

CFD WITH OPENSOURCE SOFTWARE

A COURSE AT CHALMERS UNIVERSITY OF TECHNOLOGY
TAUGHT BY HÅKAN NILSSON

Project work:

A wave motion class

Developed for OpenFOAM-2.1.x
Requires: PyFoam

Author:
Ali AL SAM

Peer reviewed by:
PEDRAM RAMIN
OLIVIER PETIT

Moving Wave class

1. Introduction

This tutorial describes how to generate moving waves in OpenFoam. This class can be used to simulate for example the effect of ocean waves on offshore structures. A mesh motion class is modified to simulate three types of wave: linear wave, second and third order Stocks moving waves. An existence OpenFoam solver that can handle mesh motion is also modified to add a driving pressure force to it. Further a post-process utility is written to calculate the horizontally average velocity profile. Finally these modifications are tested by running a moving mesh case.

2. Mesh motion Classes

In OpenFoam, The mesh motions are classified into two types: motion without any topology change (*dynamicFvMesh*) and motion with topology change (*topoChangeFvMesh*). In this tutorial we are dealing with the first category where the mesh preserves its number of nodes and the mesh connectivity during the mesh motion.

The *dynamicFvMesh* is an automatic mesh motion solver in which the motion is obtained by changing the distances between the mesh nodes (by stretching and squeezing the cells)[3]. To accomplish this, a mesh motion equation has to be solved together with mesh motion boundary conditions and a diffusivity model (to redistribute the boundary motion through the mesh volume). The automatic mesh motion solver *dynamicFvMesh* includes the following classes:

- I. StaticFvMesh.
- II. dynamicMotionSolverFvMesh.
- III. dynamicInkJetFvMesh.
- IV. dynamicRefineFvMesh.
- V. solidBodyMotionFvMesh.
- VI.

There is an already implemented class in OpenFoam that can handle a wavy mesh motion: (*src / fvMotionSolver / pointPatchFields / derived / waveDisplacement /*).

This class acts as a boundary condition for boundary wavy motion which gives cosine wave shape (Eq 1) to a specific patch.

$$\eta(x, t) = a * \cos(kx - \omega t) \quad \text{Eq1}$$

η :the surface elevation of a deep water wave,
 x : the horizontal coordinate;
 t : time;

a : the first-order wave amplitude;
k : the angular wavenumber, $k = 2\pi / \lambda$ with λ being the wavelength;
 ω : the angular frequency, $\omega = 2\pi / \tau$ where τ is the period time.

The mesh points are then moved by solving a mesh motion equation with one of the diffusivity models. All these mesh motion equations solve laplacian equation (displacementLaplacian, velocityLaplacian, laplaceFaceDecomposition, SBRStress).

The author had difficulties to use this class for horizontally periodic cases (periodic BC in both streamwise and spanwise directions) in parallel, where non matched faces area appear between the periodic patch pars (probably due to the solving of laplacian equation). This is the motivation of the current work.

3. MovingWave class

The dynamicInkJetFvMesh is used instead to generate moving waves. This class is used in cases where the resolution is not changing too much during the mesh motion, i.e. for relatively small changes [3]. The dynamicInkJetFvMesh is previously used to simulate the movement of the rubber hose in the Vigor wave energy converter [2]. Three different wave types are included in the MovingWave class: linear wave (Eq1) , second order Stocks wave (Eq2) and third order Stocks wave (Eq3).

$$\eta(x, t) = a \cos \theta + 0.5(\kappa a) \cos 2\theta \quad \text{Eq2}$$

$$\eta(x, t) = a \cos \theta + 0.5(\kappa a) \cos 2\theta + \frac{3}{8}(\kappa a)^2 \cos 3\theta \quad \text{Eq3}$$

$$\theta = (\kappa x - \omega t) \quad \text{Eq4}$$

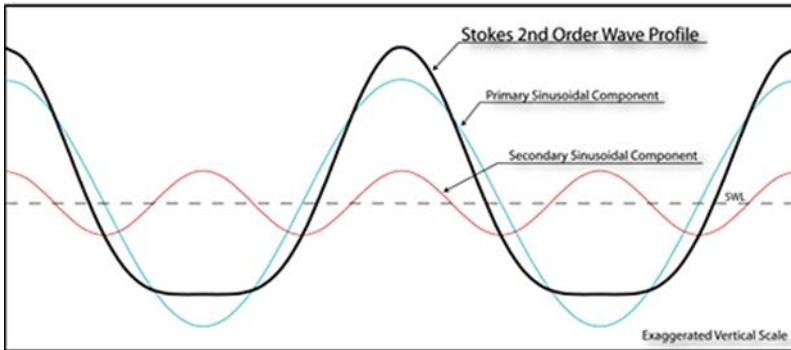


Fig1: Second order Stocks wave profile.

