Cite as: Alhamwi Alshaar, K.: Adding an Explicit Godunov Type Solid Model to solids4foam Toolbox. In Proceedings of CFD with OpenSource Software, 2024, Edited by Nilsson. H.,
http://dx.doi.org/10.17196/OS_CFD#YEAR_2024

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Adding an Explicit Godunov Type Solid Model to solids4foam Toolbox

Developed for OpenFOAM-v2012
Requires: solids4foam v2.1 toolbox, explicitSolidDynamics toolkit

Author:
Khoder Alhamwi Alshaar
IIT Bomaby
khoder.alshaar@gmail.com
alshaar@iitb.ac.in

Peer reviewed by:
Prof. Jadav Chandra Mandal
Prof. Philip Cardiff
Dr. Saeed Salehi
Björn Jarfors

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 19, 2025

Learning Outcomes

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

The reader will learn:

How to use it:

- How to use the `solids4foam` toolbox to solve fluid-solid interaction problems.

The theory of it:

- The theory behind the `explicitSolidDynamics` toolkit, which includes an explicit Riemann solver for a first-order hyperbolic system of equations governing solid dynamics.
- The difference between the solvers used in `explicitSolidDynamics` and `solids4foam`.
- An overview of the FSI coupling theory as implemented in `solids4foam`.

How it is implemented:

- An overview of the `solids4foam` code structure for implementing the FSI solver.
- The design of the solid solver in the `explicitSolidDynamics` toolkit.

How to modify it:

- How to add new solvers to solid models in `solids4foam`.
- How to integrate the solid solver from `explicitSolidDynamics` as a new subclass of the `solidModel` class.
- Detailed instructions for migrating the `explicitSolidDynamics` solvers and libraries into `solids4foam`.

Prerequisites

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

- Basic understanding of continuum mechanics and fluid-solid interaction.
- Fundamentals of Finite volume method.
- Good experience with C++ language and Object-Oriented Programming (OOP) concepts like class, objects data abstraction, inheritance, etc.
- How to run standard OpenFOAM tutorials.
- Good knowledge of OpenFOAM solvers and libraries and how to modify them.

Contents

Nomenclature

Acronyms

BC	Boundary condition	
CFD	Computational Fluid Dynamics	
FSI	Fluid-solid interaction	
RK	Runge-Kutta	

English symbols

$\boldsymbol{\mathcal{F}}$	Flux vector	
$\boldsymbol{\mathcal{L}}$	Residuals vector in dual-time	
$\boldsymbol{\mathcal{S}}$	Source term vector	
$\boldsymbol{\mathcal{U}}$	Conservative variables vector	
$\boldsymbol{b}$	Body forces	N/m^3
$\boldsymbol{E}$	Cartesian material coordinate basis vector	
$\boldsymbol{F}$	Deformation gradient tensor	
$\boldsymbol{p}$	Linear momentum	$kg/m^2/s$
$\boldsymbol{S}$	Stabilization matrix	
$\boldsymbol{t}$	Traction vector	
$\boldsymbol{u}$	Displacement	m
$\boldsymbol{v}$	Velocity	m/s
$\boldsymbol{R}$	Residuals vector	
$\boldsymbol{X}$	Cartesian material coordinates vector	
ρ	Density	kg/m^3
S	Cell face area	
t	Physical time	s

Greek symbols

Ω	Cell volume	m^3
τ	Pseudo time	s

Superscripts

$\star$	Runge-Kutta stage	
k	Pseudo-time level	
n	Physical-time level	

Subscripts

0	Material coordinate	
F	Fluid domain	
S	Solid domain	
e	Cell number	
f	Cell face	
L	Left side interpolated value	

R	Right side interpolated value
I	Cartesian direction

Chapter 1

Theory

1.1 Introduction

Fluid-solid interaction (FSI) refers to the domain of studies concerned with the interaction between a deformable solid structure and a surrounding or internal fluid, where the behavior of one significantly influences the other. It has gained significant attention in numerical analysis due to the complexity introduced by the coupling between the fluid and solid domains. This coupling presents challenges not typically encountered in standalone fluid flow or solid stress analyses. Nevertheless, the Finite-Volume Method (FVM) has become a solid foundation for the numerical simulation of fluid and solid systems [1]. This has encouraged researchers to explore the potential of FVM in addressing FSI problems, focusing on both fluid and solid domains, as well as the associated coupling procedures.

One notable development in this area is the `solids4foam` toolbox [2] is built on the OpenFOAM software framework. The toolbox features a modular structure, incorporating fluid, solid, and fluid-solid interface coupling algorithms within a unified framework. Currently, most of the solid models in `solids4foam` utilize a second-order displacement-based formulation. Among these, some models, such as the `linGeomTotalDispSolid` solid model, employ an implicit cell-centered finite-volume discretization approach with a segregated algorithm originated from the work of Demirdžić et al. [3]. Others, like the `coupledUnsLinGeomLinearElasticSolid` solid model, are based on an implicit cell-centered finite-volume discretization approach using a block-coupled algorithm [4]. Additionally, there are models like the `vertexCentredLinGeomSolid` solid model, which adopts a vertex-centered approach [5]. For a complete list of available solid models, refer to the solid models web page. This project aims to extend the `solids4foam` toolbox by introducing a novel solid model to facilitate a new approach to FSI analysis.

The proposed solid model is founded on a coupled system of first-order governing equations for linear momentum and the deformation gradient tensor. Trangenstein and Colella initially introduced this formulation [6] and further developed and refined by Lee et al. [7] and Haider et al. [8]. Its primary advantage lies in extending well-established computational fluid dynamics (CFD) solvers to structural analysis. Specifically, the proposed first-order system is hyperbolic [7], enabling the use of Riemann solvers that have demonstrated success in solving hyperbolic systems of equations [9]. To our knowledge, this formulation has not yet been applied to FSI problems.

The code development for this project is based on solids4foam v2.1 using OpenFOAM v2012. On the other hand, the new solid model is derived from the explicitSolidDynamics toolkit [10], originally implemented in OpenFOAM v7 and later ported to OpenFOAM v2012 by P. Cardiff (explicitSolidDynamics OFv2012). Additionally, the original `explicitSolidDynamics` algorithm employs a single-time multi-stage time-marching approach. In this project, we implement a dual-time, multi-stage, time-marching approach to overcome the time-step restrictions commonly encountered in solving FSI problems.

1.2 Fluid-Solid Interaction

The solution procedure for FSI problems is broadly classified into two main approaches based on the coupling strategy between the fluid and solid domains [11]: the *monolithic* approach and the *partitioned* approach. In the monolithic approach, both domains are treated as a single coupled system and solved simultaneously using a unified numerical procedure. In contrast, the partitioned approach solves each domain independently and incorporates a coupling algorithm at the fluid-solid interface.

Another way to see the difference between the monolithic and partitioned approaches lies in the definition of fluid-solid interface boundary conditions [12]. While the monolithic approach defines interface conditions implicitly, the partitioned approach uses them explicitly to communicate information between the fluid and solid solutions.

This section focuses on the *partitioned* approach, precisely the coupling procedure. Readers interested in the detailed numerical treatment of fluid and solid domains can refer to additional resources [2, 1, 11]

1.2.1 Partitioned approach

In the partitioned approach, the FSI problem is solved iteratively by treating the fluid and solid domains independently while ensuring that the interface conditions between them are satisfied. The solution process alternates between solving the fluid and solid sub-problems governed by the chosen coupling algorithm. This algorithm determines whether the coupling is *weak* (weakly-coupled approach) or *strong* (strongly-coupled approach).

The distinction between weak and strong coupling lies in the stability requirements dictated by the physics of the problem. In case of weak interaction between the fluid and solid like in aeroelastic simulations, a stable solution requires one iteration in each time-step [13, 14], see Algorithm 1.

Algorithm 1 Weakly-coupled algorithm

```
for each time-step do
    Solve the solid sub-system
    Update the solid displacement at the fluid-solid interface
    Move the fluid mesh
    Solve the fluid sub-system
    Update force at the fluid-solid interface
end for
```

The weakly-coupled algorithm does not enforce equilibrium at the fluid-solid interface at each time step, which can lead to unstable solutions. Thus, cases with strong interaction between the fluid and solid require a strongly-coupled algorithm, like incompressible flow or a high ratio of fluid/solid densities [15, 16, 14]. In this case, the strongly-coupled algorithm uses a Gauss-Seidel iteration between the fluid and solid solvers to enforce equilibrium at the fluid-solid interface [11]. See Algorithm 2. Moreover, a fixed relaxation procedure can improve the Gauss-Seidel iteration [11, 17] or include Aitken relaxation [11, 18, 17], or IQN–ILS procedure [11, 14].

1.2.2 FSI interface conditions

In addition to the standard boundary conditions defined separately for the fluid and solid domains, coupling boundary conditions must be satisfied at the fluid-solid interface. These conditions ensure the continuity of forces and motion across the interface and consist of the following [11]:

1. Kinematic Condition: The velocity and displacement of the fluid must match those of the solid at the fluid-solid interface:

$$\boldsymbol{v}_{\mathrm{F}} = \boldsymbol{v}_{\mathrm{S}}, \tag{1.1}$$

$$\boldsymbol{u}_{\mathrm{F}} = \boldsymbol{u}_{\mathrm{S}}, \tag{1.2}$$

Algorithm 2 Strongly-coupled algorithm

```
for each time-step do
    while FSI residuals < tolerance do
        Solve the solid sub-system
        Update the solid displacement at the fluid-solid interface
        Move the fluid mesh
        Solve the fluid sub-system
        Update force at the fluid-solid interface
        Update FSI residual
    end while
end for
```

where $\boldsymbol{v}$ and $\boldsymbol{u}$ represent the velocity and displacement, respectively, and the subscripts F and S refer to the fluid and solid domains.

2. Dynamic Condition: The traction forces at the interface must be in equilibrium, ensuring the continuity of stresses across the interface:

$$\boldsymbol{n} \cdot \boldsymbol{\sigma}_{\mathrm{F}} = \boldsymbol{n} \cdot \boldsymbol{\sigma}_{\mathrm{S}}, \tag{1.3}$$

where $\boldsymbol{n}$ is the interface normal vector, and $\boldsymbol{\sigma}$ represents the stress tensor for the fluid and solid domains.

1.3 Explicit Godunov Solid Solver

The primary distinction between the explicit Godunov solver and other solid solvers lies in its representation of the physical behavior of solid material using a first-order system of equations. This system encompasses the conservation equations for linear momentum $\boldsymbol{p}$ and the deformation gradient tensor $\boldsymbol{F}$. In the total lagrangian framework, these equations can be expressed in their differential form as follows:

$$\begin{aligned} \frac{\partial \boldsymbol{p}}{\partial t} - \frac{\partial (\boldsymbol{P}\boldsymbol{E_I})}{\partial \boldsymbol{X_I}} &= \rho_0 \boldsymbol{b}, \\ \frac{\partial \boldsymbol{F}}{\partial t} - \frac{\partial (\frac{1}{\rho_0}\boldsymbol{p} \otimes \boldsymbol{E}_I)}{\partial \boldsymbol{X_I}} &= 0, \end{aligned} \qquad \forall\, I = 1, 2, 3 \tag{1.4}$$

here, $\boldsymbol{P}$ is the first Piola-Kirchhoff stress tensor, ρ_0 is the material density, $\boldsymbol{b}$ represents the body forces, $\boldsymbol{E}_I$ Cartesian material coordinate basis vector in the I^{th} direction, t is the physical-time, and $\boldsymbol{X}$ is the cartesian material coordinates vector. The evolution of $\boldsymbol{F}$ must satisfy some compatibility conditions known as involutions. That is, the deformation gradient $\boldsymbol{F}$ must be curl-free:

$$\nabla \times \boldsymbol{F} = 0. \tag{1.5}$$

The above Eq. (1.4) in 3D can be combined into a single system of first-order hyperbolic equations as

$$\frac{\partial \boldsymbol{\mathcal{U}}}{\partial t} + \frac{\partial \boldsymbol{\mathcal{F}}_I}{\partial \boldsymbol{X}_I} = \boldsymbol{\mathcal{S}} \tag{1.6}$$

where $\boldsymbol{\mathcal{U}}$ represents the vector of conservative variables, $\boldsymbol{\mathcal{F}}_I$ is the flux vector in the I^{th} direction, and $\boldsymbol{\mathcal{S}}$ is the source term vector:

$$\boldsymbol{\mathcal{U}} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ F_{11} \\ F_{12} \\ F_{13} \\ F_{21} \\ F_{22} \\ F_{23} \\ F_{31} \\ F_{32} \\ F_{33} \end{bmatrix}, \quad \boldsymbol{\mathcal{F}_1} = \begin{bmatrix} P_{11} \\ P_{21} \\ P_{31} \\ \frac{1}{\rho}p_1 \\ 0 \\ 0 \\ \frac{1}{\rho}p_2 \\ 0 \\ 0 \\ \frac{1}{\rho}p_3 \\ 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{\mathcal{F}_2} = \begin{bmatrix} P_{12} \\ P_{22} \\ P_{32} \\ 0 \\ \frac{1}{\rho}p_1 \\ 0 \\ 0 \\ \frac{1}{\rho}p_2 \\ 0 \\ 0 \\ \frac{1}{\rho}p_3 \\ 0 \end{bmatrix}, \quad \boldsymbol{\mathcal{F}_3} = \begin{bmatrix} P_{13} \\ P_{23} \\ P_{33} \\ 0 \\ 0 \\ \frac{1}{\rho}p_1 \\ 0 \\ 0 \\ \frac{1}{\rho}p_2 \\ 0 \\ 0 \\ \frac{1}{\rho}p_3 \end{bmatrix}, \quad \boldsymbol{S} = \begin{bmatrix} \rho b_1 \\ \rho b_2 \\ \rho b_1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{1.7}$$

To close the system Eq. (1.6), we should provide a constitutive law equation that describes the solid material of interest, relating the first Piola-Kirchhoff stress tensor to the deformation gradient tensor.

1.3.1 Finite volume discretization

The integral form of the system of Eq.(1.6) can be written in a Total Lagrangian frame as,

$$\int_{\Omega_0} \boldsymbol{\mathcal{U}} \, \mathrm{d}\Omega_0 + \int_{\partial\Omega_0} \boldsymbol{\mathcal{F}}_I N_I \, \mathrm{d}A_0 = \int_{\Omega_0} \boldsymbol{\mathcal{S}} \, \mathrm{d}\Omega_0, \tag{1.8}$$

where Ω_0 is an arbitrary volume, $\partial\Omega_0$ is a surface enclosing the volume, N_I is the I^{th} component of the material unit normal vector, and A_0 is the surface area. The finite volume discretization of the above equation over an M-sided cell e leads to the following semi-discrete form:

$$\frac{\mathrm{d}\boldsymbol{\mathcal{U}}_e}{\mathrm{d}t} = \boldsymbol{R}(\boldsymbol{\mathcal{U}}_e), \tag{1.9}$$

where,

$$\boldsymbol{R}(\boldsymbol{\mathcal{U}}_e) = -\frac{1}{\Omega_e} \sum_{f=1}^{M} (\boldsymbol{\mathcal{F}}_I N_I \, \Delta A)_f + \boldsymbol{\mathcal{S}}_e, \tag{1.10}$$

here, $\boldsymbol{R}(\boldsymbol{\mathcal{U}}_e)$ is called the residual. The original finite volume solution procedure followed by Lee et al. [7], and later extended and implemented in OpenFOAM by Haider et al. [8], includes a single-step, two-stage, second-order Total Variation Diminishing (TVD) Runge-Kutta (RK) scheme [19] for discretizing the time derivative, see Algorithm 3. This scheme discretize Eq. (1.9) as:

$$\begin{aligned} \boldsymbol{\mathcal{U}}^{\star} &= \boldsymbol{\mathcal{U}}^{n} + \Delta t \, \boldsymbol{R}\left(\boldsymbol{\mathcal{U}}^{n}\right), \\ \boldsymbol{\mathcal{U}}^{\star\star} &= \boldsymbol{\mathcal{U}}^{\star} + \Delta t \boldsymbol{R}\left(\boldsymbol{\mathcal{U}}^{\star}\right), \\ \boldsymbol{\mathcal{U}}^{n+1} &= \frac{1}{2}\left(\boldsymbol{\mathcal{U}}^{n} + \boldsymbol{\mathcal{U}}^{\star\star}\right), \end{aligned} \tag{1.11}$$

where the $(\star)$ and $(\star\star)$ superscripts represent the first and second stages of the RK scheme. However, to solve unsteady problems, a two-stage dual-time step Runge-Kutta scheme is developed and presented in the next section.

