# INVESTIGATION OF CONJUGATE HEAT TRANSFER IN FLUID-STRUCTURE INTERACTION MODELING USING OpenFOAM

A Senior Project
presented to
the Faculty of California Polytechnic State University,
San Luis Obispo

In Partial Fulfillment
of the requirements for the degree
Bachelors of Science in Aerospace Engineering

By
Andrew Brown
October 2014

## Abstract

INVESTIGATION OF CONJUGATE HEAT TRANSFER IN FLUID-STRUCTURE INTERACTION MODELING USING OpenFOAM
Andrew Joseph Brown

Modern engineering problems are often comprised of highly interacting and complex systems. Modeling of these systems is a difficult problem and involves many types of physics. Traditional models often sought to only solve one type of physics and usually did not interact with other models. Modern computing has allowed more and more interaction of these various models. Commercially available software has been readily available as of late to solve complex fluid-structure interaction problems, but none have effectively captured heat transfer across the fluid-structure boundary. This project investigates the effectiveness of using the open source software OpenFOAM to effectively model this enhanced interaction. Progress was made towards a functional solver, though there is still work needed in order to completely model conjugate heat transfer in a fluid-structure interaction problem

# Table of Contents

## List of Tables

## List of Figures

## Nomenclature

| | | |
|---|---|---|
| Re | = | Reynold’s number |
| $\mu$ | = | shear modulus |
| $\rho$ | = | density |
| $\nu_f$ | = | kinematic viscosity |
| $\nu_s$ | = | Poisson’s ratio |

# 1 Introduction

Fluid Structure Interaction (FSI) is the interaction between a fluid and structure. When a fluid flows around a structure, that structure is deformed thereby changing the wetted area as well as the geometry. As the structure is deformed, it changes how the fluid flows around it. While this interaction can be negligible in some cases, low speed wind on a building, it can have very drastic effects in others such as aeroelastic failure where accurate modeling is critical. These interactions usually take one of two forms, steady and unsteady. An unsteady interaction is characterized by structural deformation that does not remain in a stressed state, i.e. one that returns to its original position only to be deformed again.

In recent years, FSI has become an important sub-field of numerical simulation. Prior to this, their existed a wide range of finite element codes for structural problems and computational fluid dynamics codes for fluid flow problems. Eventually, these codes were coupled to provide a solution which more accurately represents reality. This joint solution has applications in numerous fields including aerospace, bio-medical, power generation, computer design, and civil engineering.

## 1.1 FSI Methodology

There are two methodologies for solving coupled FSI systems – the monolithic, and the partitioned approach. The monolithic approach seeks to define a series of partial differential equations (PDEs) that govern the entire fluid and structural domain and then discretize the complete domain[1]. While simple in theory, this approach is difficult to actually implement due to the differing mathematical and numerical properties of the two domains[1]. Because monolithic solvers solve for all variables simultaneously during a single time step, they can be more robust and can solve a wider variety of cases. For identical reasons, they are also more computationally expensive and somewhat more difficult to code.

The partitioned approach seeks to utilize existing fluid and structural codes and couple these codes at the interface of the two domains. The fluid solver uses a separate set of equations, variables, and mesh than the solid solver[2]. The coupling happens at the boundary where the pressure from the fluid updates the boundary of the solid which is then solved for to determine displacement of the interface boundary[2]. This updated boundary is then used to solve the fluid domain. This represents sort of a staggered solution where only one field is solved per time step[2]. Most of the existing FSI codes follow a partitioned approach and can be further broken into weakly and strongly coupled implementations. One of the biggest advantages of the partitioned solver is that it makes use of already existing, highly refined fluid and structure codes[1].

Weakly coupled solvers, also called loosely coupled, start by predicting the structural motion and then solving for the fluid properties[3]. These fluid properties, primarily pressure and fluid stresses, are transferred to the interface where they act as boundary conditions for the structural solver which then solves for the structure's displacement[3]. A defining characteristic of these weakly coupled solvers is that there is "no check that the predicted structural displacements match the displacements computed at the end of the step”[3]. This characteristic in particular defines the solver as weakly coupled and gives it an explicit nature despite the potential implicit nature of the individual fluid or structural solvers[3].

A weakly coupled solver can be made into a strongly coupled solver with an additional step that computes the convergence of the predicted and computed structural displacements during each step of the solution. This iteration defines the solver as strongly coupled as well as giving it an implicit nature[3]. Both block-Newton root finding and fixed point iteration techniques are used to tighten the coupling[3] with fixed point being the most common. The fixed point technique is typically implemented with Aitken relaxation[4]. Strongly coupled solvers are more computationally expensive than their weaker counterparts as a result of the extra iterations.

Weakly coupled solvers are ideal for problems with small deformations and momentum variations between time steps as they don’t have the extra subiterations at the interface[2]. These solvers also are unable to handle cases where the density ratio of the fluid and solid approach unity[5]. The ultimate reason for these shortcomings is the “added-mass effect” related to energy conservation at the interface[2,3]. When deriving the coupling conditions for the fluid and structure, there is an “extra operator in front of the second order time derivative” for the structure[6]. This operator functions as an “added-mass” and takes into account the way in which the fluid alters the natural vibration frequencies of the solid[6]. Functionally, during solver iterations, the fluid forces are dependent on predicted structural displacements instead of the actual displacements and so error is introduced[2]. The extra iterations required to transform the solver from weakly coupled to strongly coupled have been shown to negate the influence of the “added-mass effect”[7].

## 2 OpenFoam

OpenFOAM (Open Field Operation and Manipulation) is the code suite chosen for the simulations carried out for this research. This is an open source program suite capable of tackling a wide range of both fluid and structural problems. OpenFOAM, initially developed as FOAM, was brought to the open source domain with the launch of OpenCFD Ltd. in 2004 and subsequently acquired by SGI Corp in 2011[*]. It has remained and always will be open source due to the bylaws of the managing organization which has allowed a wide range of collaboration and development with a greater audience than most commercial codes have access to. This has in turn improved and broadened the scope of the software quicker than can be accomplished with most commercial codes. The software also allows ease of comparison in the academic community as everyone has uninhibited access to the same code. Commercial users include ICE, Strömungsforschung Gmbh, who utilizes it for optimization of air conditioning and filtration, as well as move-csc who make use of OpenFOAM for turbine design and automotive aerodynamics.

[*] http://www.openfoam.com/features/

## 2.1 OpenFOAM Introduction

An object oriented approach was used when OpenFOAM was created in C++[8]. As described on the OpenFOAM website, the program "follows a highly modular code design in which collections of functionality (e.g. numerical methods, meshing, physical models,…) are each compiled into their own shared library"[*]. This structure allows the user to easily modify components of the code and adapt the solvers to specific cases. OpenFOAM can be broken into two types of applications, solvers which handle the PDE's and utilities which handle data processing[8]. Figure 2-1 shows the basic layout of the program with the libraries at the top feeding into the various program components.

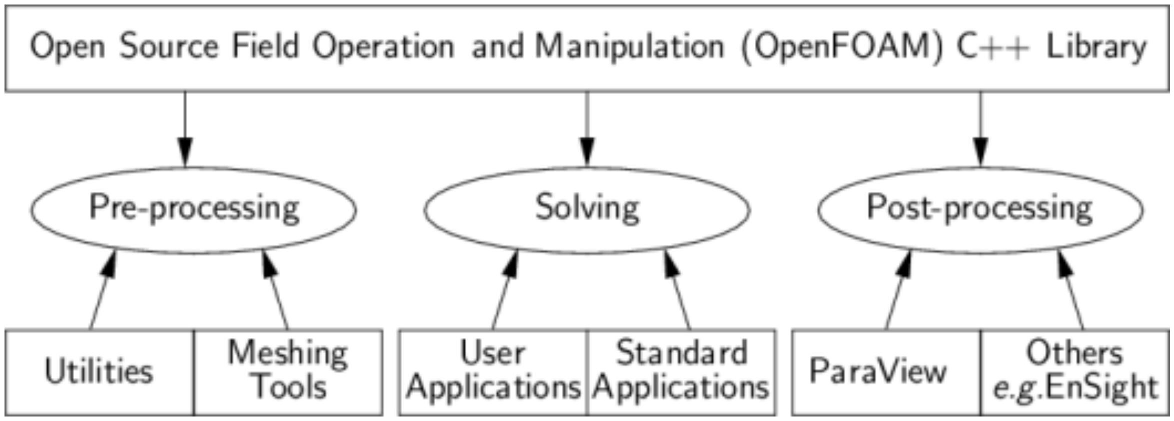


**Figure 2-1: Layout of OpenFOAM structure[†]**

In an effort to ensure the ease of modularity and user adaptability of the code, the solver syntax closely resembles the PDE that it is solving for. "For example the following equation

[*] http://www.openfoam.com/features/
[†] http://www.openfoam.org/docs/user/

$$\frac{\partial \rho U}{\partial t} + \nabla \bullet \phi U - \nabla \bullet \mu \nabla U = -\nabla p$$

is represented by the code[*]."

```
solve
(
    fvm::ddt(rho, U)
  + fvm::div(phi, U)
  - fvm::laplacian(mu, U)
    ==
    fvc::grad(p)
);
```

This syntax is very intuitive and thus simplifications or additions can easily be made to solvers by eliminating or adding terms. These solvers are also procedural in nature, which mimics the natural procedure of the solution algorithm being implemented[*]. These characteristics combine to allow implementation and modification to a multitude of engineering fields.

