Cite as: Siggeirsson, E.M.V.: Force based motion of a submerged object using immersed boundary metod.
In Proceedings of CFD with OpenSource Software, 2016,
`http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2016`

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Project work:

Force based motion of a submerged object using immersed boundary method

Developed for Foam Extend-4.x

Author:
Elias Mikael Vagn Siggeirsson
eliass@chalmers.se

Peer reviewed by:
Ishaa Markale

January 19, 2017

Contents

Learning outcomes

The main requirements of a tutorial is that it should teach the four points: How to use it, The theory of it, How it is implemented, and How to modify it. Therefore the list of learning outcomes is organized with those headers.

The reader will learn:

How to use it:

- How to change and modify a specific tutorial from the `foam extend` tutorials
- How to use the immersed boundary solver

The theory of it:

- The basic theory of the immersed boundary method

How it is implemented:

- How to implement a new class into the `solidBodyMotion` library
- How to use information in one library from a different one

How to modify it:

- How to modify the `movingCylinderInChallerIco` tutorial case to simulate a force based motion
- How to modify a already existing class

Chapter 1

Tutorial submerged object

1.1 Introduction

The purpose of this tutorial is to describe how to set up a simulation for a submerged object in water. The immersed boundary method is used in `OpenFOAM` to be able to handle large movements of the object along with calculating the fluid forces. A new class is defined in the `solidBodyMotion` library, named `forceBased`. There the motion, based on the forces is calculated. Only translation is considered for simplicity reasons as including rotation as well would increase the complexity of the coupling between the fluid and body motion solvers. This allows a simple form of the Newton's second law to be used. This tutorial handles the implementation of the new class along with describing which libraries are needed and how to set up the test case.

1.2 Theory

The tools needed for this tutorial are quite common CFD tools. The one that might be less common than the others is the immersed boundary method. For the solid body motion the Newton's second law is used.

1.2.1 Body motion

For calculating the movement of the solid body, Newton's second law is used. That states that $F = ma$. Integrating twice over time gives the following equation for translation

$$x_2 = x_1 + v_0 * \Delta t + a * (\Delta t)^2$$

where x_2 is the position calculated, x_1 is the position of previous time step, v_0 is the velocity calculated from previous time step, Δt is the time step and a is the acceleration calculated from the external forces on the object and the object mass. The forces are then obtained through the immersed boundary class in `OpenFOAM`. The forces that the immersed boundary class calculates exclude the gravitational force and needs to be added if to be included.

1.2.2 Immersed boundary method

The immersed boundary method (IBM) is a technique for defining boundary conditions in a grid that does not necessarily need to be fitted to the geometry. For more detailed description see references [**1**, **2**, **3**]. The method was originally developed for simulation of cardiac mechanics and associated blood flow by by Charles Peskin [**4**]. There are two main approaches in implementing the IBM for CFD simulations. Forcing methods and Reconstruction methods. In `OpenFoam` the former one is used. The forcing methods define a source term in the governing equation to account for the effect of the immersed boundary. By applying the forcing method to the Navier-Stokes equations the following form is obtained:

$$\frac{\partial \Phi}{\partial t} + \text{convection}(x,t,\Phi) = \text{diffusion}(x,t,\Phi) + F(x,t,\Phi)$$

where F(x,t,Φ) is the source term for the immersed boundary. The forcing method will introduce a smeared boundary between the solid and fluid domains which means the interface between the two domains will not be a clear sharp line.
The advantages of the IBM over the body fitted method are:

- A substantially simpler grid can be generated for complex geometry and large body movements without having to rely on unstructured solvers. Figure 1.1 is an example of an immersed boundary and a non-geometry fitted grid.
- Inclusion of body motion or deformation is relatively simple as the background grid is stationary and non-deforming. Therefore the IBM does not need a specific time meshing strategy.
- Implementing the IBM should be easy as the source terms are added to existing code.

