Multigrid preconditioning methods for the Darcy problem

Francesco Cattoglio
AA 2012-2013

Overview

This work has been developed as part of the Programmazione Avanzata per il Calcolo Scientifico class held at Politecnico di Milano. The original objectives were the following:

- Solve a Darcy problem in the $H(div) \times L^2$ space, with non-trivial boundary conditions, by using a direct numerical solver. The code should be written in the C++ language and should use the deal.II library [4, 5] as a tool for PDE discretization and solution.

- Replace the direct numerical solver with an iterative solver and create an ideal preconditioner for the problem. This preconditioner is only required to work in single-threaded solvers.

- Modify the code to use the MPI protocol using the tools available in the deal.II library, for faster numerical solution in multithreaded and multi-machine environments.

- Replace the ideal preconditioner with a more scalable version, a Domain Decomposition or Multigrid-based approximation of the preconditioner itself, as proposed by Arnold et al. in [1–3].

During the whole implementation process two key objectives were always pursued. First and foremost, the code should be as much reusable as possible. This also means the code should be easy to integrate back into the deal.II library, to allow others to take advantage of our work. Another important objective is having a flexible final product: the user of the software should be able to easily configure various settings, e.g. spatial dimension of the problem, desired mesh for the computation, various boundary conditions, permeability and right hand side term.

During development a few library issues have been discovered. Some of them impacted heavily on development times, slowing down the work in a considerable way. The most important issues have been:
• The dealii::MeshWorker framework has been used for system matrix assembly. However, because of a bug in the default Assembler class provided by the library, the linear constraints (namely the ones given by hanging nodes) were not correctly enforced on the system matrix. This bug has been resolved by patching manually the deal.II/meshworker/simple.h file. The patch was found in the deal.II mailing list, since a bug report had just been filed a few weeks prior to my discovery (and was not yet documented elsewhere).

• Trying to impose strong Neumann boundary conditions produced wrong results. Therefore I had to rewrite the assembly procedure to add support for weak Neumann boundary conditions (Nitsche formulation).

• When using the dealii::MeshWorker framework, a race condition caused numerical errors during the matrix assembly procedures. This race condition was an internal deal.II issue: since deal.II used an external thread-unsafe library (fparser), the whole assembly procedure became thread-unsafe. This bug was reported by me in mailing list, along with a simple fix for it.

• Reworking some parts of the code for adding MPI support became way more difficult than expected. In particular, an unexpected bug showed up when trying to impose hanging nodes constraints on distributed meshes. In fact, the bug only triggered when using Raviart–Thomas finite elements. Apparently, I was the first one to use RT elements in a distributed framework. I reported this bug along with a ‘minimal working example’, and the issue has since been fixed by one of the deal.II developers.

The deal.II libraries has two possible backends for MPI support: it can either rely on Trilinos or PETSc libraries, but using any of those backends outside of the given examples is non-trivial, therefore I am still trying to achieve a good numerical behaviour of the internal ideal preconditioner solver, since the achieved results are not yet satisfactory. The work on a Multigrid-based preconditioner has been planned but has not been developed for this reason. The detailed description of the ideas for implementing this feature are in the “Enhancements and Known Issues” section.

The Darcy problem

We are trying to solve the following problem:

\[
\begin{align*}
\mu^{-1} \mathbf{u} - \nabla p &= 0 \quad \text{in } \Omega \\
\text{div } \mathbf{u} &= f \quad \text{in } \Omega \\
p &= g \quad \text{on } \Gamma_D \\
\mathbf{u} \cdot \mathbf{n} &= h \quad \text{on } \Gamma_N
\end{align*}
\]

Where \( \mu \) is the permeability tensor (or coefficient), \( \mathbf{u} \) is the flux velocity vector, \( p \) is the pressure, \( f, g, \) and \( h \) are given functions.
The variational formulation for this problem when $g = 0$, $\Gamma_D = \partial \Omega$ is the following:

Find $u \in H(\text{div}), p \in L^2(\Omega)$ such that

$$(\mu^{-1}u, v) + (p, \text{div} v) = 0 \quad \text{for all } v \in H(\text{div})$$

$$(\text{div} u, q) = (f, q) \quad \text{for all } q \in L^2$$

For correct numerical solving of this saddle point problem we need a pair of finite elements which fulfills the inf-sup stability condition [6]. The pair used for discretization is $RT(r) \times DG(r)$. This discretization allows us to rewrite the original Darcy problem as a linear system $Dx = b$. The degrees of freedom of the problem can easily be reordered component-wise, so that the matrix $D$ can be divided in the following blocks:

$$D = \begin{pmatrix} A & B^T \\ B & 0 \end{pmatrix}$$

where $A$ is the block given by the term $a(u, v) = (\mu^{-1}u, v)$ and $B$ is the block given by the bilinear form $b(u, q) = (\text{div} u, q)$.