The flux vector in Eq. (1.10) is evaluated following a Riemann solver based on the contact algorithm [7, 8]. We can write the normal flux vector in Eq. (1.10) as follows:

$$\boldsymbol{\mathcal{F}}_I \boldsymbol{N}_I = \boldsymbol{\mathcal{F}}_N = \begin{bmatrix} \boldsymbol{t} \\ \frac{1}{\rho_0}\boldsymbol{p} \otimes \boldsymbol{N} \end{bmatrix}, \tag{1.12}$$

where, $\boldsymbol{t} = \boldsymbol{PN}$ is the traction vector. Following the contact algorithm [7, 8], the traction vector and linear momentum at the cell interface is evaluated as follows:

$$\boldsymbol{t}_f = \frac{1}{2}(\boldsymbol{t}_L + \boldsymbol{t}_R) + \frac{1}{2}\boldsymbol{S_p}(\boldsymbol{p}_L - \boldsymbol{p}_R), \tag{1.13}$$

$$\boldsymbol{p}_f = \frac{1}{2}(\boldsymbol{p}_L + \boldsymbol{p}_R) + \frac{1}{2}\boldsymbol{S_t}(\boldsymbol{t}_L - \boldsymbol{t}_R), \tag{1.14}$$

where, subscripts L and R refer to the field value interpolated at the left and right side of each cell edge following a second-order reconstruction procedure. Also, $\boldsymbol{S_t}$ and $\boldsymbol{S_p}$ are the stabilization matrices derived following the contact algorithm.

Moreover, the solution procedure consists of a constrained algorithm to satisfy the deformation gradient's curl-free condition Eq. (1.5) and an angular momentum conservation preserving algorithm. Interested readers can find more details in [7, 8].

Algorithm 3 Time update of conservation variables with single-time stepping

Require: $\boldsymbol{\mathcal{U}}_e^n$
Ensure: $\boldsymbol{\mathcal{U}}_e^{n+1}$
1: **while** time $<$ final time **do**
2: Set physical-time step: Δt
3: Store old time variables: $\boldsymbol{\mathcal{U}}^{\mathrm{n}}$
4: **for** each Rk stage **do**
5: Evaluate $\boldsymbol{R}(\boldsymbol{\mathcal{U}})$
6: Solve governing equations: $\boldsymbol{\mathcal{U}} = \boldsymbol{\mathcal{U}} + \Delta\tau\boldsymbol{\mathcal{L}}(\boldsymbol{\mathcal{U}})$
7: **end for**
8: Update conservative variables: $\boldsymbol{\mathcal{U}}^{n+1} = \frac{1}{2}(\boldsymbol{\mathcal{U}}^n + \boldsymbol{\mathcal{U}})$
9: **end while**

1.3.2 Dual-time stepping for the solid solver

Explicit schemes in unsteady simulations often require extremely small time steps to maintain stability, far smaller than those needed for acceptable accuracy [20]. This limitation can lead to an impractically large number of time steps. On the other hand, implicit schemes allow for significantly larger time steps. To address this issue, Jameson [20] proposed the dual-time stepping (DTS) method, which transforms the unsteady problem in physical time into a pseudo-steady problem in a fictitious time domain. By adding a pseudo-time derivative to Eq. (1.6), this transformation gives:

$$\frac{\partial \boldsymbol{\mathcal{U}}_e}{\partial \tau} = -\frac{\partial \boldsymbol{\mathcal{U}}_e}{\partial t} + \boldsymbol{R}(\boldsymbol{\mathcal{U}}_e). \tag{1.15}$$

Whenever $\partial \boldsymbol{\mathcal{U}}_e \,/\, \partial \tau \to 0$, the solution reaches a pseudo-time steady state within each physical time step, and recovers the original governing equation Eq. (1.9). Thus, an iterative scheme to march in pseudo-time is needed to converge to a pseudo-steady state solution up to a specified accuracy at every real-time step. In this work, following the DTS procedure shown by S. Bhat et al. [21, 22], we discretize the physical time derivative using the implicit second-order backward differencing scheme,

$$\frac{\partial \boldsymbol{\mathcal{U}}}{\partial \tau} = -\frac{3\boldsymbol{\mathcal{U}}^{n+1} - 4\boldsymbol{\mathcal{U}}^n + \boldsymbol{\mathcal{U}}^{n-1}}{2\Delta t} + \boldsymbol{R}\left(\boldsymbol{\mathcal{U}}^{\boldsymbol{n+1}}\right), \tag{1.16}$$

where, the superscript n is the physical-time level. On the other hand, we use a two-stage Runge-Kutta method, similar to Eq. (1.11), to discretize the pseudo-time derivative. First, rearranging Eq. (1.16) gives:

$$\frac{\partial \boldsymbol{\mathcal{U}}_e}{\partial \tau} = \boldsymbol{\mathcal{L}}(\boldsymbol{\mathcal{U}}_e), \tag{1.17}$$

where,

$$\boldsymbol{\mathcal{L}}(\boldsymbol{\mathcal{U}}_e) = -\frac{3\boldsymbol{\mathcal{U}}^{n+1} - 4\boldsymbol{\mathcal{U}}^n + \boldsymbol{\mathcal{U}}^{n-1}}{\Delta t} + \boldsymbol{R}\left(\boldsymbol{\mathcal{U}}^{\boldsymbol{n+1}}\right). \tag{1.18}$$

Then, the two-stage RK scheme is evaluated by:

$$\begin{aligned} \boldsymbol{\mathcal{U}}^{\star} &= \boldsymbol{\mathcal{U}}^{k} + \Delta\tau\, \boldsymbol{\mathcal{L}}\left(\boldsymbol{\mathcal{U}}^{k}\right), \\ \boldsymbol{\mathcal{U}}^{\star\star} &= \boldsymbol{\mathcal{U}}^{\star} + \Delta\tau \boldsymbol{\mathcal{L}}\left(\boldsymbol{\mathcal{U}}^{\star}\right), \\ \boldsymbol{\mathcal{U}}^{k+1} &= \frac{1}{2}\left(\boldsymbol{\mathcal{U}}^{k} + \boldsymbol{\mathcal{U}}^{\star\star}\right), \end{aligned} \tag{1.19}$$

where the superscript k represents the pseudo-time level, and $\boldsymbol{\mathcal{L}}(\boldsymbol{\mathcal{U}}_e)$ becomes,

$$\mathcal{L}\left(\boldsymbol{\mathcal{U}}\right) = -\begin{cases} \frac{\boldsymbol{\mathcal{U}}^{k}-\boldsymbol{\mathcal{U}}^{n}}{\Delta t} + \boldsymbol{R}\left(\boldsymbol{\mathcal{U}}\right) & \text{for the frst real time step,} \\ \frac{3\boldsymbol{\mathcal{U}}^{k}-4\boldsymbol{\mathcal{U}}^{n}+\boldsymbol{\mathcal{U}}^{n-1}}{2\Delta t} + \boldsymbol{R}\left(\boldsymbol{\mathcal{U}}\right) & \text{for the second real time step onwards.} \end{cases} \tag{1.20}$$

The value of $\boldsymbol{\mathcal{U}}^{k}$ is $\boldsymbol{\mathcal{U}}^{n}$ at the start of pseudo-time iteration, which eventually becomes $\boldsymbol{\mathcal{U}}^{n+1}$ as $\partial\boldsymbol{\mathcal{U}}_e \,/\, \partial\tau \to 0$. While the physics of the problem restricts the physical-time step [20], the restriction on the pseudo-time step comes from a combined effect of stability requirements of the scheme and the physical-time step [21, 22] as,

$$\Delta\tau = \mathrm{CFL}\left[min\left(\Delta\hat{\tau}, \frac{2}{3}\Delta t\right)\right], \tag{1.21}$$

where, $\Delta\hat{\tau} = h_{\min}/U_{\max}$, with $h_{\min}$ is the is the minimum grid size, and $U_{\max}$ describes the maximum wave speed. For linear elasticity cases, the maximum wave speed is given by:

$$U_{\max} = \sqrt{\frac{\lambda + 2\mu}{\rho_0}}, \tag{1.22}$$

where, λ and μ are Lame's first and second parameter that are expressed in terms of the solid physical properties as follows:

$$\lambda = \frac{\nu E}{(1+\nu)(1-2\nu)}, \quad \mu = \frac{E}{2(1+\nu)}, \tag{1.23}$$

where, E is the Young's modulus, and ν is the Poisson's ratio. This approach effectively decouples the physical-time evolution from the numerical stability constraints, enabling efficient and accurate solutions for unsteady problems. However, the pseudo-time step still needs to be smaller than the physical-time step, which explains the (2/3) factor. The dual-time stepping algorithm is given in Algorithm 4 and schematically shown in Figure 1.1.

Algorithm 4 Time update of conservation variables with dual-time step

Require: $\boldsymbol{\mathcal{U}}_e^n$
Ensure: $\boldsymbol{\mathcal{U}}_e^{n+1}$
while time $<$ final time **do**
  Set physical-time step: Δt
  Store old time variables: $\boldsymbol{\mathcal{U}}^{\text{n}}, \boldsymbol{\mathcal{U}}^{\text{n-1}}$
  while L2-norm $>$ tolerance **do**
    Store previous iteration variables: $\boldsymbol{\mathcal{U}}^k$
    Set pseudo-time step: $\Delta\tau$
    for Runge Kutta stage $= 1$ to 2 **do**
      Evaluate $\boldsymbol{R}(\boldsymbol{\mathcal{U}})$
      Solve governing equations: $\boldsymbol{\mathcal{U}} = \boldsymbol{\mathcal{U}} + \Delta\tau\boldsymbol{\mathcal{L}}(\boldsymbol{\mathcal{U}})$
    end for
    Update conservation variables: $\boldsymbol{\mathcal{U}}^{k+1} = \frac{1}{2}(\boldsymbol{\mathcal{U}}^k + \boldsymbol{\mathcal{U}})$
  end while
  $\boldsymbol{\mathcal{U}}^{n+1} = \boldsymbol{\mathcal{U}}^{k+1}$
end while

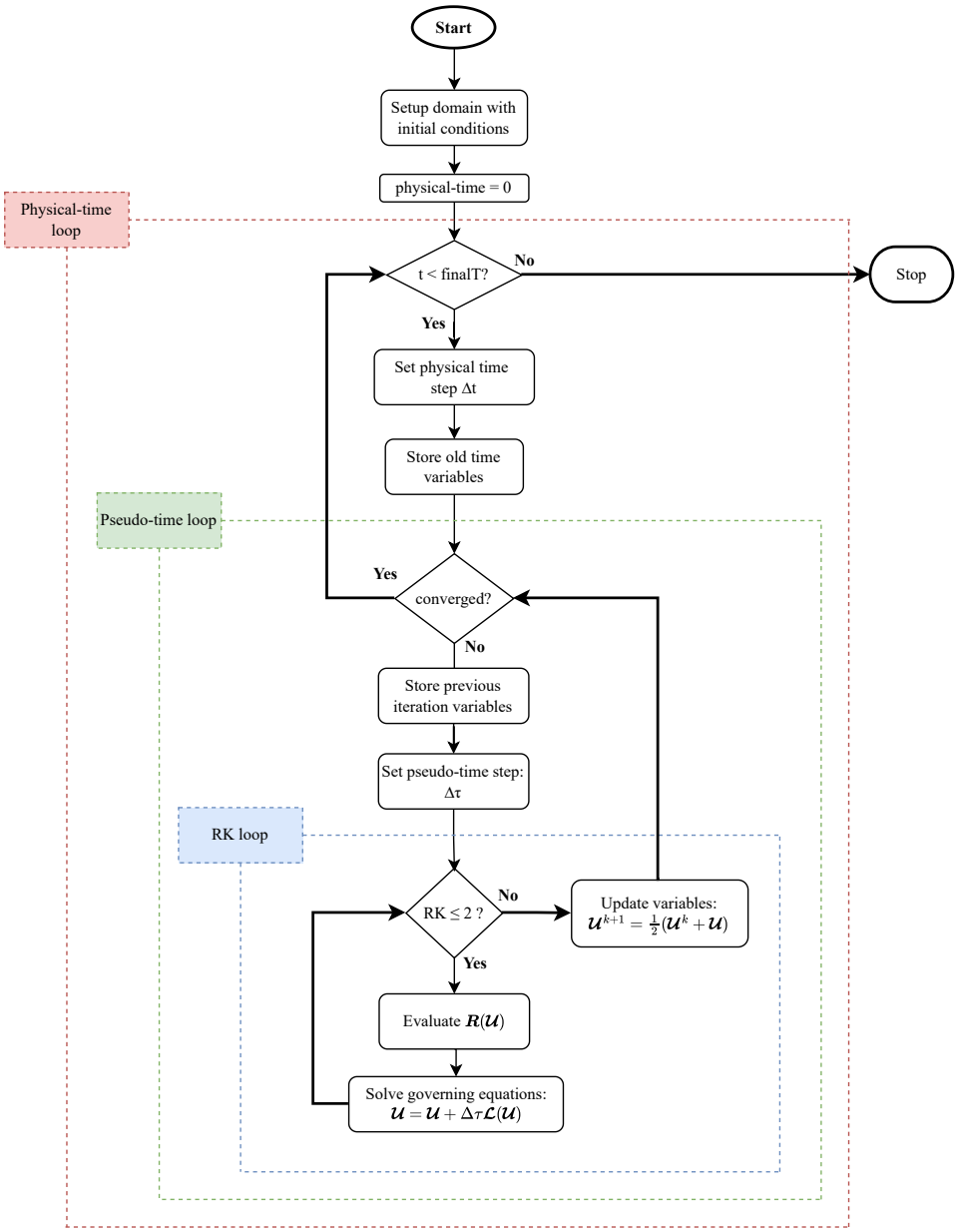

Figure 1.1: Dual-Time stepping algorithm flow chart

Chapter 2

Structure of `explicitSolidDynamics` Toolkit

The `explicitSolidDynamics` toolkit [10] has two solvers, `solidFoam` and `plasticFoam`. The first solves linear-elasticity and hyper-elasticity cases, while the second includes an algorithm to handle cases with plasticity. In this project, we consider only the `solidFoam` solver.

The directory structure of the `explicitSolidDynamics` toolkit follows a standard OpenFOAM organization:

Listing 2.1: Directory of explicitSolidDynamics toolkit

```
explicitSolidDynamics
|-- Allwmake
|-- applications
|   |-- solvers
|       |--solidFoam
|       |--placticFoam
|   |-- utilities
|-- docs
|-- src
|   |-- boundaryConditions
|   |-- mathematics
|   |-- models
|   |-- schemes
|-- tutorials
```

The solver resides in the `solidFoam` directory, which,has the following structure:

Listing 2.2: Directory of solidFoam

```
solidFoam
|-- Make
|-- compile
|-- createFields.H
|-- gEqns.H
|-- readControls.H
|-- riemannSolver.H
|-- solidFoam.C
|-- strongBCs.H
|-- updateVariables.H
```

where `solidFoam.C` has the `main()` function and includes multiple header files, which we will explore in the coming sections.

2.1 Closer Look at solidFoam.C

The file solidFoam.C Listing A.1 contains the main() function of the solver. The initial section includes the necessary libraries:

Listing 2.3: Libraries in solidFoam.C

```
35 #include "fvCFD.H"
36 #include "pointFields.H"
37 #include "operations.H"
38 #include "solidModel.H"
39 #include "mechanics.H"
40 #include "gradientSchemes.H"
41 #include "interpolationSchemes.H"
42 #include "angularMomentum.H"
```

Here, fvCFD.H and pointFields.H are standard OpenFOAM libraries, while the others are toolkit-specific libraries found in:

```
explicitSolidDynamics/src
```

The main() function, as shown in Listing 2.4, begins with standard OpenFOAM include commands, essentially copying the content of the included files. The solution procedure in this solver follows Algorithm 3. It features two main loops: the physical-time loop (line 46) and the RK loop nested within the time loop (line 64).

