, and outflow (without trabecular meshwork area) is set at the end points of anterior chamber. Mesh generation and calculation can also be performed within the software. The results are shown below:

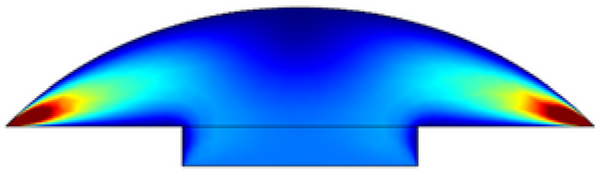

Figure 11: normal eye: 2D velocity

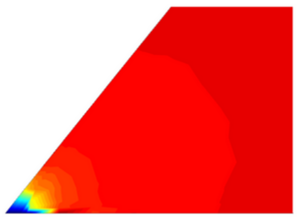

Figure 12: normal eye: 2D pressure

From the graphs, we can note that velocity is fairly consistent throughout the anterior chamber region, and the dramatic change in outflow regions is due to their relative small sizes. The intraocular pressure also remains consistent throughout the whole region except the changes that occur near the outflow area.

Since the results of 2D simulation fit the expectations, we expand the simulation onto 2D axis-symmetric model. A geometry that resembles half of the anterior chamber, along with the addition of trabecular meshwork and Schlemms canal, is created. In addition to the application of standard Navier-Stokes equation, Darcys law is also applied on the Schlemms canal area. Then simulation in 2D was performed on the model and result can be displayed in the 3D geometry created by rotating the model around the axis (center of anterior chamber). The results from 2D axis-symmetric model are shown below:

The results from 2D and 2D axis-symmetry are consistent, with relative consistent speed and pressure throughout anterior chamber region and dramatic changes in outflow region. However, the Darcys law seems to have little to no effect on intraocular pressure. Although we were unable to figure out the reasons behind it, we manually modified the inflow pressure to perform simulations to observe the fluid flow in glaucoma eye.

And a rough 3D model also shows similar result for velocity.

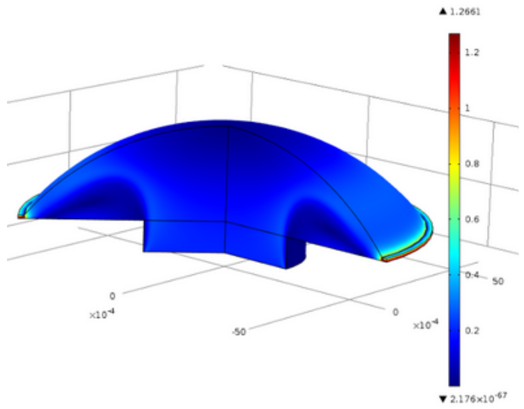

Figure 13: normal eye: 2D axis symmetry velocity

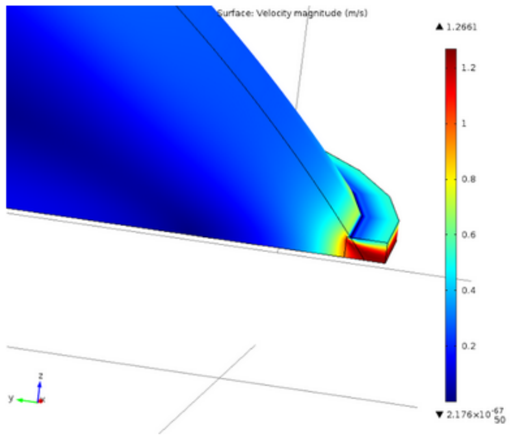

Figure 14: normal eye: 2D axis symmetry velocity (detail)