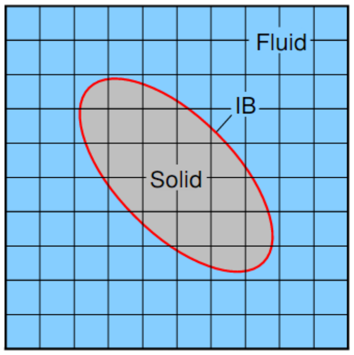

Figure 1.1: Immersed boundary on a structured grid

The disadvantages are however

- Special techniques are needed to implement the boundary condition.
- Lack of control over local meshing.
- Difficult to use the forcing methods when Neumann boundary conditions are needed.

The IBM is implemented into `OpenFOAM` using discrete forcing approach with direct imposition of boundary conditions. For a certain variable the value in the immersed boundary cell is calculated by interpolating the neighbouring cells and the specified boundary conditions for the immersed boundary point. The points and cells configurations can be seen in Figure 1.2.

1.3 Code modification

When browsing through the tutorials available in `foam extend` an obvious initial step is to build upon the `movingCylinderInChannelIco` tutorial as it simulates a moving cylinder in a channel oscillating back and forth according to a prescribed sinus wave. The tutorial is located at

`movingCylinderInChannelIco tutorial`
```
cd $FOAM_TUTORIALS/immersedBoundary/movingCylinderInChannelIco
```

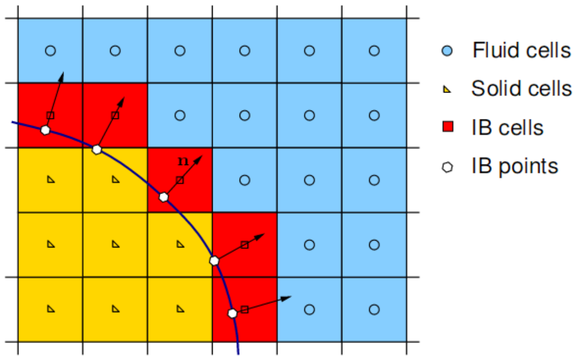

Figure 1.2: Implementation of IBM

By looking through the libraries used in the tutorial it can been seen that the `solidBodyMotion` library is the one that controls the movement of the object. The library however has no class which offers force based motion. The only available classes are dependent on a specific function which describes the object motion. Therefore a new class is created, called `forceBased`, which is built on the `translation` class. The `solidBodyMotion` library is located at

solidBodyMotion
```
cd $FOAM_SRC/dynamicMesh/meshMotion/solidBodyMotion
```

where all the classes already available can be found.
What the new class needs to do is to read the immersed boundary forces from the `immersedBoundaryForce` library and transfer that information to the `solidBodyMotion` library. From the forces the translation of the object can be calculated based on Newton's second law like described above.

1.3.1 The `forceBased` class

For simplicity reasons and to keep the original classes and libraries unchanged the `translation` class in the `soldiBodyMotion` library is copied to the users source directory. Both the class it self is copied as well as the `Make` directory but that is necessary to compile the class later on.

```
mkdir -p $WM_PROJECT_USER_DIR/src/dynamicMesh/meshMotion/solidBodyMotion/
cd $WM_PROJECT_USER_DIR/src/dynamicMesh/meshMotion/solidBodyMotion/
cp -r $FOAM_SRC/dynamicMesh/meshMotion/solidBodyMotion/translation forceBased
cp -r $FOAM_SRC/dynamicMesh/meshMotion/solidBodyMotion/{Make, lnInclude} .
```

Then the `translation` class is renamed to the `forceBased` class.

```
mv forceBased/translation.C forceBased/forceBased.C
mv forceBased/translation.H forceBased/forceBased.H
```

In the new `forceBased.*` files few modifications are needed. Changing the class names in both classes can easily be done with the following command

```
sed -i -e '/translation/forceBased/g' forceBased/forceBased.*
```

