DOLFIN Documentation

Release 2017.1.0.dev0

FEniCS Project

Mar 06, 2017

Contents

This documentation is under development

Contents:

CHAPTER 1

Installation

Quick start

Building from source

Dependencies

DOLFIN requires a compiler that supports the C++11 standard.

The required and optional DOLFIN dependencies are listed below.

Required

- Boost (http://www.boost.org), with the following compiled Boost components
  - filesystem
  - iostreams
  - program_options
  - timer
- CMake (https://cmake.org)
- Eigen3 (http://eigen.tuxfamily.org)
- FFC (https://bitbucket.org/fenics-project/ffc)
- pkg-config (https://www.freedesktop.org/wiki/Software/pkg-config/)
- zlib

Required for Python interface

- Python (including header files)
- SWIG (http://www.swig.org)
- NumPy (http://www.numpy.org)
- ply (https://github.com/dabeaz/ply)

Optional

- HDF5, with MPI support enabled
- MPI
- ParMETIS[1]
- PETSc (strongly recommended)[2]
- SCOTCH and PT-SCOTCH[1]
- SLEPc
- Suitesparse[1]
- Trilinos
- VTK

Optional for the Python interface

- petsc4py
- slepc4py
- mpi4py
- Matplotlib

[1] It is strongly recommended to use the PETSc build system to download and configure and build these libraries.

[2] Its is recommended to configuration with ParMETIS, PT-SCOTCH, MUMPS and Hypre using the `--download-parmetis --download-ptscotch --download-suitesparse --download-mumps --download-hypre`

CHAPTER 2

Using DOLFIN

Under development

CHAPTER 3

Getting help

Under development

CHAPTER 4

Demo documentation

Under development

Using the Python interface

Introductory DOLFIN demos

These demos illustrate core DOLFIN/FEniCS usage and are a good way to begin learning FEniCS. We recommend that you go through these examples in the given order.

1. Getting started: Solving the Poisson equation.
2. Solving nonlinear PDEs: Solving a nonlinear Poisson equation
3. Using mixed elements: Solving the Stokes equations
4. Using iterative linear solvers: Solving the Stokes equations more efficiently

More advanced DOLFIN demos

These examples typically demonstrate how to solve a certain PDE using more advanced techniques. We recommend that you take a look at these demos for tips and tricks on how to use more advanced or lower-level functionality and optimizations.

- Implementing a nonlinear hyperelasticity equation
- Implementing a splitting method for solving the incompressible Navier-Stokes equations
- Using a mixed formulation to solve the time-dependent, nonlinear Cahn-Hilliard equation
- Computing eigenvalues of the Maxwell eigenvalue problem

Demos illustrating specific features

How to

- work with built-in meshes
- define and store subdomains
- integrate over subdomains
- set boundary conditions on non-trivial geometries
- solve a basic eigenvalue problem
- set periodic boundary conditions
- de-singularize a pure Neumann problem by specifying the nullspace
- de-singularize a pure Neumann problem by adding a constraint
- use automated goal-oriented error control
- specify a Discontinuous Galerkin formulation
- work with c++ expressions in Python programs
- specify various finite element spaces
  - Brezzi-Douglas-Marini elements for mixed Poisson
  - the Mini element for Stokes equations

Working list of Python demos

- demos/demo_poisson.py
- demos/demo_eigenvalue.py
- demos/demo_built-in-meshes.py
- demos/demo_mixed-poisson.py
- demos/demo_biharmonic.py
- demos/demo_auto-adaptive-poisson.py
- demos/demo_cahn-hilliard.py
- demos/demo_maxwell-eigenvalues.py
- demos/demo_built-in-meshes.py
- demos/demo_hyperelasticity.py
- demos/demo_nonlinear-poisson.py
- demos/demo_nonmatching-interpolation.py

Using the C++ interface

- demos/poisson/main.cpp
- demos/eigenvalue/main.cpp

- demos/built-in-meshes/main.cpp
- demos/mixed-poisson/main.cpp
- demos/biharmonic/main.cpp
- demos/auto-adaptive-poisson/main.cpp
- demos/nonmatching-interpolation/main.cpp

Todo

Fix the toctree

CHAPTER 5

Contributing

This page provides guidance on how to contribute to DOLFIN.

Adding a demo

The below instructions are for adding a Python demo program to DOLFIN. DOLFIN demo programs are written in reStructuredText, and converted to Python/C++ code using `pylit`. The process for C++ demos is similar. The documented demo programs are displayed at http://fenics-dolfin.readthedocs.io/.