## 2.2 OpenFOAM Solver Creation

In order to build a new solver, it will be necessary to have a modicum of understanding of the solver file structure. The OpenFOAM library contains executable library files with .so extensions, or shared object files. These make up the class definitions that the solvers are based on. There are also header files (.H) known as dependency files that contain the class declaration including the name of the class and its function/s. When a new solver is created, it needs to be compiled which is accomplished with the 'wmake' command. This command is somewhat "more versatile and easier to use" than the Linux included 'make' command and can be used to compile any code, not just OpenFOAM

[*] http://www.openfoam.org/docs/user/

scripts[*]. The typical solver first contains a list of the necessary headers called by a '`# include`' statement[*]. The '`wmake`' command will systematically work through this list checking whether the source files have been updated and selectively compiling only those that have been.

The directory structure for a new application can be seen in Figure 2-2. All of the files pertaining to the new solver are placed in a directory with the name of the new solver. There is a .C file with the solver name that calls all of the necessary libraries as well as initializing the time, meshes, and material properties. This file contains the time loop as well as the calls to the individual solvers. The other .H files in the solver directory relate to the initialization and the calls within the time loop. The Make directory includes a file titled options which "contains the full directory paths to locate the header files … and library names" while the file titled "files", lists the entire path and name of the solver once it has been compiled[*].

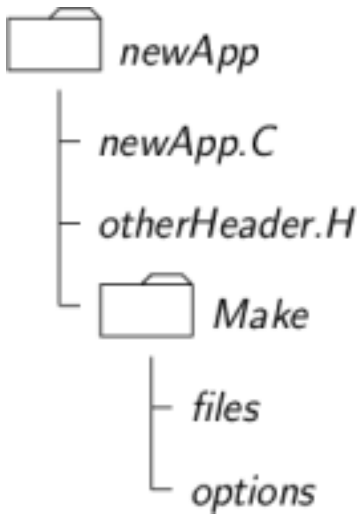


**Figure 2-2: OpenFOAM Directory Structure**[*]

[*] http://www.openfoam.org/docs/user/

### 2.2.1 icoThermFsiFoam Creation

The first step in implementing the OpenFOAM package was to create the FSI solver from a fluid and structural solver with the addition of interface coupling. The thermal FSI solver created for this research, icoThermFsiFoam, is a combination of two existing solvers with specific modifications. The structural component of the solver "is based on the stressedFoam solver and as such is limited to linear stress-strain relationships and relatively small deformations[5]." The fluid component of the solver is based on icoThermFoam which is itself a modification of the icoFoam solver such that thermal effects can be computed within the fluid.

icoFoam is a standard OpenFOAM solver designed to solve transient cases for incompressible, laminar, Newtonian fluids[*]. It does this by implementing the PISO algorithm to solve the incompressible laminar simplification of the Navier-Stokes equations[†]. The PISO algorithm stands for pressure implicit with split operator and is the method utilized in coupling the pressure and velocity equations[9]. It functions as a loop with "an implicit momentum predictor followed by a series of pressure solutions and explicit velocity corrections"[9] and is solved until the user defined tolerance is satisfied. The PISO algorithm is chosen over the SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) algorithm because it is more efficient for cases where the time step is determined by the need for temporal accuracy though the SIMPLE algorithm can be used in transient cases as well[9]. The icoFoam solver

[*] http://www.openfoam.com/features/
[†] http://www.openfoam.org/docs/user/

is “inherently transient, requiring an initial condition (such as zero velocity) and boundary conditions”[*].

The thermal modification of the icoFoam solver was completed and tested against a cavity with a moving lid as described in the literature[‡]. The first step in creating the icoThermFoam solver was to copy the icoFoam solver, modify the Make files in order to create the new solver name and recompile the files. The createFields.H file was modified to include a call for the thermal diffusivity value as well as creating the volume scalar field for temperature. Finally, the actual solver file icoThermFoam.C was altered to include the constant diffusivity, incompressible, and no source or sink convective heat transfer PDE represented by equation 2 which is a simplification of equation 1, the full form convective PDE. These modifications now complete the new solver and it can be compiled for use. All of the necessary files for the icoThermFoam solver are listed in Appendix B.

$$\frac{\partial T}{\partial t} = \nabla \bullet (D \nabla T) - \nabla \bullet (\phi T) + R \tag{1}$$

$$\frac{\partial T}{\partial t} = D \nabla^2 T - \phi \bullet \nabla T \tag{2}$$

The test case for the icoThermFoam solver is the slightly modified cavity case, which is the standard tutorial for the icoFoam solver. To modify the tutorial case, we need to add the solution schemes to the fvSchemes.H file. The lacplacian scheme for the temperature equation is Gauss linear corrected and the divergence scheme is Gauss upwind. The fvSolution.H file is updated to

[*] http://www.openfoam.org/docs/user/
[‡] http://openfoamwiki.net/index.php/How_to_add_temperature_to_icoFoam

include the solver controls for the temperature equation. The biconjugate gradient solver with diagonal incomplete-LU (asymmetric) preconditioning is selected for this case. Finally, the temperature boundary conditions are added to the initial condition directory (0). Figure 2-3 shows the temperature distribution in the cavity case as given by the literature and Figure 2-4 shows the temperature distribution in the same cavity case with the same flow parameters as solved with the icoThermFoam solver that was created.

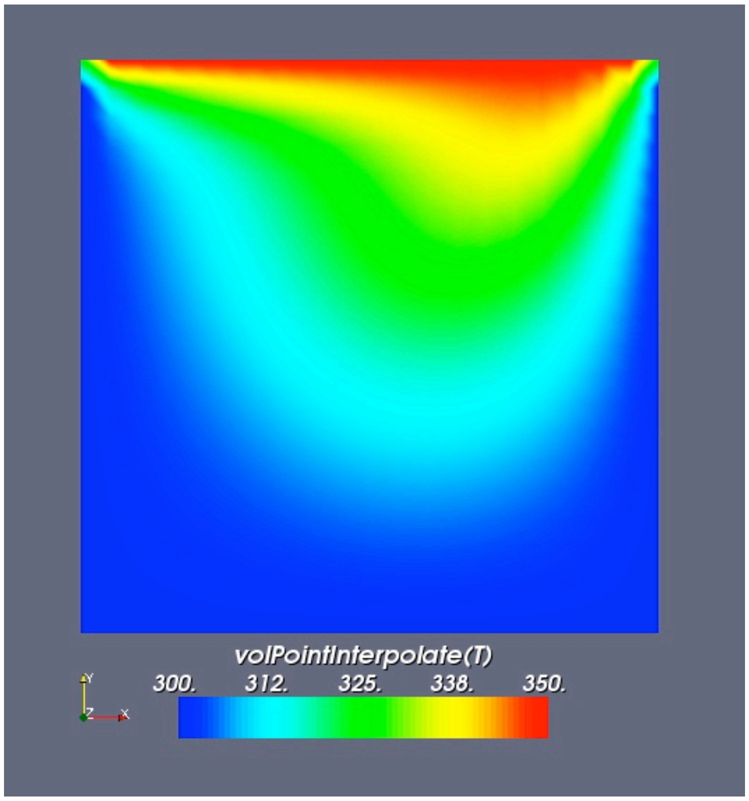


**Figure 2-3: Reference Cavity Temperature Distribution**[*]

[*] http://openfoamwiki.net/index.php/How_to_add_temperature_to_icoFoam

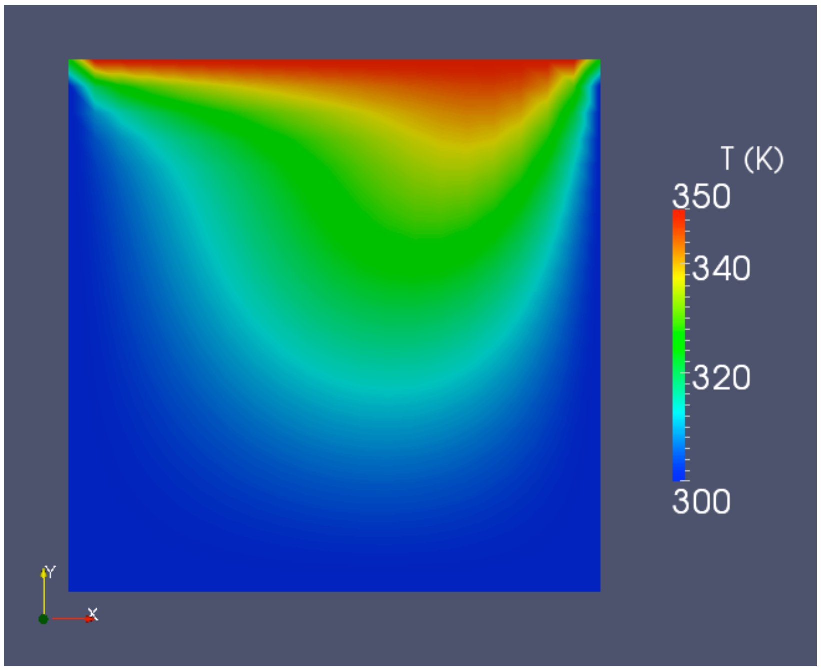


**Figure 2-4: Experimental Cavity Temperature Distribution**

The actual creation of the icoThermFsiFoam solver starts by creating a directory with the same name and copying many files from the stressedFoam and icoThermFoam solvers to this new directory. The icoFoam.C file is used as the template for the icoThermFsiFoam.C file and is amended by removing the PDE solvers and including several "`# include`" lines. This file does not contain any PDE's but includes calls for both the structural and fluid solvers as well as all of the other necessary .H files required for the coupled solver.