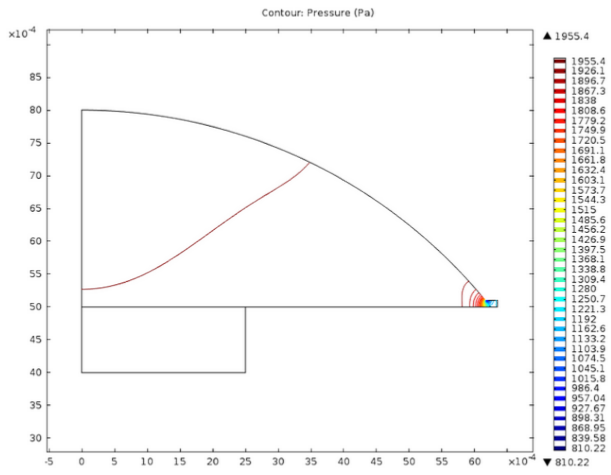

Figure 15: normal eye: 2D axis symmetry pressure

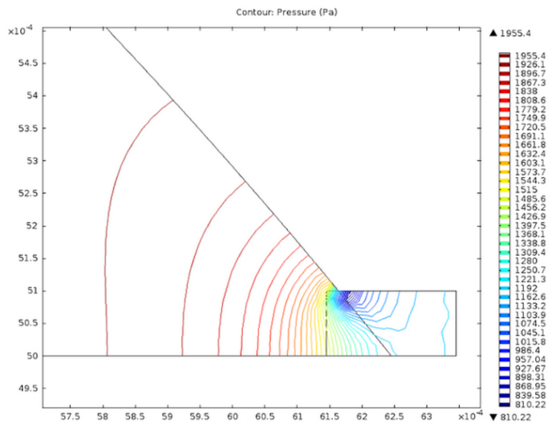
Figure 16: normal eye: 2D axis symmetry pressure (detail)

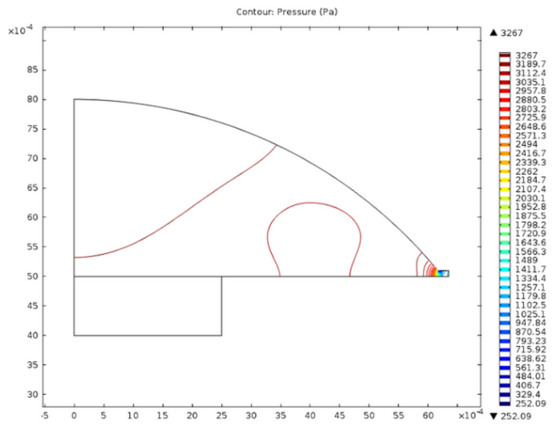
Figure 17: glaucoma eye: 2D axis symmetry pressure

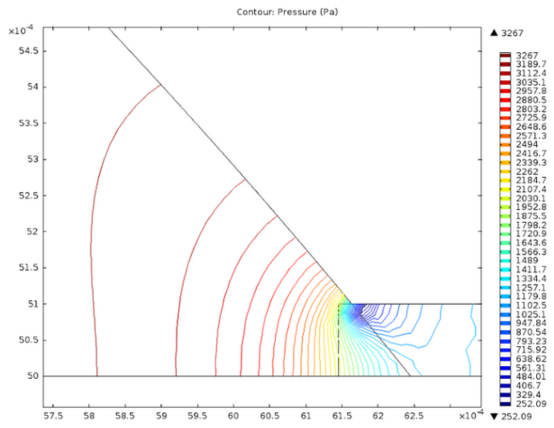
Figure 18: glaucoma eye: 2D axis symmetry pressure (detail)

5 Conclusions and Future Work

When comparing 2D simulations performed in COMSOL and Deal.II, the flow of the aqueous humor appears to be relatively consistent. The velocity stays relatively constant throughout the anterior chamber except when it begins to approach the outlet boundaries. At this point, a dramatic increase in velocity is seen. The pressure results are consistent in the sense that a decrease in pressure is seen close to the outlet regions, but it appears that COMSOL indicates the pressure is higher in the anterior chamber than Deal.II. The cause of these inconsistencies in pressure will require further investigation.