First, start with the `forceBased.H` file. To include the `immersedBoundaryForces` library in the `forceBased` class, add `immersedBoundaryForces.H` to the `forceBased.H` file, bellow the inclusion of `solidBodyMotionFunction.H`. This makes sure that the new class can take information from both libraries.

forceBased.H
```
#include "solidBodyMotionFunction.H"
#include "immersedBoundaryForces.H"
```

Now to the data and functions. Instead of the following data declaration

forceBased.H
```
        //- Velocity
        vector velocity_;

        //- Ramoing time scale
        scalar rampTime_;
```

the following is needed instead

forceBased.H
```
        vector g_;
        scalar still_;
        scalar mass_;
        mutable vector velocity_;
        mutable vector oldPosition_;
        mutable scalar currentTime_;
```

For the functions the following function declaration

forceBased.H
```
//- Velocity ramping factor resulting form rampTime_value
        scalar rampFactor() const
```

has to be replaced with the following declaration

forceBased.H
```
        //- Position calculations
        vector newPosition() const;
        vector newVelocity() const;
```

Now to the `forceBased.C` file. A new function that calculates the position of the object each iteration is needed. For simplicity reasons that function replaces the `rampTime()` function that was defined in the `translation` class. The follwing

forceBased.C
```
Foam::scalar
Foam::solidBodyMotionFunctions::translation::rampFactor() const
{
    const scalar t = time_.value();

    if (t < rampTime_)
    {
        // Ramping region
        return sin(pi/(2*rampTime_)*t);
    }
    else
    {
        // Past ramping region
        return 1;
    }
}
```

has to be replaced with the new `newPosition()` function, which is defined as

forceBased.C
```
Foam::vector
Foam::solidBodyMotionFunctions::forceBased::newPosition() const
{
        scalar t = time_.value();
    scalar dT = time_.deltaT(.value());

    if (t < (still_*dT+currentTime_) )
    {
            return oldPosition_;
    }
    else if (t< 2*(still_*dT))
    {
        if (t > currentTime_)
            {
                currentTime_ = t;
                vector gravityForce = mass_*g_;
                Info << "Gravity: " << g_ << endl;
                Info << "Gravitational Force: " << gravityForce << endl;
                vector pressureForces = imBForces_.calcForcesMoment().first().first();
                vector viscousForces = imBForces_.calcForcesMoment().first().second();
                vector totalForce = gravityForce+pressureForces+viscousForces;

                Info << "Pressure Force: " << pressureForces << endl;
                Info << "Viscous Force: " << viscousForces << endl;
                Info << "total force: " << totalForce << endl;
                Info << "Delta t: " << dT << endl;
                Info << "old positon: " << oldPosition_ << endl;

                vector velocity(0.01,0,0);
                vector dx = velocity*dT;

                Info << "old velocity: " << velocity_ << endl;
                velocity_ = vector(0,0,0);//dx/dT;//velocity_+totalForce/mass_*dT;
                Info << "new velocity: " << velocity_ << endl;

                vector newPos=oldPosition_+dx;

                Info << "new position: " << newPos << endl;
                Info << "new old positon: " << oldPosition_ << endl;
                oldPosition_ = newPos;
                return oldPosition_;
            }
            else
            {
                return oldPosition_;
            }
    }
        else
    {
            if (t > currentTime_)
            {
                        currentTime_ = t;
                        vector gravityForce = mass_*g_;
                        Info << "Gravity: " << gravityForce << endl;
            vector pressureForces = imBForces_.calcForcesMoment(.first(.first()));
            vector viscousForces = imBForces_.calcForcesMoment(.first(.second()));
                    vector totalForce = gravityForce+pressureForces+viscousForces;
            Info << "Force first: " << pressureForces << endl;
            Info << "Force second: " << viscousForces << endl;
            Info << "total force pos: " << totalForce << endl;
            Info << "Delta t: " << dT << endl;
                Info << "old position: " << oldPosition_ << endl;

            vector velocity_ = velocity_+totalForce*dT;

            vector newPos=oldPosition_+velocity_*dT+0.5*totalForce/mass_*pow(dT,2);

                    oldPosition_ = newPos;
                    Info << "new position: " << newPos << endl;
                    Info << "old position: " << oldPosition_ << endl;
            Info << "velocity, pos: " << velocity_ << endl;
                    oldPosition_ = newPos;
                    return oldPosition_;
                    }
        else
            {
            return oldPosition_;
            }
    }
}
```