Solving such a linear sistem however is non-trivial if the degrees of freedom grow too much in number, so our efforts focused on implementing a good preconditioner for the matrix $D$. An often proposed ideal preconditioner for the problem has the form

$$P = \begin{pmatrix} \Theta & 0 \\ 0 & M \end{pmatrix}$$

where $\Theta$ is the matrix associated with the bilinear form $\Lambda(u, v) = (\mu^{-1}u, v) + (\text{div} u, \text{div} v)$, which is a scalar product of the Hilbert space $H(\text{div})$, and $M$ is the pressure mass matrix. Such a preconditioner has been implemented and tested using the deal.II library.

Structure of the code

The source code has been structured in the following way:

- Some code is meant to be easily included in the deal library. That code is located in the ./dealextension/ folder.

- There are two versions of the code. The main difference between those two versions is support of MPI through the TrilinosWrappers namespace provided by deal.II. The choice of separating the code was made mostly for ease of development; the differences are somewhat small, therefore it should be possible to re-write the two versions as a single codebase and adding conditional compilation of code (e.g. adding one or more compiler flags and ifndef statements).

- The version with no MPI support is in the folder ./pacs/prec_single, while the version with MPI support is in the ./pacs/dd folder.

- The build script files are named wscript and can be found in every subfolder of the project.
Please note: the linear_algebra_factory.h file is a file written by Mattia Penati to reduce the amount of code repetition. The idea is moving some of the most common operations made during initialization of the system matrix and RHS into a separate template class. It has been successfully tested and used in the non-MPI version of the code. Note that “factory” should not be interpreted as in “factory pattern”.

**Revision system**

The revision system used for the code is Mercurial, and a copy of the repository is hosted on Bitbucket, at the address https://bitbucket.org/francesco_cat/tesi-cattoglio

Choosing Mercurial was mostly a matter of preference. We wanted the following features:

- Distributed revision system
- Handling of both source and binary files
- Easy to use and largely supported
- Allow for small and quick commits

Mercurial satisfies all the requisites we needed, and the pull/commit/push workflow fits perfectly a small project like the one I developed.

**Build system**

The build system of choice is the Waf build system. The philosophy behind Waf is quite unique among all the build tools: it is designed to be included as part of the individual software project, rather than being installed as a system-wide program (contrary to CMake or Scons, for example). For this reason, its only dependency is a working Python interpreter. The entire build system is self contained and unpacks itself in the local folder prior to first execution of the build steps. No installation or configuration of any kind is needed. If the need arises, Waf can take advantage of the full power of the Python language, since its configuration files are just Python scripts with extra functionality baked in. As a bonus, the ./waf executable file is versioned in the repository, therefore pulling the code will grant you a working copy of waf.

**Building the code**

Building the project only requires two commands. However some environment variables must be set beforehand:

- **P4EST_INC**: path to the p4est library include files
- **PETSC_INC**: path to the PETSc library include files
- **DEALII_INC**: path to the deal.II library include files
**TRILINOS_INC** : path to the Trilinos library include files

**MUMPS_INC** : path to the MUMPS library include files

**BOOST_INC** : path to the Boost library include files

**DEALII_LIB** : path to the deal.II library .so or .a files

**TRILINOS_LIB** : path to the trilinos library .so or .a files

Defining those variables is necessary because the code is currently being developed and tested on two different computers, one of them being the “Veio” computer hosted at the Politecnico di Milano. Therefore we can not assume that every library is available at the default paths /usr/ and /usr/local/. A set_env file is also provided in the root folder as an example for setting of those variables: the user can easily modify it and source it from the bash.

