

Cite as: Shayegh, A.: Block-coupled Finite Volume algorithms: A solids4Foam tutorial. In Proceedings of CFD with OpenSource Software, 2020, Edited by Nilsson. H.,  
[http://dx.doi.org/10.17196/OS\\_CFD#YEAR\\_2020](http://dx.doi.org/10.17196/OS_CFD#YEAR_2020)

## CFD WITH OPENSOURCE SOFTWARE

A COURSE AT CHALMERS UNIVERSITY OF TECHNOLOGY  
TAUGHT BY HÅKAN NILSSON

---

# Block-coupled Finite Volume algorithms: A solids4Foam tutorial

---

Developed for foam-extend 4.1  
Requires: solids4Foam

*Author:*

ALI SHAYEGH  
Shiraz University  
alishayegh@pm.me

*Peer reviewed by:*

PHILIP CARDIFF  
University College Dublin

SAI DARBHA  
Monash University

SAEED SALEHI  
Chalmers University of Technology

Licensed under CC-BY-NC-SA, <https://creativecommons.org/licenses/>

Disclaimer: This is a student project work, done as part of a course where OpenFOAM and some other OpenSource software are introduced to the students. Any reader should be aware that it might not be free of errors. Still, it might be useful for someone who would like to learn some details similar to the ones presented in the report and in the accompanying files. The material has gone through a review process. The role of the reviewer is to go through the tutorial and make sure that it works, that it is possible to follow, and to some extent correct the writing. The reviewer has no responsibility for the contents.

January 18, 2021

# Learning outcomes

This tutorial tries to teach some points about block-coupled solid models using `solids4Foam` toolbox. The present tutorial does not intend to teach all of the `solids4Foam` capabilities. The reader interested in learning `solids4Foam` in general can refer to Cardiff [\[1\]](#).

The reader will learn:

## **How to use it:**

- How to setup cases for the `solids4Foam` solver in order to use the block-coupled finite volume (FV) algorithms for solid regions;

## **The theory of it:**

- How coupled algorithms work and how they compare to segregated ones;

## **How it is implemented:**

- How the `solids4Foam` structure differs from the conventional OpenFOAM solver structure;

## **How to modify it:**

- How to implement the numerical diffusion term in a block-coupled solid model.

# Prerequisites

The reader is expected to know the following in order to get the maximum benefit out of this report:

- How to set up and run simple standard document tutorials like the cavity case;
- How to use simple shell commands like `cd`, `ls`, ...;
- The fundamentals of the finite volume method;
- What are the classes, functions and objects in C++ and how to implement them;
- How the OpenFOAM matrix `lduAddressing` works.

# Contents

<b>1</b>	<b>Preparation</b>	<b>5</b>
1.1	OpenFOAM case . . . . .	5
1.2	What is <code>solids4Foam</code> ? . . . . .	5
1.3	Preparing <code>solids4Foam</code> . . . . .	5
<b>2</b>	<b>My First Block-Coupled Simulations</b>	<b>7</b>
2.1	Block-Coupled in a Nutshell . . . . .	7
2.2	Solid Simulation . . . . .	8
2.2.1	Run . . . . .	8
2.2.2	Walk Through . . . . .	9
2.2.3	Modify . . . . .	13
2.3	FSI Simulation . . . . .	15
2.3.1	Run . . . . .	15
2.3.2	Walk Through . . . . .	17
<b>3</b>	<b>Theory</b>	<b>19</b>
3.1	Mathematical Model . . . . .	19
3.2	Equation Discretization . . . . .	19
3.3	Coupled vs Segregated . . . . .	21
<b>4</b>	<b>Implementation</b>	<b>22</b>
4.1	<code>solids4Foam</code> Structure and Implementation . . . . .	22
4.2	Block-Coupled Solid Model Implementation . . . . .	25
<b>5</b>	<b>Adding a Numerical Diffusion Term</b>	<b>29</b>
5.1	Numerical Diffusion . . . . .	29
5.2	Modified Block-Coupled Solid Model . . . . .	29
5.3	A Test Case . . . . .	30
<b>A</b>	<b>Diffusion Discretization</b>	<b>37</b>

# Nomenclature

## Acronyms

2D	Two Dimensional
CFD	Computational Fluid Dynamics
CSM	Computational Solid Mechanics
CV	Control Volume
FEM	Finite Element Method
FSI	Fluid-Solid Interaction
FV	Finite Volume
FVM	Finite Volume Method
RHS	Right-Hand Side

## English symbols

$\mathbf{f}$	Body force vector
$\mathbf{I}$	Second-order identity matrix
$\mathbf{n}$ , $n_i$	Cartesian surface unit normal
$\mathbf{T}$	Traction vector
$\mathbf{u}$	Displacement vector
$S$	Control volume surface boundary
$S_i$	Surface vector
$t$	time

## Greek symbols

$\sigma$	Stress tensor
$\delta_{ij}$	Kronecker delta function
$\Gamma$	Control volume surface boundary
$\lambda$ , $\mu$	Lamé's constants
$\Omega$	Arbitrary control volume

## Superscripts

T	Transpose (of a matrix)
exp	Explicit

## Subscripts

$f$	Summation index for faces/face centers
$n$	Normal component
$t$	Tangential component

# Chapter 1

## Preparation

### 1.1 OpenFOAM case

In OpenFOAM jargon, a *case* is a problem that is to be solved. A case is a directory (a.k.a. folder) that contains all of the information needed for solving the problem. There are a number of folders within the case directory, each of which contains some part of the information/settings needed in the form of text files (a.k.a *dictionaries*). For any problem, the classification of the information needed and the directory name in which this information is enclosed through dictionaries is as follows:

- Initial and boundary conditions: start-time directory (usually `0/` directory);
- Equation discretization and solution procedure: `system/` directory;
- Other settings (mesh, turbulence models, ...): `constant/` directory.

### 1.2 What is solids4Foam?

`solids4foam` is a toolbox for OpenFOAM with capabilities for solid mechanics and fluid-solid interactions [2]. The overall aim of the `solids4foam` project is to develop an OpenFOAM toolbox for solid mechanics and fluid-solid interactions that is [1]:

- intuitive to *use* for new users;
- easy to *understand* at the case and code level;
- straightforward to *maintain*;
- uncomplicated to *extend*.

### 1.3 Preparing solids4Foam

By the time of writing this tutorial, to get the most out of `solids4Foam`, you need first to get `foam-extend` 4.0 or 4.1. While other OpenFOAM forks are also partially compatible with `solids4Foam`, block-coupled solid models are currently supported only by `foam-extend`. When you have finished the `foam-extend` installation, it is easy to install `solids4Foam`. You can always find the most-updated `solids4Foam` installation instructions in the repository [2], therefore it will not be repeated here.

In the remainder of this tutorial, it is assumed that:

- `foam-extend` 4.1 (or 4.0) is known to the terminal. In order to test it, type the command `foamVersion`; if it returns `foamVersion: command not found`, then you have to turn OpenFOAM commands and variables on by sourcing `bashrc` file or using `aliases` if you have set any;

- After downloading *solids4Foam*, you have copied its *tutorials* directory to your *\$FOAM\_RUN* directory and renamed it to *solids4FoamTut*. Before copying, it is a good practice to run Listing 1.1.

Listing 1.1: Creating *\$FOAM\_RUN*; no effect if existing

```
mkdir -p $FOAM_RUN
```

## Chapter 2

# My First Block-Coupled Simulations

### 2.1 Block-Coupled in a Nutshell

The implicit cell-centered finite volume discretisation of the governing equations of a solid region results in a system of algebraic equations, i.e.,

$$\mathbf{A}\mathbf{D} = \mathbf{B} \quad (2.1)$$

where  $\mathbf{A}$  is the coefficient matrix,  $\mathbf{D}$  is the solution vector and contains all of the cell-center displacement vectors and  $\mathbf{B}$  is the source vector. In order to find  $\mathbf{D}$ , one can solve 3 distinct systems—one system for each displacement component, i.e.,

$$\mathbf{A}_x \mathbf{D}_x = \mathbf{B}_x \quad (2.2)$$

$$\mathbf{A}_y \mathbf{D}_y = \mathbf{B}_y \quad (2.3)$$

$$\mathbf{A}_z \mathbf{D}_z = \mathbf{B}_z \quad (2.4)$$

where  $\mathbf{D}_x$ ,  $\mathbf{D}_y$  and  $\mathbf{D}_z$  are the solution vectors containing  $x$ -component,  $y$ -component and  $z$ -component of the cell-center displacement vectors respectively. These three systems have to be solved sequentially. This solution procedure is termed a *segregated* approach [3]. For the first time, it was introduced for the solid mechanics by Demirdžić et al. [4].

Another approach to solve Eq. (2.1) is termed *coupled* through which the system is solved at once. For a typical mesh shown in Figure 2.1, using the segregated approach, the system of, say, Eq. 2.2 is written as

$$\begin{bmatrix} a_x^{11} & a_x^{12} & \dots & a_x^{19} \\ a_x^{21} & a_x^{22} & \dots & a_x^{29} \\ \vdots & \vdots & \ddots & \vdots \\ a_x^{91} & a_x^{92} & \dots & a_x^{99} \end{bmatrix} \begin{bmatrix} D_x^1 \\ D_x^2 \\ \vdots \\ D_x^9 \end{bmatrix} = \begin{bmatrix} B_x^1 \\ B_x^2 \\ \vdots \\ B_x^9 \end{bmatrix}$$

where the components of the coefficient matrix, i.e.,  $a_x^{11}$ ,  $a_x^{12}$ ,  $\dots$  are obviously scalars. However, for the coupled approach, the system of Eq. (2.1) is like

$$\begin{bmatrix} [A^{11}] & [A^{12}] & \dots & [A^{19}] \\ [A^{21}] & [A^{22}] & \dots & [A^{29}] \\ \vdots & \vdots & \ddots & \vdots \\ [A^{91}] & [A^{92}] & \dots & [A^{99}] \end{bmatrix} \begin{bmatrix} [D^1] \\ [D^2] \\ \vdots \\ [D^9] \end{bmatrix} = \begin{bmatrix} [B^1] \\ [B^2] \\ \vdots \\ [B^9] \end{bmatrix} \quad (2.5)$$



where  $[A^{ij}]$ s, are matrix (also known as *blocks* [5]) and  $[D^j]$ s and  $[B^j]$ s are vectors, e.g.,

$$[A^{11}] = \begin{bmatrix} a_{xx}^{11} & a_{xy}^{11} & a_{xz}^{11} \\ a_{yx}^{11} & a_{yy}^{11} & a_{yz}^{11} \\ a_{zx}^{11} & a_{zy}^{11} & a_{zz}^{11} \end{bmatrix}, [D^1] = \begin{bmatrix} D_x^1 \\ D_y^1 \\ D_z^1 \end{bmatrix}, [B^1] = \begin{bmatrix} B_x^1 \\ B_y^1 \\ B_z^1 \end{bmatrix}$$

where the off-diagonal components of  $[A^{ij}]$  are responsible for inter-component coupling of the displacement field. If a fully implicit equation discretisation is adopted, i.e., dependent variable (displacement vector) contributes only to the coefficient matrix of Eq. (2.5), then using the coupled approach removes the deferred correction (i.e. outer loop). Therefore the solution will be obtained in a single step instead of multiple loop trials. This is the essence of Cardiff et al.’s method [5] called *block-coupled*<sup>1</sup>. The name “block-coupled” originates from the appearance of the coefficient matrix which looks like it is filled by blocks (Eq. (2.5)). With a proper choice for the linear solver, Cardiff et al. [5] showed that their new method noticeably outperforms segregated FV methods and a commercial finite element code.

