Chapter 1

Demo problem: A two-dimensional Poisson problem

In this document, we demonstrate how to solve a 2D Poisson problem using existing objects from the `oomph-lib` library:

Two-dimensional model Poisson problem

Solve

$$\sum_{i=1}^{2} \frac{\partial^2 u}{\partial x_i^2} = f(x_1, x_2), \qquad (1)$$

in the rectangular domain $D = \{(x_1, x_2) \in [0, 1] \times [0, 2]\}$, with Dirichlet boundary conditions

$$u|_{\partial D} = u_0 \qquad (2)$$

where the function u_0 is given.

We provide a detailed discussion of the driver code `two_d_poisson.cc` which solves the problem for

$$u_0(x_1, x_2) = \tanh(1 - \alpha(x_1 \tan\Phi - x_2)), \qquad (3)$$

and

$$f(x_1, x_2) = \sum_{i=1}^{2} \frac{\partial^2 u_0}{\partial x_i^2}, \qquad (4)$$

so that $u_0(x_1, x_2)$ represents the exact solution of the problem. For large values of α the solution approaches a step function

$$u_{step}(x_1, x_2) = \begin{cases} -1 & \text{for } x_2 < x_1 \tan\Phi \\ 1 & \text{for } x_2 > x_1 \tan\Phi \end{cases}$$

which presents a serious challenge for any numerical method. The figure below compares the numerical and exact solutions for $\alpha = 1$ and $\Phi = 45^o$.

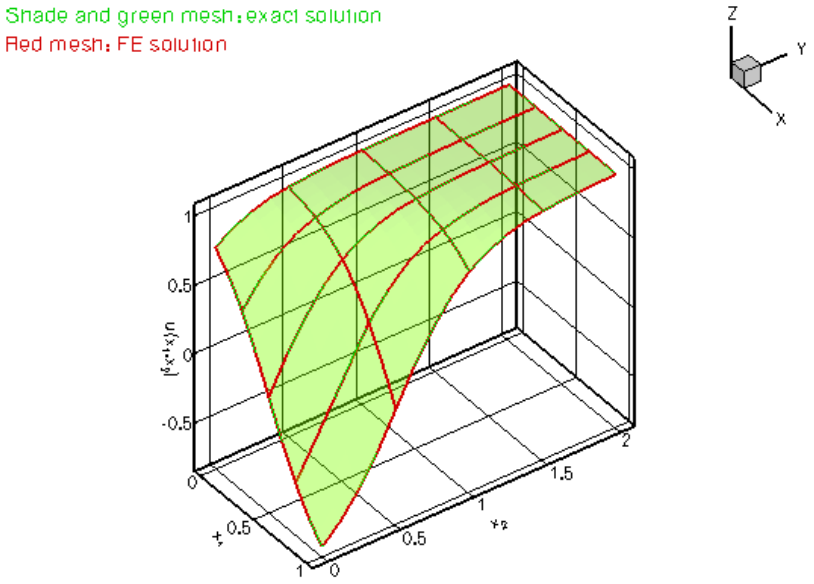

Figure 1.1 Plot of the solution

1.1 Global parameters and functions

Following our usual practice, we use a namespace, `TanhSolnForPoisson`, to define the source function (4) and the exact solution (3). Both functions permit arbitrary values of the tangent $\tan\Phi$ and the steepness parameter α, which are stored in TanhSolnForPoisson::TanPhi and TanhSolnForPoisson::Alpha, respectively, so that the "user" can set their values from the driver code.

```
//===== start_of_namespace=============================================
/// Namespace for exact solution for Poisson equation with "sharp step"
//======================================================================
namespace TanhSolnForPoisson
{

 /// Parameter for steepness of "step"
 double Alpha=1.0;

 /// Parameter for angle Phi of "step"
 double TanPhi=0.0;

 /// Exact solution as a Vector
 void get_exact_u(const Vector<double>& x, Vector<double>& u)
 {
  u[0]=tanh(1.0-Alpha*(TanPhi*x[0]-x[1]));
 }

 /// Source function required to make the solution above an exact solution
 void source_function(const Vector<double>& x, double& source)
 {
  source = 2.0*tanh(-1.0+Alpha*(TanPhi*x[0]-x[1]))*
    (1.0-pow(tanh(-1.0+Alpha*(TanPhi*x[0]-x[1])),2.0))*
    Alpha*Alpha*TanPhi*TanPhi+2.0*tanh(-1.0+Alpha*(TanPhi*x[0]-x[1]))*
    (1.0-pow(tanh(-1.0+Alpha*(TanPhi*x[0]-x[1])),2.0))*Alpha*Alpha;
 }
} // end of namespace
```

