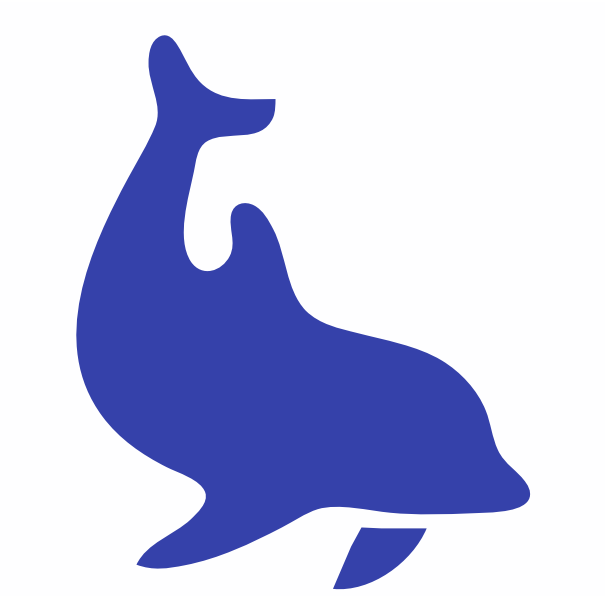
DOLFIN User Manual

September 10, 2004



Hoffman, Logg, et al.

This manual has been written by:
Johan Hoffman, Johan Jansson, Anders Logg, Andreas Mark, and Andreas Nilsson.

Contents

1 Introduction

This is a first draft for a DOLFIN manual. Contributions are most welcome.

2 Installation

In preparation.

3 Linear algebra

In preparation.

4 The multigrid solver

Multigrid solvers were invented to solve partial differential equations and in contrast to other iterative solvers its convergence rate is independent of the problem size. The multigrid solver solves a given PDE using a hierarchy of discretizations, typically Poisson's equation:

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega. \end{aligned} \tag{1}$$

4.1 Usage

First a variational formulation (a PDE object) has to be defined. Then there are three different ways to call the multigrid solver:

- Call the solver with a MeshHierachy which consisting of arbitrarily refined meshes:

```
MultigridSolver::solve(PDE poisson, Vector& x,
                       MeshHierarchy& meshes);
```

- Call the solver with a mesh which has been refined a number of times (implicitly containing a mesh hierarchy):

```
MultigridSolver::solve(PDE poisson, Vector& x,
                       Mesh mesh);
```

- Call the solver with a coarse mesh and a number specifying the number of (uniform) refinements to make:

```
MultigridSolver::solve(PDE poisson,Vector& x,
                       Mesh& mesh, int refine);
```

4.2 An example

```
Mesh     mesh("square.xml");
Function f("source");
Poisson  poisson(f);

MultigridSolver::solve(poisson, x, mesh, 5);
```

4.3 Performance

The solution time for the Multgrid solver has been compared with the Krylov solver for tolerance TOL $= 10^{-3}$. Figure 1 shows that the complexity of the multigrid solver is linear and that it is faster than the Krylov solver. For large n, the multigrid solver is still linear while the Krylov solver fails to converge. The test was performed on 1.7 GHz Celeron with 256 Mb of RAM running Debian GNU/Linux.

4.4 Limitations

The solver has only been tested in 2D and it's only written for linear elements.

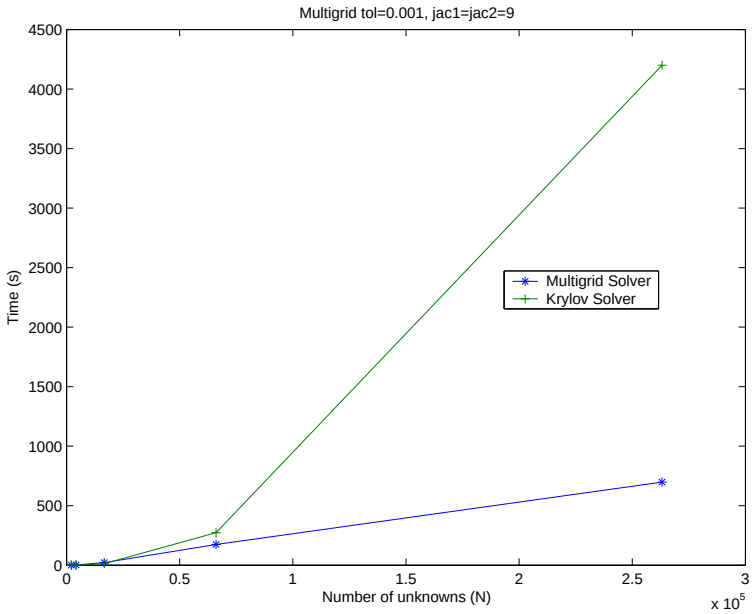


Figure 1: Plot of the solution time for the multigrid solver and the Krylov solver.