For building, the two following commands are used: from the root folder run

```
$ ./waf configure
$ ./waf build
```

The binaries will be build in the ./builds/* subfolders. Before running the executable `mx_prob`, make sure an `out/` folder exists at the executable location, or any file output from the program will fail.

The user can specify a compiler to the configure step if he wishes to. Just use

```
$ CXX=<compiler> ./waf configure
```

e.g.

```
$ CXX=mpic++ ./waf configure
```

**Implementation details**

Here we give a quick overview of the classes and methods implemented when developing the work, as well as design choices and rationale.

**Templated code**

Just like in the deal.II library, most of the code is templated on the spatial dimension, making sure that solving a problem in 2 or 3 dimensions does not require any additional coding. Since almost everything is a template, separating different classes in different compile units requires explicit template instantiation. For this reason, at the end of almost every .cc file there are a few lines making sure code is produced for the classes and methods defined in the file.
ProblemParameters class

class ProblemParameters : public dealii::ParameterHandler

ProblemParameters::ProblemParameters()

This class is a specialized dealii::ParameterHandler. The purpose of this class is to read from file all the settings of the problem we are going to solve, hold them and make them available when needed by the rest of the code. Inheriting from the dealii::ParameterHandler allows us to properly set up this object at construction time, making sure we don’t forget to initialize the object before use. The other classes hold a dealii::SmartPointer to this class, and call the get_*() methods to read the settings when needed.

ProblemFuncParser class

template<int dim>
class ProblemFuncParser : public dealii::FunctionParser<dim>

ProblemFuncParser(const std::string & function_string, const unsigned int n_components = 1)

This class is a specialized dealii::FunctionParser, which in turn is a dealii::Function class. ProblemFuncParser automatically defines some useful constants (thus avoiding some code duplication) and prevents calls to a FunctionParser object that has not been properly initialized. On construction, the class takes a std::string representing the function to be parsed and the number of components the function is supposed to have. The number of components is required even if we could deduce it from the string given as first argument. The decision of making it a required parameter allows for better handling of bad input from the user.

FunctionSmartPointer typedef

template <int dim>
struct FunctionSmartPointer {
    typedef std::map<dealii::types::boundary_id, dealii::SmartPointer<const dealii::Function<dim> > > type;
};

dealII provides dealii::FunctionMap, which is a typedef for a std::map associating cell or boundary labels to pointers to dealii::Function objects. Because of ownership concerns (if code evolves in unforeseen ways knowing who should delete the pointed object is not trivial), I added a safer typedef. The template FunctionSmartPointer::type associates labels to dealii::SmartPointers instead of plain C pointers.

Maps of type FunctionSmartPointer are used for passing mathematical functions as arguments to the classes responsible for integrating local contributes to system matrix and vectors.

6
**InitMapFromStrings function**

```cpp
template<int spatial_dim>
void InitMapFromStrings(
    typename FunctionSmartptrMap<spatial_dim>::type & map,
    const std::string& labels,
    const std::string& functions,
    const int functionDimension)
```

Correct initialization of the function maps from string parameters from file requires some tokenizing, since multiple functions are passed in a single string. The tokenizer of choice is the one provided by the Boost library. It was chosen over the C function `strtok()` for robustness and giving the code a better “C++ feeling”. Dependency from the Boost library should not compromise portability of the code in any way.

**AnyDimensionProblem class**

```cpp
class AnyDimensionProblem : public dealii::Subscriptor
AnyDimensionProblem(dealii::ParameterHandler& settings,
    const std::string prefix);
virtual void run_problem()=0;
```

This abstract class is used as a base class representing a generic problem. The interface it defines is extremely simple, but that’s all we need for defining a solving problem:

- A `dealii::ParameterHandler` for defining the problem settings and the solving parameters.
- A `run_problem()` method for starting the computations.

The `MixedProblem` inherits from this base class. This class also serves an important purpose: almost everything in the code is templated on the spatial dimension, but we still want the spatial dimension to be a parameter read by file. Holding a pointer to the base class allows us to reference the problem, regardless of the dimension chosen by the user, in an elegant way. The `prefix` parameter is the string that will be prepended to the log produced by the class and its methods.