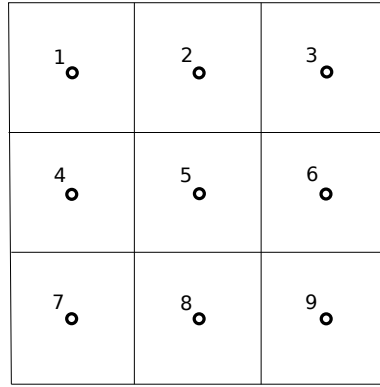


Figure 2.1: A typical mesh

## 2.2 Solid Simulation

Let us start with a specific problem. Preparing a case for `solids4Foam` is the same as for the conventional OpenFOAM solvers. There are only minor differences which are not challenging for an OpenFOAM user.

### 2.2.1 Run

• **Problem Statement** We are going to calculate the displacement field (D) for the problem shown in Figure 2.2; an elastic cantilever with a vertical load on its right end.

• **Prepare** The common practice for preparing a case in OpenFOAM is to find an existing tutorial which uses the suitable solver and then modify it based on our needs. Here, we copy `coupledCantilever2D` case from `solids4FoamTut` into the `run` directory. To do so, type the following commands line-by-line in the terminal and press enter after each line:

Listing 2.1: Find the case

```
cd $FOAM_RUN
find solids4FoamTut -type d -iname coupledCantilever2D
```

<sup>1</sup>In Cardiff et al. [5], boundary values are also treated implicitly, but we ignore that in Eq. (2.5) for the sake of simplicity.



Figure 2.2: Cantilever deflection problem; (length = 2 m and thickness = 0.1 m.)

which results in something like this:

```
solids4FoamTut/solids/linearElasticity/cantilever2d/coupledCantilever2d
```

Now, copy the above directory to your run directory:

```
cp -r solids4FoamTut/solids/linearElasticity/cantilever2d/coupledCantilever2d $FOAM_RUN
```

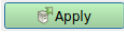
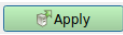
• **Run** Continue with:

```
cd coupledCantilever2d
bash Allrun
```

The simulation will start and finish within 1 s! this isn't, however, the case for all solid simulations; this case is very simple, but enough to learn how to setup and run a solid simulation case.

• **View results** Now:

```
paraFoam&
```

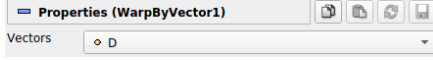
will open ParaView. Continue with  → Filters in menu bar → Search.... In the window that appears, search for warp by vector (Figure 2.3c), press Enter, choose D from Vector's drop-down list (Figure 2.3a) and press . Now you can see the deflected cantilever in the render view (Figure 2.4). You can make it colored based on the different known fields, e.g. D Magnitude; to this end, in the Properties pane, choose D under Coloring (Figure 2.3b).

### 2.2.2 Walk Through

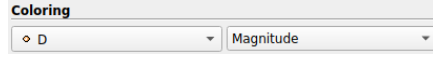
• **Start-time** Open 0/D with your favorite text editor (here I use vim). Based on the C++ convention, the lines between `/*` and `*/` and also the lines started by `//` are comments; they have no effect and, based on the editor, they may be displayed in a different color than the rest of the text. Listing 2.2 shows the first piece of the text that takes action; it is unique for each field and is not edited most of the times.

Listing 2.2: Typical header for boundary and initial conditions

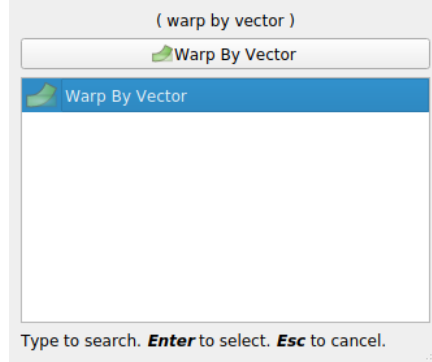
```
FoamFile
{
    version    2.0;
    format     ascii;
    class      volVectorField;
    location   "0";
    object     D;
}
```



(a) Select D vector



(b) Select coloring by D magnitude



(c) Filters search pane

Figure 2.3: Warp By Vector filter

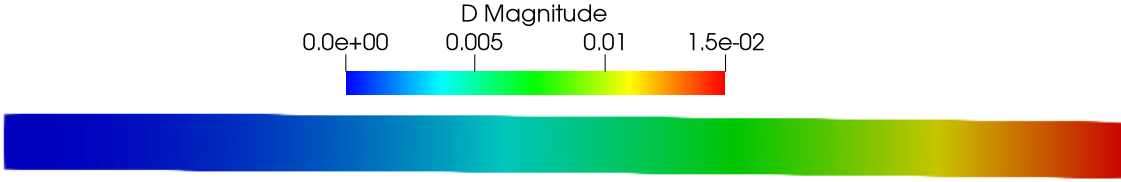


Figure 2.4: Deflected cantilever: D magnitude contours (in m)

The next line that takes action is shown in Listing 2.3; it specifies the units for the field (here *meter* for field D). The order of the units is shown in Table 2.1. The default units are SI, but they can be changed to any other system; here we will not talk more about it.

Listing 2.3: Units

```
dimensions [0 1 0 0 0 0];
```

Table 2.1: Order of units

Quantity	mass	Length	Time	Temperature	Amount	Current	Luminous intensity
Unit	kg	m	s	K	mol	Ampere	Candela

The next line (Listing 2.4) sets all of the internal cell values at the start of the present time (here it is  $t = 0$  s) i.e. the initial conditions. Therefore, we have set the three D component values equal to zero for all of the cells at  $t = 0$  s.

Listing 2.4: Initial conditions

```
internalField uniform (0 0 0);
```

In the next part of the dictionary, boundary conditions are set (Listing 2.5).

Listing 2.5: Boundary conditions

```
boundaryField
{
    ...
}
```

Currently, for the block-coupled solid model, `solids4Foam` supports the boundary conditions shown in Table 2.2. In the current problem (Figure 2.2), the cantilever has six sides; the left side is fixed in its position: `blockFixedDisplacement`; on the right hand side, a fixed value traction is exerted: `blockSolidTraction`; top and bottom sides are traction-free: `blockSolidTraction` with zero components for traction. Regarding `blockSolidTraction`, it is worth mentioning that `pressure` keyword is not referring to hydro-static pressure; instead, *applied traction* on a patch is obtained by [1]

$$\text{applied traction} = \text{traction} - \mathbf{n} * \text{pressure}$$

where  $\mathbf{n}$  is the face unit normal vector. Front and back sides has no boundary condition because the problem is 2D and no equation will be solved for the normal direction to the page, therefore empty. For each *patch*,<sup>2</sup> boundary condition is set through a sub-dictionary named by the patch name. Listing 2.6 shows such sub-dictionaries for the `right` and the `left` patches.

Table 2.2: Supported boundary conditions for block-coupled solid solver

Type	Explanation
<code>blockFixedDisplacement</code>	Fixed value displacement (Dirichlet)
<code>blockFixedDisplacementZeroShear</code>	Fixed value normal displacement and zero shear traction [2] (Mixed)
<code>blockFixedGradient</code>	Fixed value normal gradient of displacement [2]
<code>blockSolidTraction</code>	Fixed value traction [2]
<code>blockSolidVelocity</code>	Fixed value velocity [2]

- **system** Within the `system` directory, there are always at least three files (a.k.a. dictionaries): `controlDict`, `fvSolution` and `fvSchemes`.

**controlDict** as its name suggests, controls a couple of things. Listing 2.7 shows the important lines of the `controlDict` for the present case. `application` entry (line 1) takes no effect, unless for some automating scripts. The time from which the simulation starts is set through the next two entries. `startTime` (line 5) takes effect if `startFrom` (line 3) is set to `startTime`. Sometimes there are already results from a previous run and we want our simulation to continue that run. Setting `startFrom` to `latestTime`, overrides the `startTime` entry (here it is 0) and makes the simulation start from the largest existing time directory. It is not important in the present case whether `startFrom` is set to `startTime` or `latestTime`, because the only existing time directory is 0. The lines 7 and 9 play the same role as the lines 3 and 5, but for stopping the simulation. If `stopAt` is set to `writeNow`, the simulation stops after only one time step, regardless of the `endTime` entry. If it

Listing 2.6: Typical boundary condition sub-dictionaries

```

right
{
    type            blockSolidTraction;
    traction        uniform ( 0 -1e6 0 ); % Traction vector
    pressure        uniform 0;           % Normal component of traction
    value           uniform (0 0 0);     % Patch initial value
}
left
{
    type            blockFixedDisplacement;
    value           uniform (0 0 0);     % Fixed displacement vector
}

```

<sup>2</sup>A patch is a set of external cell faces with the same boundary condition; here we have six patches: right, left, top, bottom, front and back.

is set to `endTime`, the `endTime` entry determines when the simulation stops. `deltaT` determines the magnitude of the time step. `writeControl` controls how the run-time results are written out. It can be based on the simulation time, clock time or the number of time steps; consider the *banana trick* (see section 2.2.2) to see the different options. Lines 13 and 15 tell the solver to write out the results at every time step.

Listing 2.7: Important lines of `controlDict`

```

1 application    solids4Foam;    % Name of application
2
3 startFrom      startTime;      % startTime and latestTime are mostly used.
4
5 startTime      0;              % The simulation starts from it, if startFrom is set to startTime.
6
7 stopAt         endTime;        % endTime and writeNow are mostly used.
8
9 endTime        1;              % The simulation stops at it, if stopAt is set to endTime.
10
11 deltaT         1;              % Size of the time step.
12
13 writeControl   timeStep;       % How to control writing run-time results.
14
15 writeInterval  1;              % The period of writing the run-time results.

```

**fvSchemes** is a dictionary within which the discretisation schemes of the different terms are specified. The needed sub-dictionaries depend on the selected solid model. For example, a steady-state simulation of a `coupledUnsLinGeomLinearElasticSolid` model needs the entries shown in Listing 2.8. For a coupled solid model, there are new Laplacian discretisation schemes implemented in `solids4Foam`. Those are the needed schemes for creating a fully implicit block system [6]. More details about these new schemes and their implementation are described in Chapters 3 and 4.