One thing that became apparent rather quickly was the effect of using a low resolution versus a high resolution mesh in Deal.II. Due to the lengthy time of 3D simulations, low resolution meshes were initially used. This allowed the 3D simulations to complete within approximately 30 to 40 minutes. However, compared to simulations run in COMSOL, these results were quite inaccurate. When a simulation was run using a slightly more refined mesh, the velocity plot was much more consistent with 3D results in COMSOL. The tradeoff was this more refined mesh required a significant amount of time to complete compared to the 2D simulations. Ultimately, it was determined more accurate results were more important than the time lost by using a refined mesh. The issue of lengthy run times is not unsolvable. Deal.II programs can run in parallel using Trilinos and MPI, barring that Deal.II is linked to these packages during the build process. If the sparse matrices and vectors can be distributed amongst several processors, the run time of the 3D simulations will be reduced significantly.

At this point, we have mainly modeled fluid flow within the anterior chamber using the standard Navier-Stokes equations. A region was added in COMSOL meant to represent the trabecular meshwork, but the results from these simulations did not result in any changes to the flow of aqueous humor. These results were inconsistent with the results of Ferreira et al., meaning further modifications will need to be made to the current COMSOL model.

The 1D Laplace parallel solver is at the point where it can be expanded to 2D and 3D. This will require numerous extra features, although the general structure of the code is the same. While a 1D Laplace problem can only be applied over a set of nodes distributed over a line, a 2D/3D Laplace problem could be distributed over a variety of meshes. A function will need to be created to read in the file and establish the offsets and column locations. It will also require a more complicated local matrix setup as the values will vary from local element to local element.

References

[1] C.R. Canning, *Fluid Flow in the Anterior Chamber of a Human Eye.* Mathematical Medicine and Biology, 19.1, 31-60, 2002

[2] T.R.Crowder and V.J. Ervin, *Numerical Simulations of Fluid Pressure in the Human Eye.* Applied Mathematics and Computation, 219.24, 11119-11133, 2013.

[3] J.A. Ferreira, P. de Oliveira, P.M. da Silva and J.N. Murta *Numerical Simulation of Aqueous Humor Flow: from Healthy to Pathologic Situations*. Applied Mathematics and Computation, 226, 777-792, 1994.

[4] A.D. Fitt and G. Gonzalez, *Fluid Mechanics of the Human Eye: Aqueous Humour Flow in the Anterior Chamber*. Bulletin of Mathematical Biology, 68.1, 53-71, 2006.

[5] J.J. Heys, V.H. Barocas and M.J. Taravella, *Modeling Passive Mechanical Interaction between Aqueous Humor and Iris*. Journal of Biomechanical Engineering, 123.6, 540, 2001

[6] S. Roy, R. Kao and T. Sato, *Effect of High Glucose on Fibronectin Expression and Cell Proliferation in Trabecular Meshwork Cells*. Investigative Ophthalmology and Visual Science, 43.1, 170-175, 2002.

[7] A. Villamarin, S. Roy, R. Hasballa, O. Vardoulis, P. Reymond and N. Stergiopulos, *3D Simulation of the Aqueous Flow in the Human Eye*. Medical Engineering and Physics, 34.10, 1462-1470, 2012.