**MixedProblem class**

```cpp
template<int dim>
class MixedProblem : public AnyDimensionProblem
MixedProblem(dealii::ParameterHandler& settings,
    const std::string prefix);
virtual void run_problem();
```
This class is the one responsible for most of the work. See the Program flow section for more details. It is responsible for initializing and storing the mesh, the finite elements, the function maps and the constraints used both internally and by the IdealPreconditioner class, as well as initializing and storing its own sparsity pattern, system matrix and vectors.

**SystemIntegrator class**

```cpp
template <int dim>
class SystemIntegrator : public dealii::MeshWorker::LocalIntegrator<dim>
{
    SystemIntegrator(
        const typename FunctionSmartptrMap<dim>::type dirichletMap,
        const typename FunctionSmartptrMap<dim>::type neumannMap,
        const typename FunctionSmartptrMap<dim>::type muMap,
        const typename FunctionSmartptrMap<dim>::type rhsMap)

    void cell(
        dealii::MeshWorker::DoFInfo<dim> &dinfo,
        dealii::MeshWorker::IntegrationInfo<dim> &info) const

    void boundary(
        dealii::MeshWorker::DoFInfo<dim> &dinfo,
        dealii::MeshWorker::IntegrationInfo<dim> &info) const

    void face(
        dealii::MeshWorker::DoFInfo<dim> &dinfo1,
        dealii::MeshWorker::DoFInfo<dim> &dinfo2,
        dealii::MeshWorker::IntegrationInfo<dim> &info1,
        dealii::MeshWorker::IntegrationInfo<dim> &info2) const
```

This class implements cell(), boundary() and face() methods that are called during assembly by deal.II as part of the integration_loop() method. Those methods compute the local contribution to the system matrix and vector given by every cell and boundary face of the triangulation.

**IdealPreconditioner class**

```cpp
template<int dim, typename VECTOR=dealii::BlockVector<double> >
class IdealPreconditioner : public dealii::Subscriptor
{
    IdealPreconditioner(
        const dealii::DoFHandler<dim> &mixedProblemDoFHandler,
        const dealii::ConstraintMatrix &mixedProblemConstraintMatrix,
        const std::string prefix)

    void init_preconditioner(
        const typename FunctionSmartptrMap<dim>::type neumannMap,
        const typename FunctionSmartptrMap<dim>::type muMap)
```
void vmult(VECTOR& dst, const VECTOR& src) const

This class implements the ideal preconditioner discussed in “The Darcy problem” section. The deal.II iterative solvers are heavily templated and can take any class as preconditioner, as long as that class implements a vmult() method.

The constructor initializes the smart pointers to the dealii::DoFHandler and the dealii::ConstraintMatrix stored by the MixedProblem class. Before calling the vmult() method, the preconditioner is initialized by calling init_preconditioner(). The prefix parameter is the string that will be prepended to the log produced by the class and its methods.

IdealIntegrator class

template <int dim>
class IdealIntegrator : public
dealii::MeshWorker::LocalIntegrator<dim>
IdealIntegrator(
    const typename FunctionSmartptrMap<dim>::type neumannMap,
    const typename FunctionSmartptrMap<dim>::type muMap)

This class inherits from dealii::MeshWorker::LocalIntegrator. It fulfills the same role as the SystemIntegrator class, but for the of the matrix used by the IdealPreconditioner instead of the system matrix. The constructor needs less parameters because the ideal preconditioner does not need any information about Dirichlet boundary conditions or RHS function. Only the Neumann boundary conditions and the \( \mu \) function are needed for assembly.

dealii::LocalIntegrators::Darcy namespace

All the methods in this namespace have been coded with the objective of having them included in future releases of the library. They mimic the signatures of all the other methods in the dealii::LocalIntegrators namespaces.