1.2 The driver code

In order to solve the 2D Poisson problem using `oomph-lib`, we represent the mathematical problem defined by equations (1) and (2) in a specific `Problem` object, `PoissonProblem`. `oomph-lib` provides a variety of 2D Poisson elements (e.g. 2D quadrilateral elements with bi-linear, bi-quadratic and bi-cubic representations for the unknown function) and we pass the specific element type as a template parameter to the `Problem`. In the driver code, listed below, we use the `QPoissonElement<2,3>`, a nine-node (bi-quadratic) 2D Poisson element.

The next few lines of the `main()` function create a `DocInfo` object – an `oomph-lib` object that collates various items of data that can be used to label output files: Here we specify that the output files are to be written to the directory "RESLT", and that the first batch of output files should be labelled with the identifier "0". See the discussion of the postprocessing routine `doc_solution(...)` for details. [**Note:** While the ability to specify an output directory from the driver code is useful, it does rely on the "user" having created the directory before the code is executed. We could use the C++ `system(...)` function to issue a system command which creates the directory if it does not exist. Since this would make the code non-portable, we only issue a warning suggesting the likely cause of the problem if the output file cannot be opened. If you want to make absolutely sure that the output directory does exist and can be written to, you can change this forgiving behaviour with the function `DocInfo::directory_must_exist()`. This function provides access to a boolean flag which is set to `false` by default. If set to `true`, the code execution terminates with `assert(false)` if the directory specified with `DocInfo::set_directory(...)` cannot be written to.]

Next we execute the `Problem::self_test()` function to check whether the `Problem` has been correctly initialised. If this test is passed, we proceed to the solution. We choose the angle of the "step" as 45 degrees (corresponding to $\tan\Phi = 1$) and then solve the problem for a number of values of the steepness parameter α. We document each solution with the post-processing routine `doc_solution(...)` which accesses the step number and the output directory via the `DocInfo` object.

```
//===== start_of_main=====================================================
/// Driver code for 2D Poisson problem
//========================================================================
int main()
{

 //Set up the problem
 //------------------

 // Create the problem with 2D nine-node elements from the
 // QPoissonElement family. Pass pointer to source function.
 PoissonProblem<QPoissonElement<2,3> >
  problem(&TanhSolnForPoisson::source_function);

 // Create label for output
 //------------------------
 DocInfo doc_info;

 // Set output directory
 doc_info.set_directory("RESLT");

 // Step number
 doc_info.number()=0;

 // Check that we're ready to go:
 //------------------------------
 cout << "\n\n\nProblem self-test ";
 if (problem.self_test()==0)
  {
   cout << "passed: Problem can be solved." << std::endl;
  }
 else
  {
   throw OomphLibError("Self test failed",
                       OOMPH_CURRENT_FUNCTION,
                       OOMPH_EXCEPTION_LOCATION);
  }

 // Set the orientation of the "step" to 45 degrees
 TanhSolnForPoisson::TanPhi=1.0;
 // Initial value for the steepness of the "step"
 TanhSolnForPoisson::Alpha=1.0;


 // Do a couple of solutions for different forcing functions
 //---------------------------------------------------------
 unsigned nstep=4;
 for (unsigned istep=0;istep<nstep;istep++)
  {

   // Increase the steepness of the step:
   TanhSolnForPoisson::Alpha+=1.0;

   cout << "\n\nSolving for TanhSolnForPoisson::Alpha="
        << TanhSolnForPoisson::Alpha << std::endl << std::endl;

   // Solve the problem
   problem.newton_solve();

   //Output the solution
   problem.doc_solution(doc_info);
   //Increment counter for solutions
```

```
   doc_info.number()++;
  }


} //end of main
```

