Documentation for modules ale, fsi

Of the DOLFIN software package found at www.fenics.org

1 ALE

This ALE solver is based on the NSE solver. This means that it incorporates all the functionality of the NSE solver but with the added bonus of being able to move the mesh. The most important changes from NSE to ALE will be covered below. The ALE solver also uses mesh smoothing (Laplace). This is required as moving the mesh can result in bad triangles or tetrahedrons.

1.1 New BoundaryCondition Class

The boundary conditions are no longer set via the BoundaryCondition class, instead we use ALEBoundaryCondition. This class has two additions from the BoundaryCondition class

- Set boundary values according to the reference coordinate system

  This allows the user to specify boundary conditions in terms of the reference coordinate points. The reference coordinate system is just the mesh at t=0. Since the mesh will change over time, there is no way of knowing whether certain boundary conditions are valid for the duration of the simulation. The reference coordinate system however, will always be the same and will never change during the simulation.
- Set boundary values using the mesh velocity function

  To correctly specify boundary conditions, we have to account for the mesh velocity at the boundary. We can easily do this by calling the mesh velocity function.

1.2 New Function Class

- Specify an external function which acts on the boundary of the mesh.

  Allows us to move the mesh boundary.

1.2.1 Examples of the above

Below is an example of a user created C++ file for running a simulation using the ALE solver.

```
#include <dolfin.h>

using namespace dolfin;

//-----------------------------------------------------------------
class ALEExtFunction : public ALEFunction
{
  real eval(const Point& p, const Point& r, unsigned int i)
  {
    // no horizontal mesh displacement
    if (i == 0) return 0.0;

    if (i == 1) return ( 0.05*sin((p.x() + time())*3 ) );
  }
};
//-----------------------------------------------------------------------------
class ForceFunction_2D : public Function
{
  real eval(const Point& p, unsigned int i)
  {
    if (i==0) return 0.0;
    if (i==1) return 0.0;
    dolfin_error("Wrong vector component index");
```

This function lets us specify how the boundary of the mesh behaves. In this case, we have sinusoidal movement.

Point& r – is the reference coordinate point.

```
    return 0.0;
  }
};
//-----------------------------------------------------------------------
class BC_Momentum_2D : public ALEBoundaryCondition
{

  void eval(BoundaryValue& value,
              const Point& p, const Point& r,
              unsigned int i)
  {
     real bmarg = 1.0e-3;

     if (i==0){  // x direction
       if ( p.x() < (2.0 - DOLFIN_EPS - bmarg))
            value.set(0.0 + w->eval(p,r,i));
       if ( r.x() < (0.0 + DOLFIN_EPS + bmarg)){
            value.set((1 - r.y()) * r.y() * 4 + w->eval(p,r,i));
       }

    } else if (i==1){  // y direction
      if ( p.x() < (2.0 - DOLFIN_EPS - bmarg))
           value.set(0.0 + w->eval(p,r,i));
    } else{
      dolfin_error("Wrong vector component index");
    }
  }
};
//-----------------------------------------------------------------------
class BC_Continuity_2D : public ALEBoundaryCondition
{
  void eval(BoundaryValue& value,
              const Point& p, const Point& r,
              unsigned int i)
  {
    real bmarg = 1.0e-3;

    if (p.x() > (2.0 - DOLFIN_EPS - bmarg))
      value.set(0.0);
  }
};
//-----------------------------------------------------------------------
int main(int argc, char* argv[])
{
  dolfin_init(argc, argv);

  Mesh mesh("rect_ale_ns.xml");

  for (int grain = 0; grain < 4; grain++)
    mesh.refineUniformly();

  BC_Momentum_2D   bc_mom;
  BC_Continuity_2D bc_con;
  ForceFunction_2D f;
  ALEExtFunction   e;

  ALESolver::solve(mesh, f, bc_mom, bc_con, e);

  return 0;
}
```

New class ALEBoundaryCondtion.

Function* w – is the mesh velocity at the boundary.