Listing 2.4: main() function in solidFoam.C

```
46 int main(int argc, char *argv[])
47 {
48     #include "setRootCase.H"
49     #include "createTime.H"
50     #include "createMesh.H"
51     #include "readControls.H"
52     #include "createFields.H"
53
54     while (runTime.run())
55     {
56         mech.time(runTime, deltaT, max(Up_time));
57
58         lm.oldTime();
59         F.oldTime();
60         x.oldTime();
61         xF.oldTime();
62         xN.oldTime();
63
64         forAll(RKstages, stage)
65         {
66             #include "gEqns.H"
67
68             if (RKstages[stage] == 0)
69             {
70                 #include "updateVariables.H"
71             }
72         }
73
74         lm = 0.5*(lm.oldTime() + lm);
75         F = 0.5*(F.oldTime() + F);
76         x = 0.5*(x.oldTime() + x);
77         xF = 0.5*(xF.oldTime() + xF);
78         xN = 0.5*(xN.oldTime() + xN);
79
80         #include "updateVariables.H"
81
82         if (runTime.outputTime())
83         {
84             uN = xN - XN;
```

```
85              uN.write();
86
87              p = model.pressure();
88              p.write();
89          }
90
91          Info<< "Simulation completed = "
92              << (runTime.value()/runTime.endTime().value())*100 << "%" << endl;
93      }
```

Lines 58 - 62 invoke the .oldTime() function for each field, saving the values from the previous time step. Following the algorithm, these values are later used after the RK loop (lines 74 - 78). The remaining code handles data output and terminal information display.

In this project, we make major changes to solidFoam.C and gEqns.H to include the dual-time step algorithm introduced in Section 1.3.2. On the other hand, files like updateVariables will remain unchanged. Thus, we look at gEqns.H next.

2.2 Closer Look at gEqns.H

At line 66 in solidFoam.C, the file gEqns.H is included. This file (Listing 2.5) implements the angular momentum-preserving algorithm, followed by RK stage updates for fields such as cell centers (x), face centers (xF), cell nodes (xN), linear momentum (lm), and deformation gradient (F).

Listing 2.5: solidFoam gEqns.H file

```
1  // Compute right hand sides
2  rhsLm = fvc::surfaceIntegrate(tC*magSf);
3
4  if (angularMomentumConservation == "yes")
5  {
6      rhsAm = fvc::surfaceIntegrate((xF ^ tC)*magSf);
7      am.AMconservation(rhsLm, rhsLm1, rhsAm, stage);
8  }
9
10 // Update coordinates
11 x += deltaT*(lm/rho);
12 xF += deltaT*(lmC/rho);
13 xN += deltaT*(lmN/rho);
14
15 // Update linear momentum
16 lm += deltaT*rhsLm - deltaT*dampingCoeff*lm;
17 // lm += deltaT*rhsLm;// - deltaT*dampingCoeff*lm;
18
19 // Update deformation gradient tensor
20 F += deltaT*fvc::surfaceIntegrate((lmC/rho)*Sf);
```

Chapter 3

Structure of `solids4foam` Toolbox

The `solids4foam` toolbox adopts a modular concept to run solid mechanics and fluid-solid interaction simulations. The structure of this toolbox follows a standard OpenFOAM organization:

Listing 3.1: Directory of `solids4foam` toolbox

```
solids4foam
|-- Allwclean
|-- Allwmake
|-- ThirdParty
|-- applications
|   |-- scripts
|   |-- solvers
|       |--solid4Foam
|   |-- utilities
|   |-- ...
|-- etc
|-- optionalFixes
|-- src
|   |-- RBFMeshMotionSolver
|   |-- abaqusUMATs
|   |-- blockCoupledSolids4FoamTools
|   |-- solids4FoamModels
|   |-- ...
|-- tutorials
|   |-- fluidSolidInteraction
|   |-- fluids
|   |-- solids
|   |-- Alltest
|   |-- ...
```

where the primary solver is located in the `solids4Foam` directory. Additionally, the toolbox is structured around a main abstract class named `physicsModel` with three derived classes: `solidModels`, `fluidModels`, and `fluidSolidInterfaces`. These classes, along with others, are part of a main library, `solids4FoamModels`, which has the following structure:

Listing 3.2: Directory of `solids4FoamModels` library

```
solids4FoamModels
|-- fluidModels
|-- fluidSolidInterfaces
|-- materialModels
|-- physicsModel
|-- solidModels
|-- ...
```

This project focuses primarily on the `solidModels` and `fluidSolidInterfaces` classes. In the coming sections, we shall take a closer look at the crucial components of the toolbox, including the main solver and libraries.

3.1 Closer Look at solids4Foam.C

The main solver is located in solids4Foam.C Listing 3.3. An object physics of physicsModel class at line 43 is created through a runTimeSelction mechanism. The solver has no direct information on the physics of the simulation. Instead, the physics object has the information required. When the physics object is created, the solver decides what type of simulation to run: solid, fluid, or fluid-solid interaction through a runTimeSelection mechanism.

Listing 3.3: solids4Foam updateVariables.H file

```
36 int main(int argc, char *argv[])
37 {
38 #   include "setRootCase.H"
39 #   include "createTime.H"
40 #   include "solids4FoamWriteHeader.H"
41
42     // Create the general physics class
43     autoPtr<physicsModel> physics = physicsModel::New(runTime);
44
45     while (runTime.run())
46     {
47         // Update deltaT, if desired, before moving to the next step
48         physics().setDeltaT(runTime);
49
50         runTime++;
51
52         if (physics().printInfo())
53         {
54             Info<< "Time = " << runTime.timeName() << nl << endl;
55         }
56
57         // Solve the mathematical model
58         physics().evolve();
59
60         // Let the physics model know the end of the time-step has been reached
61         physics().updateTotalFields();
62
63         if (runTime.outputTime())
64         {
65             physics().writeFields(runTime);
66         }
67
68         if (physics().printInfo())
69         {
70             Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
71                 << "  ClockTime = " << runTime.elapsedClockTime() << " s"
72                 << nl << endl;
73         }
74     }
75
76     physics().end();
77
78     Info<< nl << "End" << nl << endl;
79
80     return(0);
81 }
```

The solver's physical-time loop begins with a while loop at line 45 in Listing 3.3. Several functions are invoked within this loop via the physics object. As discussed later, all these functions are either virtual or pure-virtual, with their definitions provided in the subclasses derived from the physicsModel class.

3.2 Closer Look at the solidModel Class

The solidModel class is a base class for specific modeling approaches and discretizations for solid mechanics. As mentioned before, All modeling approaches exclusively consider the second-order displacement-based formulation for which multiple discretization procedures are included, for example,

1. linGeomTotalDispSolid: solves for total displacement field $\boldsymbol{d}$ (D) using a segregated approach.
2. linGeomSolid: solves for the increment of displacement $\Delta\boldsymbol{d}$ (DD) using a segregated approach.
3. ....

More information about these approaches can be found at solidModels web page. The new solid model inherits member data and member functions from this class. However, only some of them, shown in Table 3.1, are useful to the new solid model.

Member data	//- Solid properties dictionary mutable IOdictionary solidProperties_;
	//- Derived solidModel type const word type_;
	//- Point total displacement field pointVectorField pointD_
	//- Point increment of displacement field // pointDD = pointD - pointD.oldTime() pointVectorField pointDD_;
	//- Damping coefficient const dimensionedScalar dampingCoeff_;
	//- Solution standard tolerance const scalar solutionTol_;
	//- Write frequency for residuals information const int infoFrequency_;
	//- Maximum number of momentum correctors const int nCorr_;
Member Functions	//- Return mesh const dynamicFvMesh& mesh() const
	//- Return point mesh const pointMesh& pMesh() const
	//- Return time const Time& runTime() const
	//- Return non-const reference to pointD virtual pointVectorField& pointD()
	//- Return non-const reference to pointDD virtual pointVectorField& pointDD()
	//- Set traction at specified patch virtual void setTraction
	//- Evolve the solid model virtual bool evolve() = 0;

Table 3.1: Member data and Member functions of solidModel class

The most essential function is evolve(), which implements the core mechanics of the new solid model. Another key function is setTraction(). The fluidSolidInterface class called this function to facilitate the transfer of traction at the fluid-solid interface from the fluid side to the solid side via boundary conditions. We will see more details of this function and how to modify it in Section 4.3.

3.3 Closer Look at the fluidSolidInterface Class

The fluidSolidInterface class implements the partitioned coupling algorithm between the fluid and solid systems. In fluid-solid interaction (FSI) problems, the physics pointer in Listing 3.3 points to a fluidSolidInterface object. This object ensures that the evolve() function is selected from the fluidSolidInterface models, which may implement weakly-coupled or strongly-coupled schemes, with or without acceleration.

The interfaceToInterfaceMapping class, found in the fluidSolidInterface directory, manages information transfer at the interface. This virtual base class handles the mapping or interpolation of fields between two interfaces. Its subclasses implement various methods, such as the Direct Map approach and the Radial Basis Function (RBF) method.

For simplicity, consider the weakCouplingInterface model. The evolve() function in this model is defined as follows:

Listing 3.4: evolve() function in weakCouplingInterface model

```
bool weakCouplingInterface::evolve()
{
    initializeFields();

    updateInterpolatorAndGlobalPatches();

    solid().evolve();

    updateWeakDisplacement();

    moveFluidMesh();

    fluid().evolve();

    updateForce();

    solid().updateTotalFields();

    return 0;
}
```

This function calls the evolve() function for both the solid and fluid domains. Between these calls, it invokes updateWeakDisplacement() and updateForce(). These functions are essential for transferring information between the fluid and solid at the interface. We examine the implementations of these functions in the following sections.

3.3.1 updateForce() function

This function transfers the fluid traction forces at the fluid-solid interface by setting the traction on solid interfaces Listing 3.5.

Listing 3.5: updateForce() function

```
void Foam::fluidSolidInterface::updateForce()
{
    ...
    for (label interfaceI = 0; interfaceI < nGlobalPatches_; interfaceI++)
    {
        ...
        // Transfer the field frm the fluid interface to the solid interface
        interfaceToInterfaceList()[interfaceI].transferFacesZoneToZone
        (
            fluidZone,                // from zone
            solidZone,                // to zone
            fluidZoneTotalTraction,   // from field
            solidZoneTotalTraction    // to field
        );

```

```
        // Flip traction sign after transferring from fluid to solid
        solidZoneTotalTraction = -solidZoneTotalTraction;

        // Set traction on solid
        if (coupled())
        {
            solid().setTraction
            (
                interfaceI,
                solidPatchIndices()[interfaceI],
                solidZoneTotalTraction
            );
        }
    }
}
```

First, it transfers the forces between the patches through the `transferFacesZoneToZone` function, which is part of the `interfaceToInterfaceMapping` class. Then, it forwards the traction to the solid boundary condition through `setTraction` function in the `SolidModel` class, see Section 3.2.

3.3.2 `updateWeakDisplacement()` function

This function transfers the solid mesh displacement at the fluid-solid interface to the fluid mesh Listing 3.6.

Listing 3.6: `updateWeakDisplacement()` function

```
void weakCouplingInterface::updateWeakDisplacement()
{
    // Update the residual
    updateResidual();

    forAll(fluid().globalPatches(), interfaceI)
    {
        fluidZonesPointsDisplsPrev()[interfaceI] =
            fluidZonesPointsDispls()[interfaceI];

        // No under-relaxation
        fluidZonesPointsDispls()[interfaceI] += residuals()[interfaceI];
    }
    ...
}
```

First, it updates the point-displacement `residuals()` through `updateResidual`. Then, it adds it to the fluid interface.

Chapter 4

Implementing `explicitGodunovCC` Solid Model in `solids4foam`

This chapter explains the steps required to add a new solid solver to the solids4Foam toolbox. New solvers should be Wrapped in a sub-class of the desired physics model to accomplish this object, i.e., solid, fluid, fluid-solid interface. To install and compile `solids4foam` toolbox, follow the instructions at Installing solids4foam from Source, which reads [1],

```
cd $HOME
git clone --branch v2.1 https://github.com/solids4foam/solids4foam.git
cd solids4foam && ./Allwmake -j && cd tutorials && ./Alltest
```

To install `explicitSoliddynamics` toolkit with OpenFOAM v2012, you can run the following command [2] :

```
git clone --branch openfoam-v2012 https://github.com/philipcardiff/\
explicitSolidDynamics.git
```

The accompanying files of this project are available on the course proceeding web page OSCFD course along with this report.

4.1 Adding a New `solidModel` Sub-class

We must create a new subclass of the `solidModel` class to implement a new solid solver. We start by creating an empty `testModelSolid`. First, we copy an existing solid model from the `solidModel` directory, such as `linGeomTotalDispSolid`. Then we do several modifications following these steps:

```
cd   ~/solids4foam/src/solids4FoamModels/solidModels
mkdir testModelSolid
cp  -r linGeomTotalDispSolid/. testModelSolid
cd testModelSolid/
mv linGeomTotalDispSolid.C testModelSolid.C
mv linGeomTotalDispSolid.H testModelSolid.H
sed -i 's/linGeomTotalDispSolid/testModelSolid/g' testModelSolid.H
sed -i 's/linearGeometryTotalDisplacement/testModel/g' testModelSolid.H
sed -i 's/linGeomTotalDispSolid/testModelSolid/g' testModelSolid.C
```

[1] We assume that the solids4foam toolbox is installed in the `HOME` directory in future command lines.

[2] Due to page size constraints, the symbol "\" is used to split a Linux command across two lines. When copying any such command, ensure to copy both lines together. The Linux shell reads the symbol "\" to escape the next character, meaning it will ignore the newline character when executing the command.

To compile the new solid model, we include its source file in the `solids4FoamModels` library by adding the following line:

```
solidModels/testModelSolid/testModelSolid.C
```

to the `solids4FoamModels/Make/files.openfoam` file, immediately after the line that references the `solidModel` class source file that reads:

```
solidModels/solidModel/solidModel.C
```

Now, we compile the `solids4FoamModels` library. First, we ensure that the required OpenFOAM environment, OpenFOAM v2012, is sourced before proceeding. Then, we execute the following commands:

```
cd ~/solids4foam/src/solids4FoamModels
./Allwmake
```

This will compile the updated library, incorporating the new `testModelSolid` class. Finally, we test the new solid model against one of the tutorials. For example, run the following commands:

```
run
cp -r ~/solids4foam/tutorials/solids/linearElasticity/\
cantilever2d/segregatedCantilever2d/ .

cd segregatedCantilever2d
```

We set the `solidModel` to `testModel` in the `solidProperties` dictionary and add to the same file the `solidModel` sub-dictionary with default values:

```
testModelCoeffs
{
}
```

Follow the instructions on how to run a solid tutorial using solids4foam on Solid Mechanics Tutorials, and verify the results.

4.1.1 Base `solidModel` sub-class

After verifying the results, we clean up the `testModelSolid` class by retaining only the essential components necessary to construct the class and implement the virtual functions. In the class header file, we do the following steps:

1. Remove the following includes:

```
#include "volFields.H"
#include "surfaceFields.H"
#include "pointFields.H"
#include "uniformDimensionedFields.
```

2. Remove all the Privet member data.

3. Remove the flowing privet member functions:

```
void predict();
```

4. Keep the rest of the member functions as they have a virtual prefix.

Similarly, In the class source file, we do the following steps:

1. Remove the privet member function.
2. Remove all member data definition in the constructor keeping only the following line:

```
solidModel(typeName, runTime, region)
```

3. Remove all the lines in the `evolve()` function.
4. Keep the `tractionBoundarySnGrad`, However, remove all the lines in the class implementation.

Following these steps, the resulting header and source files for `testModelSolid` are presented in Listings A.2 and A.3, respectively.

Once the cleanup is complete, we ensure the solver operates without errors by testing it with a suitable tutorial case.

In the header file in Listing A.2, note the declaration of the `evolve()` function. The definition of this function, provided in Listing A.3, contains the core implementation of the solid solver, which we will elaborate upon in the next section.

4.2 Wrapping `explicitSolidDynamics` into a Class

After preparing the base subclass, the next step is to add the components of `explicitSolidDynamics`. we begin by renaming `testModelSolid` to `explicitGodunovCCSolid`:

```
cd   ~/solids4foam/src/solids4FoamModels/solidModels/
mv testModelSolid/ explicitGodunovCCSolid
cd explicitGodunovCCSolid
mv testModelSolid.C explicitGodunovCCSolid.C
mv testModelSolid.H explicitGodunovCCSolid.H
sed -i 's/testModelSolid/explicitGodunovCCSolid/g' explicitGodunovCCSolid.H
sed -i 's/testModel/explicitGodunovCC/g' explicitGodunovCCSolid.H
sed -i 's/testModelSolid/explicitGodunovCCSolid/g' explicitGodunovCCSolid.C
```

The next step is to include the source file `explicitGodunovCCSolid.C` for compilation with the library. We update the `files.openfoam` file using the following command:

```
sed -i 's/testModel/explicitGodunovCC/g'  \
                ~/solids4foam/src/solids4FoamModels/Make/files.openfoam
```

Finally, we compile the library and run a tutorial to verify the implementation.