1.3 The problem class

The PoissonProblem is derived from oomph-lib's generic Problem class and the specific element type is specified as a template parameter to make it easy for the "user" to change the element type from the driver code.

```
//====== start_of_problem_class=====================================
/// 2D Poisson problem on rectangular domain, discretised with
/// 2D QPoisson elements. The specific type of element is
/// specified via the template parameter.
//==================================================================
template<class ELEMENT>
class PoissonProblem : public Problem
```

The problem class has five member functions, only three of which are non-trivial:

- the constructor PoissonProblem(...)
- the function actions_before_newton_solve()
- the function doc_solution(...)

The function Problem::actions_after_newton_solve() is a pure virtual member function of the Problem base class and must be provided. However, it is not required in the present problem and we leave it empty. Similarly, the problem destructor can remain empty as all memory de-allocation is handled in the destructor of the Problem base class. The Problem only stores one private data member, the pointer to the source function.

```
public:

 /// Constructor: Pass pointer to source function
 PoissonProblem(PoissonEquations<2>::PoissonSourceFctPt source_fct_pt);

 /// Destructor (empty)
 ~PoissonProblem(){}

 /// Update the problem specs before solve: Reset boundary conditions
 /// to the values from the exact solution.
 void actions_before_newton_solve();

 /// Update the problem after solve (empty)
 void actions_after_newton_solve(){}

 /// Doc the solution. DocInfo object stores flags/labels for where the
 /// output gets written to
 void doc_solution(DocInfo& doc_info);

private:

 /// Pointer to source function
 PoissonEquations<2>::PoissonSourceFctPt Source_fct_pt;

}; // end of problem class
```

[See the discussion of the 1D Poisson problem for a more detailed discussion of the function type PoissonEquations<2>::PoissonSourceFctPt.]

1.4 The Problem constructor

In the Problem constructor, we start by discretising the rectangular domain, using oomph-lib's SimpleRectangularQuadMesh object. The arguments of this object's constructor are the number of elements (whose type is specified by the template parameter), and the domain lengths in the x_1 and x_2 directions, respectively.
The subsequent lines of code pin the nodal values along the entire domain boundary. In the 1D example considered earlier, the identification of the nodes on the domain boundaries was trivial. In higher-dimensional problems, this task can become rather involved. oomph-lib's Mesh base class provides the helper function Mesh::boundary_node_pt(...) giving (pointer-based) access to nodes on specified mesh boundaries. [The total number of boundaries can be obtained from Mesh::nboundary(), while the number of nodes on a specific boundary is available from Mesh::nboundary_node(...).] The nested loops over the mesh boundaries and the nodes on these boundaries therefore provide a convenient and completely generic method of accessing all boundary nodes.
Finally we loop over all elements to assign the source function pointer, and then call the generic Problem::assign_eqn_numbers() routine to set up the equation numbers.

```
//=====start_of_constructor=======================================
/// Constructor for Poisson problem: Pass pointer to source function.
//================================================================
template<class ELEMENT>
PoissonProblem<ELEMENT>::
      PoissonProblem(PoissonEquations<2>::PoissonSourceFctPt source_fct_pt)
       :  Source_fct_pt(source_fct_pt)
{
 // Setup mesh

 // # of elements in x-direction
 unsigned n_x=4;

 // # of elements in y-direction
 unsigned n_y=4;

 // Domain length in x-direction
 double l_x=1.0;

 // Domain length in y-direction
 double l_y=2.0;

 // Build and assign mesh
 Problem::mesh_pt() = new SimpleRectangularQuadMesh<ELEMENT>(n_x,n_y,l_x,l_y);

 // Set the boundary conditions for this problem: All nodes are
 // free by default -- only need to pin the ones that have Dirichlet conditions
 // here.
 unsigned n_bound = mesh_pt()->nboundary();
 for(unsigned i=0;i<n_bound;i++)
  {
   unsigned n_node = mesh_pt()->nboundary_node(i);
   for (unsigned n=0;n<n_node;n++)
    {
     mesh_pt()->boundary_node_pt(i,n)->pin(0);
    }
  }

 // Complete the build of all elements so they are fully functional

 // Loop over the elements to set up element-specific
 // things that cannot be handled by the (argument-free!) ELEMENT
 // constructor: Pass pointer to source function
 unsigned n_element = mesh_pt()->nelement();
 for(unsigned i=0;i<n_element;i++)
  {
   // Upcast from GeneralsedElement to the present element
   ELEMENT *el_pt = dynamic_cast<ELEMENT*>(mesh_pt()->element_pt(i));

   //Set the source function pointer
   el_pt->source_fct_pt() = Source_fct_pt;
  }


 // Setup equation numbering scheme
 cout <<"Number of equations: " << assign_eqn_numbers() << std::endl;

} // end of constructor
```