By using the same idea of diffusivity model, a diffusivity parameter is used to absorb mesh deformation next to moving boundary by multiply the calculated function by a decaying exponential function ($e^{-\alpha z}$), where alpha here is a diffusivity parameter that can be selected by the user and z is the vertical coordinate, with the positive z-direction and $z = 0$ corresponding with the mean surface elevation.

3.1. Copy the dynamicInkJetFvMesh

The easiest way to start is to copy the dynamicInkJetFvMesh class to the user directory and then modify it. Since the dynamicInkJetFvMesh class is well described in previous projects [2,3], only the applied modifications will be presented in this report. We start by copying the dynamicInkJetFvMesh as following:

```
mkdir MovingWave
cd MovingWave
cp $WM_PROJECT_DIR/src/dynamicFvMesh/dynamicInkJetFvMesh/dynamicInkJetFvMesh.C
MovingWave.C
cp $WM_PROJECT_DIR/src/dynamicFvMesh/dynamicInkJetFvMesh/dynamicInkJetFvMesh.H
MovingWave.H
cp -r $WM_PROJECT_DIR/src/dynamicFvMesh/Make/ ./
```

Now we need to change the class name inside the MovingWave

```
sed -i s/dynamicInkJetFvMesh/MovingWave/g MovingWave.C
sed -i s/dynamicInkJetFvMesh/MovingWave/g MovingWave.H
rm Make/files
vi Make/files
```

add

```
MovingWave.C
LIB=$(FOAM_USER_LIBBIN)/libMovingWave
```

In Make/options add the following line

```
-I$(LIB_SRC)/dynamicFvMesh/InInclude \
```

No further change is needed to be able to compile this new class

```
wmake libso
```

So far we have not changed anything in the old class functionality and the new class acts exactly as the old one.

3.2. Modify MovingWave class

The modification can be started by changing the class private data in MovingWave.H file where

```
// Private data
scalar amplitude_;
scalar frequency_;
scalar refPlaneX_;
```

should be replaced by :

```
scalar waveAmplitude_; // the wave amplitude.
scalar waveLength_;    // the wave length .
scalar wavePeriod_;    // the period time.
scalar waveDirection_; // either in positive or negative direction (+1,-1) or (0) for
                        //stationary wave. The default value is (+1).
scalar waveType_;       // 1 for linear wave , 2 for Stocks 2nd order and 3 for 3rd
                        // order Stocks wave. The default option is 1.
scalar alpha_;          // a diffusivity parameter.
```

after saving the modification in the H file, the main change will be made in the MovingWave.C file.

The following lines in class constructor

```
amplitude_(readScalar(dynamicMeshCoeffs_.lookup("amplitude"))),
frequency_(readScalar(dynamicMeshCoeffs_.lookup("frequency"))),
refPlaneX_(readScalar(dynamicMeshCoeffs_.lookup("refPlaneX"))),
```

have to be replaced by

```
waveAmplitude_(readScalar(dynamicMeshCoeffs_.lookup("waveAmplitude"))),
waveLength_(readScalar(dynamicMeshCoeffs_.lookup("waveLength"))),
wavePeriod_(readScalar(dynamicMeshCoeffs_.lookup("wavePeriod"))),
```

```

waveDirection_(dynamicMeshCoeffs_.lookupOrDefault <scalar>("waveDirection",1.0)),
waveType_(dynamicMeshCoeffs_.lookupOrDefault <scalar>("waveType",1.0)),
alpha_(readScalar(dynamicMeshCoeffs_.lookup("alpha"))),

```

Notice that a default values are given to waveDirection_ and waveType_.