The icoThermFoam.C fluid solver is updated to include the continuity errors associated with the moving mesh instead of the original continuity error file and is renamed solveFluid.H. Many of the "`# include`" headers are stripped as they are now in the icoThermFsiFoam.C and are no longer needed in the fluid solver file. The structural solver is updated in a similar manner with the time stepping code removed as it is now in the icoThermFsiFoam.C file. All instances of `U` and `T` are replaced with `Usolid` and `Tsolid` respectively in order for the code to keep the fluid and structural parameters separate, especially since the "`U`" in "`Usolid`" refers to displacement, not velocity as it does in the fluid solver. The various other .H files are updated in order to have consistent naming schemes for the coupled solver. One of the most important files that is copied from the OpenFOAM libraries is the readCouplingProperties.H file which describes to the solver exactly how the two domains are coupled. Once all of the modifications are made to the individual files, the solver can be compiled for use. The full icoThermFsiFoam solver code can be found in Appendix A.

### 2.3 OpenFOAM Implementation

The icoThermFsiFoam solver requires a very precise case setup in order to run properly. The case name serves as the parent directory and then the fluid and solid sub-directories are created within this structure. These two domain sub-directories each contain the initial time sub-directory (0), as well as the constant and system sub-directories associated with a typical OpenFOAM case. The constant directory contains a polyMesh sub-directory which contains the blockMesh dictionary and all of the mesh files. The constant directory also contains the constant material properties for the particular domain. The system directory within the fluid directory contains the pertinent run information for the case. The same directory within the solid directory contains a soft link to the controlDict in the fluid directory which is the overall case control file as well as files relating to the different schemes needed to solve the solid domain. A visual representation of the preceding can be seen in a directory and file flowchart for an icoThermFsiFoam case can in Figure 2-5. This case setup is designed to use the fluid directory as the run directory because OpenFOAM expects to see the 0, constant and system directories as the first subdirectories it encounters (versus fluid and solid under the case name). This setup necessitates the need for soft links so that the solid information can be read from the fluid directory. When the solver writes the output files, they will also be written to the case directory. Each time directory will have the fluid output as well as a sub-directory with the solid output. Soft links to these files can be created within the solid directory using a file called linkedSolutions.

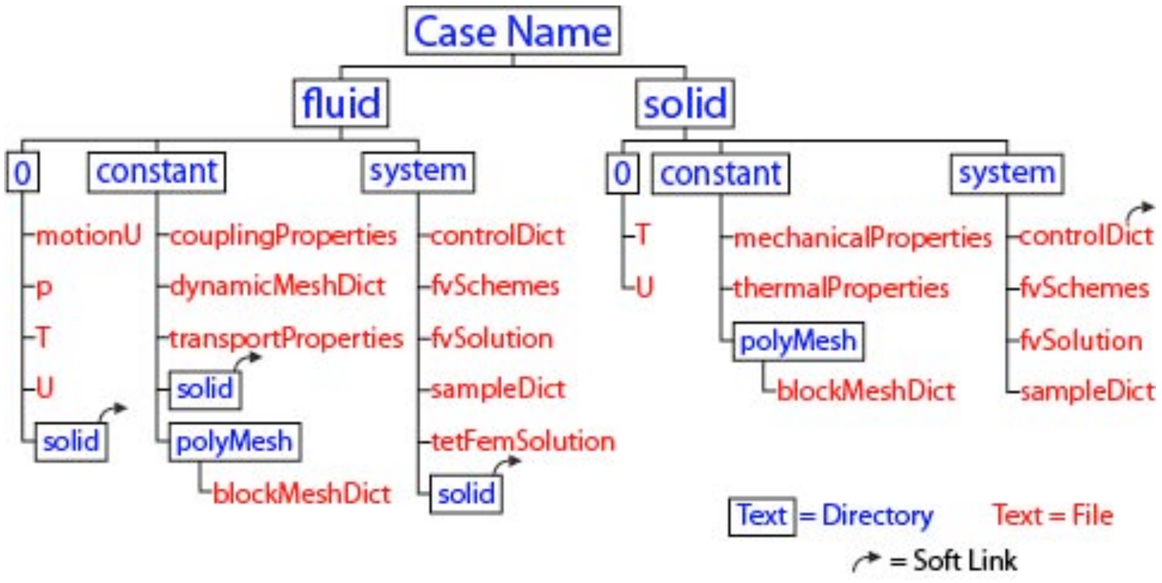


**Figure 2-5: Case Flow Chart**

## 3 Experimental Set-up

The first step in creating a CFD (or FEA) case is defining the geometry of the problem that will be solved. In OpenFOAM this can be accomplished by using the blockMesh utility which is designed for simple geometries. The geometry selected for this experiment is the Turek/Hron FSI benchmark[4] since there is existing data to compare at least the FSI results to as well as being simple and computationally "easy". The basic geometry can be seen in Figure 3-1 and Figure 3-2 with the associated dimensions summarized in

Table 3-1. The basic setup consists of a stationary circle located near the inlet with an elastic tail trailing from the downstream side. This geometry is a modified 2D version of the standard CFD benchmark involving a cylinder exposed to transverse flow.

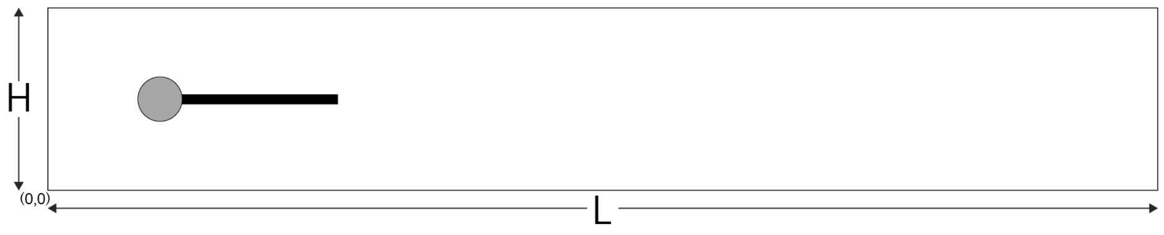


**Figure 3-1: Basic FSI Geometry**

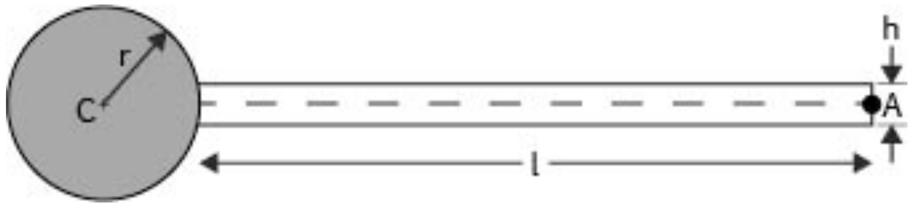


**Figure 3-2: Geometry Detail**

**Table 3-1: Summary of Primary Dimensions**[4]

| Parameter | Dimension (m) or parameter location |
|---|---|
| H | 0.41 |
| L | 2.5 |
| C | (0.2,0.2) |
| r | 0.05 |
| l | 0.35 |
| A | (0.6,0.2) |
| h | 0.02 |

Once the geometry is defined, it will be important to identify regions with significant flow phenomena so that we can make sure that we do an adequate job of meshing the domain. Since the vortex shedding of the circle and subsequently the motion of the tail are of interest, adequate cell density is required in those regions. This is implemented by adding extra blocks closely surrounding the geometry of interest in order to better control mesh density and quality. Figure 3-3 shows all of the blocks used to mesh the domain.

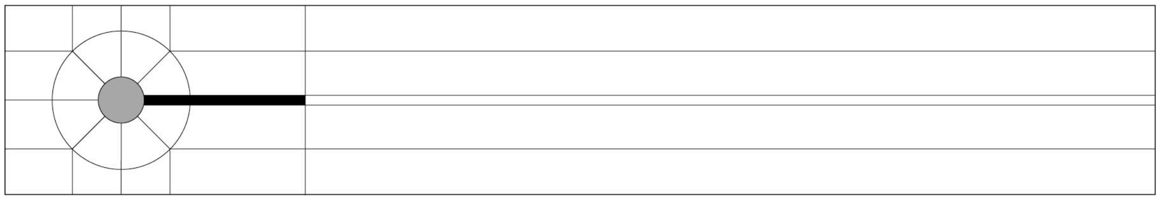


**Figure 3-3: blockMesh Setup**

To create the blockMesh dictionary the coordinates for the vertices of each block must be defined. These coordinates are summarized in Appendix C and were derived using basic geometry. The listed coordinates create a geometry 10 times larger than that given by the Turek/Hron benchmark but the blockMesh dictionary incorporates a scaling function so that the resultant domain is identical. For every block in the domain, the number of cells in each direction, x, y, and z, can be set along with grading, or the cell expansion ratio in each of the three coordinate directions. Due to the nature of the FSI problem, two separate meshes, and therefore blockMesh dictionaries, will need to be created, one for the fluid and one for the structure. Because of the coupling methodology prescribed in the icoThermFsiFoam solver, the vertices of the coupled fluid and structural meshes must be coincident. This means that the same number of cells and grading need to be specified in the appropriate directions at the fluid-structure interface.