A Appendix

Deal.II 2D Navier-Stokes Solver:

```
/* ---------------------------------------------------------------------
 * $Id: step-35.cc 30526 2013-08-29 20:06:27Z felix.gruber $
 *
 * 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
 * Modified by Alexander Cope, Centre College 2014, CSURE 2014, to
 * simulate fluid flow within the eye, assuming uniform temperature.
 */
```

```
// @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 <stdio.h>
#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>

// Finally this is as in all previous programs:
namespace Step35
{
  using namespace dealii;



  // @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;
}


// 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;
};


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]);

    }
}
```

```
template <int dim>
double Velocity<dim>::value (const Point<dim> &p,
                             const unsigned int) const
{
  if (this->comp == 0)
    {
      //const double Um = 1.5;
      //const double H = 4.1;
      //return 4.*Um*p(1)*(H - p(1))/(H*H);
      return 1.2;

    }
  else
    return 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)
{}


template <int dim>
double Pressure<dim>::value (const Point<dim> &p,
                             const unsigned int) const
{
    return 9.4;
  }

  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;
  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> 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 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_cxx1x::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_one_cell_of_gradient (const IteratorPair &SI,
                                    InitGradScratchData &scratch,
                                    InitGradPerTaskData &data);

void copy_gradient_local_to_global (const InitGradPerTaskData &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()),
      dpc (fe.dofs_per_cell),
      u_star_local (nqp),
      grad_u_star (nqp),
      u_star_tmp (nqp),
      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 &cell,
                                       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);

  void output_results (const unsigned int step);

  void assemble_vorticity (const bool reinit_prec);
};
```

```
// @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),
  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));

  create_triangulation_and_dofs (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>::
create_triangulation_and_dofs (const unsigned int n_refines)
{
  GridIn<dim> grid_in;
  grid_in.attach_triangulation (triangulation);

  {
    std::string filename = "output_BC2.ucd";
    std::ifstream file (filename.c_str());
    Assert (file, ExcFileNotOpen (filename.c_str()));
    grid_in.read_ucd (file);
  }

  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;



  //The following code was added for our specific mesh. It sets the inlet and
  //outlet boundary indicators. Everything else is treated as boundary
  //indicator 0.
  for (typename Triangulation<dim>::active_cell_iterator cell = triangulation.
      begin_active(); cell != triangulation.end(); ++cell)
    {
    for (unsigned int f=0; f<GeometryInfo<dim>::faces_per_cell; ++f)
      {
        if (cell->face(f)->at_boundary())
          {
            if (cell->face(f)->center()[0] == 4)
              {
                cell->face(f)->set_boundary_indicator (1);
              }
            else if (cell->face(f)->center()[0] == 5 &&
                ((cell->face(f)->center()[1]>5.7 &&
                cell->face(f)->center()[1]<6.24) ||(
                cell->face(f)->center()[1]<-5.7 &&
                cell->face(f)->center()[1]>-6.24)))
            // else if ((cell->face(f)->center()[1]==6.0 &&
                cell->face(f)->center()[0]>=5 && cell->face(f)->center()[0]<5.2)
                || (cell->face(f)->center()[1]==-6.0 &&
                cell->face(f)->center()[0]>=5 && cell->face(f)->center()[0]<5.2))
                     {
                       cell->face(f)->set_boundary_indicator (2);
                     }
          }
      }
    }

  boundary_indicators = triangulation.get_boundary_indicators();
  printf("Number of boundaries: %lu\n",boundary_indicators.size());

  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);

  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());
  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());
  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());
    }
  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, ZeroFunction<dim>(),
          u_n_minus_1[d]);
      vel_exact.advance_time (dt);
      VectorTools::interpolate (dof_handler_velocity, ZeroFunction<dim>(),
          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);

  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_cxx1x::get<0> (SI.iterators));
  scratch.fe_val_pres.reinit (std_cxx1x::get<1> (SI.iterators));

  std_cxx1x::get<0> (SI.iterators)->get_dof_indices
      (data.vel_local_dof_indices);
  std_cxx1x::get<1> (SI.iterators)->get_dof_indices
      (data.pres_local_dof_indices);

  data.local_grad = 0.;
  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) );
}


// @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);
  vel_exact.set_time (2.*dt);
  output_results(1);
  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));
      verbose_cout << " Projection Step" << std::endl;
      projection_step ( (n == 2));
      verbose_cout << " Updating the Pressure" << std::endl;
      update_pressure ( (n == 2));
      vel_exact.advance_time(dt);
    }
  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();

  for (unsigned int d=0; d<dim; ++d)
    {
      force[d] = 0.;
      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);

      vel_exact.set_component(d);
      boundary_values.clear();
      for (std::vector<types::boundary_id>::const_iterator
       boundaries = boundary_indicators.begin();
            boundaries != boundary_indicators.end();
           ++boundaries)
         {
         switch (*boundaries)
           {
           case 0:
              VectorTools::
              interpolate_boundary_values (dof_handler_velocity,
                                           *boundaries,
                                           ZeroFunction<dim>(),
                                           boundary_values);
                   break;
            case 1:
              VectorTools::
              interpolate_boundary_values (dof_handler_velocity,
                                           *boundaries,
                                           vel_exact,
                                          boundary_values);
              break;
            case 2:
              if (d != 0)
                VectorTools::
                interpolate_boundary_values (dof_handler_velocity,
                                             *boundaries,
                                             ZeroFunction<dim>(),
                                             boundary_values);
              break;

            default:
                      Assert (false, ExcNotImplemented());
                              }
                      }
      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;
  if (reinit_prec)
    VectorTools::interpolate_boundary_values (dof_handler_pressure, 2,
                                              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);
  data_out.build_patches (deg + 1);
  std::ofstream output (("solution_eye-" +
                        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;
```

```
}
```