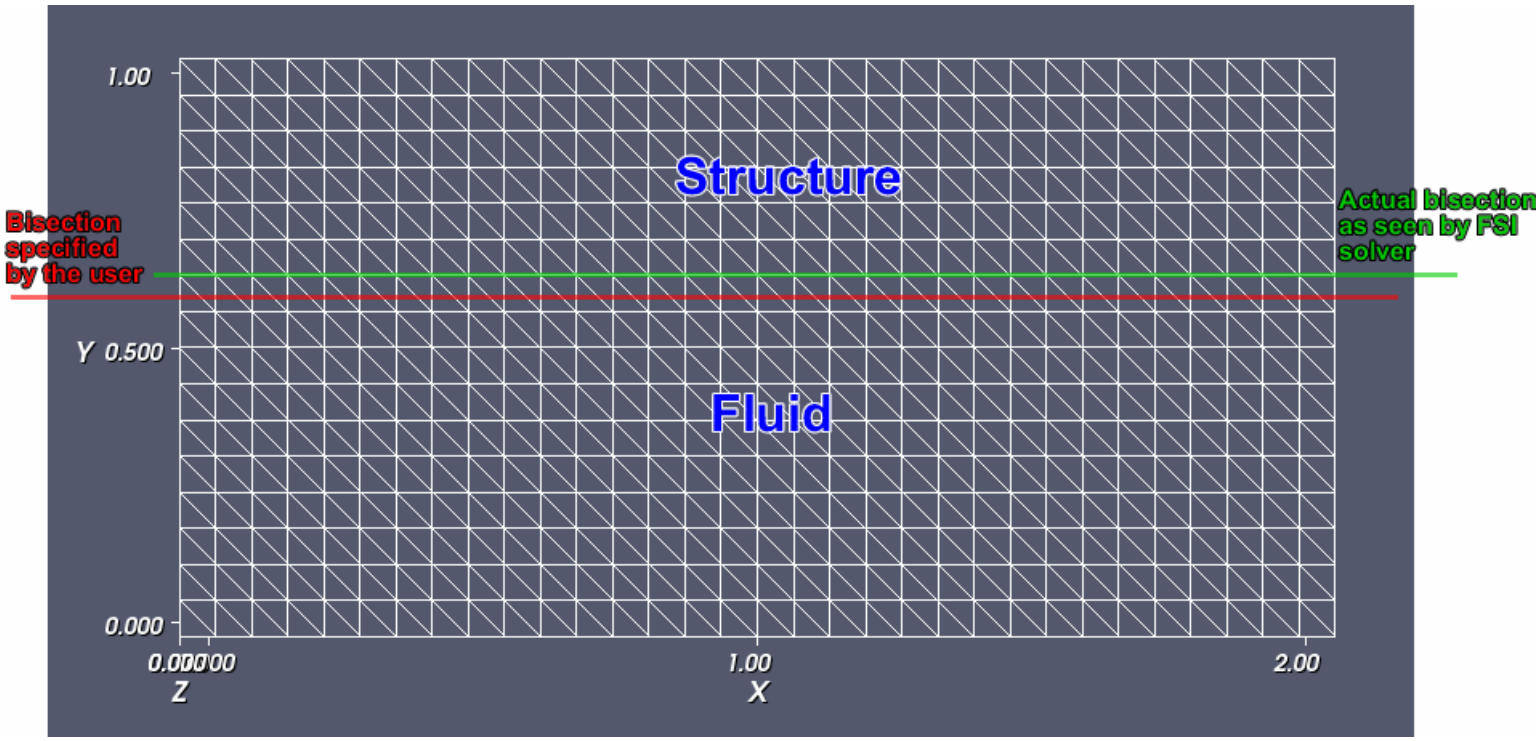
We then pass all the information to the ALE solver.

2 FSI

This module brings with it the ability to simulate fluid-structure interaction. Note that this is still in development. To use this solver, the user defines a function which bisects the mesh into regions of structure and into regions of fluid. The boundary conditions are the same as for the ale solver module and thus conditions can be specified on the reference coordinate system.
Largely, the FSI solver module combines the incompressible elasticity functionality together with the ALE solver functionality. This means that the user will specify parameters both for the structure (elasticity parameters) and the fluid (navier-stoke parameters). As opposed to the ALE module, there are no additions in terms of classes however; we will reuse the ALEBoundaryCondition class.