4.2.1 Adding `explicitSolidDynamics` libraries

Before transferring the `solidFoam` solver components into the class, it is crucial to ensure that all dependent libraries are accessible to the `explicitGodunovCCSolid` class. The `explicitSolidDynamics` toolkit consists of four libraries: `boundaryConditions`, `mathematics`, `models`, and `schemes`.

To include these libraries, we start by copying the contents of the `src` directory into the directory of `explicitGodunovCCSolid` using the following commands:

```
cd ~/solids4foam/src/solids4FoamModels/solidModels/explicitGodunovCCSolid
cp -r  ~/explicitSolidDynamics/src/.  .
```

After that, we delete all the `Make` folders and `compil` scripts in each directory:

```
rm -r boundaryConditions/Make
rm -r boundaryConditions/compile
rm -r mathematics/Make
rm -r mathematics/compile
rm -r models/Make
rm -r models/compile
rm -r models/plasticityModel
rm -r schemes/Make
rm -r schemes/compile
```

Due to the fact that `solids4foam` already has a class called `solidModel`, we should change the class `solidModel` copied from the `explicitSolidDynamics` libraries to `solidMaterialModel`:

```
cd  ~/solids4foam/src/solids4FoamModels/solidModels/\
explicitGodunovCCSolid/models

mv solidModel/ solidMaterialModel
cd solidMaterialModel
mv solidModel.C solidMaterialModel.C
mv solidModel.H solidMaterialModel.H
sed -i 's/solidModel/solidMaterialModel/g' solidMaterialModel.H
sed -i 's/solidModel/solidMaterialModel/g' solidMaterialModel.C
```

The next step is to add all source files of these libraries to the main library of `solids4FoamModels`. This is done by copying the following lines to `solids4FoamModels/Make/files.openfoam`:

```
explicitGodunovCCSolid = solidModels/explicitGodunovCCSolid
$(explicitGodunovCCSolid)/mathematics/operations/operations.C
$(explicitGodunovCCSolid)/mathematics/mechanics/mechanics.C
$(explicitGodunovCCSolid)/boundaryConditions/moving/\
movingDisplacementLinearMomentumFvPatchVectorField/\
movingDisplacementLinearMomentumFvPatchVectorField.C
$(explicitGodunovCCSolid)/boundaryConditions/moving/\
movingDisplacementNodalLinearMomentumPointPatchVectorField/\
movingDisplacementNodalLinearMomentumPointPatchVectorField.C
$(explicitGodunovCCSolid)/boundaryConditions/moving/\
movingDisplacementTractionFvPatchVectorField/\
movingDisplacementTractionFvPatchVectorField.C
$(explicitGodunovCCSolid)/boundaryConditions/moving/\
movingTractionFvPatchVectorField/\
movingTractionFvPatchVectorField.C
$(explicitGodunovCCSolid)/boundaryConditions/traction/\
tractionLinearMomentumFvPatchVectorField/\
tractionLinearMomentumFvPatchVectorField.C
$(explicitGodunovCCSolid)/boundaryConditions/traction/\
tractionTractionFvPatchVectorField/\
tractionTractionFvPatchVectorField.C
$(explicitGodunovCCSolid)/boundaryConditions/symmetric/\
symmetricLinearMomentumFvPatchVectorField/\
symmetricLinearMomentumFvPatchVectorField.C
$(explicitGodunovCCSolid)/boundaryConditions/symmetric/\
symmetricTractionFvPatchVectorField/\
symmetricTractionFvPatchVectorField.C
$(explicitGodunovCCSolid)/models/solidMaterialModel/solidMaterialModel.C
$(explicitGodunovCCSolid)/schemes/gradientSchemes/gradientSchemes.C
$(explicitGodunovCCSolid)/schemes/interpolationSchemes/interpolationSchemes.C
```

```
$(explicitGodunovCCSolid)/schemes/angularMomentum/angularMomentum.C
$(explicitGodunovCCSolid)/explicitGodunovCCSolid.C
```

Also, we delete the previous line we added earlier:

```
solidModels/explicitGodunovCCSolid/explicitGodunovCCSolid.C
```

Finally, we include the libraries in `explicitGodunovCCSolid.H`

Listing 4.1: Include libraries `explicitGodunovCCSolid.H`

```
#include "operations.H"
#include "solidMaterialModel.H"
#include "mechanics.H"
#include "gradientSchemes.H"
#include "interpolationSchemes.H"
#include "angularMomentum.H"
```

Now, we compile the code and check for errors:

```
cd ~/solids4foam/src/solids4FoamModels
./Allwmake
```

4.2.2 Adding member data

At this stage, the declarations and definitions of fields and class objects from `createFields.H` in the `solidFoam` directory can be incorporated into the `explicitGodunovCCSolid` class. The key difference is that declarations are added to the class header file, while definitions are placed in the constructor within the source file. For instance, we declare the following member data and define them as follows:

In the header file, we add declarations for the member data within the private section of the class:

Listing 4.2: declarations of member data in `explicitGodunovCCSolid.H`

```
    //Creating linear momentum fields
    volVectorField lm_;
```

In the source file, we define the fields and initialize them appropriately in the class constructor:

Listing 4.3: definition of member data in `explicitGodunovCCSolid.H`

```
    lm_
    (
        IOobject
        (
            "lm",
            runTime.timeName(),
            mesh(),
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        mesh(),
        dimensionedVector("lm", dimensionSet(1,-2,-1,0,0,0,0), vector::zero)
    ),
```

Notice that all class member data are suffixed with an underscore, following a convention in `cpp`. Similarly, we add the remaining member data to the class, as detailed in Listing A.4 and A.5. At this stage, we add only the member data definitions and declarations till line `479` in Listing A.5.

Note: The definition of some member data in `explicitSolidDynamics` relies on specific dictionaries. We replace these dependencies with entries defined according to the `solids4foam` toolbox. For instance, we replace `dampingCoeff_` with `dampingCoeff()`, a function already defined in `solidModel` class. Similarly, we redefine the angular momentum conservation switch as,

```
    angularMomentumConservation_
    (
        solidModelDict().lookupOrAddDefault<word>("angularMomentumConservation", "no")
    ),
```

Here, `solidModelDict()` retrieves the sub-dictionary, `explicitGodunovCCCoeffs` in this case, from the `solidProperties` dictionary. Similar changes are applied to the `solidMaterialModel` class in `explicitGodunovCC` to align with the dictionary structure used in `solids4foam`. In the source file, we change the definition of certain member data by deleting the following lines in the class constructor:

```
    model_(dict.lookup("solidMaterialModel")),

    rho_(dict.lookup("rho")),
    E_(dict.lookup("E")),
    nu_(dict.lookup("nu")),
    mu_(E_/(2.0*(1.0 + nu_))),
    lambda_(nu_*E_/((1.0 + nu_)*(1.0 - 2.0*nu_))),
    kappa_(lambda_ + (2.0/3.0)*mu_),

    Up_(sqrt((lambda_+2.0*mu_)/rho_)),
    Us_(sqrt(mu_/rho_))
```

and changing their definition as follows:

```
    rho_("rho",dimDensity , 0.0),
    E_("E", dimPressure, 0.0),
    nu_("nu", dimless, 0.0),
    mu_("mu", dimPressure, 0.0),
    lambda_("lambda", dimPressure, 0.0),
    kappa_("kappa_", dimPressure, 0.0),
    Up_("Up_", dimVelocity, 0.0),
    Us_("Us_", dimVelocity, 0.0)
```

and adding the following lines to the body of the constructor:

```
    // Read the mechanical laws
    const PtrList<entry> lawEntries(dict.lookup("mechanical"));

    const dictionary& materialDict = lawEntries[0].dict();

    // Read model rho, E, and nu from the material dictionary

    const word model
    (
        materialDict.lookup("type")
    );
    model_ = model;

    rho_ = dimensionedScalar(materialDict.lookup("rho"));
    E_ = dimensionedScalar(materialDict.lookup("E"));
    nu_ = dimensionedScalar(materialDict.lookup("nu"));
    mu_ =(E_/(2.0*(1.0 + nu_)));

    lambda_ = (nu_*E_/((1.0 + nu_)*(1.0 - 2.0*nu_)));

    kappa_ =lambda_ + (2.0/3.0)*mu_;

    Up_ = sqrt((lambda_+2.0*mu_)/rho_);
    Us_ = sqrt(mu_/rho_);


    correct();
```

Note: `solids4foam` already includes a class for defining solid material laws, `mechanicalLaws`. However, it does not provide direct access to its member data. The challenge is to replace the

dependency on `solidMaterialModel` with minimal modifications to the original `mechanicalLaw` class, which could be done in the future.

We perform a similar modification to redefine the density `rho` in `angularMomentum.C` file. In the class constructor, we change the following line:

```
        rho_(dict.lookup("rho"))
```

to,

```
        rho_("rho",dimDensity , 0.0)
```

and we add the following lines to the body of the constructor:

```
    const PtrList<entry> lawEntries(dict.lookup("mechanical"));
    const dictionary& materialDict = lawEntries[0].dict();

    rho_ = dimensionedScalar(materialDict.lookup("rho"));
```

Finally, skipping only `readControls.H`, we copy all the header files (`.H`) present in `solidFoam` directory:

```
cd  $HOME/explicitSolidDynamics/applications/solvers/solidFoam

cp   gEqns.H riemannSolver.H strongBCs.H updateVariables.H \
$HOME/solids4foam/src/solids4FoamModels/solidModels/explicitGodunovCCSolid
```

Some definitions included in `readControls.H` are moved to the solver class constructor accordingly. All these files can be found on the course page, as mentioned earlier, and can be copied directly to save time. However, `gEqns.H` is changed further to adapt the dual-time stepping algorithm in Section4.3.2.

Note, when we compile the code, we get a warning related to the definition of symmetric patches for the strong boundary conditions for nodal linear momentum. This warning can be avoided by removing the definitions from the class constructor and adding them to `strongBCs.H` file where it is needed, as:

```
// Boundary patches
const polyBoundaryMesh& bm = mesh().boundaryMesh();
const label& symmetricPatchID_ = bm.findPatchID("symmetric");
const label& symmetricXpatchID_ = bm.findPatchID("symmetricX");
const label& symmetricYpatchID_ = bm.findPatchID("symmetricY");
const label& symmetricZpatchID_ = bm.findPatchID("symmetricZ");
```

Now, we compile the code and check for errors:

```
cd ~/solids4foam/src/solids4FoamModels
./Allwmake
```

4.2.3 Definition of `evolve()` function

Recall that the `main()` function in `solids4Foam.C` includes a `while(runTime.run())` loop. This loop encompasses the solver's main operations, which need to be incorporated into the `evolve()` function. Therefore, the content of the `while(runTime.run())` loop in the original `main()` function from `solidFoam.C` will now form the body of the `evolve()` function.

With all the required member data in the `explicitGodunovCCSolid` class, the main solver logic can now be implemented in the `evolve()` function. By copying the content of the `while()`loop in `solid4Foam.C` to `evolve()` function, the function should look as follows:

Listing 4.4: Components of `solidFoam.C` in `evolve()`

```
bool explicitGodunovCCSolid::evolve()
{
        mech_.time(runTime_, deltaT_, max(Up_time_));

        forAll(RKstages_, stage)
        {
            #include "gEqns.H"

            if (RKstages_[stage] == 0)
            {
                #include "updateVariables.H"
            }
        }

        x_  = 0.5*(x_.oldTime() + x_);
        xF_ = 0.5*(xF_.oldTime() + xF_);
        xN_ = 0.5*(xN_.oldTime() + xN_);
        lm_ = 0.5*(lm_.oldTime() + lm_);
        F_  = 0.5*(F_.oldTime() + F_);

        #include "updateVariables.H"

        if (runTime_.outputTime())
        {
            uN_ = xN_ - XN_;
            uN_.write();
            p_ = model_.pressure();
            p_.write();
        }
    return true;
}
```

This structure ensures that the solver loop in `solidFoam.C` is properly transferred and integrated into the class-based implementation. Further refinements can be made to incorporate specific solver behaviors or configurations.

Note, we have moved the following lines form `while()` loop in the original solver to the class constructor:

```
    x_.oldTime();
    xF_.oldTime();
    lm_.oldTime();
    F_.oldTime();
    xN_.oldTime();
```

Now, compile the code and check for errors:

```
cd ~/solids4foam/src/solids4FoamModels
./Allwmake
```

At this stage, the `explicitGodunovCC` solid model can solve most of the solid problems that the `solidFoam` solver in the `explicitSolidDynamics` toolkit can handle. The only exceptions are problems that require the use of the `initialConditions` utility, which is not implemented here.

4.2.4 Additional Modifications - `empty` boundary conditions

Although the `explicitSolidDynamics` toolkit can solve 2D cases, it does not include support for the `empty` boundary condition, which is commonly used in 2D simulations. When an `empty` boundary condition is applied, OpenFOAM excludes calculations on the corresponding boundary patch. Precisely, the `updateCoeffs()` function, called during boundary condition updates, performs no operations for `empty` boundaries. This behavior is defined in the following file:

`src/finiteVolume/fieldsfvPatchFields/constraint/empty/emptyFvPatchField.C`

The implementation of `updateCoeffs()` is as follows:

```
template<class Type>
void Foam::emptyFvPatchField<Type>::updateCoeffs()
{

}
```

Additionally, when an `empty` boundary condition is specified, OpenFOAM treats the mesh as 2D. This leads to the omission of geometrical quantities in the third direction. These quantities are only needed when evaluating field gradients at boundary patches. To address this, the `gradientSchemes` class in `explicitGodunovCC` must account for `empty` boundary conditions.

This modification is straightforward. Whenever a loop iterates over the boundary patches, a condition is added to skip patches of type `empty`. In `gradientSchemes.C` file, which can be found in the following directory:

`explicitGodunovCCSolid/schemes/gradientSchemes/gradientSchemes.C`

we look up "`forAll(mesh_.boundary(), patchID)`" and do the following modifications for all the loops skipping the ones that start with "`if (Pstream::parRun())`":

```
    forAll(mesh_.boundary(), patchID)
    {
        // Check if the boundary patch is of type "empty"
        if (mesh_.boundary()[patchID].type() == "empty")
        {
            continue;
        }

        //rest of the loop
    }
```

The `gradientSchemes` class requires further modifications due to conflicts when working with `empty` BC, particularly in the `distanceMatrixLocal` function. We comment out the following lines of code inside this function, rather than removing them, to allow for potential future development:

```
if (lmN_.boundaryField().types()[patchID] == "fixedValue")
{
    const label& faceID =
        mesh_.boundary()[patchID].start() + facei;

    forAll(mesh_.faces()[faceID], nodei)
    {
        const label& nodeID = mesh_.faces()[faceID][nodei];

        d = XN_[nodeID] - X_[bCellID];
        dCd[bCellID] += d*d;

        for (int i=0; i<7; i++)
        {
            d =
                ((((i+1)*XN_[nodeID])
              + ((7 - i)*XF_.boundaryField()[patchID][facei]))/8.0)
              - X_[bCellID];
            dCd[bCellID] += d*d;
        }
    }
}
```

Now, we compile the code and check for errors:

```
cd ~/solids4foam/src/solids4FoamModels
./Allwmake
```

This adjustment ensures that the solver correctly handles `empty` boundary conditions and avoids unnecessary calculations for 2D simulations.

4.3 Enabling explicitGodunovCC with FSI

To extend the functionality of the explicitGodunovCC model of handling FSI problems, we take the following steps:

- Update/Evaluate Required Fields: Certain fields need to be updated or evaluated to ensure compatibility with solids4foam functionalities. This step ensures that the solid model provides the necessary data for FSI simulations, maintaining consistency with the toolbox's framework.
- Modify Time-Stepping Scheme: Transition from a single-time-step two-stage Runge-Kutta scheme to a dual-time-step two-stage Runge-Kutta scheme. This modification enhances the solver's stability and accuracy, enabling it to effectively handle the coupled dynamics in FSI problems.
- Interface Coupling: This is done by introducing new boundary conditions for linear momentum and traction fields. These boundary conditions facilitate coupling between the solidModel and the fluidSolidInterface model, enabling the update of linear momentum and traction fields at the coupled interface. Moreover, a modification to setTraction() function enables the update of the boundary conditions when explicitGodunovCC model is used.