All the methods can be plugged into any code using the dealii::MeshWorker framework. Those are the ones actually computing local contributes to the various matrices and vectors used in our code. They all take a dealii::FEValuesBase reference as argument: this object holds all the informations about cells and finite elements needed for numerical computation of local integrals of the variational formulation. Now, for a detailed look:

template <int dim>
void cell_matrix (FullMatrix<double> & M,
    const FEValuesBase<dim> & feValues,
    const std::vector<double> & muValues)

computes the contribute given by a cell to the system matrix

template <int dim>
void cell_preconditioner (FullMatrix<double> & M,
    const FEValuesBase<dim> & feValues,
    const std::vector<double> & muValues)
computes the contribute given by a cell to the matrix used by the ideal preconditioner

```c++
template <int dim>
void cell_rhs (Vector<double> & localVector,
               const FEValuesBase<dim> & feValues,
               const std::vector<double> & rhsValues)
```

computes the contributes given by a cell to the right hand side vector

```c++
template<int dim>
void u_times_n_residual (Vector<double> &localVector,
                         const FEValuesBase<dim> &feValues,
                         const std::vector<double> &rhsValues)
```

computes the contribute given by the Dirichlet boundaries to the system RHS vector

```c++
template<int dim>
void nitsche_matrix (FullMatrix<double> & M,
                     const FEValuesBase<dim> & feValues,
                     double penalty)
```

computes the contribute given by Neumann boundaries to the system matrix

```c++
template<int dim>
void nitsche_preconditioner (FullMatrix<double> & M,
                             const FEValuesBase<dim> & feValues,
                             double penalty)
```

computes the contribute given by Neumann boundaries to the ideal preconditioner matrix

```c++
template<int dim>
void nitsche_residual (Vector<double> & localVector,
                       const FEValuesBase<dim> & feValues,
                       const std::vector<Vector<double> > & fluxVector,
                       double penalty)
```

computes the contribute given by Neumann boundaries to the system RHS vector

**Program flow**

The class responsible for most of the work is the `MixedProblem` class. The whole computation is executed by the `MixedProblem::run_problem()` member function. The 10,000 foot view of the program flow is the following:

The `main` function reads problem settings from a configuration file specified as a command line argument (if no argument is provided, it defaults to `./in/settings.txt`). The first setting read is the desired spatial dimension for the problem. Depending the spatial dimension, the `main` function creates a new `MixedProblem<2>` or a new `MixedProblem<3>` and return a smart pointer to the `AnyDimensionProblem` base class.

The `main` then calls the `MixedProblem::run_problem()` function. In turn, the `run_problem()` executes the following procedure:
1. Builds the mesh from scratch in the `RebuildMesh()` function.

2. If required by the user, it starts the following reinit/solve/refine cycle:
   a) `InitSystem()` takes care of initializing the `DoFHandler`, setting up the constraints on the matrix, creating the sparsity pattern and resizing the system matrix and vectors.
   b) `RebuildSystem()` is the function assembling the system matrix and the RHS vector, taking advantage of the `dealii::MeshWorker` interface.
   c) `Solve()` reads the settings and sets up the solver classes as requested by the user; it then solves the problem and makes sure to assign the correct values to the constrained degrees of freedom.
   d) If the user wishes to, the problem will output a solution for the current refinement step using the `OutputSolution()` function.
   e) `RefineGrid()` uses a `dealii::KellyErrorEstimator` to give an estimation of the error, allowing subsequent local mesh refinement. The refinement parameters are read from the settings.

3. Initializes the system, rebuilds the linear system and solves the problem for the last time.

4. If the user wishes to, it outputs the final solution alongside a parameter file with the last used settings.

All the data output is in `vtu` format, therefore it can be visualized by `Paraview` or by a custom script built upon the VTK framework.

**Side objectives**

**Flexibility through configuration file**

The parameter file is a plain text file with pairs `set Key = Value`, read at program start by a class derived from `dealii::ParameterHandler`. This is an example of what the file looks like.

```
# Listing of Parameters
# ---------------------
set Dimension = 2
set Dirichlet Boundary Labels = 0
set Dirichlet Functions = y*y
set FE Degree = 1
set Grid Generator = ball
set Output File = test_boundary_layer
set Permeability Cell Labels = 0
set Permeability Functions = if((x^2 + y^2)<0.25, 1, 1e4)
```
The following is a in-deep review of each option, in an order that mirrors the program flow. If a setting is not given in the file, a default value is used instead.

**Key: Log Depth**  **Accepts:** an integer value from 0 to 10
Controls the depth of the log messages the program and the deal.II library will output to the `std::cout`. Zero disables all outputs.