2.1 Bisection Function explained

The bisection function is specified by the user. For the implementation to work correctly, the function must either return a 0 or a 1, corresponding to structure and fluid respectively. Because the implied interface which separates fluid and structure consists of edges, the bisection function was designed to be fluid biased. This means that if the implied user interface for the separation layer of the structure and a fluid happen to pass through a cell, the cell will be considered as part of a fluid. An example of what happens is shown in the figure below.



A bisection function as shown above in red can look like the following

```
class Bisect : public Function
{
   real eval(Point& p, unsigned int i)
   {
        if (p.y() > 0.555) return 0.0;
        return 1.0;
    }
}
```

This method of bisecting has both pros and cons. Pros are that is easy to get something off the ground and that the boundary can be defined using an external geometry tool since the user specified bisection line moves to the nearest edge, and edge which can be drawn by a program such as gmesh. Cons are that its usability is limited, also one needs to open the mesh in a mesh viewer such as paraview to see where the cells (and thus edges) are located.

Below is an example of a problem using the FSISolver, everything should be quite self-explanatory:

```
#include <dolfin.h>

using namespace dolfin;


// Defines the fluid domain, conversely !FluidDomain is structure
// domain.
bool FluidDomain(const Point& r)
{
  return (0.2 < r.y() && r.y() < 0.8);
}
//-----------------------------------------------------------------------
// Force term
class ForceFunction_2D : public Function
{
  real eval(const Point& p, unsigned int i)
  {
    return 0.0;
  }
};
//-----------------------------------------------------------------------
// Boundary condition for momentum equation
class BC_Momentum_2D : public ALEBoundaryCondition
{
  real amp()
  {
    real pi = 3.14;

    return (1.0+0.5*sin(4.0*pi*time())+0.1*sin(2.0*pi*time()));
  }
  //-----------------------------------------------------------------------
  void eval(BoundaryValue& value, const Point& p, const Point& r, unsigned
int i)
  {
     real bmarg = 1.0e-3;

     if (p.x() < (0.0 + DOLFIN_EPS + bmarg))
       value.set(0.0);

     if (!FluidDomain(p)) {                        // structure
       if (p.x() < (0.0 + DOLFIN_EPS + bmarg) ||  // fixed at ends
              p.x() > (2.0 - DOLFIN_EPS - bmarg))
            value.set(0.0);
     } else {                                      // fluid
       if (i == 0)
            if (p.x() < (0.0 + DOLFIN_EPS + bmarg))  // inflow
              value.set(amp()*(0.8 - p.y()) * (p.y() - 0.2) * 4);
     }
  }
};
//-----------------------------------------------------------------------
// Boundary condition for continuity equation
class BC_Continuity_2D : public ALEBoundaryCondition
```

```
{
  void eval(BoundaryValue& value, const Point& p, const Point& r, unsigned
int i)
  {
    real bmarg = 1.0e-3;

    if (p.x() > (2.0 - DOLFIN_EPS - bmarg))
      if (FluidDomain(p))
           value.set(0.0);
  }
};
//-----------------------------------------------------------------------
class BisectionFunction : public Function
{
  real eval(const Point& r, unsigned int i)
  {
    if (FluidDomain(r)) return 1; // fluid
    return 0;                     // structure
  }
};
//-----------------------------------------------------------------------
int main(int argc, char* argv[])
{
  dolfin_init(argc, argv);

  // mesh rectangle [0,0] - [2,1]
  Mesh mesh("rect_ale_ns.xml");

  //need to refine mesh
  for (int grain = 0; grain < 4; grain++)
    mesh.refine();

  BC_Momentum_2D    bc_mom;
  BC_Continuity_2D  bc_con;
  ForceFunction_2D  f;
  BisectionFunction bisect;

  real rhof = 1;          // fluid:     density
  real nu   = 1.0/3900.0; // fluid:     viscosity
  real rhos = 1;          // structure: density
  real E    = 40;         // structure: Young's modulus
  real elnu = 0.3;        // structure: Poisson's ratio
  real T    = 2.0;        // final time
  real k    = 1e-5;       // time step size

  FSISolver::solve(mesh, f, bc_mom, bc_con, bisect, rhof, rhos, E, elnu,
nu, T, k);
  return 0;
}
```

3 The end