4.3.1 Evaluating fields required by solids4Foam

To integrate explicitGodunovCC fields into the broader solids4foam functionalities, the following fields are evaluated:

```
1    //nodal displacement
2    pointD() = xN_ - XN_;
3
4    // Update the stress field based on the latest D field
5    sigma() =  symm((1.0/J_)*(P_ & F_.T()));
6
7    // Increment of point displacement
8    pointDD() = pointD() - pointD().oldTime();
```

- Nodal displacement (pointD()): Calculated as the difference between the current nodal position (xN_) and the reference position (XN_).
- Stress field (sigma()): Updated based on the relationship between the Cauchy stress (\Sigma) and the first Piola-Kirchhoff stress tensor (P_).
- Increment of displacement (pointDD()): Evaluated as the change in nodal displacement between the current and previous time steps.

These evaluations ensure that the required fields are correctly defined and updated to maintain compatibility with the solids4foam toolbox during simulations.

4.3.2 Dual-time stepping in explicitGodunovCC

We introduce a pseudo-time step and a corresponding loop to implement dual-time stepping in the explicitGodunovCC model. The pseudo-time step (pDeltaT_) is evaluated similarly to the physical-time step (deltaT_) as before, with deltaT_ replaced by pDeltaT_.

In this setup, the physical-time step is now managed by the solids4Foam solver using the function physics().setDeltaT(runTime), as shown on line 48 in Listing 3.3.

The pseudo-time step must satisfy the stability conditions discussed in Section 1.3.2. To achieve this, we update the void mechanics::time(...) function in the mechanics class as follows:

Listing 4.5: Pseudo-time update

```
void mechanics::time
(
    Time& runTime,
    dimensionedScalar& deltaT,
    dimensionedScalar Up_time
)
{
    const dimensionedScalar& h = op.minimumEdgeLength();

    deltaT = min((cfl_*h)/Up_time, 0.666*runTime.deltaT());
}
```

- `h`: The smallest edge length of the mesh, retrieved using `op.minimumEdgeLength()`.
- `cfl_`: A stability parameter (CFL number).
- `Up_time`: The maximum wave speed
- `deltaT`: The pseudo-time step is computed as the smaller of two values. The first one ensures stability based on mesh and velocity conditions. Meanwhile, the second restricts the pseudo-time step relative to the physical-time step for additional stability.

This modification ensures that the system of equations is advanced using the dual-time stepping approach while adhering to the stability requirements of the simulation.

The pseudo-time loop acts as a correction mechanism implemented using a `do-while` structure to iterate until convergence is achieved or a maximum number of corrections is reached. The implementation is shown below:

Listing 4.6: Pseudo-time loop

```
        // Pseudo time loop (Correction loop)
        do
        {
            //something
        }
        while
        (
            !converged
                (
                    iCorr,
                    pDeltaT_,
                    lm_
                )
         && ++iCorr < nCorr()
        );
```

The `converged()` function evaluates the residuals of the linear momentum field (`lm_`) to determine if the system has reached convergence. We implement it as a private member function within the `explicitGodunovCCSolid` class based on a similar function found in `solidModel` class. This function takes three parameters: `iCorr`, the current iteration number; `pDeltaT_`, the pseudo-time step size; and `lm_`, the linear momentum field. While its primary purpose is to check the residual for linear momentum, the function can be adapted to incorporate additional checks for other fields if necessary.

Finally, We revise the `evolve()` function to incorporate dual-time stepping, as described in Algorithm 4. While the overall structure of the solver algorithm remains largely unchanged, a key adjustment is replacing the use of `oldTime()` for accessing field values from the previous physical-time step with `storePrevIter()` function (lines `526-534` in Listing A.5) to save information from the last iteration and retrieves it during subsequent steps using the `prevIter()` function (lines `548-556` in Listing A.5).

We also update the `gEqns.H` file to align with the dual-time stepping methodology outlined in Algorithm 4. Refer to Listing A.6 for implementation details. To update the implementation, we

can directly replace the `evolve()` function and the `gEqns.H` file and add the `converged()` function as a private member function from the accompanying files.

Note: The angular momentum conservation algorithm is not implemented in this work because it requires further development to align with the time discretization scheme. However, the algorithm remains included in the code for potential use in future work.

Now, we compile the code and check for errors:

```
cd ~/solids4foam/src/solids4FoamModels
./Allwmake
```

4.3.3 Interface coupling

As discussed in Section 3.3.1, the `updateForce()` function invokes the `setTraction()` virtual function from the `solidModel` class. There are two overloaded versions of `setTraction()`function. The first call is defined as follows:

Listing 4.7: `setTraction` (1st call)

```
void Foam::solidModel::setTraction
(
    const label interfaceI,
    const label patchID,
    const vectorField& faceZoneTraction
)
{
    const vectorField patchTraction
    (
        globalPatches()[interfaceI].globalFaceToPatch(faceZoneTraction)
    );

    setTraction(solutionD().boundaryFieldRef()[patchID], patchTraction);
}
```

This version takes the patch ID and traction as inputs and subsequently calls the other overloaded function, adding `solutionD()` to the call. The `solutionD()` function provides a reference to the displacement field `D()`.

Considering the overloaded function `setTraction()` (second call) in Listing 4.8, it assigns a given traction vector field `traction` to a patch vector field `tractionPatch`. This occurs only if the patch matches the type `solidTractionFvPatchVectorField`. If the types match, the function casts the patch field and assigns the new traction values.

The type casting ensures that `tractionPatch` is identified as `solidTractionFvPatchVectorField` correctly. The function first checks the type of `tractionPatch` using its `type()` method and compares it with `solidTractionFvPatchVectorField::typeName`. If the types match, a reference cast using `refCast()` is performed.

The `refCast()` safely converts `tractionPatch` to `solidTractionFvPatchVectorField`, enabling access to its specific properties and functions. After casting, the `traction()` function of the `solidTractionFvPatchVectorField` class assigns the new traction values from `traction`.

This ensures type safety while operating on the appropriate patch field type. If the type does not match, the `else` branch manages the alternative case.

Listing 4.8: `setTraction` (2nd call)

```
void Foam::solidModel::setTraction
(
    fvPatchVectorField& tractionPatch,
    const vectorField& traction
)
{
    if (tractionPatch.type() == solidTractionFvPatchVectorField::typeName)
    {
        solidTractionFvPatchVectorField& patchD =
            refCast<solidTractionFvPatchVectorField>(tractionPatch);
```

```
11
12          patchD.traction() = traction;
13      }
14      else
15      {
16          ...
17      }
18  }
```

To incorporate the specific fields of the `explicitGodunovCCSolid` class, the linear momentum field `lm_b` and the traction field `t`, we redefine both of the `setTraction` functions in Listing 4.7 and 4.8 from `solidModel` class to the `explicitGodunovCCSolid` sub-class. First, we add the functions declarations to `explicitGodunovCCSolid.H`, see Listing 4.9. Then we add the definitions of both calls of the functions to `explicitGodunovCCSolid.C`.

Listing 4.9: Declaration of `setTraction` functions in `explicitGodunovCCSolid.H`

```
1               //- Set traction at specified patch
2               virtual void setTraction
3               (
4                   fvPatchVectorField& tractionPatch,
5                   const vectorField& traction
6               );
7
8               //- Set traction at specified patch
9               virtual void setTraction
10              (
11                  const label interfaceI,
12                  const label patchID,
13                  const vectorField& faceZoneTraction
14              );
```

For the first call, Listing 4.10, the fields `lm_b` and `t_b` are referenced, and the overloaded function is called similar to `solutionD()` Listing 4.8.

Listing 4.10: Redefine `setTraction` (1st call) in `explicitGodunovCCSolid.C`

```
1   void explicitGodunovCCSolid::setTraction
2   (
3       const label interfaceI,
4       const label patchID,
5       const vectorField& faceZoneTraction
6   )
7   {
8       const vectorField patchTraction
9       (
10          globalPatches()[interfaceI].globalFaceToPatch(faceZoneTraction)
11      );
12
13      volVectorField& lm_b = mesh().lookupObjectRef<volVectorField>("lm_b");
14      volVectorField& t_b = mesh().lookupObjectRef<volVectorField>("t_b");
15
16      setTraction(lm_b.boundaryFieldRef()[patchID], patchTraction);
17      setTraction(t_b.boundaryFieldRef()[patchID], patchTraction);
18
19  }
```

For the second call, Listing 4.11, we add similar conditions to that in Listing 4.8 to include linear momentum and traction fields boundary conditions, which will be introduced in the coming section.

Listing 4.11: Modified `setTraction` (2nd call) in `explicitGodunovCCSolid.C`

```
1   void explicitGodunovCCSolid::setTraction
2   (
3       fvPatchVectorField& tractionPatch,
4       const vectorField& traction
5   )
```

```
{
    if (tractionPatch.type() == explicitSolidTractionTractionFvPatchVectorField::typeName)
    {
        explicitSolidTractionTractionFvPatchVectorField& patchTraction =
            refCast<explicitSolidTractionTractionFvPatchVectorField>(tractionPatch);

        patchTraction.traction() = traction;

    }
    else if (tractionPatch.type() == explicitSolidTractionLinearMomentumFvPatchVectorField::typeName)
    {
        explicitSolidTractionLinearMomentumFvPatchVectorField& patchLinearMomentum =
            refCast<explicitSolidTractionLinearMomentumFvPatchVectorField>(tractionPatch);

        patchLinearMomentum.traction() = traction;
    }
    else
    {
        ...
    }
}
```

This modification extends the function by checking additional traction patch types, casting them to the appropriate classes, and assigning the traction values to the corresponding fields.

4.3.4 Coupling interface boundary conditions

The FSI algorithm necessitates the introduction of new boundary conditions for the linear momentum and traction fields, which utilize the `traction()` function called in `setTraction()` (see Listing 4.11).

In OpenFOAM, boundary conditions (BCs) are categorized into basic, constrained, and derived types. Derived BCs are implemented by extending the functionality of basic types. In `solids4foam`, the `solidTraction` BC is a derived type of the basic class `fixedGradient`. This is because, in a displacement-based formulation, the displacement BC is of the Neumann type, requiring the gradient evaluation of the field at the boundary. Conversely, for the momentum-deformation-based formulation, the BCs are of the Dirichlet type, which requires setting a fixed value for the field at the boundary.

To develop the new BCs, we copy and modify the implementation of `solidTraction` as follows:

```
cd ~/solids4foam/src/solids4FoamModels/solidModels/fvPatchFields
mkdir explicitSolidTractionTraction
cp -r solidTraction/. explicitSolidTractionTraction
cd explicitSolidTractionTraction

mv solidTractionFvPatchVectorField.H \
explicitSolidTractionTractionFvPatchVectorField.H

mv solidTractionFvPatchVectorField.C \
explicitSolidTractionTractionFvPatchVectorField.C

sed -i 's/fixedGradient/fixedValue/g' \
explicitSolidTractionTractionFvPatchVectorField.H

sed -i 's/fixedGradient/fixedValue/g' \
explicitSolidTractionTractionFvPatchVectorField.C

sed -i 's/solidTraction/explicitSolidTractionTraction/g' \
explicitSolidTractionTractionFvPatchVectorField.H

sed -i 's/solidTraction/explicitSolidTractionTraction/g' \
```

```
explicitSolidTractionTractionFvPatchVectorField.C
```

Next, we remove all lines involving gradient evaluation or related calls. This process is repeated for the linear momentum BC, ensuring that `explicitSolidTractionLinearMomentum` is set as the `typeName`:

```
cd ~/solids4foam/src/solids4FoamModels/solidModels/fvPatchFields
mkdir explicitSolidTractionLinearMomentum
cp -r solidTraction/. explicitSolidTractionLinearMomentum
cd explicitSolidTractionLinearMomentum

mv solidTractionFvPatchVectorField.H \
explicitSolidTractionLinearMomentumFvPatchVectorField.H

mv solidTractionFvPatchVectorField.C \
explicitSolidTractionLinearMomentumFvPatchVectorField.C

sed -i 's/fixedGradient/fixedValue/g' \
explicitSolidTractionLinearMomentumFvPatchVectorField.H

sed -i 's/fixedGradient/fixedValue/g' \
explicitSolidTractionLinearMomentumFvPatchVectorField.C

sed -i 's/solidTraction/explicitSolidTractionLinearMomentum/g' \
explicitSolidTractionLinearMomentumFvPatchVectorField.H

sed -i 's/solidTraction/explicitSolidTractionLinearMomentum/g' \
explicitSolidTractionLinearMomentumFvPatchVectorField.C
```

Now, we add the specific details for updating the traction in accordance with the BCs defined in the `explicitSolidDynamics` toolkit. We append the following lines to the `updateCoeffs()` function in the source file:
For `explicitSolidTractionTraction`:

```
1    this->operator==(traction_);
```

For `explicitSolidTractionLinearMomentum`:

```
1    const fvsPatchField<vector>& lm_M_ =
2    patch().lookupPatchField<surfaceVectorField, vector>("lm_M");
3
4    const fvsPatchField<vector>& t_M_ =
5        patch().lookupPatchField<surfaceVectorField, vector>("t_M");
6
7    const fvsPatchField<tensor>& S_t_ =
8        patch().lookupPatchField<surfaceTensorField, tensor>("S_t");
9
10   fvsPatchField<vector> lm_C(lm_M_);
11   lm_C = lm_M_ + (S_t_ & ((traction_) - t_M_));
12   this->operator==(lm_C);
```

These modifications ensure compatibility of the new boundary conditions with the explicit dynamics framework, enabling the FSI coupling. Finally, we include the BC's header files to `solidModel.C`:

```
#include "explicitSolidTractionTractionFvPatchVectorField.H"
#include "explicitSolidTractionLinearMomentumFvPatchVectorField.H"
```

We also add their source files to the main library `solids4FoamModels` `Make` files:

```
$(solidFvPatchFields)/explicitSolidTractionTraction\
```

```
    /explicitSolidTractionTractionFvPatchVectorField.C
$(solidFvPatchFields)/explicitSolidTractionLinearMomentum\
/explicitSolidTractionLinearMomentumFvPatchVectorField.C
```

Now, we compile the code and check for errors:

```
cd ~/solids4foam/src/solids4FoamModels
./Allwmake
```

With these modifications, the development of the new solid model `explicitGodunovCC` is complete. The integration ensures compatibility with `solids4foam` functionalities, enabling advanced handling of FSI problems through dual-time stepping and coupling of linear momentum and traction boundary conditions. The updates include modifications to the solver structure, addition of new boundary conditions, and refinement of field evaluation to seamlessly adapt to the momentum-deformation-based formulation.

Chapter 5

FSI Benchmarks

5.1 Verification: Perpendicular flap

The fluid flows through a two-dimensional rectangular channel measuring 6 units in length (x) and 4 units in height (y) Figure 5.1. At the inlet ($x = 0$), a constant inflow velocity of 10 m/s is prescribed in the x-direction [23, 24]. At the outlet ($x = 6$), an outflow boundary condition is applied using a zero-gradient assumption for velocity and pressure. The top wall ($y = 4$), bottom wall ($y = 0$), and The surface of the flap are no-slip boundaries where the velocity is set to zero. The fluid density is $1.0\,\mathrm{kg/m^3}$, and its kinematic viscosity is $1.0\,\mathrm{m^2/s}$.

A solid, elastic flap is fixed at the center of the channel's bottom wall ($x = 0, y = 0$). The flap is 1 unit long in the y-direction and 0.1 units thick in the x-direction. It oscillates due to the fluid pressure exerted on its surface. The solid's density is $3.0 \times 10^3\,\mathrm{kg/m^3}$, and it is characterized by a Young's modulus of $4.0 \times 10^6\,\mathrm{kg/ms^2}$ and a Poisson ratio of 0.3. The fluid and solid properties are summarized in Table 5.1

Table 5.1: Properties of the Fluid and Solid [23, 24]

Property	**Value**	**Units**
Fluid Properties		
Fluid Density	1	$\mathrm{kg/m^3}$
Kinematic Viscosity	1	$\mathrm{m^2/s}$
Inlet Velocity	10	m/s
Solid Properties		
Solid Density	3×10^3	$\mathrm{kg/m^3}$
Young's Modulus	4×10^6	$\mathrm{kg/ms^2}$
Poisson's Ratio	0.3	–

The toolbox includes this tutorial in its directory; however, it is configured to run using displacement-based approaches. Some modifications are necessary to adapt it for the `explicitGodunov` solid model. We begin by copying the tutorial to the working directory with the following commands:

```
cd ~/solids4foam/tutorials/fluidSolidInteraction/
cp -r perpendicularFlap $FOAM_RUN
cd $FOAM_RUN/perpendicularFlap
```

After copying, we add the required field dictionaries using the command:

```
touch  0/solid/lm_b 0/solid/t_b 0/solid/lmN
```

Then, we update these files as shown in Listings A.7, A.8, and A.9 to define the necessary boundary and initial conditions.