1.5 "Actions before solve"

We use `Problem::actions_before_newton_solve()` to update the boundary conditions in response to possible changes in the problem parameters. We use the exact solution, specified in `TanhSolnForPoisson::get_exact_u(...)`, to determine the boundary values that are appropriate for the current values of α and $\tan\Phi$.

```
//========================================start_of_actions_before_newton_solve===
/// Update the problem specs before solve: (Re-)set boundary conditions
/// to the values from the exact solution.
//================================================================
template<class ELEMENT>
void PoissonProblem<ELEMENT>::actions_before_newton_solve()
{
 // How many boundaries are there?
 unsigned n_bound = mesh_pt()->nboundary();
 //Loop over the boundaries
 for(unsigned i=0;i<n_bound;i++)
  {
   // How many nodes are there on this boundary?
   unsigned n_node = mesh_pt()->nboundary_node(i);

   // Loop over the nodes on boundary
   for (unsigned n=0;n<n_node;n++)
    {
```

```
     // Get pointer to node
     Node* nod_pt=mesh_pt()->boundary_node_pt(i,n);

     // Extract nodal coordinates from node:
     Vector<double> x(2);
     x[0]=nod_pt->x(0);
     x[1]=nod_pt->x(1);

     // Compute the value of the exact solution at the nodal point
     Vector<double> u(1);
     TanhSolnForPoisson::get_exact_u(x,u);

     // Assign the value to the one (and only) nodal value at this node
     nod_pt->set_value(0,u[0]);
    }
  }
} // end of actions before solve
```

[See the discussion of the 1D Poisson problem for a more detailed discussion of the pure virtual functions `Problem::actions_before_newton_solve()` and `Problem::actions_after_newton_solve()`.]