The fluid mesh fills the white portion of Figure 3-3 while the solid mesh fills only the black portion. The gray portion is not meshed as it is defined to act as a non-deforming rigid body, or wall, which will shed vortices. Once the two blockMesh dictionaries are fully created, the user can run the blockMesh utility from within the case directory to create the requisite mesh domains for the solver to solve on. The checkMesh utility can be used to diagnose problems with the blockMesh dictionary and evaluate the quality of the mesh.

After the geometry is created, the next step in running a simulation is setting the various properties on the domain. These boundary and initial conditions are set inside of the 0 time step directory. Each parameter (i.e. temperature, velocity, or pressure) has its own file and both the boundary and initial conditions are set within the single file. These parameters are defined on each patch (which were set in the blockMesh dictionary) of the domain and can be set to fixed values, periodic behavior, or any one of a number of other conditions. Symmetry planes are also set in these files by defining a symmetry plane for a given patch for each parameter. Finally, the last step is to specify all of the material properties in the 'constant' directory. The icoThermFsiFoam solver is designed to be run from the fluid directory so the user can either enter that directory and run the program or type: `icoThermFsiFoam –case fluid` from the case name directory.

## 3.1 Simulation Setup Parameters

For this simulation, laminar flow must be maintained in the fluid and the solid should be flexible enough that it will result in non-trivial displacement. The laminar flow requirement generally leads to the use of glycerin in physical experiments while a flexible solid could be made from polybutadiene or polypropylene. Table 3-2 shows the material properties for these three materials. For the sake of this simulation, simplified parameters were used which closely resemble these physical materials as can be seen in the same table.

**Table 3-2: Material Properties**

| Material | $\rho\ (10^3\ \frac{kg}{m^3})$ | $\nu_s$ | $\mu\ (10^6\ \frac{kg}{ms^2})$ |
|---|---|---|---|
| Polybutadiene | 0.91 | 0.5 | 0.53 |
| Polypropylene | 1.1 | 0.42 | 317 |
| | $\rho\ (10^3\ \frac{kg}{m^3})$ | $\nu_f\ (10^{-3}\ \frac{m^2}{s})$ | |
| Glycerin | 1.26 | 1.13 | |
| Simulation Properties | | | |
| | $\rho\ (10^3\ \frac{kg}{m^3})$ | $\nu_s$ | $\mu\ (10^6\ \frac{kg}{ms^2})$ |
| Solid | 1.0 | 0.4 | 0.5 |
| | $\rho\ (10^3\ \frac{kg}{m^3})$ | $\nu_f\ (10^{-3}\ \frac{m^2}{s})$ | |
| Fluid | 1.0 | 1.0 | |

# 4 Validation

## 4.1 Fluid solver grid independence

To have confidence in the results of any CFD simulation, it is important to determine whether the numerical error is acceptable. This numerical error will theoretically approach zero as the number of mesh points approaches infinity. The convergence of the CFD solution will also be less sensitive to mesh coarseness as the number of mesh points increases. It can be unpractical to have too fine a mesh as computation time increases substantially with no justifiable gain in accuracy.

The Richardson extrapolation technique as outlined in http://www.grc.nasa.gov/WWW/wind/valid/tutorial/spatconv.html was used to calculate the grid convergence for the fluid solver. An initial mesh was selected and then made more coarse by halving the number of grid points and more fine by doubling the number of grid points. Simulations were performed on these three grids with the steady state stagnation pressure at the leading edge of the sphere used as the quantity of comparison. This quantity was chosen due to its steady nature and because the pressure force is used to update the solid boundary. The results can be seen in Figure 4-1 which show that the initial mesh lies within the GCI (Grid Convergence Index) band and is therefore acceptable for future simulations.

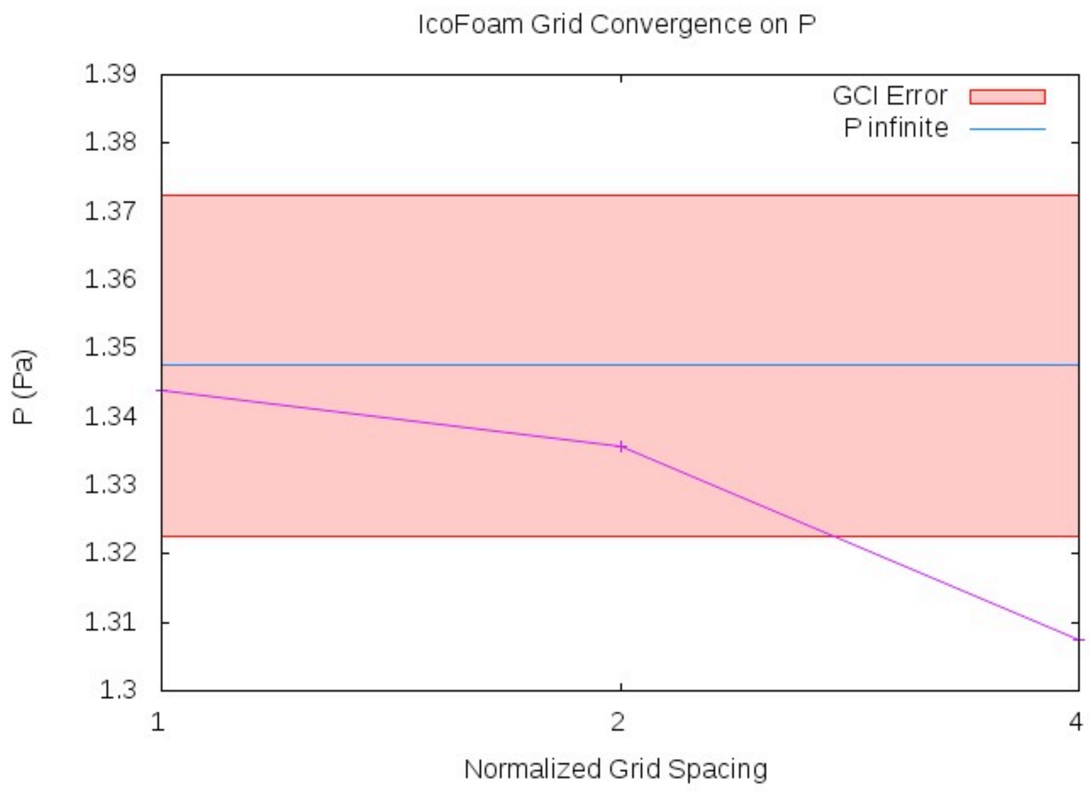


**Figure 4-1: Grid Independence of icoFoam on Turek-Hron geometry**

## 4.2 FSI solver validation

The icoThermFsiFoam solver created in section 2.2.1 does not fully implement heat transfer between the solid and fluid during an entire experimental run. The reason for this is that OpenFOAM does not contain the regionCoupling boundary condition for finite area solvers, which is the type of solver that the structural solver is. The regionCoupling boundary condition allows conjugate heat transfer between a solid and fluid domain. Due to the available moving mesh utilities in OpenFOAM and the need for a relatively robust structural solver, the choice was made to continue using a finite area based structural solver.

## 5 Complications

Several attempts were made to construct a finite area version of the regionCoupling boundary as many of the boundary conditions that are packaged with OpenFOAM are available in both finite area and finite volume forms. Ultimately, a functional finite area version of the regionCoupling boundary condition was unable to be built. This meant that the only time there was any coupling between the fluid and solid regions was at the inception of a simulation due to the initial conditions. Other similar boundary conditions were investigated but none were able to pass gradient information through a moving mesh at a solid/fluid boundary from a finite volume solver to a finite area solver.

When this project was started, OpenFOAM-1.5-dev was the most current available version of OpenFOAM that was capable of solving FSI simulations. During the course of the project, another branch of OpenFOAM was developed as OpenFOAM-1.6-ext. This branch still maintained some of the ability to solve FSI simulations but completely reformulated the structural solver and reworked the coupling between the fluid and structural solvers. There is a chance that with significantly more time, the thermal component of this project could be worked into the new framework of OpenFOAM-1.6-ext but it would mean the abandonment of all of the work undertaken in the OpenFOAM-1.5-dev framework. Future work should be carried out in the framework of OpenFOAM-1.6-ext as this is the most currently supported version of OpenFOAM that is capable of performing FSI simulations.

# 6 Discussion

## 6.1 How can it be used in the future?

FSI solvers, and particularly those that can simulate conjugate heat transfer, can be used to investigate any system where determining the interaction between a fluid and structure has a significant engineering implications. Historically, aeroelasticity has been a major driver for these types of solvers, however, other engineering fields are realizing the great potential for this type of analysis. FSI solvers utilizing heat transfer between the fluid and solid could be very useful in the design of future energy production systems such as those that convert wave energy to electricity. Biomedical engineering is already embracing the basic FSI abilities in modeling fluid movement in veins and arteries. Adding the ability to accurately predict heat transfer between the fluid and artery could help determine absorption rates of medicine based on temperature.