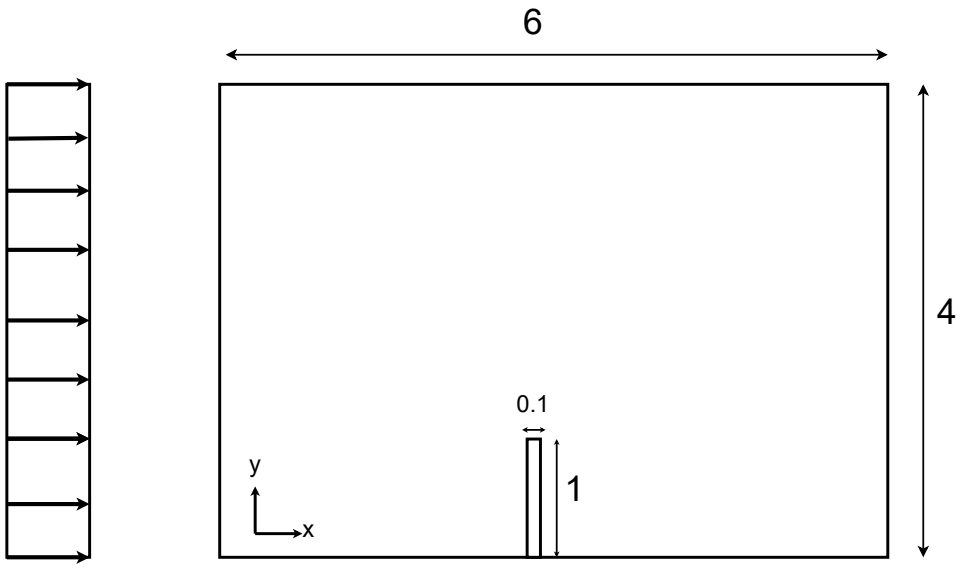

Figure 5.1: Perpendicular flap geometry

Then, we modify `constant/solid/solidProperties` dictionary by setting the appropriate options to enable the `explicitGodunov` model functionality:

Listing 5.1: `solidProperties` dictionary

```
1
2  solidModel    explicitGodunovCC;;
3
4  explicitGodunovCCCoeffs
5  {
6      // Maximum number of momentum correctors
7      nCorrectors             10000;
8
9      // Solution tolerance for displacement
10     solutionTolerance       1e-4;
11
12     // Relative solution tolerance for displacement in outer FSI iterations
13     relConvergenceTolerance 1e-4;
14
15     angularMomentumConservation     no;
16
17     incompressiblilityCoefficient   1.0;
18
19     dampingCoeff  dampingCoeff [ 0 0 -1 0 0 0 0 ] 0;
20
21     // Write frequency for the residuals
22     infoFrequency           100;
23 }
```

Then, we update the `system/solid/controlDict` file by adding the following entries:

```
cfl 0.4;
timeStepping variable;
```

Finally, we configure the `interpolationSchemes` in the `system/solid/fvSchemes` file by setting:

```
default linear;
```

Now, we run the tutorial using:

```
./Allrun
```

The simulation results are presented in Figure 5.2 and 5.3. Figure 5.2 depicts the displacement in the x-direction plotted against time. The plot compares the results of the newly implemented explicit solid model with those reported in [24, 23]. The plot demonstrates that the new solid model aligns well with the literature providing the verification of the algorithm and code development.

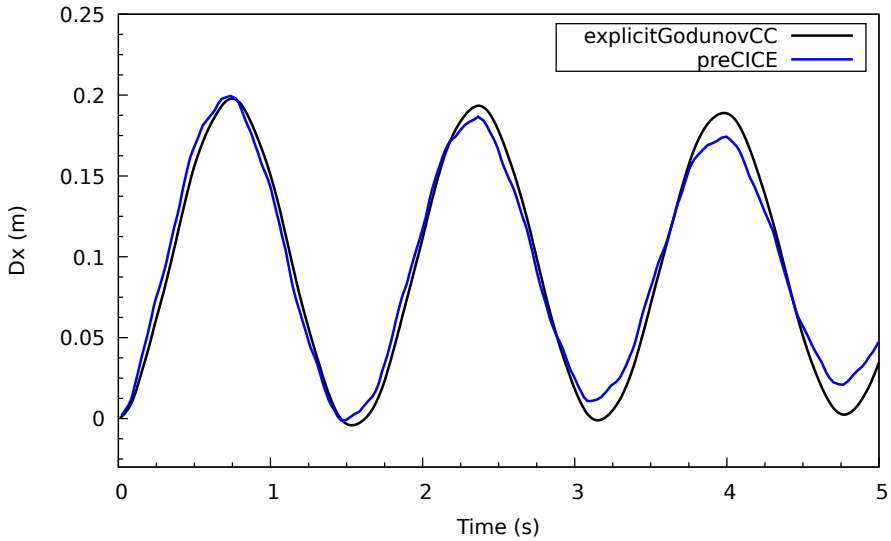

Figure 5.2: Time vs Displacement in the x-direction

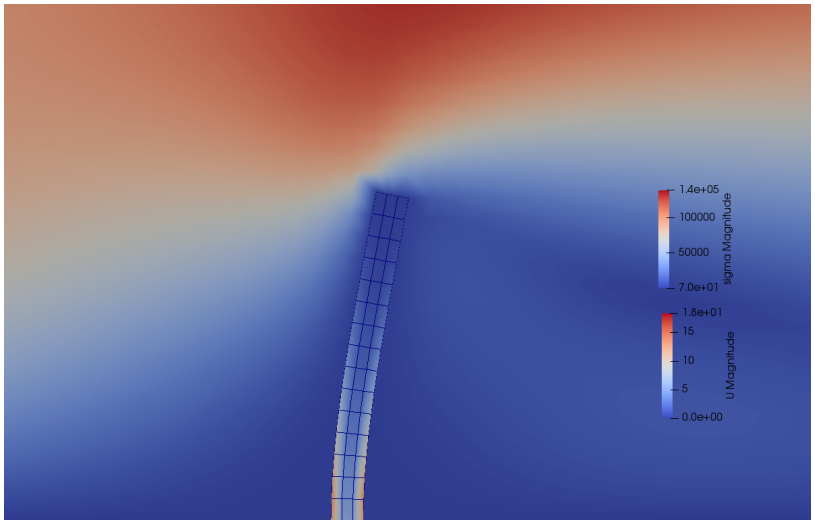

Figure 5.3: Perpendicular flap contour plot

Chapter 6

Conclusion

This project introduced a new solid model to the `solids4foam` toolbox, based on the explicit Godunov cell-centered formulation originally implemented in OpenFOAM for the `explicitSolidDynamics` toolkit.

The new solid model, named `explicitGodunovCC`, incorporates a novel dual-time stepping algorithm designed specifically to meet the requirements of fluid-solid interaction (FSI) problems. This algorithm combines second-order backward Euler time discretization for the physical-time derivative with a two-stage Runge-Kutta (RK) discretization for the pseudo-time derivative.

To validate the implementation and development of the `explicitGodunovCC` solid model, the report presented a benchmark test case. The results demonstrated strong agreement with established literature, confirming the accuracy and robustness of the model.

The primary objective of this report is to serve as a step-by-step tutorial for readers interested in developing new models for solving FSI problems, particularly using the `solids4foam` toolbox.

Bibliography

[1] P. Cardiff and I. Demirdžić, "Thirty years of the finite volume method for solid mechanics," *Archives of Computational Methods in Engineering*, vol. 28, no. 5, pp. 3721–3780, 2021.

[2] 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.

[3] I. Demirdzic, P. Martinovic, and A. Ivankovic, "Numerical simulation of thermal deformation in welded workpiece," *Zavarivanje*, vol. 31, no. 5, pp. 209–219, 1988.

[4] 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.

[5] P. CARDIFF, "Implementing a block-coupled implicit vertex-centred finite volume approach for solid mechanics in openfoam."

[6] J. A. Trangenstein and P. Colella, "A higher-order godunov method for modeling finite deformation in elastic-plastic solids," *Communications on Pure and Applied mathematics*, vol. 44, no. 1, pp. 41–100, 1991.

[7] C. H. Lee, A. J. Gil, and J. Bonet, "Development of a cell centred upwind finite volume algorithm for a new conservation law formulation in structural dynamics," *Computers & Structures*, vol. 118, pp. 13–38, 2013.

[8] J. Haider, C. H. Lee, A. J. Gil, and J. Bonet, "A first-order hyperbolic framework for large strain computational solid dynamics: an upwind cell centred total lagrangian scheme," *International Journal for Numerical Methods in Engineering*, vol. 109, no. 3, pp. 407–456, 2017.

[9] R. J. LeVeque, *Finite volume methods for hyperbolic problems*. Cambridge university press, 2002, vol. 31.

[10] J. Haider, "ExplicitSolidDynamics toolkit for OpenFOAM," 2019. [Online]. Available: https://github.com/jibranhaider/explicitSolidDynamics

[11] 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.

[12] G. Hou, J. Wang, and A. Layton, "Numerical methods for fluid-structure interaction—a review," *Communications in Computational Physics*, vol. 12, no. 2, pp. 337–377, 2012.

[13] C. Farhat, K. G. Van der Zee, and P. Geuzaine, "Provably second-order time-accurate loosely-coupled solution algorithms for transient nonlinear computational aeroelasticity," *Computer methods in applied mechanics and engineering*, vol. 195, no. 17-18, pp. 1973–2001, 2006.

[14] J. Degroote, K.-J. Bathe, and J. Vierendeels, "Performance of a new partitioned procedure versus a monolithic procedure in fluid–structure interaction," *Computers & Structures*, vol. 87, no. 11-12, pp. 793–801, 2009.

[15] P. Causin, J.-F. Gerbeau, and F. Nobile, "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, pp. 4506–4527, 2005.

[16] J. Degroote, P. Bruggeman, R. Haelterman, and J. Vierendeels, "Stability of a coupling technique for partitioned solvers in fsi applications," *Computers & Structures*, vol. 86, no. 23-24, pp. 2224–2234, 2008.

[17] U. Küttler and W. A. Wall, "Fixed-point fluid–structure interaction solvers with dynamic relaxation," *Computational mechanics*, vol. 43, no. 1, pp. 61–72, 2008.

[18] B. M. Irons and R. C. Tuck, "A version of the aitken accelerator for computer iteration," *International Journal for Numerical Methods in Engineering*, vol. 1, no. 3, pp. 275–277, 1969.

[19] C.-W. Shu and S. Osher, "Efficient implementation of essentially non-oscillatory shock-capturing schemes," *Journal of computational physics*, vol. 77, no. 2, pp. 439–471, 1988.

[20] A. Jameson, "Time dependent calculations using multigrid, with applications to unsteady flows past airfoils and wings," in *10th Computational fluid dynamics conference*, 1991, p. 1596.

[21] S. Bhat and J. C. Mandal, "Contact preserving riemann solver for incompressible two-phase flows," *Journal of Computational Physics*, vol. 379, pp. 173–191, 2019.

[22] S. P. Bhat and J. C. Mandal, "An improved hllc-type solver for incompressible two-phase fluid flows," *Computers & Fluids*, vol. 244, p. 105570, 2022.

[23] A. Zanella, L. Abergo, F. Caccia, M. Morelli, and A. Guardone, "Towards an open-source framework for fluid–structure interaction using su2, mbdyn and precice," *Journal of Computational and Applied Mathematics*, vol. 429, p. 115211, 2023.

[24] PreCICE, "Perpendicular flap tutorial using PreCICE," 2019. [Online]. Available: https://precice.org/tutorials-perpendicular-flap.html

Study questions

1. What is the difference between monolithic and partitioned approaches in fluid-structure interaction (FSI)?
2. Under what conditions should a strongly-coupled approach be used instead of a weakly-coupled approach in FSI?
3. What distinguishes a displacement-based formulation from a momentum-deformation-based formulation in solid dynamics systems?
4. How is the pseudo-time step determined?
5. What are the primary classes in the `solids4foam` toolbox?
6. Which functions are responsible for enforcing FSI interface boundary conditions in a partitioned algorithm?
7. why do we have to keep virtual functions when re-implement a solidModel sub-class?
8. How to save fields information from previous time and previous iteration?
9. what function is responsible of setting the traction at solid boundary ?

Appendix A

Code Listing

A.1 explicitSolidDynamics

Listing A.1: `solidFoam.C` file

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Copyright (C) 2011-2018 OpenFOAM Foundation
     \\/     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 3 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, see <http://www.gnu.org/licenses/>.

Application
    solidFoam

Description
    A solid mechanics solver based on a Total Lagrangian mixed formulation
    comprising of conservation laws for linear momentum and deformation
    gradient of the system.

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

#include "fvCFD.H"
#include "pointFields.H"
#include "operations.H"
#include "solidModel.H"
#include "mechanics.H"
#include "gradientSchemes.H"
#include "interpolationSchemes.H"
#include "angularMomentum.H"

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

```
45
46 int main(int argc, char *argv[])
47 {
48     #include "setRootCase.H"
49     #include "createTime.H"
50     #include "createMesh.H"
51     #include "readControls.H"
52     #include "createFields.H"
53
54     while (runTime.run())
55     {
56         mech.time(runTime, deltaT, max(Up_time));
57
58         lm.oldTime();
59         F.oldTime();
60         x.oldTime();
61         xF.oldTime();
62         xN.oldTime();
63
64         forAll(RKstages, stage)
65         {
66             #include "gEqns.H"
67
68             if (RKstages[stage] == 0)
69             {
70                 #include "updateVariables.H"
71             }
72         }
73
74         lm = 0.5*(lm.oldTime() + lm);
75         F = 0.5*(F.oldTime() + F);
76         x = 0.5*(x.oldTime() + x);
77         xF = 0.5*(xF.oldTime() + xF);
78         xN = 0.5*(xN.oldTime() + xN);
79
80         #include "updateVariables.H"
81
82         if (runTime.outputTime())
83         {
84             uN = xN - XN;
85             uN.write();
86
87             p = model.pressure();
88             p.write();
89         }
90
91         Info<< "Simulation completed = "
92             << (runTime.value()/runTime.endTime().value())*100 << "%" << endl;
93     }
94
95     Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
96         << "  ClockTime = " << runTime.elapsedClockTime() << " s"
97         << nl << endl;
98
99     return 0;
100 }
101
102 // ************************************************************************* //
```