The `forcebased.C` file is set to run with a stationary object for the first `still_` time steps and then with a constant velocity of 0.01 for the next `still_` time steps and then to be set loose.
The whole constructor then needs to be changed to the following

forceBased.C

```
Foam::solidBodyMotionFunctions::forceBased::forceBased
(
    const dictionary& SBMFCoeffs,
    const Time& runTime
)
:
    solidBodyMotionFunction(SBMFCoeffs, runTime),
        g_(0,0,0),
    velocity_(0,0,0),
    oldPosition_(0,0,0),
    currentTime_(0.0),
    imBForces_("test", time_.lookupObject<objectRegistry>(polyMesh::defaultR
egion), SBMFCoeffs_, true)

{
    read(SBMFCoeffs);
}
```

Since the new position is calculated in a separated function the `transformation` function is changed to

forceBased.C

```
Foam::septernion
Foam::solidBodyMotionFunctions::forceBased::transformation() const
{
    septernion TR;

        TR = septernion
        (
            newPosition(),
            quaternion::I
        );

    return TR;
}
```

and the velocity function changed to

forceBased.C

```
Foam::septernion
Foam::solidBodyMotionFunctions::forceBased::velocity() const
{
    septernion TV
    (
        velocity_,
        quaternion::I/time_.deltaT().value()
    );

    Info<< "solidBodyMotionFunctions::forceBased::transformation(): "
        << "Time = " << time_.value() << " velocity: " << TV << endl;

    return TV;
}
```

To be able to read in the values that are specified in the input file a small modification to the `read` class is needed.

forceBased.C

```
        rampTime_ = readScalar(SBMFCoeffs_.lookup("rampTime"));
```

is replaced with

forceBased.C

```
    SBMFCoeffs_.lookup("gravity") >> g_;
    SBMFCoeffs_.lookup("mass") >> mass_;
    SBMFCoeffs_.lookup("stationary") >> still_;
```

to be able to read in the object mass and to define how many timesteps the object is set to be still before taking the next movement.
To be able to compile the new class some modifications are needed to the `Make` directory. A new `Make/files` file is needed that only refers to the new class.

Make/files

```
forceBased/forceBased.C
LIB = $(FOAM_USER_LIBBIN)/libforceBasedMotion
```

Few additions are then needed to the `Make/options` file to be able to read in the immersed boundary forces into the new class. After those additions the file looks like this

Make/options

```
EXE_INC = \
    -I$(LIB_SRC)/dynamicMesh/dynamicMesh/lnInclude \
    -I$(LIB_SRC)/immersedBoundary/immersedBoundaryForce/lnInclude \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/postProcessing/functionObjects/forces/lnInclude \
    -I$(LIB_SRC)/surfMesh/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude \
    -I$(LIB_SRC)/immersedBoundary/immersedBoundaryTurbulence/lnInclude \
    -I$(LIB_SRC)/dynamicMesh/meshMotion/solidBodyMotion/lnInclude

LIB_LIBS = \
    -lforces \
    -lincompressibleTransportModels \
    -lincompressibleRASModels \
    -lincompressibleLESModels \
    -lbasicThermophysicalModels \
    -lspecie \
    -lcompressibleRASModels \
    -lcompressibleLESModels \
    -lfiniteVolume \
    -lmeshTools \
    -lsurfMesh \
    -lsampling \
    -ldynamicMesh \
    -L$(FOAM_USER_LIBBIN) \
    -limmersedBoundary \
    -limmersedBoundaryTurbulence \
    -limmersedBoundaryForceFunctionObject
```