1.6 Post-processing

The function `doc_solution(...)` writes the FE solution and the corresponding exact solution, defined in `TanhSolnForPoisson::get_exact_u(...)` to disk. The `DocInfo` object specifies the output directory and the label for the file names. [See the discussion of the 1D Poisson problem for a more detailed discussion of the generic `Mesh` member functions `Mesh::output(...)`, `Mesh::output_fct(...)` and `Mesh::compute_error(...)`].

```
//==============start_of_doc=====================================================
/// Doc the solution: doc_info contains labels/output directory etc.
//================================================================================
template<class ELEMENT>
void PoissonProblem<ELEMENT>::doc_solution(DocInfo& doc_info)
{

 ofstream some_file;
 char filename[100];

 // Number of plot points: npts x npts
 unsigned npts=5;

 // Output solution
 //-----------------
 sprintf(filename,"%s/soln%i.dat",doc_info.directory().c_str(),
         doc_info.number());
 some_file.open(filename);
 mesh_pt()->output(some_file,npts);
 some_file.close();


 // Output exact solution
 //----------------------
 sprintf(filename,"%s/exact_soln%i.dat",doc_info.directory().c_str(),
         doc_info.number());
 some_file.open(filename);
 mesh_pt()->output_fct(some_file,npts,TanhSolnForPoisson::get_exact_u);
 some_file.close();

 // Doc error and return of the square of the L2 error
 //---------------------------------------------------
 double error,norm;
 sprintf(filename,"%s/error%i.dat",doc_info.directory().c_str(),
         doc_info.number());
 some_file.open(filename);
 mesh_pt()->compute_error(some_file,TanhSolnForPoisson::get_exact_u,
                          error,norm);
 some_file.close();

 // Doc L2 error and norm of solution
 cout << "\nNorm of error   : " << sqrt(error) << std::endl;
 cout << "Norm of solution: " << sqrt(norm) << std::endl << std::endl;

} // end of doc
```

1.7 Comments and exercises

1. In its current form, the number of elements and the dimensions of the domain are hard-coded in the `Problem` constructor. Change the `Problem` constructor so that these quantities become input parameters that can

be set from the `main()` function.

2. Note how the accuracy of the FE solution decreases as the steepness of the "step" is increased:

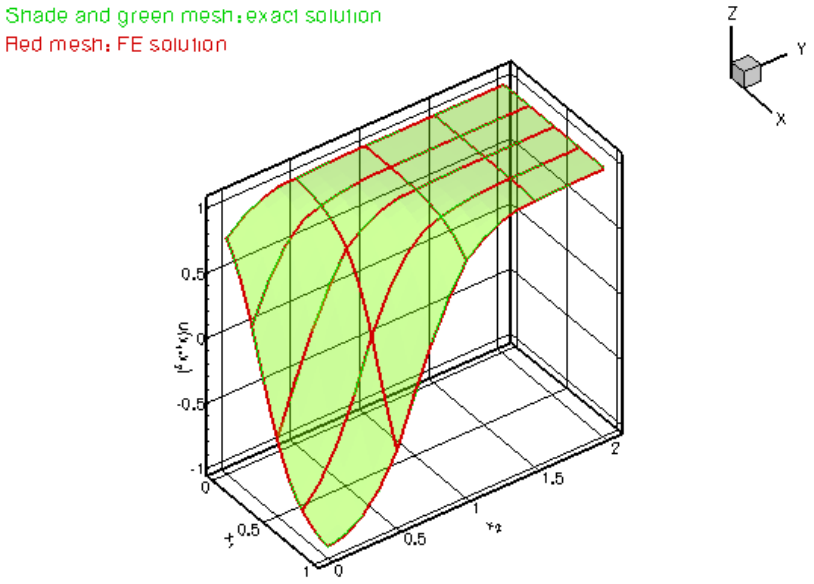

Figure 1.2 Plot of the solution for different values of the steepness parameter

How many elements are required to resolve the solution with $\alpha = 10$ as accurately as in the case of $\alpha = 1$? [**Note:** Since the solution has steep gradients only in a very narrow region, uniform mesh refinement is an extremely wasteful method of improving the accuracy of the computed solution. `oomph-lib` provides powerful mesh adaptation routines which perform fully-automatic mesh refinement and unrefinement, based on a posteriori error estimates of the solution. We will demonstrate these in `another example.`]

3. Repeat the numerical experiments with different element types. Replace the nine-node Poisson element, `QPoissonElement<2,3>`, by its lower- and higher-order counterparts `QPoissonElement<2,2>` and `QPoissonElement<2,4>`, respectively. Compare the total number of degrees of freedom, the errors, and the run-times for the different discretisations.