The next modification is optional since it is only an output message during the run time tells the user which parameters have been used to deform the mesh. However this output message can be changed as following:

```

Info<< "Performing a dynamic mesh calculation: " << endl
<< "amplitude: " << amplitude_
<< " frequency: " << frequency_
<< " refPlaneX: " << refPlaneX_ << endl;

```

Replaced by

```

Info<< "Performing a dynamic mesh calculation: " << endl;
Info<< "WaveAmplitude: " << waveAmplitude_ << endl;
Info<< " WaveLength: " << waveLength_ << endl;
Info<< " WavePeriod: " << wavePeriod_ << endl;
if (waveDirection_==0){Info<< "WaveDirection:"<<""<< "    Stationary wave"<<endl;}
else if(waveDirection_==1.0){Info<< "WaveDirection:"<<""<< "    negative wave"<<endl;}
else {Info<< "WaveDirection:"<<""<< "    positive wave"<<endl;}
if(waveType_==2.0){Info<< "WaveType:"<<""<< "    second order Stocks wave"<<endl;}
else if (waveType_==3.0){Info<< "WaveType:"<<""<< "    third order Stocks wave"<<endl;}
else {Info<< "WaveType:"<<""<< "    linear wave"<<endl;}

```

The update() member function in the original class, which update the mesh vortices coordinate, is modified now to perform one of the three types of wave motions.

Unlike the Vigor wave mesh motion class [2], the MovingWave class uses the original (newPointsreplace) function which replaces one of the stationary mesh coordinates by the calculated one according to the specified motion equation. This function is kept because it runs perfectly in parallel without any additional modification.

The scalingFunction in dynamicInkJetFvMesh class should be omitted by removing these lines:

```

scalar scalingFunction =
0.5*
(
::cos(constant::mathematical::twoPi*frequency_*time().value())

```

```

- 1.0
);

Info<< "Mesh scaling. Time = " << time().value() << " scaling: "
<< scalingFunction << endl;

```

and the newPointsreplace function can be changed as following:
replace

```

newPoints.replace
(
    vector::X,
    stationaryPoints_.component(vector::X)*
    (
        1.0
        + pos
        (
            - (stationaryPoints_.component(vector::X))
            - refPlaneX_
        )*amplitude_*scalingFunction
    )
);

```

by

```

if(waveType_==2.0) //Stocks 2nd order wave, Eq 2
{
    newPoints.replace
    (vector::Z,
    stationaryPoints_.component(vector::Z)+Foam::exp(-
    (alpha_*mag(stationaryPoints_.component(vector::Z))))*
    waveAmplitude_*(cos((constant::mathematical::twoPi)*stationaryPoints_.component(vector:
    :X)/waveLength_ - waveDirection_*time().value()/wavePeriod_)
    +0.5*(waveAmplitude_*constant::mathematical::twoPi/waveLength_)*
    cos(2*((constant::mathematical::twoPi)*stationaryPoints_.component(vector::X)/waveLength
    _ - waveDirection_*time().value()/wavePeriod_)));};

else if(waveType_==3.0) //Stocks 3rd order wave , Eq3
{
    newPoints.replace
    (vector::Z,
    stationaryPoints_.component(vector::Z)+Foam::exp(-
    (alpha_*mag(stationaryPoints_.component(vector::Z))))*

```

```

waveAmplitude_*(cos((constant::mathematical::twoPi)*stationaryPoints_.component(vector::X)/waveLength_ - waveDirection_*time().value()/wavePeriod_)
+0.5*(waveAmplitude_*constant::mathematical::twoPi/waveLength_)*
cos(2*((constant::mathematical::twoPi)*stationaryPoints_.component(vector::X)/waveLength_ - waveDirection_*time().value()/wavePeriod_))
+(3/8)*pow((waveAmplitude_*constant::mathematical::twoPi/waveLength_),2)*
cos(3*((constant::mathematical::twoPi)*stationaryPoints_.component(vector::X)/waveLength_ - waveDirection_*time().value()/wavePeriod_))) );}
else
    //linear wave, Eq1
    {newPoints.replace
    (vector::Z,
stationaryPoints_.component(vector::Z)+ waveAmplitude_* Foam::exp(-
(alpha_*mag(stationaryPoints_.component(vector::Z))))*
cos((constant::mathematical::twoPi)*stationaryPoints_.component(vector::X)/waveLength_ -
waveDirection_*time().value()/wavePeriod_));}