Creating the demo program

1. Create a directory for the demo under `demo/documented/`, e.g. `demo/documented/foo/python/`.
2. Write the demo in reStructuredText (rst), with the actual code in 'code blocks' (see other demos for guidance). The demo file should be named `demo_foo-bar.py.rst`.
3. Convert the rst file to to a Python file using `pylit` (pylit is distributed with DOLFIN in `utils/pylit`)

   ```
   ../../../../utils/pylit/pylit.py demo_foo-bar.py.rst
   ```

   This will create a file `demo_foo-bar.py`. Test that the Python script can be run.

Adding the demo to the documentation system

1. Add the demo to the list in `doc/source/demos.rst`.
2. To check how the documentation will be displayed on the web, in `doc/` run `make html` and open the file `doc/build/html/index.html` in a browser.

Make a pull request

1. Create a git branch and add the `demo_foo-bar.py.rst` file to the repository. Do not add the `demo_foo-bar.py` file.
2. If there is no C++ version, edit `test/regression/test.py` to indicate that there is no C++ version of the demo.
3. Make a pull request at https://bitbucket.org/fenics-project/dolfin/pull-requests/ for your demo to be considered for addition to DOLFIN. Add the `demo_foo-bar.py.rst` file to the repository, but do not add the `demo_foo-bar.py` file.

CHAPTER 6

API documentation

Under development

CHAPTER 7

Developer resources

DOLFIN development takes place on Bitbucket.

C++ coding style guide

Naming conventions

Class names

Use camel caps for class names:

```
class FooBar
{
  ...
};
```

Function names

Use lower-case for function names and underscore to separate words:

```
foo();
bar();
foo_bar(...);
```

Functions returning a value should be given the name of that value, for example:

```
class Array:
{
public:

  /// Return size of array (number of entries)
  std::size_t size() const;

};
```

In the above example, the function should be named `size` rather than `get_size`. On the other hand, a function not returning a value but rather taking a variable (by reference) and assigning a value to it, should use the `get_foo` naming scheme, for example:

```
class Parameters:
{
public:

  /// Retrieve all parameter keys
  void get_parameter_keys(std::vector<std::string>& parameter_keys) const;

};
```

Variable names

Use lower-case for variable names and underscore to separate words:

```
Foo foo;
Bar bar;
FooBar foo_bar;
```

Enum variables and constants

Enum variables should be lower-case with underscore to separate words:

```
enum Type {foo, bar, foo_bar};
```

We try to avoid using `#define` to define constants, but when necessary constants should be capitalized:

```
#define FOO 3.14159265358979
```

File names

Use camel caps for file names if they contain the declaration/definition of a class. Header files should have the suffix `.h` and implementation files should have the suffix `.cpp`:

```
FooBar.h
FooBar.cpp
```

Use lower-case for file names that contain utilities/functions (not classes).

Miscellaneous

Indentation

Indentation should be two spaces and it should be spaces. Do **not** use tab(s).

Comments

Comment your code, and do it often. Capitalize the first letter and don't use punctuation (unless the comment runs over several sentences). Here's a good example from `TopologyComputation.cpp`:

```
// Check if connectivity has already been computed
if (connectivity.size() > 0)
  return;

// Invalidate ordering
mesh._ordered = false;

// Compute entities if they don't exist
if (topology.size(d0) == 0)
  compute_entities(mesh, d0);
if (topology.size(d1) == 0)
  compute_entities(mesh, d1);

// Check if connectivity still needs to be computed
if (connectivity.size() > 0)
  return;

...
```

Always use `//` for comments and `///` for documentation (see *Documenting the interface (Programmer's reference)*). Never use `/* ... */`, not even for comments that runs over multiple lines.

Integers and reals

Use `std::size_t` instead of `int` (unless you really want to use negative integers or memory usage is critical).

```
std::size_t i = 0;
double x = 0.0;
```

Placement of brackets and indent style

Use the BSD/Allman style when formatting blocks of code, i.e., curly brackets following multiline control statements should appear on the next line and should not be indented:

```
for (std::size_t i = 0; i < 10; i++)
{
  ...
}
```

For one line statements, omit the brackets:

```
for (std::size_t i = 0; i < 10; i++)
  foo(i);
```