1.7.1 Header files and precompiled meshes

We have repeatedly stressed that `oomph-lib` `Mesh` objects are (and any user-written ones should be) templated by the element type, so that meshes can be used with all finite elements that are derived from the same geometric element (2D quad elements from the `QElement` family, say). Typically, the element type is specified in the driver code. Consequently, the compiler must instantiate the `Mesh` object for a particular element type when the driver code is compiled – there is no point in trying to "pre-compile" a `Mesh` object for "all possible element types". The source code for `Mesh` objects is therefore usually contained in a single (header) file which must be included in the driver code. The first few lines of the driver code `two_d_poisson.cc` illustrate the technique:

```
//Driver for a simple 2D poisson problem

//Generic routines
#include "generic.h"

// The Poisson equations
#include "poisson.h"

// The mesh
#include "meshes/simple_rectangular_quadmesh.h"
```

The code uses objects from the `generic` and `poisson` libraries whose function prototypes are contained in the header files `generic.h` and `poisson.h`, located in the `oomph-lib include` directory. All objects in these libraries are fully instantiated and no re-compilation is required – we simply link against the libraries which are located in `oomph-lib's lib` directory. The mesh header files (which include the entire source code for each

mesh) are located in the include (sub-)directory `include/meshes`, and are included into the driver code with a C++ include directive

While this strategy greatly facilitates code reuse, it can incur significant compile-time overheads as the (possibly very lengthy) mesh sources must be recompiled whenever the driver code is changed. During code development, this overhead can become unacceptable. To avoid the constant re-compilation of the mesh sources, all `oomph-lib` mesh objects are contained in two separate source files. In the case of the `SimpleRectangularMesh`, the class definition and function prototypes are contained in the small auxiliary header file `simple_rectangular_quadmesh.template.h`, while the actual function definitions are contained in `simple_rectangular_quadmesh.template.cc`. These are the only sources that the mesh-writer has to provide. The header file `simple_rectangular_quadmesh.h` is generated (automatically) by concatenating the two $*$.template.$*$ files and all three files are contained in the mesh include directory. This allows the "user" to pre-compile the mesh for a specific element type (or for a range of specific elements) to produce a separate object file that can be linked against when the driver code is built.

The procedure is illustrated in the alternative source code `two_d_poisson2.cc` and the associated mesh file, `two_d_poisson2_mesh.cc`. In the original version of the code, `two_d_poisson.cc`, the mesh was instantiated with the element type `QPoisson Element<2,3>` and we will assume that this is the only element type required in the driver code. We force the instantiation of the `SimpleRectangularQuadMesh` for this element type by employing the C++ "template" statement in the mesh file, `two_d_poisson2_mesh.cc`, which is listed in its entirety here:

```
// LIC// ====================================================================
// LIC// This file forms part of oomph-lib, the object-oriented,
// LIC// multi-physics finite-element library, available
// LIC// at http://www.oomph-lib.org.
// LIC//
// LIC// Copyright (C) 2006-2022 Matthias Heil and Andrew Hazel
// LIC//
// LIC// This library is free software; you can redistribute it and/or
// LIC// modify it under the terms of the GNU Lesser General Public
// LIC// License as published by the Free Software Foundation; either
// LIC// version 2.1 of the License, or (at your option) any later version.
// LIC//
// LIC// This library is distributed in the hope that it will be useful,
// LIC// but WITHOUT ANY WARRANTY; without even the implied warranty of
// LIC// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
// LIC// Lesser General Public License for more details.
// LIC//
// LIC// You should have received a copy of the GNU Lesser General Public
// LIC// License along with this library; if not, write to the Free Software
// LIC// Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
// LIC// 02110-1301  USA.
// LIC//
// LIC// The authors may be contacted at oomph-lib@maths.man.ac.uk.
// LIC//
// LIC//====================================================================
// Mesh builder: This builds the mesh with a specific element
// so that the mesh sources don't have to be re-compiled over
// and over again in the driver code... Useful strategy if
// the driver code is under development and uses a complicated
// mesh with long compile times

// Include Poisson elements/equations
#include "poisson.h"

// Include the full source for mesh (this is the automatically generated
// combined header file that contains the templated header and the
// templated sources)
#include "meshes/simple_rectangular_quadmesh.h"

namespace oomph
{
  // Force build of specific mesh
  template class SimpleRectangularQuadMesh<QPoissonElement<2, 3»;
} // namespace oomph
```