```

the new class MovingWave is now ready to compile

```

wclear libso
wmake libso

```

4. Mesh motion solver

There are five solvers in OpenFoam that can handle mesh motion [1]:

- I. pimpleDyMFoam: Transient solver for incompressible, flow of Newtonian fluids using the PIMPLE algorithm.
- II. rhoCentralDyMFoam: Density-based compressible flow solver based on central-upwind schemes of Kurganov and Tadmor.
- III. sonicDyMFoam: Transient solver for trans-sonic/supersonic, laminar or turbulent flow of a compressible gas .
- IV. interDyMFoam: Solver for 2 incompressible, isothermal immiscible fluids using a VOF (volume of fluid) phase-fraction based interface capturing approach.
- V. icoUncoupledKinematicParcelDyMFoam: Transient solver for the passive transport of a single kinematic particle cloud.

Usually in moving wave simulations the fully periodic boundary conditions (periodic BCs in both streamwise and spanwise) are preferred to simulate a flow over infinitely large area.

In this tutorial the pimpleDyMFoam solver is modified by adding a driving pressure force to it. The driving pressure force is calculated by correcting the velocity field every time step to insure fixed mass flow rate (as in channelFoam solver [1]).

4.1. Modify an existing solver

Here we start by copy the original pimpleDyMFoam as following:

```
cd $WM_PROJECT_USER_DIR/applications/solvers/  
cp -r  
$WM_PROJECT_DIR/applications/solvers/incompressible/pimpleFoam/pimpleDy  
MFoam myPimpleDyMFoam  
cd myPimpleDyMFoam  
wclean  
mv pimpleDyMFoam.C myPimpleDyMFoam.C
```

change the Make/files as following:

```
myPimpleDyMFoam.C  
EXE = $(FOAM_USER_APPBIN)/myPimpleDyMFoam
```

Now we can compile the new solver by using wmake. To modify the new solver we need to do some changes. First add #include "IFstream.H" to be able to read the pressure gradient term during runtime.

```
#include "fvCFD.H"  
#include "singlePhaseTransportModel.H"  
#include "turbulenceModel.H"  
#include "dynamicFvMesh.H"  
#include "pimpleControl.H"  
#include "IObasicSourceList.H"  
#include "IFstream.H"
```

In the main loop include the creatGradP file as following:

```
#include "setRootCase.H"  
#include "createTime.H"  
#include "createDynamicFvMesh.H"
```

```
#include "initContinuityErrs.H"
#include "createFields.H"
#include "readTimeControls.H"
#include "createGradP.H"
pimpleControl pimple(mesh);
```

now we need to copy the createGradP.H file from the channelFoam solver:

```
cp
$WM_PROJECT_DIR/applications/solvers/incompressible/channelFoam/createGra
dP.H ./
```

here the createFields.H file need a modification to initialize the velocity that we want to keep it constant throughout the simulation. In the end of the createFields file add:

```
Info<< "\nReading transportProperties\n" << endl;
IOdictionary transportProperties
(
    IOobject
    (
        "transportProperties",
        runTime.constant(),
        mesh,
        IOobject::MUST_READ_IF_MODIFIED,
        IOobject::NO_WRITE,
        false
    )
);

dimensionedVector Ubar
(
    transportProperties.lookup("Ubar")
);

dimensionedScalar magUbar = mag(Ubar);
vector flowDirection = (Ubar/magUbar).value();
```

inside the while loop and just after the turbulence correction loop we can add the pressure gradient correction as following:

```
// Extract the velocity in the flow direction
dimensionedScalar magUbarStar =
(flowDirection & U).weightedAverage(mesh.V());

// Calculate the pressure gradient increment needed to
// adjust the average flow-rate to the correct value
dimensionedScalar gragPplus =
(magUbar - magUbarStar)/rAU.weightedAverage(mesh.V());

U += flowDirection*rAU*gragPplus;

gradP += gragPplus;

Info<< "Uncorrected Ubar = " << magUbarStar.value() << tab
<< "pressure gradient = " << gradP.value() << endl;
```

Now we add the correction of pressure to the velocity equation. Inside UEqn.H change as following:

```
tmp<fvVectorMatrix> UEqn
(
    fvm::ddt(U)
  + fvm::div(phi, U)
  + turbulence->divDevReff(U)
  ==
  flowDirection*gradP
);
```

The solver is ready now to compile.