Header file layout

Header files should follow the below template:

```
// Copyright (C) 2008 Foo Bar
//
// This file is part of DOLFIN.
//
// DOLFIN is free software: you can 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 3 of the License, or
// (at your option) any later version.
//
// DOLFIN is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with DOLFIN. If not, see <http://www.gnu.org/licenses/>.
//
// Modified by Bar Foo 2008

#ifndef __FOO_H
#define __FOO_H

namespace dolfin
{

  class Bar; // Forward declarations here

  /// Documentation of class

  class Foo
  {
  public:

    ...

  private:

    ...

  };

}

#endif
```

Implementation file layout

Implementation files should follow the below template:

```
// Copyright (C) 2008 Foo Bar
//
// This file is part of DOLFIN.
//
// DOLFIN is free software: you can 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 3 of the License, or
// (at your option) any later version.
```

```
//
// DOLFIN is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with DOLFIN. If not, see <http://www.gnu.org/licenses/>.
//
// Modified by Bar Foo 2008

#include <dolfin/Foo.h>

using namespace dolfin;

//-----------------------------------------------------------------------------
Foo::Foo() : // variable initialization here
{
  ...
}
//-----------------------------------------------------------------------------
Foo::~Foo()
{
  // Do nothing
}
//-----------------------------------------------------------------------------
```

The horizontal lines above (including the slashes) should be exactly 79 characters wide.

Including header files and using forward declarations

Do not use `#include <dolfin.h>` or `#include <dolfin/dolfin_foo.h>` inside the DOLFIN source tree. Only include the portions of DOLFIN you are actually using.

Include as few header files as possible and use forward declarations whenever possible (in header files). Put the `#include` in the implementation file. This reduces compilation time and minimizes the risk of cyclic dependencies.

Explicit constructors

Make all one argument constructors (except copy constructors) explicit:

```
class Foo
{
  explicit Foo(std::size_t i);
};
```

Virtual functions

Always declare inherited virtual functions as virtual in the subclasses. This makes it easier to spot which functions are virtual.

```
class Foo
{
  virtual void foo();
  virtual void bar() = 0;
};

class Bar : public Foo
{
  virtual void foo();
  virtual void bar();
};
```

Use of libraries

Prefer C++ strings and streams over old C-style `char*`

Use `std::string` instead of `const char*` and use `std::istream` and `std::ostream` instead of `FILE`. Avoid `printf`, `sprintf` and other C functions.

There are some exceptions to this rule where we need to use old C-style function calls. One such exception is handling of command-line arguments (`char* argv[]`).

Prefer smart pointers over plain pointers

Use `std::shared_ptr` and `std::unique_ptr` in favour of plain pointers. Smart pointers reduce the likelihood of memory leaks and make ownership clear. Use `unique_ptr` for a pointer that is not shared and `shared_ptr` when multiple pointers point to the same object.

Documenting the interface (Programmer's reference)

The DOLFIN Programmer's Reference is generated for the DOLFIN C++ library and Python module from the source code using the documentation tool Sphinx. This page describes how to generate the DOLFIN documentation locally and how to extend the contents of the Programmer's Reference.

How to locally build the DOLFIN documentation

The DOLFIN documentation can be generated and built from the DOLFIN source directly as follows:

- Make sure that Sphinx is installed.
- Build DOLFIN (for instructions, see installation_from_source).
- Build the documentation by running:

    make doc

  in the DOLFIN build directory.

For `make doc` to successfully run, the DOLFIN Python module must be installed.

How to improve and extend the DOLFIN Programmer's reference

The documentation contents are extracted from specially formatted comments (docstring comments) in the source code, converted to reStructuredText, and formatted using Sphinx. The syntax used for these specially formatted comments is described below.

To document a feature,

1. Add appropriate docstring comments to source files (see *Syntax for docstring comments*).
2. If you made changes to C++ header files or docstrings in `dolfin_dir/dolfin/swig/*.i` you should update the `dolfin_dir/dolfin/swig/codeexamples.py` file with an example snippet if applicable and run the script `dolfin_dir/dolfin/swig/generate.py` to update the `dolfin_dir/dolfin/swig/docstrings.i` file.
3. Build the documentation as described in *How to locally build the DOLFIN documentation* to check the result.

Syntax for docstring comments

As Sphinx does not allow sections in the markup for class/function documentation, we use *italics* (`*italics*`) and definition lists to group information. This is to keep the markup as simple as possible since the reST source for the Python documentation of classes and functions will be used 'as is' in the docstrings of the DOLFIN module.