Now the new class should be ready for compilation. That is done through the `wmake` command along with the `libso` tag which tels the compiler to compile it as a library.

compile library

```
wmake libso
```

1.4 Case Setup

To set up the simulation the tutorial `movingCylinderInChannelIco` is used since it is close to what is needed and only simple modifications are performed. Begin by copying the tutorial to the users run directory (note that the first two lines are one command)

```
cp -r $FOAM_TUTORIALS/immersedBoundary/movingCylinderInChannelIco
$FOAM_RUN/movingCylinderInChannelForceBasedIco
cd $FOAM_RUN/movingCylinderInChannelIco
```

For the immersed boundary force calulations to take place, `libforceBasedMotion.so` has to be added to the libraries in the `controlDict` file. All the libraries included in the `controlDict` file are then

controlDict

```
    "liblduSolvers.so"
    "libimmersedBoundary.so"
    "libimmersedBoundaryDynamicFvMesh.so"
    "libforceBasedMotion.so"
```

To be able to see each time step in post-processing the write interval in the controlDict file has to be changed. The time step in the test case is a bit to large and therefore it is decreased by a factor of 10

controlDict
```
        deltaT                          0.01;
        writeControl        runTime;
        writeInterval        0.01;
```

The other file where a modification is needed is the dynamicMeshDict file. The settings for the ibCylinder are then changed to the following

dynamicMeshDict
```
                ibCylinder
        {
            solidBodyMotionFunction     forceBased;
            forceBasedCoeffs
            {
                    gravity (0 0 0);
                    mass    1;
                    stationary    20;

                    patches ( ibCylinder );

                    pName       p;
                    UName       U;
                    rhoName     rhoInf;
                    rhoInf      1000;

                    log         true;
                    CofR        ( 0 0 0 );

            //        Aref 0.5;
                    Uref 1;
            }
        }
```

The settings that are specified here are defined in a way that there is no gravity included, the mass is 1kg and the stationary is set to 20 which means that the simulations will take 20 time steps before moving the object with a constant velocity for the next 20 time steps and then sets the object loose after that. This is done so the flow will be fully developed before moving the object. Additionally the density is changed to represent water or 1000.
Now the boundary conditions are next as the tutorial case that is used is for a channel flow. For this case however a larger domain with zero gradient is used instead of the walls. Also the inlet velocity is lowered to account for the increase in density to ensure that the Reynolds number is not to high. Under the boundary field the following boundaries are needed to be changed to the following

0_org/U
```
    in
    {
        type fixedValue;
        value uniform (0.01 0 0);
    }

    out
    {
        type inletOutlet;
        inletValue uniform (0 0 0);
        value uniform (0.01 0 0);
    }

    top
    {
        type zeroGradient;
    }

    bottom
    {
        type zeroGradient;
    }
```

The other boundary file, `0_org/p` is unchanged from the tutorial case.
In the `save` directory there are two files. One is the `boundary` file which defines the boundary patches of the domain while the other is the `blockMeshDict` which defines the size and the number of nodes in the domain. As the domain is changed, through the `blockMeshDict` file, the patches definition in `boundary` has to be changed correspondingly. For this specific tutorial the following changes are needed

save/blockMeshDict

```
vertices
(
    (-2   -1.0   0)
    (2    -1.0   0)
    (2     1.0   0)
    (-2    1.0   0)
    (-2   -1.0   0.1)
    (2    -1.0   0.1)
    (2     1.0   0.1)
    (-2    1.0   0.1)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (200 100 1) simpleGrading (1 1 1)
);
```