1D Laplace Solver:

```
/* Author: Alex Cope
   Date: July 17th, 2014
   Compiler: Compiled on darter using CC wrapper
   Solves 1D Laplace Problem
*/

#include <cstdlib>
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <cmath>

#include "AztecOO_config.h"
#include "Epetra_ConfigDefs.h"
#include "Epetra_MpiComm.h"
#include "Epetra_Map.h"
#include "Epetra_Vector.h"
#include "Epetra_CrsMatrix.h"
#include "AztecOO.h"
#include "mpi.h"

using namespace std;

int main(int argc, char* argv[])
{
  int n=5; //number of global elements
  int offset[n]; //establishes the offset, ie. how many nonzero values per row
  for (int i=0;i<n;++i)
    {
      if (i==0 || i==(n-1)) //last two rows will have one nonzero values
                {
                  offset[i]=2;
                }
      else //all other rows will have three nonzero values
                {
                  offset[i]=3;
                }
    }
```

```
//Setup MPI and EPETRA communication

MPI_Init(&argc, &argv);
Epetra_MpiComm Comm(MPI_COMM_WORLD);


//Establish map with n global elements
int NumGlobalElements=n;

int NumProc=Comm.NumProc();


//This algorithm only works if the ((NumGlobalElements/2)+1==NumProc.
//The following if-statement checks to make sure this is true.
//If not, the code exits.

if ((NumGlobalElements%2==0 && NumProc!=(NumGlobalElements/2)) ||
    (NumGlobalElements%2!=0 && NumProc!=((NumGlobalElements/2)+1)) )
  {
    std::cout<<"This program requires that the number of procceses be equal to
       "<<std::endl;
    exit(1);
  }
int MyPID=Comm.MyPID();
int* MyGlobalElements;
int MyElements;
if (NumGlobalElements%2!=0)
  {
    //If the number of global elements is odd,
    //then the last process will only have one row
    if (MyPID==NumProc-1)
      {
        MyElements=1;
        MyGlobalElements=new int[MyElements];
        MyGlobalElements[0]=NumGlobalElements-1;
      }
    else
      {
        MyElements=2;
        MyGlobalElements=new int[MyElements];
        MyGlobalElements[0]=MyPID*2;
        MyGlobalElements[1]=MyPID*2+1;
      }
  }
else
  {
    MyElements=2;
    MyGlobalElements=new int[MyElements];
    MyGlobalElements[0]=MyPID*2;
    MyGlobalElements[1]=MyPID*2+1;
  }

Epetra_Map Map(-1, MyElements, MyGlobalElements,0,Comm);


int NumMyElements=Map.NumMyElements();

//Establish vectors b (right-hand side vector) and x (solution vector)

Epetra_Vector x(Map);
Epetra_Vector b(Map);

if (MyGlobalElements[NumMyElements-1]==NumGlobalElements-1)
  {
    double value=100.0;
    int indices=NumGlobalElements-1;
    b.ReplaceGlobalValues(1,&value,&indices);
  }


//Setup sparse matrix A using mapping and offset