Most information can be put in the three sections:

- *Arguments*, which are formatted using definition lists following this structure:

  ```
  *Arguments*
      <name> (<type>)
          <description>
      <name2> (<type>)
          <description>
  ```

  For example:

  ```
  *Arguments*
      dim (int)
          some dimension.
      d (double)
          some value.
  ```

- *Returns*, which is formatted in a similar fashion:

  ```
  *Returns*
      <return type>
          <description>
  ```

  For example:

  ```
  *Returns*
      int
          Some random integer.
  ```

- *Example*, a very small code snippet that shows how the class/function works. It does not necessarily have to be a stand-alone program.

An example of how to document a feature

To make matters more concrete let's consider the case of writing documentation for the member function `closest_cell` of the DOLFIN `Mesh` class. The Python interface to this class is generated by Swig and it is not extended in the Python layer. Writing documentation for other classes and functions in DOLFIN which are not extended or added in the Python layer follow a similar procedure.

The Mesh::closest_cell function is defined in the file dolfin_dir/dolfin/mesh/Mesh.h, and the comment lines and function definition look as follows:

```
/// Computes the index of the cell in the mesh which is closest to the
/// point query.
///
/// *Arguments*
///     point (_Point_)
///         A _Point_ object.
///
/// *Returns*
///     uint
///         The index of the cell in the mesh which is closest to point.
///
/// *Example*
///     .. code-block:: c++
///
///         UnitSquare mesh(1, 1);
///         Point point(0.0, 2.0);
///         info("%d", mesh.closest_cell(point));
///
///     output::
///
///         1
dolfin::uint closest_cell(const Point& point) const;
```

Note that the documentation of a function or class is placed above the definition in the source code. The structure and content follow the guidelines in the previous section.

The Point object is a class like Mesh and it is defined in the FEniCS interface. To insert a link to the documentation of this class use leading and trailing underscore i.e., _Point_. When parsing the comment lines this string will be substituted with either :cpp:class:`Point` or :py:class:`Point` depending on whether documentation for the C++ or Python interface is being generated. The return type, in this case dolfin::uint, will automatically be mapped to the correct Python type when generating the documentation for the Python interface. Note that if you are writing documentation for one of the functions/classes which are added to the Python layer manually you have to add manually the correct links and types.

The example code uses C++ syntax because it is located in the C++ header file. Translating this code to a correct Python equivalent is rather difficult. It is therefore necessary to add example code using the Python syntax manually. This code should be put in the dolfin_dir/dolfin/swig/codeexamples.py which contains a simple dictionary of example code. The dictionary containing only the example code for the example above should look as follows:

```
codesnippets = {
"Mesh":{
"dolfin::uint closest_cell(const Point& point) const":
"""
.. code-block:: python

    >>> mesh = dolfin.UnitSquare(1, 1)
    >>> point = dolfin.Point(0.0, 2.0)
    >>> mesh.closest_cell(point)
    1
"""}
}
```

The first dictionary contains dictionaries for all classes with code examples for each function. Note that the full C++ function signature has been used to identify the function to which the code example belongs.

After adding the documentation to the `Mesh.h` file and Python code example to the `codeexamples.py` file, you have to run the script `dolfin/dolfin/swig/generate.py` to generate the `dolfin/dolfin/swig/docstrings.i` file and then build DOLFIN to update the docstrings in the `dolfin` Python module.

Why is the documentation procedure so elaborate?

The procedure for writing documentation might seem cumbersome so let's have a look at the design considerations which have led to this ostensible case of overengineering.

The Python interface is (partially) generated automatically using Swig from the C++ implementation of DOLFIN. Some classes are extended when building (see the `dolfin/dolfin/swig/*post.i` files) while others are added or extended manually in the Python layer defined in `dolfin/site-packages/dolfin`. While this approach saves a lot of work when implementing the Python interface it puts some constraints on the way the documentation can be handled. In addition we have the following design goals for writing and maintaining the documentation:

Avoid duplicate text In many cases the documentation of a feature will be virtually identical for the C++ and Python interfaces, and since the Python interface is generated from the C++ code, the documentation should be in the C++ source code. To avoid that the documentation on these pages and the comments in the source code (and the implementation itself) diverge, the documentation should be automatically generated from the C++ source code. Therefore the comments should be written using Sphinx markup.

Help in the Python interpreter The documentation of a class/function when running `help(dolfin.foo)` in the Python interpreter should be identical to what can be found online. In practice this means that we have to generate the `dolfin/dolfin/swig/docstrings.i` file using the comments extracted from the C++ source before building the Python interface with Swig.

Simple markup Since the documentation is written directly in the C++ source code, we want markup to be simple such that we have 'code with comments' rather than 'comments with code'. Another reason for preferring simple markup is that it is the raw docstring which will be available from the Python interpreter.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search