**Key: Dimension**  **Accepts:** an integer value 2 or 3
The spatial dimension for the problem the user wants to solve.

**Key: Grid Generator**  **Accepts:** one of: 'ball', 'cube', 'lshape', 'file'
This allows the user to choose the shape of the mesh for the problem; 3 test cases are supported (n-dimensional ball, cube or lshape) as well as allowing the user to load a gmsh file. The cube option actually calls `GridGenerator::hyper_rectangle()` with “colorized” boundaries, (see deal.II docs*) meaning that every side (or face in 3D) has a different boundary label, allowing for easy testing of different boundary conditions.

*http://goo.gl/SL3AJX

**Key: Grid Filename**  **Accepts:** file path
The file containing the input mesh. Only used if “file” is choosen as Grid Generator.

**Key: Grid Refinement**  **Accepts:** an integer value from 0 to 10
Right after the mesh is created (or loaded by file) a global refinement step is done. This parameter lets the user decide how many levels of refinement should be applied. Beware: this also applies to grid imported from file. If you load a mesh from file, chances are that you don’t want to refine it again before starting computations, so you might want to leave this to 0.

**Keys:** Dirichlet Boundary Labels, Neumann Boundary Labels, Permeability Cell Labels, RHS Cell Labels  
**Accepts:** a properly formatted string
Meshes have labels associated to boundaries and cells. While both ball and lshape get default ‘0’ labels both for boundaries and cells, creating a cube or loading a mesh from file gives the user the possibility of setting different labels. The user can map those labels to functions. The user should specify here which labels will have a function associated to them. The user can specify as many labels as he wishes, separated by a ‘$’ sign. Each given label must have a corresponding associated function. e.g. if the users wants to impose Neumann BCs on the boundaries labelled as 1 and 3, he would write: `set Neumann Boundary Labels = 1 $ 3`

**Keys:** Dirichlet Functions, Neumann Functions, Permeability Functions, RHS Functions  
**Accepts:** a properly formatted string
The user can specify here a different function for the corresponding given labels. The functions have to be separated by a '$' sign, too. The functions are passed to the dealii::FunctionParser class, for more information read the relevant docs \(^a\). The spatial variables are the usual x, y and z. The two constants 'e' and 'pi' are also defined. Beware: Neumann Boundary functions must be vector functions. You just have to specify each component separated by a ';'. e.g. imposing two different functions would look like this: 

```
set Neumann Functions = 2*x; 2*y $ cos(pi*x); sin(pi*x).
```

\(^a\)http://goo.gl/18Ek0g

**Key:** FE Degree  **Accepts:** an integer value from 0 to 2  
The degree of the Raviart–Thomas \(\times DG\) elements used for solving the Darcy problem.

**Key:** Grid Adaptivity  **Accepts:** an integer value from 0 to 10  
This controls how many times the reinit/solve/refine cycle will be executed before the final solving procedure. Zero disables the cycle (no grid refinement).

**Key:** Refinement Percentage  **Accepts:** a float value in \((0, 1]\)  
The percentage of the mesh cells that will be refined in each step. Zero is not allowed since it would not make sense.

**Key:** Coarsening Percentage  **Accepts:** a float value in \([0, 1)\)  
The percentage of the mesh cells that will be coarsened in each refinement step. If coarsening percentage is not smaller than refinement percentage, an exception is thrown.

**Key:** Linear Solver  **Accepts:** one of: ‘noprec’, ‘ideal’ or ‘direct’  
This setting controls how the problem is solved. Direct uses a direct solver, noprec uses a GMRES with no preconditioner, while ideal uses a GMRES with the ideal preconditioner implemented as part of the project.

**Key:** Max Solver Iterations  **Accepts:** an integer value from 0 to 10000  
This setting controls the maximum number of iterations the main GMRES solver makes before throwing an exception. Zero has a special meaning: it sets the solver to make at most \(n\) iterations, where \(n\) is the size of the system matrix. Not used if ‘direct’ is chosen as Linear Solver.

**Key:** Solver Tollerance  **Accepts:** a float value in \((0, 1]\)  
The tollerance for the GMRES solver. Not used if ‘direct’ is chosen as Linear Solver.

**Key:** Output File  **Accepts:** file name  
Allows the user to choose the name for file output. You should only specify the name, with no path nor extension. The program will output in the \(.\text{/out/}\) folder. If you output intermediate steps, the name of them will be `filename.#.vtu` or `filename.#.pvtu`. You can actually specify a subfolder, but the pvtu files for the MPI version of the code will not have correct references to single files produced by different MPI processes.
Key: Output Solution  
Accepts: an integer value from 0 to 2
Solution output choice: 0 means no file output at all, 1 means “only save last solution step”, 2 means “save last solution as well as every refinement step”.