Listing 2.8: `fvSchemes`

```

d2dt2Schemes
{
    default      steadyState;
}

ddtSchemes
{
    default      steadyState;
}

laplacianSchemes
{
    default              none;
    fvmBlockLaplacian(D) pointGaussLeastSquaresLaplacian;
    fvmBlockLaplacianTranspose(D) pointGaussLeastSquaresLaplacianTranspose;
    fvmBlockLaplacianTrace(D) pointGaussLeastSquaresLaplacianTrace;
}

```

**fvSolution** contains solution procedure settings (Listing 2.9). The solution procedure determines how to solve the block-coupled system of equations obtained from the discretisation stage. Currently, it is the best practice to use *direct* linear solvers. The theory behind it is related to rank deficiency of the coefficient matrix (Eq. (2.5)). The OpenFOAM library contains only *iterative* linear solver implementations. However, it is possible to use direct solvers by linking with external libraries, such as Eigen, MUMPS, PETSc or Trilinos [7]. The current implementation of `solids4Foam` is linked to the Eigen library. In Listing 2.9, we tell the solver to use a direct sparse system solver, `EigenSparseLU`, to solve the block system of equations for `D`, `blockD`.

Listing 2.9: fvSolution

```

solvers
{
    blockD                                % Dependent variable
    {
        // Direct solver
        solver                            EigenSparseLU; % Solver
    }
}

```

• **constant** The main difference in file names between a `solids4Foam` case and other conventional OpenFOAM solvers' cases appears in the `constant` directory. For a solid simulation case, there are at least four files in this directory: `physicsProperties`, `solidProperties`, `mechanicalProperties` and `g`, and one directory, `polyMesh`.

**physicsProperties** is a dictionary within which the simulation type is assigned, e.g. `fluid`, `solid` or `fluidSolidInteraction`.

**solidProperties** determines the formulation of solid region equations that are to be solved. The coupled solid model currently implemented in `solids4Foam` is named `coupledUnsLinearGeometryLinearElastic`. The source code [2] says that this model is a “Mathematical model where linear geometry i.e. small strains and small rotations are assumed”; only Hookean solids i.e. `linearElastic` mechanical law can be used [2].

**mechanicalProperties** is where we assign material properties, e.g. Young's modulus and the constitutive law, e.g. `linearElastic` for a Hookean elastic solid.

**g** is simply where we turn on/off the gravitational effect. It will be turned off if the `value` entry is assigned to `(0 0 0)`.

**polyMesh** is a directory containing the mesh data. Before running the case, there is only one dictionary named `blockMeshDict` in `polyMesh`. This is the dictionary read by the command `blockMesh` which writes out the necessary mesh files to `polyMesh`. For more information about how to modify the `blockMeshDict` and what files are needed for a mesh to be defined in OpenFOAM, see OpenFOAM user guide [8].

**Bonus: Banana Trick** How to know what are the different options accessible for each entry in a dictionary (e.g. `writeControl` in `controlDict`)? There is no drop-down menu to select another option for each entry like GUI-based packages. So, one way is to look at the source code and see what are the available options... wait! There is still a much simpler way to see the options. Write an arbitrary word (which you are sure that is not used in OpenFOAM source code, like *banana* or *dummy*) as the value of an entry, e.g. `banana` for `writeControl` in `controlDict` of the cantilever case. OpenFOAM is designed such that it returns an error for this unknown entry and suggests the possible choices. There are of course some situations that it returns only a warning, or it returns an error but doesn't suggest the true possible choices. To check this out, try the *banana trick* for one of the boundary type entries in `0/D`.

### 2.2.3 Modify

Let us modify and re-run the cantilever case based on our knowledge from the previous section. Assume that the right and the left sides of the cantilever are fixed in the different heights (Figure 2.5). Does this figure show a physical displacement? Let us examine this different set of boundary

conditions. To do so, change the `left` and the `right` boundary conditions in `0/D` based on Listing 2.10.

Listing 2.10: New `left` and `right` boundary condition sub-dictionaries

```
right
{
    type      blockFixedDisplacement;
    value     uniform (0 0.1 0);
}
left
{
    type      blockFixedDisplacement;
    value     uniform (0 -0.1 0);
}
```

Now, run the following in order to reset the case as it was before running:

```
foamCleanTutorials
```

Now, running the case (Listing 2.11) will end up with the displacement results shown in Figure 2.6. The first command in Listing 2.11 creates the mesh based on the `blockMeshDict`. The second line calls the solver.

Listing 2.11: Running the case

```
blockMesh
solids4Foam
```



Figure 2.5: Two-sides-fixed cantilever schematic

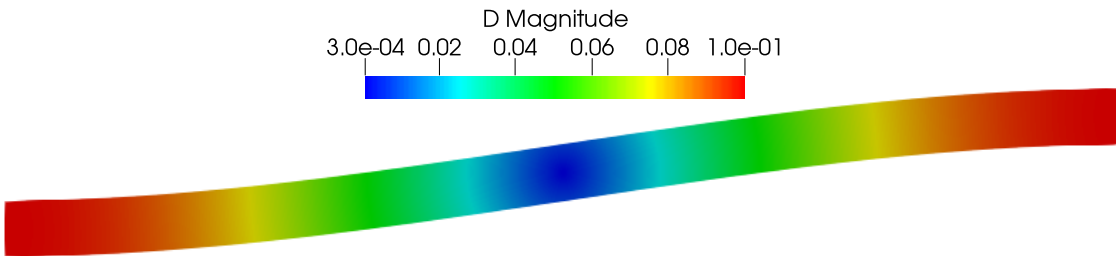


Figure 2.6: Two-sides-fixed cantilever deflection (in m)

## 2.3 Fluid-Solid Interaction<sup>3</sup> Simulation

### 2.3.1 Run

• **Problem statement** Dam-break is a simple two-phase case which can be found in all of the OpenFOAM forks. A schematic of this case is shown in Figure 2.7. The problem is that the dam breaks (or more accurately disappears!) at  $t = 0$  s and we want to calculate the fluid and solid behavior after that. Note that there is a flexible solid block behind the dam which interacts with the fluid.

• **Prepare** Like the 2D cantilever case, in order to copy the case to `$FOAM_RUN`, first find it by the commands shown in Listing 2.12.

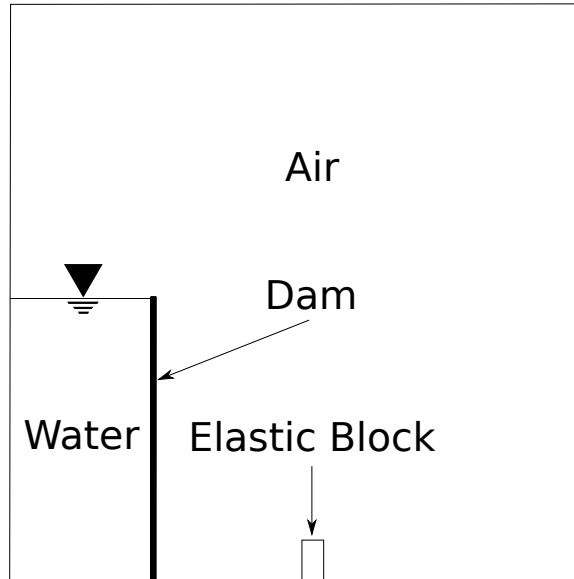


Figure 2.7: Flexible dam break problem

Listing 2.12: Find the case

```
cd $FOAM_RUN
find solids4FoamTut -type d -iname flexibleDamBreak
```

which results in something like:

```
solids4FoamTut/fluidSolidInteraction/flexibleDamBreak
```

Now, copy this directory to your `$FOAM_RUN`:

```
cp -r solids4FoamTut/fluidSolidInteraction/flexibleDamBreak $FOAM_RUN
```

• **Run** An Allrun script is enclosed within the case; simply execute it by:

```
cd flexibleDamBreak
bash Allrun
```

The simulation will finish within (approximately) 10 *min*.

<sup>3</sup>a.k.a FSI.



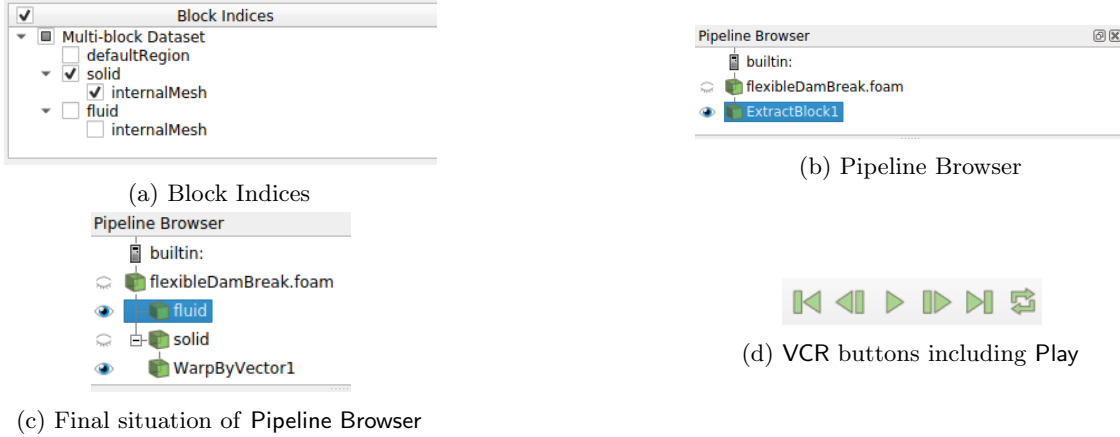




Figure 2.8: Visualization settings

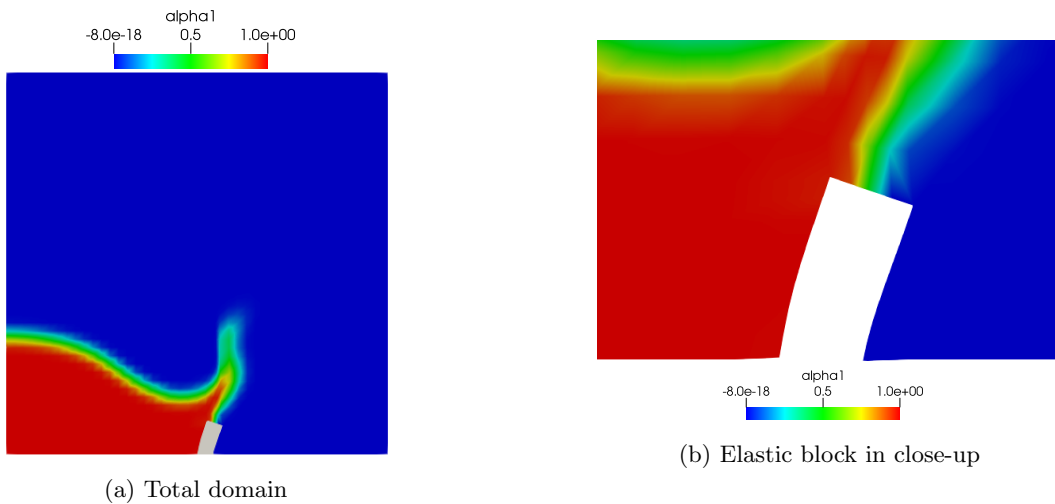
#### • View results Now:

paraFoam&

will open ParaView. Continue with  → Filters in menu bar → Search... . In the window that appears, search for **extract block** (an illustration of the search pane is shown in Figure 2.3c), press Enter, choose **fluid** under Block Indices (Figure 2.8a) and press . It will create a new object in the Pipeline Browser named **ExtractBlock1** (Figure 2.8b). In order to make this object more readable, click on it and then press F2, rename it to **fluid** and press Enter. In order to make the fluid region colored by the amount of water mass fraction, select **fluid** in Pipeline Browser → in the Properties pane, choose **alpha1** under Coloring (Figure 2.3b).