A.2 explicitGodunovCC

Listing A.2: testModelSolid.H file

```
1 /*---------------------------------------------------------------------------*\
2 License
```

```
3       This file is part of solids4foam.
4
5       solids4foam is free software: you can redistribute it and/or modify it
6       under the terms of the GNU General Public License as published by the
7       Free Software Foundation, either version 3 of the License, or (at your
8       option) any later version.
9
10      solids4foam is distributed in the hope that it will be useful, but
11      WITHOUT ANY WARRANTY; without even the implied warranty of
12      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
13      General Public License for more details.
14
15      You should have received a copy of the GNU General Public License
16      along with solids4foam.  If not, see <http://www.gnu.org/licenses/>.
17
18  Class
19      testModelSolid
20
21  Description
22      Mathematical model where linear geometry is assumed i.e. small strains and
23      small rotations are assumed, and the total displacement is the primary
24      unknown.
25
26      The stress is calculated by the run-time selectable mechanical law.
27
28  Author
29      Philip Cardiff, UCD.  All rights reserved.
30
31  SourceFiles
32      testModelSolid.C
33
34  \*---------------------------------------------------------------------------*/
35
36  #ifndef testModelSolid_H
37  #define testModelSolid_H
38
39  #include "solidModel.H"
40  // #include "volFields.H"
41  // #include "surfaceFields.H"
42  // #include "pointFields.H"
43  // #include "uniformDimensionedFields.H"
44
45  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
46
47  namespace Foam
48  {
49
50  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
51
52  namespace solidModels
53  {
54
55  /*---------------------------------------------------------------------------*\
56                         Class testModelSolid Declaration
57  \*---------------------------------------------------------------------------*/
58
59  class testModelSolid
60  :
61      public solidModel
62  {
63      // Private data
64
65      // Private Member Functions
66
67          //- Disallow default bitwise copy construct
68          testModelSolid(const testModelSolid&);
69
70          //- Disallow default bitwise assignment
```

```
71          void operator=(const testModelSolid&);
72
73
74  protected:
75
76      // Protected member functions
77
78          //- Return nonlinear geometry enumerator
79          virtual nonLinearGeometry::nonLinearType nonLinGeom() const
80          {
81              return nonLinearGeometry::LINEAR_GEOMETRY;
82          }
83
84
85  public:
86
87      //- Runtime type information
88      TypeName("testModel");
89
90      // Constructors
91
92          //- Construct from components
93          testModelSolid
94          (
95              Time& runTime,
96              const word& region = dynamicFvMesh::defaultRegion
97          );
98
99      // Destructor
100
101         virtual ~testModelSolid()
102         {}
103
104
105     // Member Functions
106
107         // Access
108
109             //- Each solidModel must indicate whether D or DD is the primary
110             //  solution variable
111             virtual volVectorField& solutionD()
112             {
113                 // This model solves for D
114                 return D();
115             }
116
117         // Edit
118
119             //- Evolve the solid solver and solve the mathematical model
120             virtual bool evolve();
121
122             //- Traction boundary surface normal gradient
123             virtual tmp<vectorField> tractionBoundarySnGrad
124             (
125                 const vectorField& traction,
126                 const scalarField& pressure,
127                 const fvPatch& patch
128             ) const;
129 };
130
131
132 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
133
134 } // End namespace solidModel
135
136 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
137
138 } // End namespace Foam
```

```
139
140 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
141
142 #endif
143
144 // ************************************************************************* //
```

Listing A.3: testModelSolid.C file

```
1  /*---------------------------------------------------------------------------*\
2  License
3      This file is part of solids4foam.
4
5      solids4foam is free software: you can redistribute it and/or modify it
6      under the terms of the GNU General Public License as published by the
7      Free Software Foundation, either version 3 of the License, or (at your
8      option) any later version.
9
10     solids4foam is distributed in the hope that it will be useful, but
11     WITHOUT ANY WARRANTY; without even the implied warranty of
12     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
13     General Public License for more details.
14
15     You should have received a copy of the GNU General Public License
16     along with solids4foam.  If not, see <http://www.gnu.org/licenses/>.
17
18 \*---------------------------------------------------------------------------*/
19
20 #include "testModelSolid.H"
21 #include "fvm.H"
22 #include "fvc.H"
23 #include "fvMatrices.H"
24 #include "addToRunTimeSelectionTable.H"
25 #include "momentumStabilisation.H"
26 #include "backwardDdtScheme.H"
27
28 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
29
30 namespace Foam
31 {
32
33 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
34
35 namespace solidModels
36 {
37
38 // * * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //
39
40 defineTypeNameAndDebug(testModelSolid, 0);
41 addToRunTimeSelectionTable(solidModel, testModelSolid, dictionary);
42
43
44 // * * * * * * * * * * * *  Private Member Functions * * * * * * * * * * * * * //
45
46 // * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * //
47
48 testModelSolid::testModelSolid
49 (
50     Time& runTime,
51     const word& region
52 )
53 :
54     solidModel(typeName, runTime, region)
55 {
56 }
57
58
59 // * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * //
```

```
60
61
62 bool testModelSolid::evolve()
63 {
64     return true;
65 }
66
67
68 tmp<vectorField> testModelSolid::tractionBoundarySnGrad
69 (
70     const vectorField& traction,
71     const scalarField& pressure,
72     const fvPatch& patch
73 ) const
74 {
75
76 }
77
78
79 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
80
81 } // End namespace solidModels
82
83 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
84
85 } // End namespace Foam
86
87 // ************************************************************************* //
```

Listing A.4: `explicitGodunovCCSolid.H` file

```
1  /*---------------------------------------------------------------------------*\
2  License
3      This file is part of solids4foam.
4
5      solids4foam is free software: you can redistribute it and/or modify it
6      under the terms of the GNU General Public License as published by the
7      Free Software Foundation, either version 3 of the License, or (at your
8      option) any later version.
9
10     solids4foam is distributed in the hope that it will be useful, but
11     WITHOUT ANY WARRANTY; without even the implied warranty of
12     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General
13     Public License for more details.
14
15     You should have received a copy of the GNU General Public License
16     along with solids4foam. If not, see <http://www.gnu.org/licenses/>.
17
18 Original Authors
19     Jibran Haider - explicitSolidDynamics toolkit
20     Philip Cardiff, UCD - solids4foam toolbox
21
22     Original source code retrieved from:
23     https://github.com/jibranhaider/explicitSolidDynamics/
24
25 Modifications by
26     Khoder Alhamwi Alshaar, IITB - Modified to implement the explicit Godunov
27     scheme for solid mechanics as part of the solids4foam project.
28     Additional modifications include the integration of a dual-time stepping
29     algorithm, redefinition the setTraction() and converge() functions,
30     and developing new baoundary conditions for linear momentum and traction
31     fields.
32
33     Note: The angular momentum algorithm has not been adapted to the new
34     time discretization scheme. As a result, it may not provide accurate
35     solutions if used in its current form.
36
37 Class
```

```
38      explicitGodunovCCSolid
39
40  Description
41      A solid model based on the explicitSolidDynamics toolkit.
42      Solves a first-order system of total Lagrangian mixed formulation
43      comprising conservation laws for linear momentum and deformation
44      gradient.
45
46  SourceFiles
47      explicitGodunovCCSolid.C
48
49  \*---------------------------------------------------------------------------*/
50
51  #ifndef explicitGodunovCCSolid_H
52  #define explicitGodunovCCSolid_H
53
54  #include "solidModel.H"
55  #include "operations.H"
56  #include "solidMaterialModel.H"
57  #include "mechanics.H"
58  #include "gradientSchemes.H"
59  #include "interpolationSchemes.H"
60  #include "angularMomentum.H"
61
62  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
63
64  namespace Foam
65  {
66
67  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
68
69  namespace solidModels
70  {
71
72  /*---------------------------------------------------------------------------*\
73                         Class explicitGodunovCCSolid Declaration
74  \*---------------------------------------------------------------------------*/
75
76  class explicitGodunovCCSolid
77  :
78      public solidModel
79  {
80      // Private data
81   // Private data
82          Time& runTime_;
83
84      // Reading dictionaries
85      // Mechanical properties
86      IOdictionary mechanicalProperties_;
87
88      // Control dictionary
89      IOdictionary controlDict_;
90
91      scalar incompressibilityCoefficient_;
92      // Stabilisation parameter for near incompressibility
93      const scalar& beta_;
94
95      // Angular momentum conservation
96      const word angularMomentumConservation_;
97
98      //Creating mesh parameters
99
100     // Mesh-related fields
101     operations op_;
102
103     // Point mesh
104     // pointMesh pMesh(mesh); // available pMesh()
105
```

```
106     // Material face area
107     const surfaceScalarField& magSf_;
108
109     // Material face area normal vector
110     const surfaceVectorField& Sf_;
111
112     // Minimum edge length
113     const dimensionedScalar h_;
114
115     //Creating mesh coordinate fields
116
117     // Material cell center coordinates
118     const volVectorField& C_;
119
120     // Spatial cell center coordinates
121     volVectorField x_;
122     // material cell center coordinates
123     const volVectorField X_;
124
125     // Spatial nodal coordinates
126     pointVectorField xN_;
127
128     // Material nodal coordinates
129     pointVectorField XN_;
130
131     // Spatial face center coordinates
132     surfaceVectorField xF_;
133
134     //Creating mesh normal fields
135
136     // Material normals
137     const surfaceVectorField N_;
138
139     // Spatial normals
140     surfaceVectorField n_;
141
142     //Creating linear momentum fields
143     // Cell linear momentum
144     volVectorField lm_;
145
146     // Nodal linear momentum
147     pointVectorField lmN_;
148
149     //Creating strain measure fields
150
151     // Deformation gradient tensor
152     volTensorField F_;
153
154     // Cofactor of deformation
155     volTensorField H_;
156
157     // Jacobian of deformation
158     volScalarField J_;
159
160     // Creating constitutive model
161
162     // Solid model class
163     solidMaterialModel model_;
164
165     // Density
166     const dimensionedScalar& rho_;
167
168     // Pressure
169     volScalarField p_;
170
171     // First Piola Kirchhoff stress tensor
172     volTensorField P_;
173     volVectorField Px_;
```

```
174     volVectorField Py_;
175     volVectorField Pz_;
176 
177     // Creating fields for wave speeds
178 
179     // Continuum mechanics class
180     mechanics mech_;
181 
182     // Longitudinal wave speed
183     volScalarField Up_;
184 
185     // Wave speed for time increment
186     volScalarField Up_time_;
187 
188     // Shear wave speed
189     volScalarField Us_;
190 
191     // Creating fields for gradient
192     // Gradient class
193     gradientSchemes grad_;
194 
195     // Gradient of cell linear momentum
196     volTensorField lmGrad_;
197 
198     // Gradients of first Piola Kirchhoff stress tensor
199     volTensorField PxGrad_;
200     volTensorField PyGrad_;
201     volTensorField PzGrad_;
202 
203     //Creating fields for reconstruction
204     // Reconstruction of linear momentum
205     surfaceVectorField lm_M_;
206     surfaceVectorField lm_P_;
207 
208     // Reconstruction of PK1 stresses
209     surfaceTensorField P_M_;
210     surfaceTensorField P_P_;
211 
212     // Reconstruction of traction
213     surfaceVectorField t_M_;
214     surfaceVectorField t_P_;
215 
216     // Creating fields for riemann solver
217     // Surface tensors
218     surfaceTensorField S_lm_;
219     surfaceTensorField S_t_;
220 
221     // Contact traction
222     surfaceVectorField tC_;
223 
224     // Contact linear momentum
225     surfaceVectorField lmC_;
226 
227     // Volumetric fields
228     volVectorField t_b_;
229     volVectorField lm_b_;
230 
231     // Creating fields for the constrained procedure
232     // Constrained class
233     interpolationSchemes interpolate_;
234 
235     // Cell-averaged linear momentum
236     volVectorField lmR_;
237 
238     // Local gradient of cell-averaged linear momentum
239     volTensorField lmRgrad_;
240 
241 
```

```
242     // Creating fields for angular momentum
243     // Angular momentum class
244     angularMomentum am_;
245 
246     // RHS of linear momentum equation
247     volVectorField rhsLm_;
248     volVectorField rhsLm1_;
249 
250     // RHS of angular momentum equation
251     volVectorField rhsAm_;
252 
253 
254     // Creating fields for post-processing
255     // Nodal displacement field
256     pointVectorField uN_;
257 
258     //Creating variables for time
259     // Time increment
260     dimensionedScalar deltaT_;
261     // Time increment (pesudo)
262     dimensionedScalar pDeltaT_;
263 
264     // Runge-Kutta stage
265     scalarList RKstages_;
266 
267     // Private Member Functions
268     //- Check for solution convergence
269     bool converged
270     (
271         const int iCorr,
272         const dimensionedScalar pDeltaT,
273         const GeometricField<vector, fvPatchField, volMesh>& vf
274     );
275         //- Disallow default bitwise copy construct
276         explicitGodunovCCSolid(const explicitGodunovCCSolid&);
277 
278         //- Disallow default bitwise assignment
279         void operator=(const explicitGodunovCCSolid&);
280 
281 
282 protected:
283 
284     // Protected member functions
285 
286         //- Return nonlinear geometry enumerator
287         virtual nonLinearGeometry::nonLinearType nonLinGeom() const
288         {
289             return nonLinearGeometry::LINEAR_GEOMETRY;
290         }
291 
292 
293 public:
294 
295     //- Runtime type information
296     TypeName("explicitGodunovCC");
297 
298     // Constructors
299 
300         //- Construct from components
301         explicitGodunovCCSolid
302         (
303             Time& runTime,
304             const word& region = dynamicFvMesh::defaultRegion
305         );
306 
307     // Destructor
308 
309         virtual ~explicitGodunovCCSolid()
```

```
        {}


    // Member Functions

        // Access

            //- Each solidModel must indicate whether D or DD is the primary
            //  solution variable
            virtual volVectorField& solutionD()
            {
                // This model solves for D
                return D();
            }

        // Edit

            //- Evolve the solid solver and solve the mathematical model
            virtual bool evolve();

            //- Traction boundary surface normal gradient
            virtual tmp<vectorField> tractionBoundarySnGrad
            (
                const vectorField& traction,
                const scalarField& pressure,
                const fvPatch& patch
            ) const;

            //- Set traction at specified patch
            virtual void setTraction
            (
                fvPatchVectorField& tractionPatch,
                const vectorField& traction
            );

            //- Set traction at specified patch
            virtual void setTraction
            (
                const label interfaceI,
                const label patchID,
                const vectorField& faceZoneTraction
            );
};


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

} // End namespace solidModel

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

} // End namespace Foam

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

#endif

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

Listing A.5: `explicitGodunovCCSolid.C` file

```
/*---------------------------------------------------------------------------*\
License
    This file is part of solids4foam.

    solids4foam 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 3 of the License, or (at your
```

```
8       option) any later version.
9
10      solids4foam is distributed in the hope that it will be useful, but
11      WITHOUT ANY WARRANTY; without even the implied warranty of
12      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
13      General Public License for more details.
14
15      You should have received a copy of the GNU General Public License
16      along with solids4foam.  If not, see <http://www.gnu.org/licenses/>.
17
18  \*---------------------------------------------------------------------------*/
19
20  #include "explicitGodunovCCSolid.H"
21  #include"addToRunTimeSelectionTable.H"
22  #include "explicitSolidTractionTractionFvPatchVectorField.H"
23  #include "explicitSolidTractionLinearMomentumFvPatchVectorField.H"
24
25
26  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
27
28  namespace Foam
29  {
30
31  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
32
33  namespace solidModels
34  {
35
36  // * * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //
37
38  defineTypeNameAndDebug(explicitGodunovCCSolid, 0);
39  addToRunTimeSelectionTable(solidModel, explicitGodunovCCSolid, dictionary);
40
41
42  // * * * * * * * * * * * *  Private Member Functions * * * * * * * * * * * * * //
43
44  bool explicitGodunovCCSolid::converged
45  (
46      const int iCorr,
47      const dimensionedScalar pDeltaT,
48      const GeometricField<vector, fvPatchField, volMesh>& vf
49  )
50  {
51      // We will check three residuals:
52      // - relative linear momentum residual
53
54      bool converged = false;
55
56      // Calculate residual based on the relative change of vf
57      scalar denom = 0.0;
58
59      // Denom is linear momentum increment
60          denom = gMax
61          (
62  #ifdef OPENFOAM_NOT_EXTEND
63              DimensionedField<scalar, volMesh>
64  #else
65              Field<scalar>
66  #endif
67              (
68                  mag(vf.internalField() - vf.oldTime().internalField())
69              )
70          );
71
72      if (denom < SMALL)
73      {
74          denom =
75          max
```

```
        (
            gMax
            (
#ifdef OPENFOAM_NOT_EXTEND
                DimensionedField<scalar, volMesh>(mag(vf.internalField()))
#else
                mag(vf.internalField())
#endif
            ),
            SMALL
        );
    }

    const scalar residualvf =
        gMax
        (
#ifdef OPENFOAM_NOT_EXTEND
            DimensionedField<scalar, volMesh>
            (
                mag(vf.internalField() - vf.prevIter().internalField())
            )
#else
            mag(vf.internalField() - vf.prevIter().internalField())
#endif
        )/denom;

    // If one of the residuals has converged to an order of magnitude
    // less than the tolerance then consider the solution converged
    // force at least 1 outer iteration
   if (residualvf < solutionTol())
    {
        Info<< "    Converged" << endl;
        converged = true;
    }