Integration into deal.II library

The deal.II library provides the user with a few important tools for code simplification and reusal. The most interesting framework provided by the library is the dealii::MeshWorker interface (see docs\(^1\)). The MeshWorker is a collection of useful classes and functions for looping over the cells of the mesh. An easy way to use the framework has been shown in a previous section. Using the meshworker framework provides a few benefits, the most important one being the automated parallelization of the matrix assembly procedure (via pthreads and Intel TBB library). The second benefit is modularity: separating the code that computes local contribution is good for the library user since he has more tools available and ready to be plugged in the code, and it also greatly benefits the library developer because building the library as logically separated blocks makes it easier to add features and maintain the code. We think that the code added into the dealii::LocalIntegrators::Darcy namespace could be approved for inclusion in the deal.II library with minimal changes. We also plan to suggest a few small modifications to the other existing integration functions to expand a bit on their functionality.

Results on test cases

In this section we show three interesting test cases. The configuration files used can be found in the tests/ subfolder.

3D with MPI

The first test case is a simple problem on a 3D l-shape run on MPI with 7 processes, to show how the mesh gets distributed and refined. In this problem we have omogeneous boundary conditions and omogeneous permeability function. The mesh has gone through four refinement steps and the two pictures are of the final result of the computation.

\(^1\)http://goo.gl/nzLsNe
The first picture shows how the mesh is correctly refined around the central edge of the shape, and how the mesh is distributed to the MPI processes. The second picture has a slice to display the computed results for the pressure variable.

Convergence wise the results are fair enough. The ideal preconditioner does not perform as we expected (this is an unresolved issue with MPI) but overall the results aren’t terrible: if the computation is run with the preconditioned GMRES solver iterations span from 37 to 75, while if the same problem is run with no preconditioner iterations quickly grow from 53 to 873. We are still investigating the issue.

**2D with boundary layer**

The second test is chosen to show how the program deals with problems for which exact solution of the pressure has a boundary layer near the mesh border. The test is run on a 2D ball mesh and the permeability and RHS functions have been chosen to produce a strong gradient while keeping the exact solution smooth.
The results are really nice: the refinement captures perfectly the boundary layer and the numerical solution looks really nice. The preconditioner also show the predicted behaviour: after all the mesh refinements the iterations of the GMRES solver only grow from the initial 14 to 15, while the size of the solved matrix increased almost three times.

This result was achieved running the MPI code with only one MPI process. This is to show that the MPI issue only appears when more than one process is used.

2D with discontinuous permeability

The last test is used to check how the ideal preconditioner deals with sharp changes in the permeability of the problem. In the square mesh the upper half has a permeability value of $10^4$ while the lower half has a value of 1. On the right side we imposed a Neumann boundary condition, imposing the normal flux to be equal to 1. On the left side we impose a homogeneous Dirichlet boundary while on the upper and lower boundaries a no normal flux condition is applied.

The first picture shows the pressure profile with the mesh refinement, and allows us to check that the refinement is correctly executed. There is one critical point in this problem: we forced a certain flux but the permeability is discontinuous. The second is a zoom on the flux near this critical point, and we can see how the fluxes go from the low permeability area to the high permeability one, while respecting the divergenceless constraint (this should be a given but we had to go through a few hops for this to actually work).

This example was solved by the non-MPI version of the code.

Enhancements and Known Issues

Multigrid method

deal.ii has a good framework for geometrical multigrid methods. A noteworth property of those methods is that they are extremely flexible: when the framework is production
ready, adding a new method can be as simple as defining a new smoother and choosing
the type of cycle to be executed.