This source file can be pre-compiled into an object file, `two_d_poisson2_mesh.o`, say.

The driver code only needs to include the templated header file (which contains the class definition and the function prototypes) so that the first few lines of the modified driver code look like this:

```
//Driver for a simple 2D poisson problem

//Generic routines
#include "generic.h"

// The Poisson equations
#include "poisson.h"

// Include the templated mesh header only -- the mesh is
// precompiled and instantiated with the required element type
// in the separate file, two_d_poisson2_mesh.cc to avoid
```

```
// recompilation.
#include "meshes/simple_rectangular_quadmesh.template.h"
```

The driver code can now be compiled separately (without having to recompile the mesh sources every time) and the correctly instantiated version of the `SimpleRectangularQuadMesh` can be made available by including `two_d_poisson2_mesh.o` during the linking phase.

1.7.2 How to choose the linear solver for the Newton method

`oomph-lib` treats all problems as nonlinear problems and provides steady (and unsteady) Newton solvers to solve the system of nonlinear algebraic equations that arise from the spatial (and temporal) discretisation of the governing equations. Typically, the repeated assembly of the Jacobian matrix and the solution of the linear systems during the Newton iteration provides the major part of the computational work. Within this framework linear problems are simply special cases of nonlinear problems for which the Newton method converges in one iteration. The assembly of the Jacobian matrix and the solution of the linear system is performed by `oomph-lib's LinearSolver` objects. These typically provide interfaces to general purpose linear solvers such as `SuperLUSolver` (our default solver). The list of solvers includes:

- `SuperLUSolver`: An interface to Demmel, Eistenstat, Gilbert, Li & Liu's serial SuperLU solver. See http://crd.lbl.gov/~xiaoye/SuperLU/ for details.
- `HSL_MA42`: An interface to the MA42 frontal solver from the HSL library. See http://www.hsl.rl.ac.uk for details.
- `FD_LU`: An extremely inefficient solver which computes the Jacobian matrix by finite differencing and stores it in a dense matrix. This solver is mainly provided to facilitate sanity checks during code development. (The residuals are easier to compute than the the Jacobian matrix!)
- ...and many others. See the Linear Solvers Tutorial for a more detailed discussion of `oomph-lib's` various direct and iterative solvers.

By default the `oomph-lib` Newton solvers use `SuperLU` with compressed row storage for the Jacobian matrix as the linear solver. To change the linear solver to another type you can over-write the `Problem's` pointer to its linear solver. For instance, to change the linear solver to `HSL_MA42`, add the following lines to the `Problem` constructor:

```
/// Build a linear solver: Use HSL's MA42 frontal solver
Problem::linear_solver_pt() = new HSL_MA42;

/// Switch on full doc for frontal solver
static_cast<HSL_MA42*>(Problem::linear_solver_pt())->doc_stats()=true;
```

`HSL_MA42` can document various statistics such the memory usage etc. This is enabled with the second command. Other solvers have similar member functions. See the full documentation of all oomph-lib classes for details.

We provide an example code two_d_poisson_compare_solvers.cc which can be used to explore the performance of various linear solvers for the 2D Poisson problem considered above.

1.8 Source files for this tutorial

- The source files for this tutorial are located in the directory:

  demo_drivers/poisson/two_d_poisson/

- The driver code is:

  demo_drivers/poisson/two_d_poisson/two_d_poisson.cc

1.9 PDF file

A pdf version of this document is available.