Now select again **flexibleDamBreak.foam** from Pipeline Browser and repeat the Extract Block filter. This time, however, select **solid** under Block Indices. Rename the created object in Pipeline Browser to **solid**. Finally, select **solid** from Pipeline Browser and create a **Warp By Vector** filter with the Vectors set to **pointD** (Figure 2.3a). At the end, Pipeline Browser should look like Figure 2.8c.

Now you can watch the simulation animation by pressing the Play button, (Figure 2.8d). The solution at  $t = 0.195$  s is shown in Figure 2.9.

Figure 2.9: Flexible dam break solution at  $t = 0.195$  s

### 2.3.2 Walk Through

In section 2.2 we learned about the dictionaries necessary for a solid region simulation. In a fluid-solid interaction case, we have both solid and fluid regions. We assume that the reader is already familiar with setting up the dictionaries related to the fluid region. Obviously, the settings for the solid region is the same as before. Here we will have a look at the additional dictionaries existing in an FSI case.

- **Start-time** Looking inside 0/ using the `ls` command shows that there are not files, but two directories, i.e. `solid` and `fluid`. This is the directory structure for multi-region simulations, like FSI, where we have two regions, e.g. fluid and solid. Obviously, our previous discussion in section 2.2.2 regarding the start-time directory is valid for 0/solid directory as well. In the current problem (Figure 2.7), the elastic block is fixed at its bottom and there is no pre-defined traction exerting on its interface with fluid, unless that exerting by the fluid which is calculated during the simulation. Hence the bottom is `blockFixedDisplacement` and the interface is `blockSolidTraction` with zero components for traction (Table 2.2). Front and back sides are `empty` as the problem is 2D.

For the fluid region, there are four dictionaries needed in 0/fluid: `U` for velocity, `pd` for dynamic pressure,  $P - \rho gh$ , where  $P$  is pressure,  $\rho$  is density,  $g$  is gravitational acceleration and  $h$  is height, `alpha1` for liquid volume fraction and `pointMotionU` for fluid mesh.

- **system** The `system` directory contains a `controlDict` and two directories, i.e. `fluid` and `solid` which enclose the solution methodology settings for fluid and solid region respectively. For parallel runs, the domain decomposition method is selected through the `decomposeParDict` included in the `fluid/solid` directory. While not used, a `decomposeParDict` has to be present in `system` itself in order to keep the solver happy [2].

- **constant** Inside `constant`, there are two directories corresponding to the two regions, i.e., `solid` and `fluid`. There are also two dictionaries, i.e., `physicsProperties` and `fsiProperties`.

**fsiProperties** specifies what approach is used to couple the solid and fluid sub-domains [1]. Currently, the following approaches are supported [1]:

- `fixedRelaxation`
- `Aitken`
- `IQNILS`

Each of these methods use a Dirichlet-Neumann coupling approach, where the fluid interface stresses (viscous and pressure) are passed to the solid interface, and the solid interface displacements/velocities are passed to the fluid interface [1]. A sample for `fsiProperties` with its mandatory and optional entries is shown in Listing 2.13.

**fluid** directory contains the different setting for the fluid region;

- `dynamicFvMesh` specifies how the fluid mesh moves during the simulation;
- `fluidProperties` specifies the fluid model;
- `transportProperties` specifies the fluid transport model, the fluid density, viscosity and surface tension;
- `turbulenceProperties` specifies the fluid simulation type, i.e. `laminar`, `RAS` or `LES`.

Listing 2.13: fsiProperties

```

fluidSolidInterface    Aitken;

AitkenCoeffs
{
    solidPatch interface;      % Interface patch name of solid

    fluidPatch interface;      % Interface patch name of fluid

    outerCorrTolerance 1e-6;    % Stop criteria for FSI outer loop

    nOuterCorr 20;             % Max iteration for FSI outer loop

    coupled yes;

    //couplingStartTime 1;      % If coupled is set to no, coupling starts
    //                          from this time.

    //relaxationFactor 0.4;      % Under-relaxation factor for passing the solid
    //                          interface displacement/velocity to the fluid
    //                          interface

    //interfaceTransferMethod directMap; % Method for transferring
    //                          information between the
    //                          interfaces; other possible options
    //                          are GGI and RBF.

    //interpolatorUpdateFrequency 0;

    //interfaceDeformationLimit 0;
}

```

# Chapter 3

## Theory

### 3.1 Mathematical Model

Adopting a Lagrangian approach for the analysis of a solid domain, the so-called *convection* terms drop out and the solid momentum balance reads [9]

$$\frac{\partial^2(\rho \mathbf{u})}{\partial t^2} = \nabla \cdot \boldsymbol{\sigma} + \rho \mathbf{f} \quad (3.1)$$

where  $\rho$  is the density,  $\mathbf{u}$  is the displacement vector,  $\boldsymbol{\sigma}$  is the stress tensor and  $\mathbf{f}$  is the body force per unit mass. For a Hookean solid we have [10]

$$\boldsymbol{\sigma} = \sigma_{ij} = \mu \partial_i u_j + \mu \partial_j u_i + \lambda \delta_{ij} \partial_k u_k \quad (3.2)$$

where  $\lambda$  and  $\mu$  are Lamé's constants,  $u_i$  is displacement vector and  $\delta_{ij}$  is Kronecker delta function. In Eq. (3.2) and hereafter, in the context of Einstein notation,  $\partial_i$  means  $\frac{\partial}{\partial x_i}$ .

### 3.2 Equation Discretization

In integral form, Eq. (3.1) becomes

$$\int_{\Omega} \frac{\partial^2(\rho \mathbf{u})}{\partial t^2} dV = \oint_{\Gamma} \mathbf{n} \cdot \boldsymbol{\sigma} dS + \int_{\Omega} \rho \mathbf{f} dV \quad (3.3)$$

where  $\Omega$  is an arbitrary control volume and  $\Gamma$  is volume surface boundary. The time derivative term can be discretized like [11]

$$\int_{\Omega} \frac{\partial^2(\rho \mathbf{u})}{\partial t^2} dV = \frac{u^n - 2u^{oo} + u^o}{(\Delta t)^2} V_{\Omega} \quad (3.4)$$

where superscripts  $n$ ,  $o$  and  $oo$  are representative of new, old and old-old times, i.e.,  $(t + \Delta t)$ ,  $t$  and  $(t - \Delta t)$  respectively,  $\Delta t$  is the time step and  $V_{\Omega}$  is the volume of the selected CV. Eq. (3.4) shows a first-order discretization in time; second-order schemes can also be used [11]. For the body force term, second-order discretization reads

$$\int_{\Omega} \rho \mathbf{f} dV = (\rho \mathbf{f})_C V_{\Omega} \quad (3.5)$$

where the C subscription means the center of the selected CV. We call the first integral in the RHS of Eq. (3.3) the *traction term* as its integrand is the traction vector. In order to discretize the traction term, some researchers [12, 5] suggest to decompose the traction vector first to normal and tangential

components. From the elementary linear algebra we know that if we decompose an arbitrary vector  $\mathbf{T}$ , the component that is aligned with another arbitrary vector  $\mathbf{n}$  can be calculated by

$$\mathbf{T}_n = \mathbf{n}\mathbf{n} \cdot \mathbf{T} \quad (3.6)$$

Assuming  $\mathbf{T}$  as the traction vector and  $\mathbf{n}$  as the unit surface normal vector, we have

$$\mathbf{T} = \mathbf{n} \cdot \boldsymbol{\sigma} = \mathbf{T}_n + \mathbf{T}_t$$

where

$$\begin{aligned} \mathbf{T}_n &= \mathbf{n}\mathbf{n} \cdot \mathbf{T} \\ &= n_i n_j (n_k \sigma_{kj}) \end{aligned} \quad (3.7)$$

Using Eq. (3.2) for  $\sigma_{kj}$  we have

$$\begin{aligned} \mathbf{T}_n &= n_i n_j n_k (\mu \partial_k u_j + \mu \partial_j u_k + \lambda \delta_{kj} \partial_p u_p) \\ &= \mu n_i n_j n_k \partial_k u_j + \mu n_i n_k n_j \partial_j u_k + \lambda n_i n_j n_j \partial_p u_p \\ &= \mu n_k \partial_k (n_i n_j u_j) + \mu n_j \partial_j (n_i n_k u_k) + \lambda n_i \partial_p u_p \end{aligned} \quad (3.8)$$

Note that  $n_j n_j = 1$ . Let us define  $\nabla_t = \nabla - \mathbf{n}\mathbf{n} \cdot \nabla$  or in index notation

$$(\partial_t)_p = \partial_p - n_p n_m \partial_m \quad (3.9)$$

therefore, the term  $n_i \partial_p u_p$  in Eq. (3.8) can be re-written as

$$n_i \partial_p u_p = n_i (\partial_t)_p u_p + n_p n_i n_m \partial_m u_p \quad (3.10)$$

Substituting Eq. (3.10) in Eq. (3.8) yields

$$\begin{aligned} \mathbf{T}_n &= \mu n_k \partial_k (n_i n_j u_j) + \mu n_j \partial_j (n_i n_k u_k) + \lambda n_i (\partial_t)_p u_p + \lambda n_p n_i n_m \partial_m u_p \\ &= \mu n_k \partial_k (n_i n_j u_j) + \mu n_j \partial_j (n_i n_k u_k) + \lambda n_m \partial_m (n_i n_p u_p) + \lambda n_i (\partial_t)_p u_p \end{aligned} \quad (3.11)$$

Like the traction vector, let us decompose the displacement vector,

$$\begin{aligned} \mathbf{u} &= \mathbf{u}_n + \mathbf{u}_t \\ &= n_p n_q u_q + (u_t)_p \end{aligned} \quad (3.12)$$

Now let us use this decomposition and Eq. (3.9) to rewrite the term  $(\partial_t)_p u_p$  in Eq. (3.11),

$$\begin{aligned} (\partial_t)_p u_p &= (\partial_p - n_p n_m \partial_m) (n_p n_q u_q) + (\partial_t)_p (u_t)_p \\ &= [n_p \partial_p (n_q u_q) - \cancel{n_p n_p n_m \partial_m (n_q u_q)}] + (\partial_t)_p (u_t)_p \end{aligned} \quad (3.13)$$