5 Mesh management

In preparation.

6 The log system

The purpose of the log system is to provide a simple and clean interface for logging messages, including warnings and errors.

The following functions / macros are provided for logging:

```
dolfin_info();
dolfin_debug();
dolfin_warning();
dolfin_error();
dolfin_assert();
```

Examples of usage:

```
dolfin_info("Created vector of size %d.", x.size());
dolfin_debug("Opened file");
dolfin_warning("Unknown cell type.");
dolfin_error("Out of memory.");
dolfin_assert(i < m);
```

Note that in order to pass additional arguments to the last three functions (which are really macros, in order to automatically print information about file names, line numbers and function names), the variations `dolfin_debug1()`, `dolfin_debug2()` and so on, must be used.

As an alternative to `dolfin_info()`, C++ style output to cout (`dolfin::cout`, and not `std::cout`) can be used. These messages will be delivered to the same destination as messages by use of the function `dolfin_info()`.

Examples of usage:

```
cout << "Assembling matrix: " << A << endl;
cout << "Refining grid: " << grid << endl;
```

The `dolfin_assert()` macro should be used for simple tests that may occur often, such as checking indexing in vectors. The check is turned on only if DEBUG is defined.

To notify progress by a progress session, use the class Progress.

Examples of usage:

```
Progress p("Assembling", grid.noCells());

for (CellIterator c(grid); !c.end(); ++c) {
  ...
  p++;
}
```

Progress also supports the following usage:

```
p = i;    // Specify step number
p = 0.5;  // Specify percentage
p.update(t/T, "Time is t = %f", t);
```

7 Handling parameters

In preparation.

8 Writing a new module / solver

In preparation.

9 Installing DOLFIN

In preparation.

10 Contributing to DOLFIN

If you have created a new module, fixed a bug somewhere, or have done a small change which you want to contribute to DOLFIN, then the best way to do so is to send a patch to the maintainer. A patch is a file which describes how to transform a file or directory structure into another. The patch is built by comparing a version which both parties have against the modified version which only you have.

The tool used to create a patch is called `diff` and the tool used to apply the patch is called `patch`. These tools are free software and are standard in most UNIX-style operating systems.

10.1 Creating a patch using CVS

The simplest way to create a patch is through CVS. Assuming that you have previously checked out a CVS version of DOLFIN and that you are positioned at the top of the DOLFIN source tree, a patch can be created using the command

```
$ cvs diff > dolfin-logg-2004-01-16.patch
```

Choose a suitable name for the patch to simplify identification, for example your name and today's date.

10.2 Creating a patch without using CVS

Example:

Let's say you have started from the DOLFIN release 0.3.10. You have a directory structure under `dolfin-0.3.10` where you have made some modifications to some files which you think could be useful to other DOLFIN users.

1. Clean up your modified directory structure to remove temporary and binary files which will be rebuilt anyway:

   ```
   $ make clean
   ```

2. Rename the directory to something else:

   ```
   $ mv dolfin-0.3.10 dolfin-0.3.10mod
   ```

3. Unpack the version of DOLFIN which you started from:

   ```
   $ tar xzvf dolfin-0.3.10.tgz
   ```

4. You should now have two DOLFIN directory structures in your current directory:

   ```
   $ ls
   dolfin-0.3.10
   dolfin-0.3.10mod
   ```

5. Use the `diff` tool to create the patch:

   ```
   $ diff -u --new-file --recursive dolfin-0.3.10 dolfin-0.3.10mod >
   mypatch.patch
   ```

   "`-u`" means "unified", a format in which to describe the patch.

   "`--new-file`" accepts the creation of new files in addition to modifications.

   "`--recursive`" means to recursively compare files through the directory structure.

6. The patch now exists as `mypatch.patch` and can be distributed to other people who have dolfin-0.3.10 to easily create your modified version. If the patch is large, compressing it with for example `gzip` is advisable.

10.2.1 Applying a patch

Example:

Let's say the patch has been built relative to DOLFIN release 0.3.10.

1. Unpack the version of DOLFIN which the patch is built relative to:

   ```
   $ tar xzvf dolfin-0.3.10.tgz
   ```

2. Check that you have the patch `mypatch.patch` and the DOLFIN directory structure in the current directory.

   ```
   $ ls
   dolfin-0.3.10
   mypatch.patch
   ```

3. Enter the DOLFIN directory structure:

   ```
   $ cd dolfin-0.3.10
   ```

4. Apply the patch:

   ```
   $ patch -p1 < ../mypatch.patch
   ```

   "`-p1`" strips the leading directory from the filename references in the patch, to match the fact that we are applying the patch from inside the directory.

5. The modified version now exists as `dolfin-0.3.10`