## 6.2 What else still needs to be done?

The next step in creating a robust FSI solver in OpenFOAM with heat transfer would be to fully implement the regionCoupling boundary condition for finite area solvers in version 1.5-dev. Further work could also include strongly coupling the fluid and structural solvers by checking the convergence between the predicted and actual computed structural displacements during each step of the simulation. With any computer modeling, it will be necessary to check the predicted results with actual experimental data to ensure that the model is accurate. As there is no currently existing heat transfer in FSI benchmark data, it makes the most sense to build off of existing FSI benchmark geometries as they feature the more complicated physics. Guidelines could then be developed in order to build a physical experiment based on the computer modeling.

## References

1 Masarati, P., Mategazza, P., “A Conservative Mesh-Free Approach For Fluid Structure Problems,” *International Conference on Computational Methods for Coupled Problems in Science and Engineering*, edited by M. Papadrakakis, E. Oñate, B. Schrefler, CIMNE, Barcelona, 2005, pp. 24-27.

2 Förster, C., Wall, W., Ramm, E., “Artificial added mass instabilities in sequential staggered coupling of nonlinear structures and incompressible viscous flows” *Computation Methods in Applied Mechanics and Engineering,* vol. 196, no. 7, 2007, pp. 1278-1293.

3 Campbell, R., “Fluid-Structure Interaction and Inverse Design Simulations for Flexible Turbomachinery,” Ph.D. Dissertation, College of Engineering, The Pennsylvania State University, State College, PA, 2010.

4 Turek, S., Hron, J., Razzaq, M., Wobker, H., Schäfer, M., “Numerical Benchmarking of Fluid Structure Interaction: A Comparison of Different Discretization and Solution Approaches,” *Fluid Structure Interaction II,* Vol. 73, 2010, pp. 413-424.

5 Maus, K., “Constructing solvers for weakly coupled FSI problems using OpenFOAM-1.5-dev,” PhD Course in CFD with OpenSource Software 2009, Chalmers University of Technology, Göteborg, Sweden, 2009.

6 Causin, P., Gerbeau, J.F., Nobile, F., “Added-mass effect in the design of partitioned algorithms for fluid-structure problems,” *Computer Methods in Applied Mechanics and Engineering*, Vol. 194, No. 42-44, 2005, pp. 4506-4527

7 Deparis, S., Fernández, M., Formaggia, L., “Acceleration of a Fixed Point Algorithm for Fluid-Structure Interaction Using Transpiration Conditions,” ESAIM: Mathematical Modeling and Numerical Analysis, Vol. 37, 2003 pp. 601-616.

8 Tapia, X., “Modeling of wind flow over complex terrain using OpenFoam,” M.S. Thesis, University of Gävle, Gävle, Sweden, 2009.

9 Jasak, H., “Error Analysis and Estimation for the Finite Volume Method with Applications to Fluid Flows,” Ph.D. Dissertation, Department of Mechanical Engineering, Imperial College of Science and Medicine, London, England, 1996.

# Appendix A – icoThermFsiFoam Solver Code

## icoThermFsiFoam.C

```
/*---------------------------------------------------------------------------
------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright held by original author
     \\/     M anipulation  |
---------------------------------------------------------------------------
--------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
the
    Free Software Foundation; either version 2 of the License, or (at
your
    option) any later version.

    OpenFOAM 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 General Public
License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Application
    icoThermFsiFoam

Description
    Transient FSI solver for incompressible, laminar flow of Newtonian
fluids and
    linear elastic, small-strain deformation solids.

\*---------------------------------------------------------------------------
------*/

#include "fvCFD.H"
#include "dynamicFvMesh.H"
#include "tractionDisplacementFvPatchVectorField.H"
#include "patchToPatchInterpolation.H"
#include "tetFemMatrices.H"
#include "faceTetPolyPatch.H"
#include "tetPolyPatchInterpolation.H"
#include "fixedValueTetPolyPatchFields.H"
#include "pointFields.H"
```

```
#include "volPointInterpolation.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

int main(int argc, char *argv[])
{

#   include "setRootCase.H"

#   include "createTime.H"
#   include "createDynamicFvMesh.H"
#   include "createStressMesh.H"
#   include "createFields.H"

#   include "readMechanicalProperties.H"
#   include "readThermalProperties.H"
#   include "createStressFields.H"
#   include "readCouplingProperties.H"
#   include "readTimeControls.H"

#   include "initContinuityErrs.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

    Info<< "\nStarting time loop\n" << endl;

    for (runTime++; !runTime.end(); runTime++)
    {
        Info<< "Time = " << runTime.timeName() << nl << endl;

#       include "readPISOControls.H"
#       include "CourantNo.H"

#   include "readTimeControls.H"
#   include "setDeltaT.H"
//Main solver code:
#   include "setPressure.H"
#   include "solveSolid.H"
#   include "setMotion.H"
#   include "solveFluid.H"

        runTime.write();

        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }

    Info<< "End\n" << endl;

    return(0);
}
```

```
//
********************************************************************
** //
```

**tractionDisplacementFvPatchVectorField.C**

```
/*--------------------------------------------------------------------
------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright held by original author
     \\/     M anipulation  |
--------------------------------------------------------------------
--------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
the
    Free Software Foundation; either version 2 of the License, or (at
your
    option) any later version.

    OpenFOAM 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 General Public
License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

\*--------------------------------------------------------------------
------*/

#include "tractionDisplacementFvPatchVectorField.H"
#include "addToRunTimeSelectionTable.H"
#include "volFields.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

namespace Foam
{

// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * *
* * * //

tractionDisplacementFvPatchVectorField::
tractionDisplacementFvPatchVectorField
(
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF
)
:
    fixedGradientFvPatchVectorField(p, iF),
    traction_(p.size(), vector::zero),
    pressure_(p.size(), 0.0)
{
    fvPatchVectorField::operator=(patchInternalField());
    gradient() = vector::zero;
}


tractionDisplacementFvPatchVectorField::
tractionDisplacementFvPatchVectorField
(
    const tractionDisplacementFvPatchVectorField& tdpvf,
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF,
    const fvPatchFieldMapper& mapper
)
:
    fixedGradientFvPatchVectorField(tdpvf, p, iF, mapper),
    traction_(tdpvf.traction_, mapper),
    pressure_(tdpvf.pressure_, mapper)
{}


tractionDisplacementFvPatchVectorField::
tractionDisplacementFvPatchVectorField
(
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF,
    const dictionary& dict
)
:
    fixedGradientFvPatchVectorField(p, iF),
    traction_("traction", dict, p.size()),
    pressure_("pressure", dict, p.size())
{
    fvPatchVectorField::operator=(patchInternalField());
    gradient() = vector::zero;
}


tractionDisplacementFvPatchVectorField::
tractionDisplacementFvPatchVectorField
(
    const tractionDisplacementFvPatchVectorField& tdpvf
)
:
    fixedGradientFvPatchVectorField(tdpvf),
    traction_(tdpvf.traction_),
    pressure_(tdpvf.pressure_)
{}


tractionDisplacementFvPatchVectorField::
```

```
tractionDisplacementFvPatchVectorField
(
    const tractionDisplacementFvPatchVectorField& tdpvf,
    const DimensionedField<vector, volMesh>& iF
)
:
    fixedGradientFvPatchVectorField(tdpvf, iF),
    traction_(tdpvf.traction_),
    pressure_(tdpvf.pressure_)
{}


// * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * *
* * * //

void tractionDisplacementFvPatchVectorField::autoMap
(
    const fvPatchFieldMapper& m
)
{
    fixedGradientFvPatchVectorField::autoMap(m);
    traction_.autoMap(m);
    pressure_.autoMap(m);
}


// Reverse-map the given fvPatchField onto this fvPatchField
void tractionDisplacementFvPatchVectorField::rmap
(
    const fvPatchVectorField& ptf,
    const labelList& addr
)
{
    fixedGradientFvPatchVectorField::rmap(ptf, addr);

    const tractionDisplacementFvPatchVectorField& dmptf =
        refCast<const tractionDisplacementFvPatchVectorField>(ptf);

    traction_.rmap(dmptf.traction_, addr);
    pressure_.rmap(dmptf.pressure_, addr);
}


// Update the coefficients associated with the patch field
void tractionDisplacementFvPatchVectorField::updateCoeffs()
{
    if (updated())
    {
        return;
    }

    const dictionary& mechanicalProperties =
        db().lookupObject<IOdictionary>("mechanicalProperties");

    const dictionary& thermalProperties =
        db().lookupObject<IOdictionary>("thermalProperties");
    dimensionedScalar rho(mechanicalProperties.lookup("rho"));
    dimensionedScalar rhoE(mechanicalProperties.lookup("E"));
    dimensionedScalar nu(mechanicalProperties.lookup("nu"));

    dimensionedScalar E = rhoE/rho;
    dimensionedScalar mu = E/(2.0*(1.0 + nu));
    dimensionedScalar lambda = nu*E/((1.0 + nu)*(1.0 - 2.0*nu));
    dimensionedScalar threeK = E/(1.0 - 2.0*nu);

    Switch planeStress(mechanicalProperties.lookup("planeStress"));

    if (planeStress)
    {
        lambda = nu*E/((1.0 + nu)*(1.0 - nu));
        threeK = E/(1.0 - nu);
    }

    vectorField n = patch().nf();

    const fvPatchField<tensor>& gradU =
        patch().lookupPatchField<volTensorField, tensor>("grad(U)");

    gradient() =
    (
        (traction_ - pressure_*n)/rho.value()
      - (n & (mu.value()*gradU.T() - (mu + lambda).value()*gradU))
      - n*tr(gradU)*lambda.value()
    )/(2.0*mu + lambda).value();


    Switch thermalStress(thermalProperties.lookup("thermalStress"));

    if (thermalStress)
    {
        dimensionedScalar alpha(thermalProperties.lookup("alpha"));
        dimensionedScalar threeKalpha = threeK*alpha;

        const fvPatchField<scalar>& T =
            patch().lookupPatchField<volScalarField, scalar>("T");

        gradient() += n*threeKalpha.value()*T/(2.0*mu +
lambda).value();
    }

    fixedGradientFvPatchVectorField::updateCoeffs();
}


// Write
void tractionDisplacementFvPatchVectorField::write(Ostream& os) const
{
    fvPatchVectorField::write(os);
    traction_.writeEntry("traction", os);
    pressure_.writeEntry("pressure", os);
    writeEntry("value", os);
}
```