Now, combining Eq. (3.11) and Eq. (3.13) yields

$$\begin{aligned} \mathbf{T}_n &= \mu n_k \partial_k (n_i n_j u_j) + \mu n_j \partial_j (n_i n_k u_k) + \lambda n_m \partial_m (n_i n_p u_p) + \lambda n_i (\partial_t)_p (u_t)_p \\ &= (2\mu + \lambda) \mathbf{n} \cdot \nabla \mathbf{u}_n + \lambda \mathbf{n} \text{tr}(\nabla_t \mathbf{u}_t) \end{aligned} \quad (3.14)$$

A similar procedure can be followed to yield

$$\mathbf{T}_t = \mu \mathbf{n} \cdot \nabla \mathbf{u}_t + \mu \nabla_t u_n \quad (3.15)$$

where  $u_n = \mathbf{n} \cdot \mathbf{u}$  [5]. Now the traction term in Eq. (3.3) can be re-written as

$$\begin{aligned} \oint_{\Gamma} \mathbf{n} \cdot \boldsymbol{\sigma} dS &= \oint_{\Gamma} \mathbf{T} dS \\ &= \oint_{\Gamma} (\mathbf{T}_n + \mathbf{T}_t) dS \\ &= \underbrace{\oint_{\Gamma} [(2\mu + \lambda)\mathbf{n} \cdot \nabla \mathbf{u}_n + \mu \mathbf{n} \cdot \nabla \mathbf{u}_t] dS}_{\text{Normal derivative terms}} + \underbrace{\oint_{\Gamma} [\lambda \mathbf{n} tr(\nabla_t \mathbf{u}_t) + \mu \nabla_t u_n] dS}_{\text{Tangential derivative terms}} \end{aligned} \quad (3.16)$$

The first integral can be discretized using central differencing and the over-relaxed approach for treatment of non-orthogonality (Appendix A). The so-called non-orthogonal terms obtained from this approach and also the second integral can be discretized implicitly using the finite area method (see Cardiff et al. [5]).

From Eqs. (3.2) and (3.3), the traction term can be also represented as

$$\begin{aligned} \oint_{\Gamma} \mathbf{T} d\Gamma &= \oint_{\Gamma} \mathbf{n} \cdot [\mu \nabla \mathbf{u} + \mu (\nabla \mathbf{u})^T + \lambda \mathbf{I} tr(\nabla \mathbf{u})] d\Gamma \end{aligned} \quad (3.17)$$

where the superscript T stands for the transpose and  $\mathbf{I}$  is the second-order identity matrix. Using the procedure described in obtaining Eq. (3.16), the discretised form of each term in Eq. (3.17) can be written as

$$\oint_{\Gamma} \mu \mathbf{n} \cdot \nabla \mathbf{u} d\Gamma = \oint_{\Gamma} [\mu \mathbf{n} \cdot \nabla \mathbf{u}_n + \mu \mathbf{n} \cdot \nabla \mathbf{u}_t] d\Gamma \quad (3.18)$$

$$\oint_{\Gamma} \mu \mathbf{n} \cdot (\nabla \mathbf{u})^T d\Gamma = \oint_{\Gamma} [\mu \mathbf{n} \cdot \nabla \mathbf{u}_n + \mu \nabla_t u_n] d\Gamma \quad (3.19)$$

$$\oint_{\Gamma} \lambda \mathbf{n} \cdot \mathbf{I} tr(\nabla \mathbf{u}) d\Gamma = \oint_{\Gamma} [\lambda \mathbf{n} \cdot \nabla \mathbf{u}_n + \lambda \mathbf{n} tr(\nabla_t \mathbf{u}_t)] d\Gamma \quad (3.20)$$

The `solids4Foam` implementation discretises the traction term through three steps corresponding to Eqs. (3.18) to (3.20) instead of discretising it at once using Eq. (3.16). The implementation details are given in section 4.2.

### 3.3 Coupled vs Segregated

The critical differences between a coupled method and a segregated one for multi-physics simulations can be summarized as follows:

- (a) Although coupled method performs more calculations per time step comparing with segregated method, it is still totally more time efficient [13, 14]. There are even some cases in which the segregated methods simply fail to retrieve a solution, but coupled methods return an accurate solution [15, 14]. In addition, segregated methods suffer from instability more than coupled methods [14];
- (b) In case of intensive coupling, e.g. between displacement components in fluid-solid interaction problems, segregated methods suffer from slow convergence rates [16];
- (c) In unsteady problems, with coupled method, Courant number criteria is not as tight as it is in segregated method. Therefore the time step can be taken greater whereas the accuracy is the same [14, 17].

## Chapter 4

# Implementation

There are few open-source packages that offer fluid and solid analysis in the same framework [9]. FEM dominates the field of computational solid mechanics (CSM), whereas FVM is the most popular technique in CFD [9]. When dealing with the problems which involve both solids and fluids, like fluid-solid interaction problems, this can be a challenge [9]. Coupling the different packages is an option; however performing these multi-physics analyses in one package offers a number of advantages regarding code development and solver efficiency [9].

Within the OpenFOAM framework, CSM were investigated for the first time by Weller et al. [18] in their formative paper where they presented an analysis for a classical linear elasticity problem, i.e. plate-hole problem [9]. Later developments through this framework were primarily concerned with numerical procedures and no special attention was directed to code design [9].

The exceptions were Tuković et al. [19] and Cardiff et al. [9] who directly addressed the code design. The outcome of these two works is the **solids4Foam** toolbox which is constantly undergoing development and addition of new features (see section 1.2). This tool provides a general code structure that may be easily adopted and extended to related CSM applications. In particular, **solids4Foam** provides the possibility of combining of different solid and fluid models for FSI problems.

### 4.1 solids4Foam Structure and Implementation

The directory structure of **solids4Foam** resembles OpenFOAM, see Figure 4.1. In OpenFOAM, different mathematical models are hard-coded within different solvers. For example, if one wants to simulate a compressible flow, **icoFoam** cannot be used, since the implemented model within this solver is the simplified Navier-Stokes for incompressible flows. Similarly, if one wants to simulate an unsteady turbulent flow, **simpleFoam** cannot be used, since the implemented form of Navier-Stokes equations in this solver is simplified for steady flows. In OpenFOAM, by calling a solver through the terminal, the specific implementation of that solver is executed. In **solids4Foam**, however, there is only one solver named **solids4Foam**. Each solid or fluid model is implemented within a specific **class**. When **solids4Foam** is executed for a specific problem, it reads the model(s) specified by the user and then call for the related class(es); therefore the models are *run-time* selectable instead of being hard-coded within the solver.

Now let us examine the solver source code **solids4Foam.C** shown in Listing 4.1. We will dig into the code line-by-line. Line numbers shown on the left of the Listing 4.1 are consistent with the original file. It is natural that this code depends on many other codes. Based on my experience, however, re-writing those codes here is confusing and ugly. It also will lengthen the document unnecessarily. Instead, we will refer to the necessary files containing those codes when needed, and you can find them easily.

- Lines 38 and 39 are the necessary headers. A header, declares the names, i.e. makes the compiler know what are the meanings of the names that will appear in the rest of the code. We

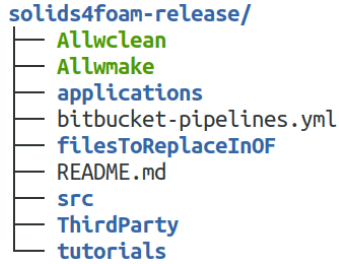
Listing 4.1: solids4Foam.C

```

38 #include "fvCFD.H"
39 #include "physicsModel.H"
40
41 // * * * * *
42
43 int main(int argc, char *argv[])
44 {
45     # include "setRootCase.H"
46     # include "createTime.H"
47     # include "solids4FoamWriteHeader.H"
48
49     // Create the general physics class
50     autoPtr<physicsModel> physics = physicsModel::New(runTime);
51
52     while (runTime.run())
53     {
54         // Update deltaT, if desired, before moving to the next step
55         physics().setDeltaT(runTime);
56
57         runTime++;
58
59         Info<< "Time = " << runTime.timeName() << nl << endl;
60
61         // Solve the mathematical model
62         physics().evolve();
63
64         // Let the physics model know the end of the time-step has been reached
65         physics().updateTotalFields();
66
67         if (runTime.outputTime())
68         {
69             physics().writeFields(runTime);
70         }
71
72         Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
73         << "   ClockTime = " << runTime.elapsedClockTime() << " s"
74         << nl << endl;
75     }
76
77     physics().end();
78
79     Info<< nl << "End" << nl << endl;
80
81     return(0);
82 }

```



Figure 4.1: Directory structure of *solids4Foam*

will specially have a closer look at `physicsModel.H` soon. The role of “`#include`<sup>1</sup>” keyword in these two lines is as if the lines of the two files `fvCFD.H` and `physicsModel.H` were copied here.

- Any C++ program contains one (and only one) `main` function. In line 43 this function begins. To know what syntax is used to call functions in C++, see Stroustrup [20].
- Include directives in lines 45 to 47 have a similar effect as of the lines 38 and 39. More specifically, line 45 checks whether we are at an OpenFOAM case directory. Line 46 creates the `runTime` object, an object of class `Time` which is declared in `foamTime.H`. This object controls time during the simulations [21]; we will see that soon in, for example, line 52. Line 47 prints out a banner including the names of the authors.
- Line 50 creates a `physics` object. It is initialized by the output of the `New` function. Looking into `New` function in `physicsModel.C` reveals that the following sequence of operations happens when it is called [2]:
  - a. Read `physicsProperties` dictionary and look up the keyword `type`;
  - b. If `type` is set to, say, `fluid`, look for a dictionary named `fluidProperties`. For `solid` and `fluidSolidInteraction`, look for `solidProperties` and `fsiProperties` dictionaries respectively.;
  - c. Lastly, create a run-time selectable<sup>2</sup> model<sup>3</sup>.
- Regarding the line 52, `run()` is a member function of class `Time`. In one simple sentence, this function examines whether the present time is smaller than “`endTime - 0.5 * deltaT`” (not `endTime` itself). If so `run()` returns `true`, otherwise, it returns `false`. Therefore, this function can be used to control a loop over the time. To do so, the source code comments (`foamTime.H`) suggest a pseudo code like Listing 4.2. This is actually what the lines 52 to 75 do. Therefore,

Listing 4.2: `runTime` loop pseudo code

```

While endTime is not reached
  Go to the next time step
  Solve
  Write the results

```

If `runTime.run()` (line 52) returns `true`, the simulation continues, otherwise, the process exits from the `while` loop.

- Line 55 consists of the object `physics`, the operator `()` and the method `setDeltaT(runTime)`. Remember that `physics` is an object of class `autoPtr<physicsModel>`. Whenever you construct an object of type `autoPtr<something>`, your object has only one member data: A

<sup>1</sup>Technically known as *include directive*.

<sup>2</sup>The description of the run-time selection mechanism of OpenFOAM is out of the scope of this document; the passionate reader can refer to Gaden [22].

