FEniCS Course

Lecture 18: FEniCS C++ programming

Contributors
Anders Logg

Two interfaces

Python: quick and easy but sometimes slow if low-level user-intervention is necessary.

C++: potentially faster than the Python interface but not for standard problems, requires quite a bit more programming expertise.

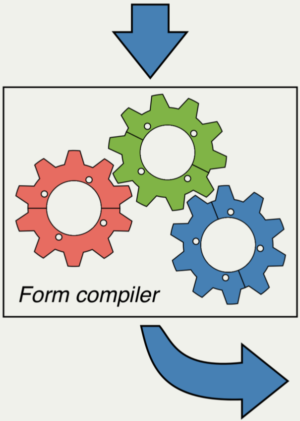
Central classes and functions

- `Mesh`, `Cell`, `Vertex`
- `Matrix`, `Vector`, `LinearSolver`
- `Assembler`, `DirichletBC`
- `FunctionSpace`, `Function`
- `LinearVariationalProblem`, `LinearVariationalSolver`
- `NonlinearVariationalProblem`, `NonlinearVariationalSolver`
- `assemble`, `solve`

But no `dot`, `grad` or `dx`!

Code generation

`ffc -O3` "$-\Delta u = f$"




```cpp
/// Tabulate the tensor for the contribution from a local cell
virtual void tabulate_tensor(double*  A,
                             const double * const *  w,
                             const double*  vertex_coordinates,
                             int cell_orientation) const
{
  // Number of operations (multiply-add pairs) for Jacobian data:      3
  // Number of operations (multiply-add pairs) for geometry tensor:    8
  // Number of operations (multiply-add pairs) for tensor contraction: 11
  // Total number of operations (multiply-add pairs):                  22

  // Compute Jacobian
  double J[4];
  compute_jacobian_triangle_2d(J, vertex_coordinates);

  // Compute Jacobian inverse and determinant
  double K[4];
  double detJ;
  compute_jacobian_inverse_triangle_2d(K, detJ, J);

  // Set scale factor
  const double det = std::abs(detJ);

  // Compute geometry tensor
  const double G0_0_0 = det*(K[0]*K[0] + K[1]*K[1]);
  const double G0_0_1 = det*(K[0]*K[2] + K[1]*K[3]);
  const double G0_1_0 = det*(K[2]*K[0] + K[3]*K[1]);
  const double G0_1_1 = det*(K[2]*K[2] + K[3]*K[3]);

  // Compute element tensor
  A[0] = 0.499999999999999*G0_0_0 + 0.5*G0_0_1 + 0.5*G0_1_0 + 0.5*G0_1_1;
  A[1] = -0.499999999999999*G0_0_0 - 0.5*G0_1_0;
  A[2] = -0.5*G0_0_1 - 0.5*G0_1_1;
  A[3] = -0.499999999999999*G0_0_0 - 0.5*G0_0_1;
  A[4] = 0.499999999999999*G0_0_0;
  A[5] = 0.5*G0_0_1;
  A[6] = -0.5*G0_1_0 - 0.5*G0_1_1;
  A[7] = 0.5*G0_1_0;
  A[8] = 0.5*G0_1_1;
}
```

Writing a form file

```
element = FiniteElement("Lagrange", triangle, 1)

u = TrialFunction(element)
v = TestFunction(element)
f = Coefficient(element)

a = dot(grad(u), grad(v))*dx
L = f*v*dx
```

Similar to Python (and UFL is actually a Python extension).

Replace `FunctionSpace` by `FiniteElement`.

Calling the form compiler

Bash code

```
ffc -l dolfin MyForms.ufl
```

This generates `MyForms.h` from `MyForms.ufl`.

Type `man fcc` for command-line options (such as optimizations).

Including forms

C++ code

```
#include "MyForms.h"

int main()
{
  ...

  MyForms::BilinearForm a(V, V);
  MyForms::LinearForm L(V);

}
```

Define forms in `.ufl` file.
Generate C++ code using FFC.
Include generated C++ code in C++ program.
Instantiate form objects in C++ program.