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

makePatchTypeField(fvPatchVectorField,
tractionDisplacementFvPatchVectorField);

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

} // End namespace Foam

//
**********************************************************************
** //
```

**tractionDisplacementFvPatchVectorField.H**

```
/*----------------------------------------------------------------------
------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright held by original author
     \\/     M anipulation  |
------------------------------------------------------------------------
--------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
the
    Free Software Foundation; either version 2 of the License, or (at
your
    option) any later version.

    OpenFOAM 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 General Public
License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Class
    tractionDisplacementFvPatchVectorField

Description
    Fixed traction boundary condition for the standard linear elastic,
fixed
    coefficient displacement equation (stressedFoam).
```

```
SourceFiles
    tractionDisplacementFvPatchVectorField.C

\*---------------------------------------------------------------------------
------*/

#ifndef tractionDisplacementFvPatchVectorField_H
#define tractionDisplacementFvPatchVectorField_H

#include "fvPatchFields.H"
#include "fixedGradientFvPatchFields.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

namespace Foam
{

/*---------------------------------------------------------------------------
------*\
                   Class tractionDisplacementFvPatch Declaration
\*---------------------------------------------------------------------------
------*/

class tractionDisplacementFvPatchVectorField
:
    public fixedGradientFvPatchVectorField
{

    // Private Data

        vectorField traction_;
        scalarField pressure_;


public:

    //- Runtime type information
    TypeName("tractionDisplacement");


    // Constructors

        //- Construct from patch and internal field
        tractionDisplacementFvPatchVectorField
        (
            const fvPatch&,
            const DimensionedField<vector, volMesh>&
        );

        //- Construct from patch, internal field and dictionary
        tractionDisplacementFvPatchVectorField
        (
            const fvPatch&,
            const DimensionedField<vector, volMesh>&,
            const dictionary&
        );

        //- Construct by mapping given
        //  tractionDisplacementFvPatchVectorField onto a new patch
        tractionDisplacementFvPatchVectorField
        (
            const tractionDisplacementFvPatchVectorField&,
            const fvPatch&,
            const DimensionedField<vector, volMesh>&,
            const fvPatchFieldMapper&
        );

        //- Construct as copy
        tractionDisplacementFvPatchVectorField
        (
            const tractionDisplacementFvPatchVectorField&
        );

        //- Construct and return a clone
        virtual tmp<fvPatchVectorField> clone() const
        {
            return tmp<fvPatchVectorField>
            (
                new tractionDisplacementFvPatchVectorField(*this)
            );
        }

        //- Construct as copy setting internal field reference
        tractionDisplacementFvPatchVectorField
        (
            const tractionDisplacementFvPatchVectorField&,
            const DimensionedField<vector, volMesh>&
        );

        //- Construct and return a clone setting internal field
reference
        virtual tmp<fvPatchVectorField> clone
        (
            const DimensionedField<vector, volMesh>& iF
        ) const
        {
            return tmp<fvPatchVectorField>
            (
                new tractionDisplacementFvPatchVectorField(*this, iF)
            );
        }


    // Member functions

        // Access

            virtual const vectorField& traction() const
            {
                return traction_;
            }
            virtual vectorField& traction()
            {
                return traction_;
            }

            virtual const scalarField& pressure() const
            {
                return pressure_;
            }

            virtual  scalarField& pressure()
            {
                return pressure_;
            }


        // Mapping functions

            //- Map (and resize as needed) from self given a mapping
object
            virtual void autoMap
            (
                const fvPatchFieldMapper&
            );

            //- Reverse map the given fvPatchField onto this
fvPatchField
            virtual void rmap
            (
                const fvPatchVectorField&,
                const labelList&
            );


        //- Update the coefficients associated with the patch field
        virtual void updateCoeffs();

        //- Write
        virtual void write(Ostream&) const;
};


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

} // End namespace Foam

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

#endif

//
**********************************************************************
** //
```

## createStressMesh.H

```
    Foam::Info<< "Create stressMesh for time = "
    << runTime.timeName() << Foam::nl << Foam::endl;

    Foam::fvMesh stressMesh
    (
        Foam::IOobject
        (
            "solid",
            runTime.timeName(),
            runTime,
            Foam::IOobject::MUST_READ
        )
    );

    Foam::pointMesh pStressMesh(stressMesh);

    Foam::volPointInterpolation cpi
    (
      stressMesh,
      pStressMesh
    );
```

## createFields.H

```
    Info<< "Reading transportProperties\n" << endl;

    IOdictionary transportProperties
    (
        IOobject
        (
            "transportProperties",
            runTime.constant(),
            mesh,
            IOobject::MUST_READ,
            IOobject::NO_WRITE
        )
    );

    dimensionedScalar nu
    (
        transportProperties.lookup("nu")
    );

    dimensionedScalar rhoFluid
    (
      transportProperties.lookup("rho")
    );

    Info<< "Reading field p\n" << endl;
    volScalarField p
    (
        IOobject
        (
            "p",
            runTime.timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh
    );


    Info<< "Reading field U\n" << endl;
    volVectorField U
    (
        IOobject
        (
            "U",
            runTime.timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh
    );

    Info<< "Reading field T\n" << endl;
    volScalarField T
    (
      IOobject
      (
          "T",
          runTime.timeName(),
          mesh,
          IOobject::MUST_READ,
          IOobject::AUTO_WRITE
      ),
      mesh
    );

#   include "createPhi.H"


    label pRefCell = 0;
    scalar pRefValue = 0.0;
    setRefCell(p, mesh.solutionDict().subDict("PISO"), pRefCell,
pRefValue);
```

**readMechanicalProperties.H**

```
    Info<< "Reading mechanical properties\n" << endl;

    IOdictionary mechanicalProperties
    (
        IOobject
        (
            "mechanicalProperties",
            runTime.constant(),
            stressMesh,
            IOobject::MUST_READ,
            IOobject::NO_WRITE
        )
    );

    dimensionedScalar rho(mechanicalProperties.lookup("rho"));
    dimensionedScalar rhoE(mechanicalProperties.lookup("E"));
    dimensionedScalar nuS(mechanicalProperties.lookup("nu"));

    Info<< "Normalising E : E/rho\n" << endl;
    dimensionedScalar E = rhoE/rho;

    Info<< "Calculating Lame's coefficients\n" << endl;

    dimensionedScalar mu = E/(2.0*(1.0 + nuS));
    dimensionedScalar lambda = nuS*E/((1.0 + nuS)*(1.0 - 2.0*nuS));
    dimensionedScalar threeK = E/(1.0 - 2.0*nuS);

    Switch planeStress(mechanicalProperties.lookup("planeStress"));

    if (planeStress)
    {
        Info<< "Plane Stress\n" << endl;

        //- change lambda and threeK for plane stress
        lambda = nuS*E/((1.0 + nuS)*(1.0 - nuS));
        threeK = E/(1.0 - nuS);
    }
    else
    {
        Info<< "Plane Strain\n" << endl;
    }
    Info<< "mu = " << mu.value() << " Pa/rho\n";
    Info<< "lambda = " << lambda.value() << " Pa/rho\n";
    Info<< "threeK = " << threeK.value() << " Pa/rho\n";
```

**readThermalProperties.H**

```
    Info<< "Reading thermal properties\n" << endl;

    IOdictionary thermalProperties
    (
        IOobject
        (
            "thermalProperties",
            runTime.constant(),
            stressMesh,
            IOobject::MUST_READ,
            IOobject::NO_WRITE
        )
    );

    Switch thermalStress(thermalProperties.lookup("thermalStress"));

    dimensionedScalar threeKalpha
    (
        "threeKalpha",
        dimensionSet(0, 2, -2 , -1, 0),
        0
    );
    dimensionedScalar DT
    (
        "DT",
        dimensionSet(0, 2, -1 , 0, 0),
        0
    );

    if (thermalStress)
    {
        dimensionedScalar C(thermalProperties.lookup("C"));
        dimensionedScalar rhoK(thermalProperties.lookup("k"));
        dimensionedScalar alpha(thermalProperties.lookup("alpha"));