5. Test case

The modifications in the dynamicFvMesh and pimpleMyDFOam are now tested by running a moving wave case

5.1. Geometry and mesh

The test case is (200m x 200m x 150m) box. The geometry is discretized by using (50 x 50 x 50) cells. The cyclic BC's are applied in the inlet, outlet and sides patches. The lower patch is supposed to represent the moving waves, therefore movingWallVelocity BC is given to this patch in 0/U file. The wave characteristics are controlled by the dynamicMeshDict. The example of input is shown below:

```
dynamicFvMeshLibs ("libMovingWave.so");
dynamicFvMesh  MovingWaveVelocity;
motionSolverLibs  ("libfvMotionSolvers.so");
MovingWaveVelocityCoeffs
{
    waveAmplitude 2.5;
    waveLength 50;
    wavePeriod 2;
    waveDirection 1.0;
    waveType 1.0;
    alpha 0.02;
}
```

Notice that the last parameter (alpha) is an arbitrary and has to adjust according to domain size. The mesh will be deformed to generate the moving waves, see Fig 2 and 3.

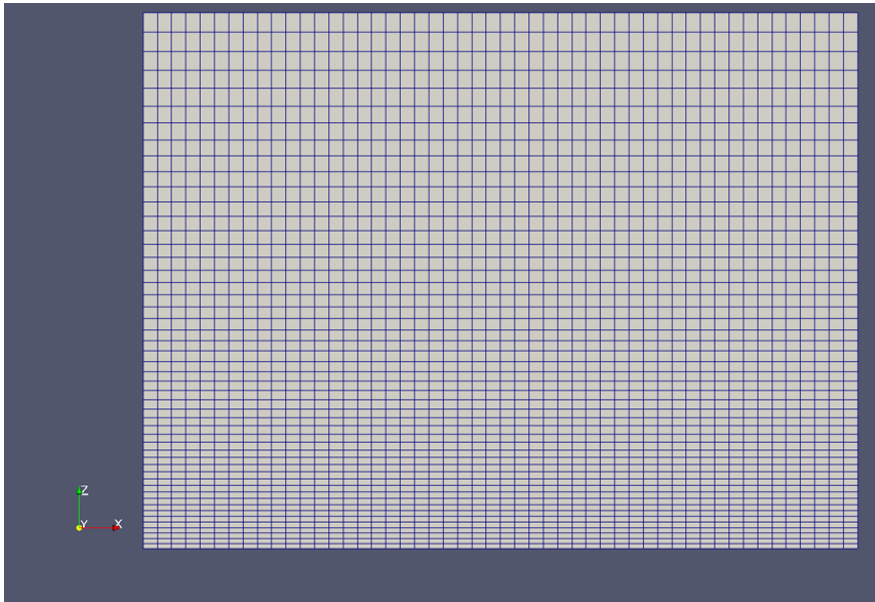


Figure 2 : The original mesh

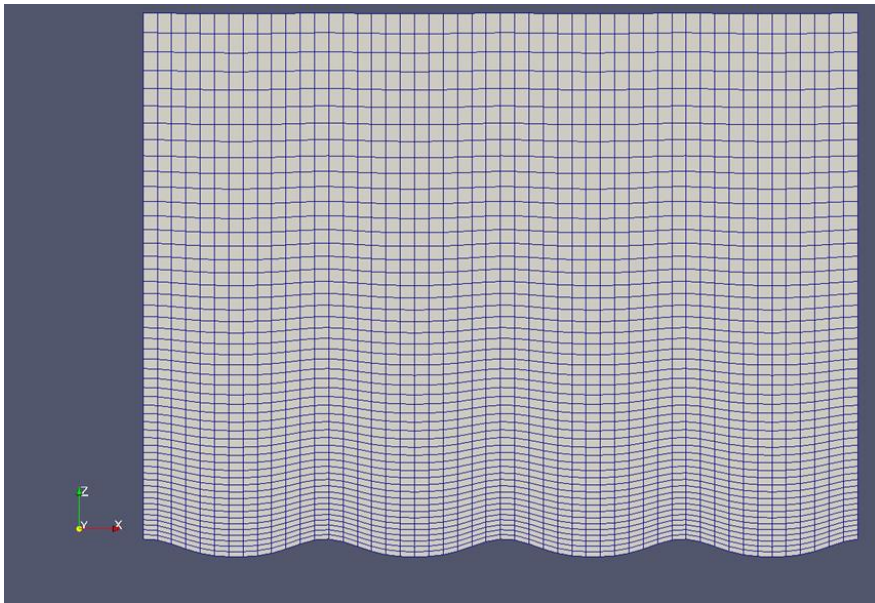


Figure 3: The deformed mesh

6. Velocity profile

There are many useful post process utilities and classes in OpenFoam. The sample utility for example samples field data either through a 1D line for plotting on graphs or a 2D plane for displaying as isosurface images. The sampling locations are specified for a case through a sampleDict dictionary in the case system directory.

However, there is a need for a simple utility that can sample a field profile that each point of it represents a plane average of that field. Such profile is usually used in atmospheric boundary layer calculation. The velocity profile along the depth of the atmospheric boundary layer is an example.

The purpose of this part of the report is to provide an example of such simple utility that can save a lot of post processing time and can be easily modified to meet the user needs. The suggested utility reads the velocity field (can be replaced by the U or UMean or any other vector or scalar quantity by doing simple modifications) from the last time directory. It calculates the minimum and the maximum geometry heights and divided this distance into several layers. The velocity field is then averaged in each of these layers by assuming that the cell-centre value is constant over the whole cell. Finally we get one velocity for each height interval. Below the utility and its explanation are presented.