Attaching coefficients

C++ code

```
L.f = f;

L.set_coefficient(0, f);

L.set_coefficient("f", f);

L.set_coefficients(...);
```

A `Coefficient` is either an `Expression`, a `Constant` or a `Function`.

Assembly and solve

C++ code

```
assemble(A, a);
assemble(b, L);
```

C++ code

```
solve(a == L, u, bc);
solve(F == 0, u, bc, J);
```

Using shared pointers

C++ code

```
Mesh* mesh = new Mesh(); // raw pointer
delete mesh;             // delete necessary
```

C++ code

```
std::shared_ptr<Mesh> mesh(new Mesh())
```

C++ code

```
auto mesh = std::make_shared<Mesh>();
```

Some FEniCS (DOLFIN) C++ functions expect a shared pointer, while others expect a reference.

Building the program (hard option)

```
cmake_minimum_required(VERSION 3.5)

set(PROJECT_NAME demo_poisson)
project(${PROJECT_NAME})

# Set CMake behavior
cmake_policy(SET CMP0004 NEW)

# Get DOLFIN configuration data
find_package(DOLFIN REQUIRED)
...

# Add executable
add_executable(${PROJECT_NAME} main.cpp)

# Target libraries
target_link_libraries(${PROJECT_NAME} ${DOLFIN_LIBRARIES})
```

Building the program (easy option)

- Copy `CMakeLists.txt` from one of the DOLFIN demos
- Edit: add source files and rename executable
- Make an out-of-source build:

C++ code

```
mkdir build
cd build
cmake ..
make
./myprogram
```

Solving Poisson's equation

We return to the simple Poisson equation and investigate how to write a FEniCS C++ solver:

$$\begin{aligned} -\Delta u &= f && \text{in } \Omega \\ u &= u_0 && \text{on } \Gamma_D \\ \partial u/\partial n &= g && \text{on } \Gamma_N \end{aligned}$$

We will take

$$f(x,y) = 10 \cdot \exp(-((x-0.5)^2 + (y-0.5)^2)/0.02)$$

Variational problem

Find $u \in V$ such that

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x + \int_{\Gamma_N} g v \,\mathrm{d}s$$

for all $v \in \hat{V}$

Form file: `Poisson.ufl`

```
element = FiniteElement("Lagrange", triangle, 1)

u = TrialFunction(element)
v = TestFunction(element)
f = Coefficient(element)
g = Coefficient(element)

a = inner(grad(u), grad(v))*dx
L = f*v*dx + g*v*ds
```

Calling the form compiler

Bash code

```
ffc -l dolfin Poisson.ufl
```

This generates `Poisson.h` from `Poisson.ufl`.

C++ implementation: `main.cpp` (1/3)

C++ code

```
#include <dolfin.h>
#include "Poisson.h"

using namespace dolfin;

class Source : public Expression
{
  void eval(Array<double>& values,
            const Array<double>& x) const
  {
    double dx = x[0] - 0.5;
    double dy = x[1] - 0.5;
    values[0] = 10*exp(-(dx*dx+dy*dy)/0.02);
  }
};
```

C++ implementation: main.cpp (2/3)

C++ code

```
int main()
{
  auto mesh =
   std::make_shared<UnitSquareMesh>(32, 32);
  auto V
   std::make_shared<Poisson::FunctionSpace>(mesh);
  auto u0 =
   std::make_shared<Constant>(0.0);
  auto boundary =
   std::make_shared<DirichletBoundary>();

  DirichletBC bc(V, u0, boundary);

  Poisson::BilinearForm a(V, V);
  Poisson::LinearForm L(V);

  auto f = std::make_shared<Source>();
  auto g = std::make_shared<dUdN>();
  L.f = f;
  L.g = g;
```

C++ implementation: `main.cpp` (3/3)

C++ code

```
  Function u(V);
  solve(a == L, u, bc);

  File file("poisson.pvd");
  file << u;

  plot(u);

  interactive();

  return 0;
}
```