<sup>3</sup>fluid, solid or FSI model depending on the specified `type` in `physicsProperties`.

pointer named `ptr_` which points to an object of class `something`; naturally, the initialization of this member data happens through the constructors, see `autoPtrI.H`. The operator `()` returns a reference to the object pointed to by `ptr_`, see `autoPtrI.H`. `setDeltaT(runTime)` changes `deltaT`, if desired, based on the user settings [2].

- Remember that `runTime` in line 57 is not simply an integer; therefore `++` is an overloaded operator, the definition of which can be found in `foamTime.C`.
- Line 59 prints out the value of the current time.
- In line 62, the mathematical model is solved [2]. There are many fluid and solid models implemented in `solids4Foam`. A coupled solid model will be investigated in section 4.2.
- Line 65 takes effect only in solid mechanical laws. It lets the physics model know the end of the time step is reached [2].
- Lines 67 to 70 check, based on the user settings, whether it is the time to write out the current results.
- Lines 72 to 74 write out some information about the elapsed time [2].
- Line 77 prints out the information about whether the solid model momentum equation is converged or the maximum number of momentum correctors is reached [2].
- Line 79 simply writes out the word `End` when the simulation is ended.

## 4.2 Block-Coupled Solid Model Implementation

In page 13 we discussed the coupled solid model currently included in `solids4Foam`. Here we will show how the `coupledUnsLinGeomLinearElasticSolid` model is implemented. Each solid or fluid model is implemented through a separate class. Here we want to know how the block-coupled solution procedure is implemented within the source code of this model. This is implemented in the `evolve()` function (Listing 4.3) and is called in line 62 of Listing 4.1. Concerning the block-coupled methodology, the limitation of the original OpenFOAM `lduAddressing` is that only face-neighbour cells are treated implicitly. Fully implicit discretization of the solid momentum equation demands for treating the point-neighbour cells implicitly as well [2]. It has been made possible through a new mesh class for solid momentum equation named `solidPolyMesh`. The member data of this class which contains all of the needed information of the mesh is `extendedMesh_`. With this introduction, we start to inspect the source code of `evolve()` function (Listing 4.3) and some other source codes related to it line-by-line. The line numbers shown on the left of the Listings 4.3 to 4.6 are consistent with their original files.

Listing 4.3: `evolve()` function

```

172     bool coupledUnsLinGeomLinearElasticSolid::evolve()
173     {
194         // Create source vector for block matrix
195         vectorField blockB(solutionVec_.size(), vector::zero);
196
197         // Create block system
198         BlockLduMatrix<vector> blockM(extendedMesh_);
199
200         // Grab block diagonal and set it to zero

```

```

201     Field<tensor>& d = blockM.diag().asSquare();
202     d = tensor::zero;
203
204     // Grab linear off-diagonal and set it to zero
205     Field<tensor>& l = blockM.lower().asSquare();
206     Field<tensor>& u = blockM.upper().asSquare();
207     u = tensor::zero;
208     l = tensor::zero;
209
210     // Insert coefficients
211     // Laplacian
212     // non-orthogonal correction is treated implicitly
213     BlockLduMatrix<vector> blockMatLap =
214     BlockFvm::laplacian(extendedMesh_, muf_, D(), blockB);
215
216     // Laplacian transpose == div(mu*gradU.T())
217     BlockLduMatrix<vector> blockMatLapTran =
218     BlockFvm::laplacianTranspose(extendedMesh_, muf_, D(), blockB);
219
220     // Laplacian trace == div(lambda*I*tr(gradU))
221     BlockLduMatrix<vector> blockMatLapTrac =
222     BlockFvm::laplacianTrace(extendedMesh_, lambdaf_, D(), blockB);
223
224     // Add diagonal contributions
225     d += blockMatLap.diag().asSquare();
226     d += blockMatLapTran.diag().asSquare();
227     d += blockMatLapTrac.diag().asSquare();
228
229     // Add off-diagonal contributions
230     u += blockMatLap.upper().asSquare();
231     u += blockMatLapTran.upper().asSquare();
232     u += blockMatLapTrac.upper().asSquare();
233     l += blockMatLap.lower().asSquare();
234     l += blockMatLapTran.lower().asSquare();
235     l += blockMatLapTrac.lower().asSquare();
236     extendedMesh_.insertBoundaryConditions
237     (
238         blockM, blockB, muf_, lambdaf_, D()
239     );
240
241     // Add terms temporal and gravity terms to the block matrix and source
242     extendedMesh_.addFvMatrix
243     (
244         blockM,
245         blockB,
246         rho()*fvm::d2dt2(D()) - rho()*g(),
247         true
248     );
249     solverPerfD =
250     BlockLduSolver<vector>::New
251     (
252         D().name(),
253         blockM,
254         mesh().solutionDict().solver("blockD")
255     )->solve(solutionVec_, blockB);
256     return true;
257 }
258

```

- The source vector of the block system is defined in 195.
- In line 198, the coefficient matrix of the block system, `blockM`, is initialized. This line calls `BlockLduMatrix(const lduMesh& ldu)`, one of the constructors of class `BlockLduMatrix<vector>`. Note that `extendedMesh_` is of type `solidPolyMesh`, while the constructor accepts `lduMesh&`; however, it is fine since the former class inherits from the latter.
- Through lines 201 to 208, the components of `blockM` are set to zero. This is done using three tensors, i.e. `d`, `l` and `u` which are the references to diagonal, lower and upper part of `blockM_`.

- The traction term is discretized through the lines 219 to 228, corresponding to Eqs. (3.18) to (3.20). For the purpose of illustration, `BlockFvm::laplacian` function in line 220 which corresponds to Eq. (3.18) will be investigated. This function is defined in `BlockFvmDivSigma.C` (Listing 4.4). The return value of this function is generated through multiple steps. First of all,

Listing 4.4: `BlockFvm::laplacian` function

```

46 namespace BlockFvm
47 {
69     tmp<BlockLduMatrix<vector> >
70     laplacian
71     (
72         const solidPolyMesh& solidMesh,
73         const surfaceScalarField& muf,
74         GeometricField<vector, fvPatchField, volMesh>& U,
75         Field<vector>& blockB
76     )
77     {
78         return fv::blockLaplacian::New
79         (
80             U.mesh(),
81             U.mesh().schemesDict().laplacianScheme
82             (
83                 "fvmBlockLaplacian(" + U.name() + ')'
84             )
85         )().fvmBlockLaplacian(solidMesh, muf, U, blockB);
86     }
130 } // End namespace BlockFvm

```

New function of class `blockLaplacian`, defined in `blockLaplacianScheme.C` is called. It returns a pointer to a new `blockLaplacian` object [2]. The return type is `tmp<blockLaplacian>`. The `()` in line 85 is an operator defined for `tmp<T>` class in `tmp.C` and returns `T&`; here it returns an object of class `blockLaplacian&`. Finally, the `fvmBlockLaplacian` method of this object is called. In `blockLaplacianScheme.H`, it can be seen that this method is declared `virtual`. The derived class which contains the definition of this method can be found in `pointGaussLsBlockLaplacianScheme.C`. More important lines of `fvmBlockLaplacian` function body are shown in Listing 4.5. The function `insertCoeffsNorm` calculates the contribution

Listing 4.5: `fvmBlockLaplacian` function body

```

241 {
242     tmp<BlockLduMatrix<vector> > tBlockM
243     (
244         new BlockLduMatrix<vector>(solidMesh)
245     );
246     BlockLduMatrix<vector>& blockM = tBlockM();
273     // Insert coeffs due to normal derivative terms
274     insertCoeffsNorm(solidMesh, muf, U, blockB, blockM);
275
276     // Insert coeffs due to tangential derivative terms from the non-orthogonal
277     // corrections
278     if (!U.mesh().orthogonal())
279     {
280         insertCoeffsTang(solidMesh, muf, U, blockB, blockM);
281     }
286     return tBlockM;
287 }

```

of the orthogonal-like terms to the coefficient matrix, `blockM`. Listing 4.6 shows how this contribution is calculated and added to the diagonal, the upper and the lower part of `blockM`. If

Listing 4.6: insertCoeffsNorm function

```

110 // Normal derivative terms
111 const tensor coeff = I*faceMu*faceMagSf*faceDeltaCoeff;
112
113 d[own] -= coeff;
114 d[nei] -= coeff;
115
116 const label varI = fvMap[faceI];
117 u[varI] += coeff;
118 l[varI] += coeff;

```

the mesh is non-orthogonal, `insertCoeffsTang` function acts as well (Listing 4.5) and calculates the contribution of the non-orthogonal-like terms to `blockM`.

- `BlockFvm::laplacianTranspose` and `BlockFvm::laplacianTrace` functions in Lines 224 and 228 of listing 4.3 operate in the same way as `BlockFvm::laplacian` does, except that their `insertCoeffsTang` is called not only to calculate the contribution of the non-orthogonal-like terms, but also for the tangential derivatives appeared in Eqs. (3.19) and (3.20).
- Through the lines 231 to 241, the above mentioned contributions to the block coefficient matrix are added together and the final `blockM` is obtained.
- The contribution of the boundary cells' equations to the system is added in lines 264 to 267. Each boundary condition has its own implementation which is called through this function.
- Finally, the contributions of the temporal and the gravitational terms are added to the system through the lines 270 to 276. The `addFvMatrix` method is implemented in `solidPolyMesh.C`.
- The block system is solved through the lines 294 to 300 using the run-time selected solver.

## Chapter 5

# Adding a Numerical Diffusion Term

### 5.1 Numerical Diffusion

The main bottleneck of the block-coupled methodology in solid mechanics is that the underlying discretization is not always stable. Cardiff et al. [5] have shown that when a coupled FV solid model is adopted, spurious oscillations may appear in the converged solution. As a workaround, they have suggested adding a numerical diffusion term to the force on each face [5]. This has already been implemented in the segregated solid models. For example, the momentum equation for the `linearGeometry` model looks like Listing 5.1. The last term in the momentum equation, i.e., `mechanical().RhieChowCorrection(DD(), gradDD())` is the numerical diffusion term. Here we want to add this term to `coupledUnsLinearGeometryLinearElastic` solid model and create a new model.

Listing 5.1: `linGeomSolid.C`

```
94 // Linear momentum equation total displacement form
95 fvVectorMatrix DDEqn
96 (
97     rho()*fvm::d2dt2(DD())
98     + rho()*fvc::d2dt2(D().oldTime())
99     == fvm::laplacian(impKf_, DD(), "laplacian(DDD,DD)")
100     - fvc::laplacian(impKf_, DD(), "laplacian(DDD,DD)")
101     + fvc::div(sigma(), "div(sigma)")
102     + rho()*g()
103     + mechanical().RhieChowCorrection(DD(), gradDD())
104 );
```

### 5.2 Numerically Modified Block-Coupled Solid Model

The procedure of adding the numerical diffusion term to the coupled solid model is started by going to the directory of the solid models and copying the base model directory with a new name (Listing 5.2).