After doing this modification the `blockMeshDict` file has to be moved to the `polyMesh` directory.

constant/polyMesh/blockMesh

```
cp save/blockMeshDict constant/polyMesh/
```

Then due to the fact that the total number of grid points has been changed the patches definition has to be recalculated

save/boundary

```
   ibCylinder
    {
        type            immersedBoundary;
        nFaces          0;
        startFace       3650;

        internalFlow    no;
    }
    in
    {
        type            patch;
        nFaces          25;
        startFace       3650;
    }
    out
    {
        type            patch;
        nFaces          25;
        startFace       3675;
    }
    top
    {
        type            patch;
        nFaces          75;
        startFace       3700;
    }
    bottom
    {
        type            patch;
        nFaces          75;
        startFace       3775;
    }
    frontAndBack
    {
        type            empty;
        nFaces          3750;
        startFace       3850;
    }
```

The modification of the `boundary` file is moved to the right location when the `Allrun` script is executed. The modifications to the `blockMeshDict` and `boundary` files are not necessary to be able to run the code and can be done in a different manner than described above. The changes only have to be consistent

1.5 Running the code

It is quite simple to run this case as the only thing needed is to run the provided script called `Allrun`

Allrun

```
./Allrun
```

1.6 Post-processing

To post-process the results in Paraview the following command can be used

paraFoam

```
paraFoam -nativeReader
```

All the needed data should be imported by pressing the green `Apply` button in Paraview. Then the variable that is shown can be changed to `cellIbMask` so the immersed boundary is shown like in Figure 1.3.

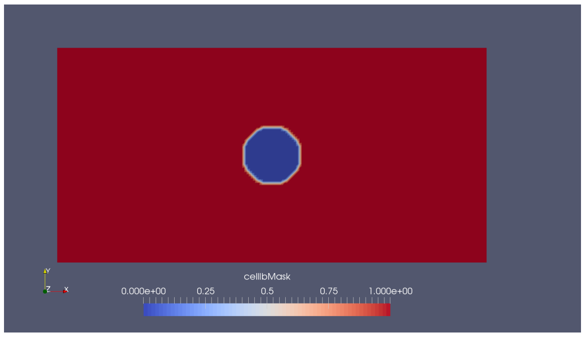

Figure 1.3: The case setup

1.6.1 Results

As mentioned before the forces oscillate but the reason for that is unknown for me as I have spent quite some time trying to figure it out. I however know that other `openFoam` users have experienced similar problems using this solver for similar things. Lee et al [**5**] discusses a similar behavior where oscillations are giving problems when using the immersed boundary method. The simulations that generates the results from Figure 1.4 are set up in a way that the object is stationary for the 20 first time steps, then the object is set to have a constant velocity which is of the same magnitude as the main flow and then after of total of 40 time steps the object is set loose and the forces define the motion. The figure is created by using the `plot.py` code given as part of the tutorial files. The reason for the oscillations are that when the object is set loose there is a force in the negative x-direction which causes the object to move upstream. This generates force pushing the object downstream with a much higher velocity than the main flow velocity. That again causes the forces to be in the negative x-direction and move the object upstream, resulting in an oscillation of the objects movement.

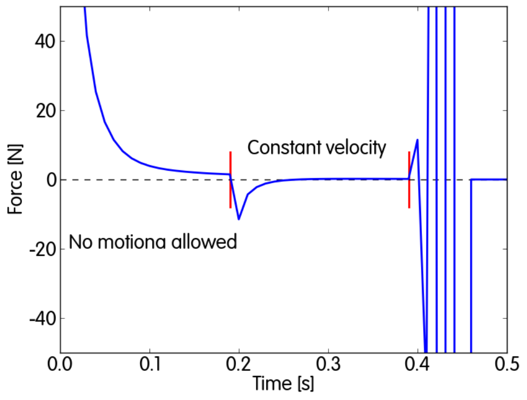

Figure 1.4: Force based motion of the object

For comparison the force signal from the tutorial case is shown in Figure 1.5. As can be seen in the figure the forces are oscillating in that case but since the motion is prescribed it has no effect.