The user can implement a new smoother by inheriting from the following deal.II ab-
abstract class:

```cpp
template <class VECTOR>
class MGSmootherBase : public Subscriptor
{
public:
  virtual ~MGSmootherBase();
  virtual void clear() = 0;
  virtual void smooth (const unsigned int level,
                       VECTOR &u, const VECTOR &rhs) const = 0;
};
```

In [2], a simple but effective smoother is proposed: the smoother solves the same
variational formulation as the ideal preconditioner, but it only solves it locally, on a patch
of quads built around each vertex of the triangulation. deal.II implements a function to
help building such a smoother: the `dealii::DoFTools::make_vertex_patches()` computes
the patches surrounding every vertex at a certain level of the triangulation. From the
deal.II documentation:

Create an incidence matrix that for every vertex on a given level of a multilevel
DoFHandler flags which degrees of freedom are associated with the adjacent
cells. This data structure is a matrix with as many rows as there are vertices
on a given level, as many rows as there are degrees of freedom on this level,
and entries that are either true or false. [...] The default settings are those for
Arnold-Falk-Winther type smoothers for divergence and curl conforming finite
elements with essential boundary conditions. Other applications are possible,
in particular changing `boundary_patches` for non-essential boundary conditions.

This means the library provides the programmer with an extremely effective tool for
our purposes. We can implement the desired smoother object in the following way:

1. For every level of the multigrid method, the smoother holds a collection of the local
   matrices associated with patches built around every vertex in the triangulation.

2. During initialization, for each multigrid level the smoother reads the global matrix,
calls the corresponding `make_vertex_patches()` function, saves the returned inci-
dence matrix and uses its indices to extract local matrices from the global matrix,
storing them.

3. When the `smooth()` method is called by the framework, for every stored vertex
   patch the smoother:
   a) copies the values of the `global` source vector corresponding to the degrees of
      freedom in the patch into a `local` source vector

b) uses this local vector to solve the local problem

c) adds the local solution to the global destination vector

The most important advantages of this method are the following:

- Since one only has to solve local problems, this approach scales really well when the mesh size increases and can be applied almost effortlessly to distributed meshes.

- The local matrices are really small and can be factorized during initialization if needed, making solution of the local problem extremely fast.

- If the triangulation is regular enough, a vertex patch always contains a maximum amount of degrees of freedom; this amount does not grow with mesh refinement. We can then assert that solving a local matrix takes a fixed amount of time, and the complexity of a whole multigrid cycle is $\Theta(n)$, $n$ being the number of the vertices or of the degrees of freedom.

**Faster and easier testing**

It would be nice to add support for the functionality provided by the deal.II class `MultipleParameterLoop`. This class allows a user to easily launch several runs of the same problem with multiple parameter combinations (see deal.II docs for more information). Adding a helper class should be easy enough, since the problem solving is already automated by calling the `run_problem()` function. Another feature that shouldn’t take too much time to implement is adding support for automated tests. Since running the program only depends on a configuration file, it should be easy to add a “test” target for the Waf build system that launches several test cases and makes sure no regression bugs are created when adding features to the code.

**Checking of error norms**

At the beginning, when all the problem parameters where hardcoded in the program, the MixedProblem class had a method to compute $L^2$ and $H^1$ error norms. After implementing user-defined functions however that code got commented out and was never modified for allowing user-defined exact solutions. This happened because the most interesting test cases didn’t had a known exact solution, making that code of little use. It should be possible to re-introduce this feature without any particular issue.

**Serious MPI issue**

Current the code doesn’t behave as expected when using an ideal preconditioner with MPI. The preconditioner loses on performance if mesh is refined and if more MPI processes are added to the computation. Since we are still doing a direct solve of the preconditioner matrix, this is a really unexpected behaviour: direct solving should always give the same result, no matter the MPI processes count.

³http://goo.gl/gATdah
Small 3D ball issue

The 3D triangulation generated by the dealii::GridGenerator::hyper_ball function seems to contain errors (a part of the boundary is assigned to interior faces). For this reason, solving a problem in 3D with the Grid Generator=ball setting throws an ‘unimplemented exception’. As a workaround you can however provide a mesh file created by an external program.

What deal.II version?

As mentioned already, since a few bugs were found and fixed while the code was still being written, the MPI version only works with the current svn version of deal.II, the 8.1.pre version. On the other hand, a regression bug is present in the svn, so the non-MPI version of the code requires the 8.0 version of deal.II. Hopefully, the newer version will be the only one required when 8.1 is finally released and the regression fixed.

It’s never enough!

Even if the program strives to be as much foolproof as possible, we could use some extra checks and asserts, better error messages and handling, etcetera.

References