Listing 5.2: Copying the files

```
$ cd path/to/solids4foam-release/src/solids4FoamModels/solidModels
$ cp coupledUnsLinGeomLinearElasticSolid coupledStabilised
```

In 5.2, we called the new directory `coupledStabilised` which contains the source code of the new model. Now we have to rename the source code and header files accordingly (Listing 5.3).

Listing 5.3: Making the files ready

```
$ cd coupledStabilised
$ mv coupledUnsLinGeomLinearElasticSolid.H coupledStabilised.H
$ mv coupledUnsLinGeomLinearElasticSolid.C coupledStabilised.C
$ sed -i "s/coupledUnsLinGeomLinearElasticSolid/coupledStabilised/g" \
coupledStabilised*
$ sed -i "s/coupledUnsLinearGeometryLinearElastic/coupledStabilised/g" \
coupledStabilised.H
```

The last command in Listing 5.3 specifies the run-time name of the model which we select through `solidProperties` dictionary when we want to use this new model, i.e., `coupledStabilised`.

Adding the numerical diffusion term is easily done by adding the lines shown by “// Added” in Listing 5.4 to `coupledStabilised.C`. Since the numerical diffusion term is calculated explicitly, iterations are needed within each time step [5]. This is why the whole body of the `evolve()` function is placed into a `for` loop. The loop iterates three times, while more iterations can be implemented as well.

Listing 5.4: `coupledStabilised.C`

```
172 bool coupledStabilised::evolve()
173 {
174     for (int i = 0; i<3; i++){ // Added
271         extendedMesh_.addFvMatrix
272         (
273             blockM,
274             blockB,
275             rho()*fvm::d2dt2(D()) - rho()*g()
276             - mechanical().RhieChowCorrection(D(), gradD()), // Added
277             true
278         );
368     } // Added
369     return true;
370 }
```

In order to compile the new model, we need to modify the files within the `Make` directory of the library. To do so, first we go to the appropriate directory by

```
$ cd path/to/solids4foam-release/src/solids4FoamModels/
```

Now the following line should be added to `Make/files`:

```
solidModels/coupledStabilised/coupledStabilised.C
```

It can be added in `Make/files` anywhere before this line:

```
LIB = $(FOAM_USER_LIBBIN)/libsolids4FoamModels
```

Now running the script `Allwmake` compiles the new model (Listing 5.5). Note that issuing this command returns a lot of warnings, but we do not mind as long as there is no error!

Listing 5.5: Running `Allrun`

```
$ bash Allrun
```

## 5.3 A Test Case

Here we examine the “T member” case provided in `solids4Foam` tutorial directory in order to compare the results of the coupled model before and after adding the numerical diffusion term. In order to run this case, you need first to follow Listing 5.6.

Listing 5.6: Preparing the “T member” case

```
$ cp -r solids4FoamTut/solids/linearElasticity/narrowTmember/ $FOAM_RUN
$ cp $FOAM_TUTORIALS/mesh/cfMesh/tetMesh/cutCubeOctree/system/meshDict system/
```

In order to create a quality mesh by `cfmesh`, modify the file `system/meshDict` according to Listing 5.7 (file header is not included).

Listing 5.7: meshDict

```
surfaceFile "mesh.stl";

maxCellSize 0.03;

boundaryCellSize 0.006;

minCellSize 0.006;

localRefinement
{
    "tractionFree.*"
    {
        cellSize 0.003;
    }
}

boundaryLayers
{
    patchBoundaryLayers
    {
        "tractionFree.*"
        {
            thicknessRatio 2;
            maxFirstLayerThickness .001;
            nLayers 5;
        }
    }
}
```

Now issue the following commands:

```
$ blockMesh
$ surfaceMeshTriangulate file.stl
$ surfaceFeatureEdges file.stl mesh.stl
$ tetMesh
```

The mesh will be created and look like Figure 5.1.

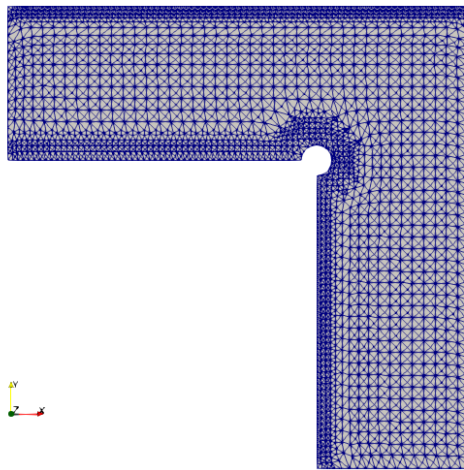


Figure 5.1: T-section mesh.

Inspecting `constant/solidProperties` shows that the `coupledUnsLinearLinearGeometry` model is



used. We make a copy of the whole case with a different name in order to test the modified model as well:

```
$ cp -r narrowTmember narrowTmember-stabilised
```

The narrowTmember case will be run by issuing

```
$ cd $FOAM_RUN/narrowTmember
$ solids4Foam
```

When it is finished, go to the next case by

```
cd $FOAM_RUN/narrowTmember-stabilised
```

Modify solidProperties and fvSchemes according to Listings 5.8 and 5.9 (file headers are not included).

Listing 5.8: solidProperties for narrowTmember-stabilised case

```
solidModel    coupledStabilised;

coupledStabilisedCoeffs
{}
```

Listing 5.9: fvSchemes for narrowTmember-stabilised case

```
d2dt2Schemes
{
    default    steadyState;
}

ddtSchemes
{
    default    steadyState;
}

gradSchemes
{
    default            none;
    grad(D)            leastSquares;
}

divSchemes
{
    default            none;
    fvmDiv(sigma)      banana;//pointGaussLeastSquares;
}

laplacianSchemes
{
    default            none;
    fvmBlockLaplacian(D)    pointGaussLeastSquaresLaplacian;
    fvmBlockLaplacianTranspose(D)    pointGaussLeastSquaresLaplacianTranspose;
    fvmBlockLaplacianTrace(D)    pointGaussLeastSquaresLaplacianTrace;
    laplacian(DDD,DD)    Gauss linear corrected;
    laplacian(DD,D)    Gauss linear corrected;
}

snGradSchemes
{
    default            none;
    snGrad(D)          orthogonal;
}

interpolationSchemes
{
    default            none;
    mu                linear;
    lambda             linear;
}
```

```

interpolate(impK)      linear;
interpolate(grad(DD))  linear;
interpolate(grad(D))   linear;
}

// ***** //

```

Finally, run the case by issuing

```
$ solids4Foam
```

Figure 5.2 shows the comparison between  $\sigma_{yy}$  of the two cases along the horizontal line of  $y = 0.045$  m. The choice of this line is due to that Cardiff et al. [5] reported the presence of the  $\sigma_{yy}$  oscillations at the upper part of the T-member. From Figure 5.2, it is obvious that adding the numerical diffusion

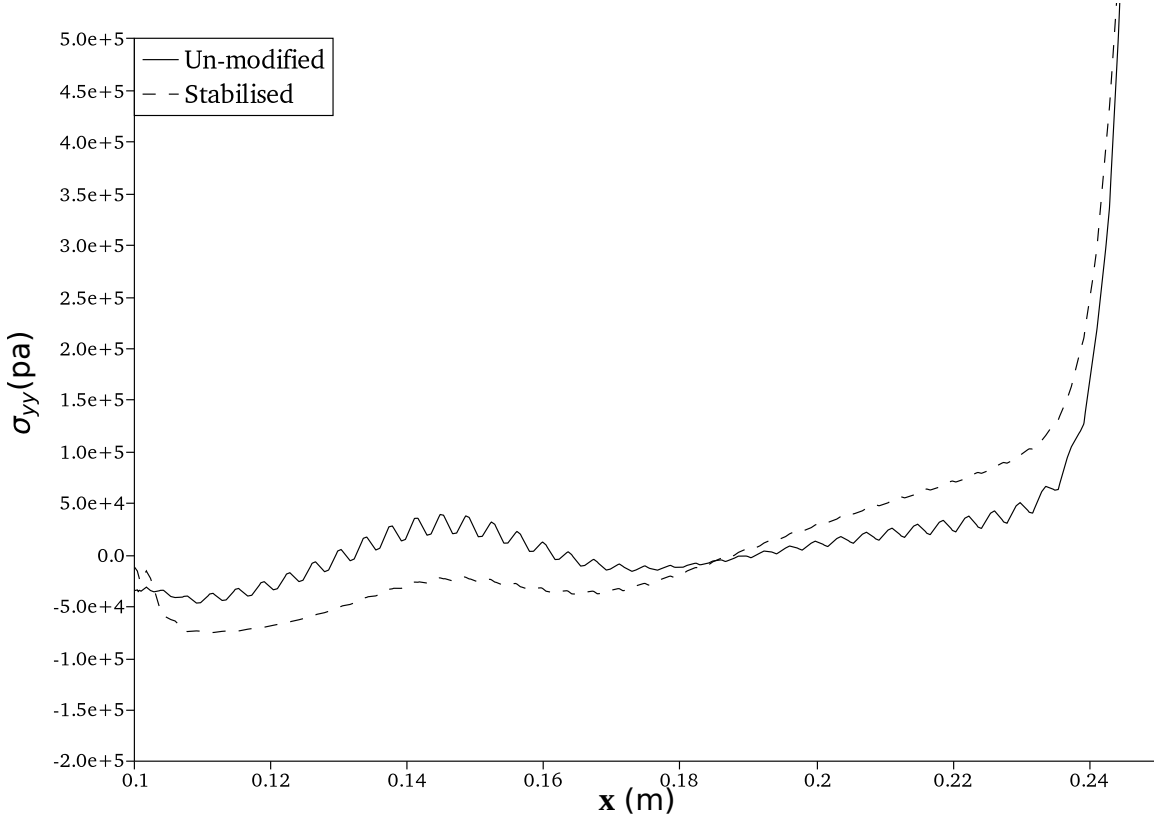


Figure 5.2: Comparing the stabilised and the un-modified results.

term helps model dampen the oscillations significantly. It should be recalled that the term added to the discretised equation (Listing 5.4) is explicit. The required iterations within each time-step may destroy the advantages of block-coupled over segregated. However, this term could be implemented implicitly [5].