1.7 Conclusion

Originally the purpose of this project was to simulate s submerged wing using the immersed boundary method but due to challenges during the project work the simulated geometry was simplified to a cylinder as the methodology should be the same regardless off the geometry used. Using the cylinder

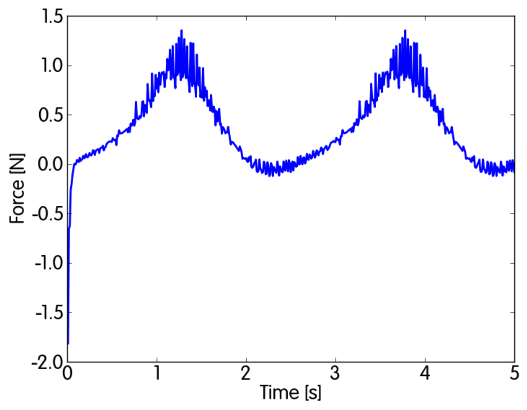

Figure 1.5: Forces for tutorial case

meant as well that the force should only be in the defined x-direction and therefore a bit easier to debug the code as well as to find some fundamental errors. The simulations however were not without troubles as they diverged due to the oscillating force, resulting in non-physical movements. The work however shows how to couple the `solidBodyMotion` and the `immersedBoundaryForce` libraries to calculate the forces acting on the object using the immersed boundary solver.

1.7.1 Future Work

As the project was not successful in its original plan it leaves room for future improvements.

- Find the source of the oscillations in the current code
- Couple an inbuilt `6Dof` solver to the `immersedBoundaryForce` library
- Replace the cylinder with a with a wing to see if the wing can "float" due to the fluid forces and gravity

Study questions

1. Describe the Immersed Boundary Method (IBM) briefly.
2. How is the IBM accounted for in the governing equations i.e. the Navier-Stokes equations? (Just a simple answer)
3. What files need a modification to be able to compile a new class in a user workspace?

Use attached files

Copy the `forceBased` to the user workspace using following commands

Copy class
```
cp -r forceBased $WM_PROJECT_USER_DIR/src/dynamicMesh/meshMotion/solidBodyMotion/
cp -r Make $WM_PROJECT_USER_DIR/src/dynamicMesh/meshMotion/solidBodyMotion/
```

and the test case

Copy test
```
cp -r movingCylinderInChannelIco $FOAM_RUN
```

After doing this the new class needs to be compiled

Compile class
```
cd $WM_PROJECT_USER_DIR/src/dynamicMesh/meshMotion/solidBodyMotion/
wmake libso
```

Now the class should be ready to use.

Bibliography

[1] A. Pinelli, Immersed boundary method-fluid structure interaction. (2016). Lecture. KTH, Stockholm. URL: `http://www.flow.kth.se/sites/flow.kth.se/files/FLOW-School-IBandFluidStr.pdf`

[2] Wim-Paul Breugem, Lecutre 1: A first introduction to Immersed boundary method. Lecture. Delft University of Technology, The Netherlands. URL: `http://www.flow.kth.se/sites/flow.kth.se/files/slides_ibm.pdf`

[3] H. Jasak and Z. Tukovic, Immersed boundary method in FOAM, theory, implementation, and use, (2015). Lecture. Chalmers University of Technology. URL `http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2015/HrvojeJasak/ImmersedBoundary.pdf`

[4] C.S. Peskin, Numerical analysis of blood flow in the heart, J. Comput. Phys. 25 (1977) 220-252

[5] J. Lee, J. Kim, H. Choi and K. Yang, Soruces of spurious force oscillations from an immersed boundary method for moving-body problems, J. Comput. Phys. 230 (2011) 2677-2695