        Info<< "Normalising k : k/rho\n" << endl;
        dimensionedScalar k = rhoK/rho;

        Info<< "Calculating thermal coefficients\n" << endl;

        threeKalpha = threeK*alpha;
        DT.value() = (k/C).value();

        Info<< "threeKalpha = " << threeKalpha.value() << " Pa/rho\n";
    }
```

**createStressFields.H**

```
    Info<< "Reading field U\n" << endl;
    volVectorField Usolid
    (
        IOobject
        (
            "U",
            runTime.timeName(),
            stressMesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        stressMesh
    );


    volScalarField* Tptr = NULL;

    if (thermalStress)
    {
        Info<< "Reading field T\n" << endl;
        Tptr = new volScalarField
        (
            IOobject
            (
                "T",
                runTime.timeName(),
                stressMesh,
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
            ),
            stressMesh
        );
    }

    volScalarField& Tsolid = *Tptr;
```

**readCouplingProperties.H**

```
    Info << "Reading coupling properties" << endl;
    IOdictionary couplingProperties
    (
        IOobject
        (
            "couplingProperties",
            runTime.constant(),
            mesh,
            IOobject::MUST_READ,
            IOobject::NO_WRITE
        )
    );

    // Read solid patch data
    word solidPatchName(couplingProperties.lookup("solidPatch"));

    label solidPatchID =
        stressMesh.boundaryMesh().findPatchID(solidPatchName);


    // Read fluid patch data
    word fluidPatchName(couplingProperties.lookup("fluidPatch"));

    label fluidPatchID =
        mesh.boundaryMesh().findPatchID(fluidPatchName);


    if (solidPatchID < 0 || fluidPatchID < 0)
    {
        FatalErrorIn(args.executable())
            << "Problem with patch interpolation definition"
            << abort(FatalError);
    }


    // Create interpolators
    patchToPatchInterpolation interpolatorFluidSolid
    (
        mesh.boundaryMesh()[fluidPatchID],
        stressMesh.boundaryMesh()[solidPatchID]
    );

    patchToPatchInterpolation interpolatorSolidFluid
    (
        stressMesh.boundaryMesh()[solidPatchID],
        mesh.boundaryMesh()[fluidPatchID]
    );
    // Grab solid patch field
    tractionDisplacementFvPatchVectorField& tForce =
        refCast<tractionDisplacementFvPatchVectorField>
        (
            Usolid.boundaryField()[solidPatchID]
        );


    // Grab motion field

    // Read fluid patch data
    word movingRegionName(couplingProperties.lookup("movingRegion"));

    const fvMesh& motionMesh =
        runTime.objectRegistry::lookupObject<fvMesh>(movingRegionName);

    tetPointVectorField& motionU =
        const_cast<tetPointVectorField&>
        (

motionMesh.objectRegistry::lookupObject<tetPointVectorField>
            (
                "motionU"
            )
        );


    fixedValueTetPolyPatchVectorField& motionUFluidPatch =
        refCast<fixedValueTetPolyPatchVectorField>
        (
            motionU.boundaryField()[fluidPatchID]
        );

    tetPolyPatchInterpolation tppi
    (
        refCast<const faceTetPolyPatch>(motionUFluidPatch.patch())
    );
```

**setPressure.H**

```
{
    // Setting pressure on solid patch
    Info << "Setting pressure" << endl;

    scalarField solidPatchPressure =
        interpolatorFluidSolid.faceInterpolate
        (
            p.boundaryField()[fluidPatchID]
        );

    solidPatchPressure *= rhoFluid.value();

    tForce.pressure() = solidPatchPressure;


    vector totalPressureForce =
        sum
        (
            p.boundaryField()[fluidPatchID]*
            mesh.Sf().boundaryField()[fluidPatchID]
        );


    Info << "Total pressure force = " << totalPressureForce << endl;
}
```

## solveSolid.H

```
/*---------------------------------------------------------------------------
------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright held by original author
     \\/     M anipulation  |
---------------------------------------------------------------------------
--------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
the
    Free Software Foundation; either version 2 of the License, or (at
your
    option) any later version.

    OpenFOAM 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 General Public
License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Application
    solveSolid

Description
    Transient/steady-state segregated finite-volume solver of linear-
elastic,
    small-strain deformation of a solid body, with optional thermal
    diffusion and thermal stresses.

    Simple linear elasticity structural analysis code.
    Solves for the displacement vector field U, also generating the
    stress tensor field sigma.

\*---------------------------------------------------------------------------
------*/
```

```
{
#       include "readStressedFoamControls.H"

        int iCorr = 0;
        scalar initialResidual = 0;

        do
        {
            volTensorField gradU = fvc::grad(Usolid);

            if (thermalStress)
            {
                solve
                (
                    fvm::ddt(Tsolid) == fvm::laplacian(DT, Tsolid)
                );
            }

            fvVectorMatrix UEqn
            (
                fvm::d2dt2(Usolid)
             ==
                fvm::laplacian(2*mu + lambda, Usolid,
"laplacian(DU,U)")

              + fvc::div
                (
                    mu*gradU.T() + lambda*(I*tr(gradU)) - (mu +
lambda)*gradU,
                    "div(sigma)"
                )
            );

            if (thermalStress)
            {
                UEqn += threeKalpha*fvc::grad(Tsolid);
            }

            //UEqn.setComponentReference(1, 0, vector::X, 0);
            //UEqn.setComponentReference(1, 0, vector::Z, 0);

            initialResidual = UEqn.solve().initialResidual();

        } while (initialResidual > convergenceTolerance && ++iCorr <
nCorr);

#       include "calculateStress.H"

}


//
**********************************************************************
** //
```

**readStressedFoamControls.H**

```
    const dictionary& stressControl =
        stressMesh.solutionDict().subDict("stressedFoam");

    int nCorr(readInt(stressControl.lookup("nCorrectors")));
    scalar convergenceTolerance(readScalar(stressControl.lookup("U")));
```

**calculateStress.H**

```
    if (runTime.outputTime())
    {
        volTensorField gradU = fvc::grad(Usolid);

        volSymmTensorField sigma =
            rho*(2.0*mu*symm(gradU) + lambda*I*tr(gradU));

        if (thermalStress)
        {
            sigma = sigma - I*(rho*threeKalpha*Tsolid);
        }

        volScalarField sigmaEq
        (
            IOobject
            (
                "sigmaEq",
                runTime.timeName(),
                stressMesh,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            sqrt((3.0/2.0)*magSqr(dev(sigma)))
        );

        Info<< "Max sigmaEq = " << max(sigmaEq).value()
            << endl;

        volScalarField sigmaxx
        (
            IOobject
            (
                "sigmaxx",
                runTime.timeName(),
                stressMesh,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            sigma.component(symmTensor::XX)
        );

        volScalarField sigmayy
        (
            IOobject
            (
                "sigmayy",
                runTime.timeName(),
                stressMesh,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            sigma.component(symmTensor::YY)
        );

        volScalarField sigmazz
        (
            IOobject
            (
                "sigmazz",
                runTime.timeName(),
                stressMesh,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            sigma.component(symmTensor::ZZ)
        );

        Info<< "Max sigmazz = " << max(sigmazz).value()
            << endl;

        volScalarField sigmaxy
        (
            IOobject
            (
                "sigmaxy",
                runTime.timeName(),
                stressMesh,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            sigma.component(symmTensor::XY)
        );

        volScalarField sigmaxz
        (
            IOobject
            (
                "sigmaxz",
                runTime.timeName(),
                stressMesh,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            sigma.component(symmTensor::XZ)
        );

        volScalarField sigmayz
        (
            IOobject
            (
                "sigmayz",
                runTime.timeName(),
                stressMesh,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            sigma.component(symmTensor::YZ)
        );

        runTime.write();
    }
```

## setMotion.H

```
{
    // Setting mesh motion

    pointVectorField solidPointsDispl =
        cpi.interpolate(Usolid - Usolid.oldTime());

    vectorField newPoints =
        stressMesh.points()
      + solidPointsDispl.internalField();

    stressMesh.movePoints(newPoints);


    vectorField fluidPatchPointsDispl =
        interpolatorSolidFluid.pointInterpolate
        (
            solidPointsDispl.boundaryField()[solidPatchID].
            patchInternalField()
        );

    motionUFluidPatch ==
        tppi.pointToPointInterpolate
        (
            fluidPatchPointsDispl/runTime.deltaT().value()
        );

    mesh.update();

#   include "volContinuity.H"

    Info << "Motion magnitude: mean = "
        << average(mag(Usolid.boundaryField()[solidPatchID]))
        << " max = "
        << max(mag(Usolid.boundaryField()[solidPatchID])) << endl;
}
```

## solveFluid.H

```
/*----------------------------------------------------------------------
------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright held by original author
     \\/     M anipulation  |
-----------------------------------------------------------------------
--------
License
    This file is part of OpenFOAM.
    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
the
    Free Software Foundation; either version 2 of the License, or (at
your
    option) any later version.

    OpenFOAM 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 General Public
License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Application
    solveFluid

Description
    Transient solver for incompressible, laminar flow of Newtonian
fluids
    with thermal transport.