# Bibliography

- [1] P. Cardiff, “Solid mechanics and fluid-solid interaction using the solids4foam toolbox.” [https://www.researchgate.net/publication/335126451\\_Solid\\_mechanics\\_and\\_fluid-solid\\_interaction\\_using\\_the\\_solids4foam\\_toolbox](https://www.researchgate.net/publication/335126451_Solid_mechanics_and_fluid-solid_interaction_using_the_solids4foam_toolbox), 07 2019. Accessed: 2020-10-17.
- [2] “solids4Foam repository.” [https://bitbucket.org/philip\\_cardiff/solids4foam-release/src/master/](https://bitbucket.org/philip_cardiff/solids4foam-release/src/master/). Accessed: 2020-10-17.
- [3] P. Cardiff and I. Demirdžić, “Thirty years of the finite volume method for solid mechanics,” *arXiv preprint arXiv:1810.02105*, 2020.
- [4] I. Demirdzic, P. Martinovic, and A. Ivankovic, “Numerical simulation of thermal deformation in welded workpiece,” *Zavarivanje*, vol. 31, no. 5, pp. 209–219, 1988.
- [5] P. Cardiff, Ž. Tuković, H. Jasak, and A. Ivanković, “A block-coupled finite volume methodology for linear elasticity and unstructured meshes,” *Computers & structures*, vol. 175, pp. 100–122, 2016.
- [6] P. Cardiff, Ž. Tuković, H. Jasak, and A. Ivanković, “A block-coupled finite volume methodology for linear elasticity and unstructured meshes,” *Computers & Structures*, vol. 175, pp. 100 – 122, 2016.
- [7] P. Cardiff, A. Karač, P. Jaeger, H. Jasak, J. Nagy, A. Ivanković, and Ž. Tuković, “An open-source finite volume toolbox for solid mechanics and fluid-solid interaction simulations,” 08 2018.
- [8] “OpenFOAM User Guide.” <https://www.openfoam.com/documentation/user-guide/>, 2020. Accessed: 2020-10-20.
- [9] P. Cardiff, A. Karač, P. De Jaeger, H. Jasak, J. Nagy, A. Ivanković, and Ž. Tuković, “An open-source finite volume toolbox for solid mechanics and fluid-solid interaction simulations,” *arXiv preprint arXiv:1808.10736*, 2018.
- [10] M. H. Sadd, *Elasticity: theory, applications, and numerics*. Academic Press, 2014.
- [11] H. Jasak and H. Weller, “Application of the finite volume method and unstructured meshes to linear elasticity,” *International journal for numerical methods in engineering*, vol. 48, no. 2, pp. 267–287, 2000.
- [12] Ž. Tuković, A. Ivanković, and A. Karač, “Finite-volume stress analysis in multi-material linear elastic body,” *International journal for numerical methods in engineering*, vol. 93, no. 4, pp. 400–419, 2013.
- [13] M. Darwish, I. Sraj, and F. Moukalled, “A coupled finite volume solver for the solution of incompressible flows on unstructured grids,” *Journal of Computational Physics*, vol. 228, no. 1, pp. 180–201, 2009.
- [14] F. Pimenta and M. A. Alves, “A coupled finite-volume solver for numerical simulation of electrically-driven flows,” *Computers & Fluids*, vol. 193, p. 104279, 2019.

- [15] G. G. Ferreira, P. L. Lage, L. F. L. Silva, and H. Jasak, “Implementation of an implicit pressure–velocity coupling for the eulerian multi-fluid model,” *Computers & Fluids*, vol. 181, pp. 188–207, 2019.
- [16] I. González, A. Naseri, J. Chiva, J. Rigola, and C. Pérez-Segarra, “An enhanced finite volume based solver for thermoelastic materials in fluid-structure coupled problems,” in *6th European Conference on Computational Mechanics (ECCM 6), 7th European Conference on Computational Fluid Dynamics (ECFD 7)*, Glasgow, UK, vol. 15, 2018.
- [17] M. Riella, R. Kahraman, and G. Tabor, “Fully-coupled pressure-based two-fluid solver for the solution of turbulent fluid-particle systems,” *Computers & Fluids*, vol. 192, p. 104275, 2019.
- [18] H. G. Weller, G. Tabor, H. Jasak, and C. Fureby, “A tensorial approach to computational continuum mechanics using object-oriented techniques,” *Computers in physics*, vol. 12, no. 6, pp. 620–631, 1998.
- [19] Ž. Tuković, A. Karač, P. Cardiff, H. Jasak, and A. Ivanković, “Openfoam finite volume solver for fluid-solid interaction,” *Transactions of FAMENA*, vol. 42, no. 3, pp. 1–31, 2018.
- [20] B. Stroustrup, *Programming: Principles and Practice Using C++ (2nd Edition)*. 2014.
- [21] “foam-extend-4.1 repository.” <https://sourceforge.net/p/foam-extend/foam-extend-4.1/ci/master/tree/>. Accessed: 2020-12-01.
- [22] D. Gaden, “runTimeSelection mechanism.” [http://openfoamwiki.net/index.php/OpenFOAM\\_guide/runTimeSelection\\_mechanism](http://openfoamwiki.net/index.php/OpenFOAM_guide/runTimeSelection_mechanism), 2010. Accessed: 2020-11-30.
- [23] J. H. Ferziger, M. Perić, and R. L. Street, *Computational methods for fluid dynamics*, vol. 3. Springer, 2002.
- [24] F. Moukalled, L. Mangani, M. Darwish, *et al.*, *The finite volume method in computational fluid dynamics*, vol. 113. Springer, 2016.
- [25] H. Jasak, *Error analysis and estimation for the finite volume method with applications to fluid flows*. PhD thesis, Imperial College London (University of London), 1996.
- [26] I. Demirdžić, “On the discretization of the diffusion term in finite-volume continuum mechanics,” *Numerical Heat Transfer, Part B: Fundamentals*, vol. 68, no. 1, pp. 1–10, 2015.

# Study questions

1. What is an OpenFOAM case? What are the dictionaries?
2. To what extent are the different OpenFOAM forks compatible with `solids4Foam`?
3. What is the appropriate ParaView filter for visualizing the solid deformation?
4. What is the appropriate boundary condition for fluid-solid interface?
5. When it comes to solving FSI problems, What is the advantage of using `solids4Foam` instead of coupling between a CFD package for the fluid region and a FEM package for the solid region?
6. How does the `solids4Foam` structure differ from the OpenFOAM structure?
7. What is the main bottleneck of coupled FV solid models?

# Appendix A

## Diffusion Discretization

Finite Volume (FV) discretization of a diffusion term starts from integrating the term over the presumed control volume (CV):

$$\begin{aligned} \oint_{\Gamma} \mu n_i \partial_i u_j dS &= \sum_f \int \mu n_i \partial_i u_j dS \\ &= \sum_f (\mu S_i \partial_i u_j)_f \end{aligned} \quad (\text{A.1})$$

where  $\Gamma$  is the surface surrounding the control volume,  $u_j$  is the dependent variable,  $\mu$  is diffusivity and  $n_i$  is surface unit normal. In Eq. (A.1), mid-point rule is used [23] to approximate the surface integrals. To keep the second-order accuracy of the discretisation practice,  $f$  has to be chosen as the face center of each face of the control volume [24].

Now, we have to evaluate  $\partial_i u_j$  – the gradient – at faces. Jasak [25] presented a sensible approach for this. He decomposed  $S_i$ , the face straddled by two cells P and N as  $S_i = \Delta_i + k_i$ , where  $\Delta_i$  is in direction of the line connecting the cell centers of P and N. Referring to Eq. (A.1), Jasak re-wrote the product under summation as:

$$\begin{aligned} S_i \partial_i u_j &= \Delta_i \partial_i u_j + k_i \partial_i u_j \\ &= \Delta \frac{u_{jN} - u_{jP}}{d} + k_i \partial_i u_j \\ &= \Delta \frac{u_{jN} - u_{jP}}{d} + (S_i - \Delta_i) \partial_i u_j \\ &= S_i (\partial_i u_j)^{\text{exp}} + \underbrace{[\Delta \frac{u_{jN} - u_{jP}}{d} - \Delta_i (\partial_i u_j)^{\text{exp}}]}_{\text{Stabilisation term}} \end{aligned} \quad (\text{A.2})$$

where  $\Delta$  is the magnitude of  $\Delta_i$  and “exp” indicates explicit evaluation, i.e. a linear interpolation between the cell-center gradient values straddling the face. Generally, at a face straddled by cells P and N, linear interpolation is calculated by [25]:

$$(\partial_i u_j)_f = g_N (\partial_i u_j)_N + g_P (\partial_i u_j)_P \quad (\text{A.3})$$

where  $g_N$  and  $g_P$  are weights. The weighted linear interpolation of Eq. (A.3) keeps the second-order accuracy of the method [25] if the weights are calculated based on the distances of the face from the cell centers shared that face. Then, the first term in the bracket in Eq. (A.2) is treated implicitly. While One has many choices for  $k_i$  and  $\Delta_i$ , Jasak investigated three special cases and gave special names to them [26] as shown in Figure A.1. From the figure, it is perceived that for over-relaxed approach,  $\Delta_i$  is calculated like this:

$$\Delta_i = \frac{S_j S_j}{S_j d_j} d_i \quad (\text{A.4})$$

where  $d_i$  is the vector connecting the two cell centers. Jasak [25] showed that over-relaxed approach results in a better stability and a better convergence behavior comparing the other two methods.

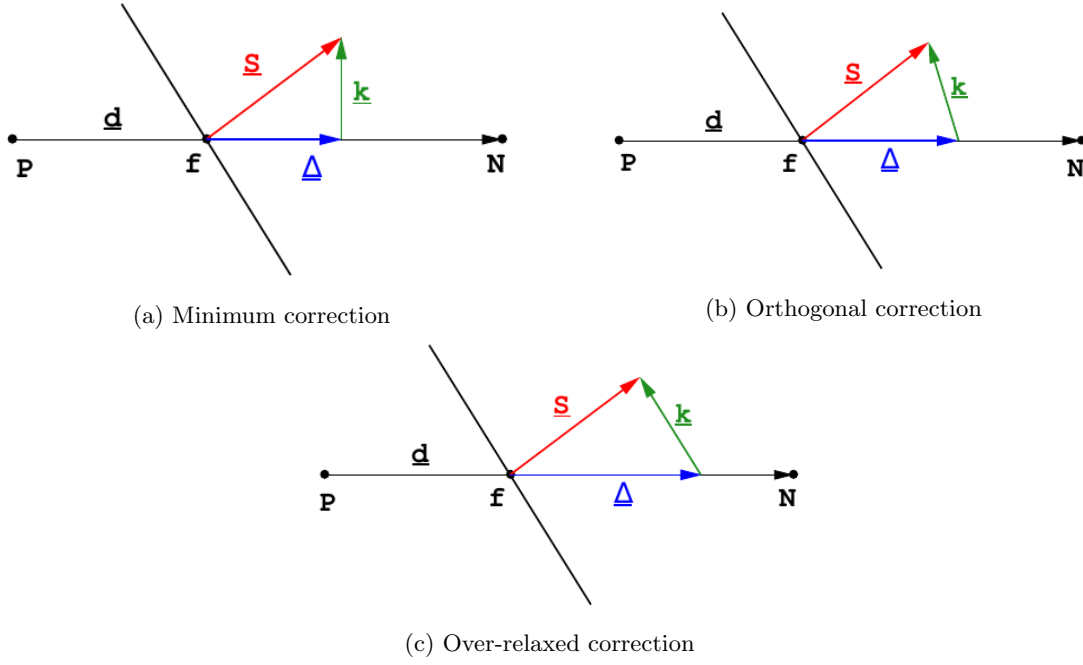


Figure A.1: Area vector decomposition approaches, reproduced from Jasak [25]