Epetra_CrsMatrix A(Copy,Map,offset);


//This establishes the number of nonzero values in this process
int sum=0;
for (int i=0;i<NumMyElements;++i)
  {
    sum=sum+offset[MyGlobalElements[i]];
  }

double* aij=new double[sum];

//This is the local matrix for the 1D Laplace problem.
double local_mat[4]={1.0,-1.0,-1.0,1.0};


int local_mat_index;
```

```
//This part is a little tricky. The basic idea is with the exception of the
//first process, the first element in the aij value will be -1, which
//corresponds to local_mat[2]
if (MyGlobalElements[0]==0)
  {
    local_mat_index=0;
  }
else
  {
    local_mat_index=2;
  }

//Establishes nonzero values found in the rows the controlled by this process
int cur_row=MyGlobalElements[0];
for (int j=0;j<sum;++j)
 {
   if (local_mat_index==1)
     {
       cur_row++;
     }
   aij[j]=aij[j]+local_mat[local_mat_index];

   ++local_mat_index;
   if (local_mat_index==4 && cur_row!=(NumGlobalElements-1))
     {
       local_mat_index=0;
       --j;
     }
 }

//Sets up the column indices for the nonzero values found in aij.

int col_loc[sum];
int off;
int col_loc_index;
int tmp=0;
for (int i=0;i<NumMyElements;++i)
  {
    col_loc_index=MyGlobalElements[i];
    off=offset[col_loc_index];

    if (col_loc_index!=0)
      --col_loc_index;
    for (int j=tmp;j<tmp+off;++j)
```

```
      {
        col_loc[j]=col_loc_index;
        col_loc_index++;
      }
    tmp=off;

  }

//Copies the necessary aij and col_loc values and puts them into the
//global matrix.

int tmp2=0;
for (int i=0;i<NumMyElements;++i)
  {
    off=offset[MyGlobalElements[i]];
    double aij_tmp[off];
    int col_loc_tmp[off];
    for (int j=tmp2;j<off+tmp2;++j)
      {
        aij_tmp[j-tmp2]=aij[j];
        col_loc_tmp[j-tmp2]=col_loc[j];
      }
    A.InsertGlobalValues(MyGlobalElements[i],off,aij_tmp,col_loc_tmp);
    tmp2=off;
  }


//Create boudary conditions
if (MyGlobalElements[0]==0)
  {
    double bcd;
    int bcd_index;
    bcd=0.0;
    bcd_index=1;
    A.ReplaceGlobalValues(0,1,&bcd,&bcd_index);
  }

else if (MyGlobalElements[NumMyElements-1]==(NumGlobalElements-1))
  {
    double bcd;
    int bcd_index;
    bcd=0.0;
    bcd_index=n-2;
    A.ReplaceGlobalValues(NumGlobalElements-1,1,&bcd,&bcd_index);
  }
  //Finalize sparse matrix A
  A.FillComplete();
  std::cout<<A<<std::endl;


  // Set up linear problem and solve. Print out x with solution to problem
  Epetra_LinearProblem problem(&A,&x,&b);
  AztecOO solver(problem);
  solver.SetAztecOption(AZ_precond, AZ_Neumann);
  solver.Iterate(100,1.0E-8);

  std::cout<<"Solved x: "<<x<<std::endl;



  MPI_Finalize();
  return 0;
}
```