```
#include "fvCFD.H"
#include <iostream>
#include <fstream>
using namespace std;
// ***** //
int main(int argc, char *argv[])
{
# include "setRootCase.H"
# include "createTime.H"
# include "createMesh.H"
// ***** //
Info<< "Time = " << runTime.timeName() << endl;
// ***** //
// ***** //
Info << "Reading field U" << endl;
volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
```

```

    mesh,
    IOObject::MUST_READ,
    IOObject::NO_WRITE
),
mesh
);
// ***** //
// ***** //

//divide the geometry into 1000 layers
int nlevels=1000;
//initialize the domain height
double zmin=0,zmax=0;
// initialize an array of vectors of "nlevels" elements for velocity
vector AvUMean[nlevels-1];
// initialize an array of scalar of "nlevels" elements for cells volume
scalar nV[nlevels-1];
// a loop to initialize the arrays
for (int i=0;i<nlevels;i++)
{
    AvUMean[i]=vector::zero;
    nV[i]=0;
}
//loop over mesh cells to calculate the Zmin and Zmax
forAll(mesh.cells(),celli)
{
    if (mesh.C()[celli].z()<zmin){zmin=mesh.C()[celli].z();}
    if (mesh.C()[celli].z()>zmax){zmax=mesh.C()[celli].z();};
}
Info<<"zmin="<<"<<zmin<<endl;
Info<<"zmax="<<"<<zmax<<endl;
forAll(mesh.cells(),celli)
{
    // for each cell get center, calculate the index (toc)
    const vector& cellCenter = mesh.C()[celli];
    i= int(nlevels*mag((cellCenter.z()-zmin)/(zmax-zmin)));
    i=min(max(i,0),nlevels-1);
    // sum the (velocity*cell volume) at each layer
    AvUMean[i]=AvUMean[i]+(U[celli]*mesh.V()[celli]);
    //sum the volume at each layer
    nV[i]=nV[i]+mesh.V()[celli];
}
ofstream UMean;

```

```

//open file to write the averaged velocity
UMean.open ("UMean.data");
// loop over toc
for (i=0;i<nlevels;i++)
{
//only for the layers that include cells
if (nV[i])
{
    AvUMean[i]=AvUMean[i]/(nV[i]);
    UMean << i*(zmax-zmin)/nlevels << " " << mag(AvUMean[i]) << endl;
}
}
UMean.close();
Info<< "Done!" << endl;
return 0;
}

// ***** //
The Make/options
// ***** //
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude
EXE_LIBS = \
    -lfiniteVolume \
    -lgenericPatchFields \
    -lmeshTools
// ***** //

```

```

// ***** //
The Make/files
// ***** //
postprocess.C
EXE = $(FOAM_USER_APPBIN)/postProcess

```