\*----------------------------------------------------------------------
------*/

{
        fvVectorMatrix UEqn
        (
            fvm::ddt(U)
          + fvm::div(phi, U)
          - fvm::laplacian(nu, U)
        );

        solve(UEqn == -fvc::grad(p));

        // --- PISO loop

        for (int corr=0; corr<nCorr; corr++)
        {

#     include "TEqn.H"

            volScalarField rUA = 1.0/UEqn.A();

            U = rUA*UEqn.H();
            phi = (fvc::interpolate(U) & mesh.Sf())
                + fvc::ddtPhiCorr(rUA, U, phi);

            adjustPhi(phi, U, p);

            for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
            {
                fvScalarMatrix pEqn
                (
                    fvm::laplacian(rUA, p) == fvc::div(phi)
                );

                pEqn.setReference(pRefCell, pRefValue);
                pEqn.solve();

                if (nonOrth == nNonOrthCorr)
                {
                    phi -= pEqn.flux();
                }
            }

#           include "movingMeshContinuityErrs.H"

            U -= rUA*fvc::grad(p);
            U.correctBoundaryConditions();
        }
}


//
************************************************************************
** //
```

## TEqn.H

```
// this file will be called to solve the temperature (T) component of
the fluid

solve
(
    fvm::ddt(T)
  + fvm::div(phi, T)
  - fvm::laplacian(DT, T)
);
```

## Make/files

```
tractionDisplacement/tractionDisplacementFvPatchVectorField.C
icoThermFsiFoam.C

EXE = $(FOAM_USER_APPBIN)/icoThermFsiFoam
```

## Make/options

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -ItractionDisplacement \
    -I$(LIB_SRC)/dynamicFvMesh/lnInclude \
    $(WM_DECOMP_INC) \
    -I$(LIB_SRC)/tetDecompositionFiniteElement/lnInclude

EXE_LIBS = \
    -lfiniteVolume \
    -ldynamicFvMesh \
    $(W_DECOMP_LIBS) \
    -llduSolvers
```

# Appendix B – icoThermFoam solver code

## icoThermFoam.C

```
/*---------------------------------------------------------------------------
------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright held by original author
     \\/     M anipulation  |
---------------------------------------------------------------------------
--------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
the
    Free Software Foundation; either version 2 of the License, or (at
your
    option) any later version.

    OpenFOAM 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 General Public
License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Application
    icoThermFoam

Description
    Transient solver for incompressible, laminar flow of Newtonian
fluids with
    thermal transport.

\*---------------------------------------------------------------------------
------*/

#include "fvCFD.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

int main(int argc, char *argv[])
{

#   include "setRootCase.H"
#   include "createTime.H"
#   include "createMesh.H"
#   include "createFields.H"
#   include "initContinuityErrs.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * //

    Info<< "\nStarting time loop\n" << endl;

    for (runTime++; !runTime.end(); runTime++)
    {
        Info<< "Time = " << runTime.timeName() << nl << endl;

#       include "readPISOControls.H"
#       include "CourantNo.H"

        fvVectorMatrix UEqn
        (
            fvm::ddt(U)
          + fvm::div(phi, U)
          - fvm::laplacian(nu, U)
        );

        solve(UEqn == -fvc::grad(p));

        // --- PISO loop

        for (int corr=0; corr<nCorr; corr++)
        {

#     include "TEqn.H"

            volScalarField rUA = 1.0/UEqn.A();

            U = rUA*UEqn.H();
            phi = (fvc::interpolate(U) & mesh.Sf())
                + fvc::ddtPhiCorr(rUA, U, phi);

            adjustPhi(phi, U, p);

            for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
            {
                fvScalarMatrix pEqn
                (
                    fvm::laplacian(rUA, p) == fvc::div(phi)
                );

                pEqn.setReference(pRefCell, pRefValue);
                pEqn.solve();

                if (nonOrth == nNonOrthCorr)
                {
                    phi -= pEqn.flux();
                }
            }
#           include "continuityErrs.H"

            U -= rUA*fvc::grad(p);
            U.correctBoundaryConditions();
        }

        runTime.write();

        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }

    Info<< "End\n" << endl;

    return(0);
}


//
**********************************************************************
** //
```

**createFields.H**

```
    Info<< "Reading transportProperties\n" << endl;

    IOdictionary transportProperties
    (
        IOobject
        (
            "transportProperties",
            runTime.constant(),
            mesh,
            IOobject::MUST_READ,
            IOobject::NO_WRITE
        )
    );

    dimensionedScalar nu
    (
        transportProperties.lookup("nu")
    );

    dimensionedScalar DT
    (
      transportProperties.lookup("DT")
    );

    Info<< "Reading field p\n" << endl;
    volScalarField p
    (
        IOobject
        (
            "p",
            runTime.timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh
    );


    Info<< "Reading field U\n" << endl;
    volVectorField U
    (
        IOobject
        (
            "U",
            runTime.timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh
    );

    Info<< "Reading field T\n" << endl;
    volScalarField T
    (
      IOobject
      (
          "T",
          runTime.timeName(),
          mesh,
          IOobject::MUST_READ,
          IOobject::AUTO_WRITE
      ),
      mesh
    );

#   include "createPhi.H"


    label pRefCell = 0;
    scalar pRefValue = 0.0;
    setRefCell(p, mesh.solutionDict().subDict("PISO"), pRefCell,
pRefValue);
```

### TEqn.H

```
// this file will be called to solve the temperature (T) component of
the fluid

solve
(
    fvm::ddt(T)
  + fvm::div(phi, T)
  - fvm::laplacian(DT, T)
);
```

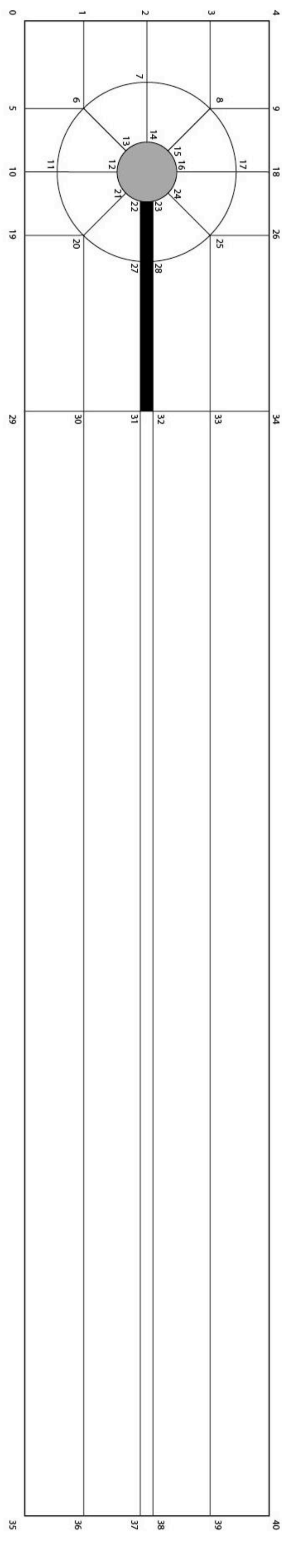
### Make/files

```
icoThermFoam.C

EXE = $(FOAM_USER_APPBIN)/icoThermFoam
```

**Make/options**

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude

EXE_LIBS = \
    -lfiniteVolume \
    -llduSolvers
```

## Appendix C – blockMesh Vertice Diagram and Coordinates



| **Coordinate #** | **X-Coordinate** | **Y-Coordinate** |
|---|---|---|
| 0 | 0.000000 | 0.000000 |
| 1 | 0.000000 | 0.939340 |
| 2 | 0.000000 | 2.000000 |
| 3 | 0.000000 | 3.060660 |
| 4 | 0.000000 | 4.100000 |
| 5 | 0.939340 | 0.000000 |
| 6 | 0.939340 | 0.939340 |
| 7 | 0.500000 | 2.000000 |
| 8 | 0.939340 | 3.060660 |
| 9 | 0.939340 | 4.100000 |
| 10 | 2.000000 | 0.000000 |
| 11 | 2.000000 | 0.500000 |
| 12 | 2.000000 | 1.500000 |
| 13 | 1.646450 | 1.646450 |
| 14 | 1.500000 | 2.000000 |
| 15 | 1.646450 | 2.353550 |
| 16 | 2.000000 | 2.500000 |
| 17 | 2.000000 | 3.500000 |
| 18 | 2.000000 | 4.100000 |
| 19 | 3.060660 | 0.000000 |
| 20 | 3.060660 | 0.939340 |
| 21 | 2.353550 | 1.646450 |
| 22 | 2.489898 | 1.900000 |
| 23 | 2.489898 | 2.100000 |
| 24 | 2.353550 | 2.358550 |
| 25 | 3.060660 | 3.060660 |
| 26 | 3.060660 | 4.100000 |
| 27 | 3.496660 | 1.900000 |
| 28 | 3.496660 | 2.100000 |
| 29 | 6.000000 | 0.000000 |
| 30 | 6.000000 | 0.939340 |
| 31 | 6.000000 | 1.900000 |
| 32 | 6.000000 | 2.100000 |
| 33 | 6.000000 | 3.060660 |
| 34 | 6.000000 | 4.100000 |
| 35 | 25.000000 | 0.000000 |
| 36 | 25.000000 | 0.939340 |
| 37 | 25.000000 | 1.900000 |
| 38 | 25.000000 | 2.100000 |
| 39 | 25.000000 | 3.060660 |
| 40 | 25.000000 | 4.100000 |