    // Print residual information
    if (iCorr == 0)
    {
        Info<< "    Corr, res, pDeltaT" << endl;
    }
    else if (iCorr % infoFrequency() == 0 || converged || iCorr >= nCorr() - 1)
    {
        Info<< "    " << iCorr
            << ", " << residualvf
            << ", " << pDeltaT.value() << endl;

        if (iCorr >= nCorr())
        {
            Warning
                << "Max iterations reached within the momentum loop"
                << endl;
            converged = true;
        }
    }

    return converged;
 }


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

explicitGodunovCCSolid::explicitGodunovCCSolid
(
    Time& runTime,
    const word& region
)
:
    solidModel(typeName, runTime, region),
```

```
144     runTime_(runTime),
145     //reading dicts
146     mechanicalProperties_
147     (
148         IOobject
149         (
150             "mechanicalProperties",
151             runTime.constant(),
152             mesh(),
153             IOobject::MUST_READ_IF_MODIFIED,
154             IOobject::NO_WRITE
155         )
156     ),
157 
158     controlDict_
159     (
160         IOobject
161         (
162             "controlDict",
163             runTime.system(),
164             mesh(),
165             IOobject::MUST_READ_IF_MODIFIED,
166             IOobject::NO_WRITE
167         )
168     ),
169 
170     incompressibilityCoefficient_
171     (
172         solidModelDict().lookupOrAddDefault<scalar>("incompressiblilityCoefficient", 1)
173     ),
174 
175     beta_(incompressibilityCoefficient_),
176 
177     angularMomentumConservation_
178     (
179         solidModelDict().lookupOrAddDefault<word>("angularMomentumConservation", "no")
180     ),
181 
182 //---------------------------------------------------------------------
183 
184     op_(mesh()),
185     magSf_(mesh().magSf()),
186     Sf_(mesh().Sf()),
187     h_(op_.minimumEdgeLength()),
188 
189     // Creating mesh coordinate fields
190     C_(mesh().C()),
191 
192     x_
193     (
194         IOobject("x", mesh()),
195         C_
196     ),
197     X_
198     (
199         IOobject("X", mesh()),
200         C_
201     ),
202 
203     xN_
204     (
205         IOobject("xN", mesh()),
206         pMesh(),
207         dimensionedVector("xN", dimensionSet(0,1,0,0,0,0,0), vector::zero)
208     ),
209 
210     XN_(xN_),
211 
```

```
212     xF_(mesh().Cf()),
213
214     // Creating mesh normal fields
215     N_((Sf_ / mesh().magSf()).ref()),
216     n_(N_),
217
218     // Creating linear momentum fields
219     lm_
220     (
221         IOobject
222         (
223             "lm",
224             runTime.timeName(),
225             mesh(),
226             IOobject::READ_IF_PRESENT,
227             IOobject::AUTO_WRITE
228         ),
229         mesh(),
230         dimensionedVector("lm", dimensionSet(1,-2,-1,0,0,0,0), vector::zero)
231     ),
232     lmN_
233     (
234         IOobject
235         (
236             "lmN",
237             runTime.timeName(),
238             mesh(),
239             IOobject::MUST_READ,
240             IOobject::AUTO_WRITE
241         ),
242         pMesh()
243     ),
244
245     F_
246     (
247         IOobject("F", mesh()),
248         mesh(),
249         tensor::I
250     ),
251
252     H_
253     (
254         IOobject("H", mesh()),
255         det(F_)*op_.invT(F_)
256     ),
257     J_
258     (
259         IOobject("J", mesh()),
260         det(F_)
261     ),
262
263     // Creating constitutive model
264     model_
265     (
266         F_,
267         mechanicalProperties_
268     ),
269
270     rho_(model_.density()),
271     p_(model_.pressure()),
272     P_(model_.piola()),
273     Px_(op_.decomposeTensorX(P_)),
274     Py_(op_.decomposeTensorY(P_)),
275     Pz_(op_.decomposeTensorZ(P_)),
276
277     mech_
278     (
279         F_,
```

```
280         controlDict_
281     ),
282
283     Up_
284     (
285         IOobject("Up", mesh()),
286         mesh(),
287         model_.Up()/beta_
288     ),
289
290     Up_time_
291     (
292         IOobject("Up_time", mesh()),
293         mesh(),
294         model_.Up()
295     ),
296
297     Us_
298     (
299         IOobject("Us", mesh()),
300         mesh(),
301         model_.Us()*beta_
302     ),
303
304     grad_(mesh()),
305
306     lmGrad_(grad_.gradient(lm_)),
307
308     PxGrad_(grad_.gradient(Px_)),
309     PyGrad_(grad_.gradient(Py_)),
310     PzGrad_(grad_.gradient(Pz_)),
311
312     // Reconstruction of linear momentum
313     lm_M_(
314         IOobject("lm_M", mesh()),
315         mesh(),
316         dimensionedVector("lm_M", lm_.dimensions(), vector::zero)
317     ),
318
319     lm_P_(lm_M_),
320
321     // Reconstruction of PK1 stresses
322     P_M_(
323         IOobject("P_M", mesh()),
324         mesh(),
325         dimensionedTensor("P_M", P_.dimensions(), tensor::zero)
326     ),
327     P_P_(P_M_),
328
329     // Reconstruction of traction
330     t_M_
331     (
332         IOobject("t_M", mesh()),
333         P_M_ & N_
334     ),
335     t_P_((P_P_ & N_).ref()),
336
337     S_lm_(mech_.Smatrix_lm()),
338     S_t_(mech_.Smatrix_t()),
339
340     // Contact traction
341     tC_(t_M_),
342     // Contact linear momentum
343     lmC_(lm_M_),
344     t_b_
345     (
346         IOobject
347         (
```

```
348             "t_b",
349             runTime.timeName(),
350             mesh(),
351             IOobject::MUST_READ,
352             IOobject::AUTO_WRITE
353         ),
354         mesh()
355     ),
356
357     lm_b_
358     (
359         IOobject
360         (
361             "lm_b",
362             runTime.timeName(),
363             mesh(),
364             IOobject::MUST_READ,
365             IOobject::AUTO_WRITE
366         ),
367         mesh()
368     ),
369
370     // Constrained class
371     interpolate_(mesh()),
372
373     // Cell-averaged linear momentum
374     lmR_(interpolate_.surfaceToVol(lmC_)),
375
376     // Local gradient of cell-averaged linear momentum
377     lmRgrad_(grad_.localGradient(lmR_, lmC_)),
378
379
380     // Creating fields for angular momentum
381     // Angular momentum class
382     am_(mesh(), mechanicalProperties_),
383
384     // RHS of linear momentum equation
385     rhsLm_(
386         IOobject("rhsLm", mesh()),
387         mesh(),
388         dimensionedVector("rhsLm", dimensionSet(1,-2,-2,0,0,0,0), vector::zero)
389     ),
390     rhsLm1_(rhsLm_),
391
392     // RHS of angular momentum equation
393     rhsAm_(
394         IOobject("rhsAm", mesh()),
395         mesh(),
396         dimensionedVector("rhsAm", dimensionSet(1,-1,-2,0,0,0,0), vector::zero)
397     ),
398         // Nodal displacement field
399     uN_(
400         IOobject
401         (
402             "uN",
403             runTime.timeName(),
404             mesh(),
405             IOobject::NO_READ,
406             IOobject::AUTO_WRITE
407         ),
408         pMesh(),
409         dimensionedVector("uN", dimLength, vector::zero)
410     ),
411
412
413     // Time increment
414     deltaT_(
415         "deltaT",
```

```
        dimTime,
        runTime.deltaTValue()
    ),
    // Time increment
    pDeltaT_(
        "pDeltaT",
        dimTime,
        runTime.deltaTValue()
    ),

    // Runge-Kutta stage
    RKstages_(2)


{
    Info << "Reading data from dictionaries ..." << endl;
    if
    (
        angularMomentumConservation_ != "yes" && angularMomentumConservation_ != "no"
    )
    {
        FatalErrorIn("angularMomentumConservation_")
            << "Valid type entries are 'yes' or 'no' "
            << "for angularMomentumConservation"
            << abort(FatalError);
    }
    Info << "Angular Momentum Conservation: " << angularMomentumConservation_ << endl;
    Info << "dampingCoeff: " << dampingCoeff().value() << endl;



    // Assign mesh points to the primitive field of xN_
    xN_.primitiveFieldRef() = mesh().points();
    XN_ = xN_;


    RKstages_[0] = 0;
    RKstages_[1] = 1;


    Info << "Printing data ..." << endl;

    // Print material properties
    model_.printMaterialProperties();

    // Print global linear and angular momentum
    am_.printGlobalMomentum(lm_,x_);

    // Print centroid of geometry
    mech_.printCentroid();
    #include "updateVariables.H"
    #include "riemannSolver.H"

    x_.oldTime();
    xF_.oldTime();
    lm_.oldTime();
    F_.oldTime();
    xN_.oldTime();

    lm_.oldTime().oldTime();
    F_.oldTime().oldTime();
    x_.oldTime().oldTime();
    xF_.oldTime().oldTime();
    xN_.oldTime().oldTime();

    // Set the printInfo
    physicsModel::printInfo() = bool
    (
```

```
484         runTime.timeIndex() % infoFrequency() == 0
485      || mag(runTime.value() - runTime.endTime().value()) < SMALL
486     );
487
488     Info<< "Frequency at which info is printed: every " << infoFrequency()
489         << " time-steps" << endl;
490
491 }
492
493
494 // * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * * * //
495
496
497 bool explicitGodunovCCSolid::evolve()
498 {
499     Info<< "starting of evolve function" << endl;
500     // Mesh update loop
501     do
502     {
503         int iCorr = 0;
504
505         if (physicsModel::printInfo())
506         {
507             Info<< "Evolving solid solver form explicitGodunovCCSolid" << endl;
508         }
509
510         // Pseudo time loop (Correction loop)
511         do
512         {
513             if (angularMomentumConservation_ == "yes")
514             {
515                 x_.storePrevIter();
516                 xF_.storePrevIter();
517             }
518
519             F_.storePrevIter();
520             lm_.storePrevIter();
521             xN_.storePrevIter();
522
523             mech_.time(runTime_, pDeltaT_, max(Up_time_));
524
525             forAll(RKstages_, stage)
526             {
527                 #include "gEqns.H"
528
529                 if (RKstages_[stage] == 0)
530                 {
531                     #include "updateVariables.H"
532                 }
533             }
534
535             if (angularMomentumConservation_ == "yes")
536             {
537                 x_ = 0.5*(x_.prevIter() + x_);
538                 xF_ = 0.5*(xF_.prevIter() + xF_);
539             }
540
541             lm_ = 0.5*(lm_.prevIter() + lm_);
542             F_ = 0.5*(F_.prevIter() + F_);
543             xN_ = 0.5*(xN_.prevIter() + xN_);
544
545             #include "updateVariables.H"
546
547             pointD() = xN_ - XN_;
548
549         }
550         while
551         (
```

```
            !converged
                (
                    iCorr,
                    pDeltaT_,
                    lm_
                )
         && ++iCorr < nCorr()
        );

        // Update the stress field based on the latest D field
        sigma() =  symm((1.0/J_) * (P_ & F_.T()));

        // Increment of point displacement
        pointDD() = pointD() - pointD().oldTime();

    }
    while (mesh().update());

    return true;
}


tmp<vectorField> explicitGodunovCCSolid::tractionBoundarySnGrad
(
    const vectorField& traction,
    const scalarField& pressure,
    const fvPatch& patch
) const
{
   Info<<"tractionBoundarySnGrad is notimplimented";
}


void explicitGodunovCCSolid::setTraction
(
    fvPatchVectorField& tractionPatch,
    const vectorField& traction
)
{
    if (tractionPatch.type() == explicitSolidTractionTractionFvPatchVectorField::typeName)
    {
        explicitSolidTractionTractionFvPatchVectorField& patchTraction =
            refCast<explicitSolidTractionTractionFvPatchVectorField>(tractionPatch);

        patchTraction.traction() = traction;

    }
    else if (tractionPatch.type() == explicitSolidTractionLinearMomentumFvPatchVectorField::typeName)
    {
        explicitSolidTractionLinearMomentumFvPatchVectorField& patchLinearMomentum =
            refCast<explicitSolidTractionLinearMomentumFvPatchVectorField>(tractionPatch);

        patchLinearMomentum.traction() = traction;
    }
    else
    {
        FatalErrorIn
        (
            "void Foam::solidModel::setTraction\n"
            "(\n"
            "    fvPatchVectorField& tractionPatch,\n"
            "    const vectorField& traction\n"
            ")"
        )   << "Boundary condition "
            << tractionPatch.type()
            << " for patch " << tractionPatch.patch().name()
            << " should instead be type "
            << explicitSolidTractionTractionFvPatchVectorField::typeName
```

```
            << " or "
            << explicitSolidTractionLinearMomentumFvPatchVectorField::typeName
            << abort(FatalError);
    }
}


void explicitGodunovCCSolid::setTraction
(
    const label interfaceI,
    const label patchID,
    const vectorField& faceZoneTraction
)
{
    const vectorField patchTraction
    (
        globalPatches()[interfaceI].globalFaceToPatch(faceZoneTraction)
    );

    volVectorField& lm_b = mesh().lookupObjectRef<volVectorField>("lm_b");
    volVectorField& t_b = mesh().lookupObjectRef<volVectorField>("t_b");

    setTraction(lm_b.boundaryFieldRef()[patchID], patchTraction);
    setTraction(t_b.boundaryFieldRef()[patchID], patchTraction);

}


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

} // End namespace solidModels

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

} // End namespace Foam

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

Listing A.6: gEqns.H file

```
// Compute right hand sides
rhsLm_ = fvc::surfaceIntegrate(tC_*magSf_);

//! angula momentum required to be updated to consider  dual time step?
if (angularMomentumConservation_ == "yes")
{
    rhsAm_ = fvc::surfaceIntegrate((xF_ ^ tC_)*magSf_);
    am_.AMconservation(rhsLm_, rhsLm1_, rhsAm_, stage , pDeltaT_);

    if (runTime_.timeIndex() == 0)
    {
        // Update coordinates
        x_ += pDeltaT_*(lm_/rho_) - pDeltaT_*( (x_.prevIter() - x_.oldTime())/(deltaT_));
        xF_ += pDeltaT_*(lmC_/rho_) - pDeltaT_*( (xF_.prevIter() - xF_.oldTime())/(deltaT_));
    }
    else
    {
        // Update coordinates
        x_ += pDeltaT_*(lm_/rho_)   - pDeltaT_ * ( (3*x_.prevIter() - 4* x_.oldTime()   + x_.oldTime()
     .oldTime())/(2*deltaT_));
        xF_ += pDeltaT_*(lmC_/rho_) - pDeltaT_ * ( (3*xF_.prevIter() - 4* xF_.oldTime() + xF_.oldTime
     ().oldTime())/(2*deltaT_));
    }
}

if (runTime_.timeIndex() == 0)
{

```

```
    xN_ += pDeltaT_*(lmN_/rho_)- pDeltaT_*( (xN_.prevIter() - xN_.oldTime())/(deltaT_));

    lm_ += pDeltaT_*rhsLm_ - pDeltaT_*( (lm_.prevIter() - lm_.oldTime())/(deltaT_));
    lm_ -= pDeltaT_*dampingCoeff()*lm_;

    F_ += pDeltaT_*fvc::surfaceIntegrate((lmC_/rho_)*Sf_) - pDeltaT_*( (F_.prevIter() - F_.oldTime())/(deltaT_)) ;
}

else
{
    xN_ += pDeltaT_*(lmN_/rho_) - pDeltaT_ * ( (3*xN_.prevIter() - 4* xN_.oldTime() + xN_.oldTime().oldTime())/(2*deltaT_));

    lm_ += pDeltaT_*rhsLm_ - pDeltaT_*( (3*lm_.prevIter() - 4* lm_.oldTime() + lm_.oldTime().oldTime())/(2*deltaT_));
    lm_ -= pDeltaT_*dampingCoeff()*lm_;

    F_ += pDeltaT_*fvc::surfaceIntegrate((lmC_/rho_)*Sf_) - pDeltaT_*( (3*F_.prevIter() - 4* F_.oldTime() + F_.oldTime().oldTime())/(2*deltaT_));
}
```

A.3 perpendicular flap benchmark dictionaries

Listing A.7: `lm_b` dictionary

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    location    "0";
    object      lm_b;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [1 -2 -1 0 0 0 0];

internalField   uniform (0 0 0);

boundaryField
{
    interface
    {
        type            explicitSolidTractionLinearMomentum;
        traction        uniform ( 0 0 0 );
        pressure        uniform 0;
        value           uniform (0 0 0);
    }
    bottom
    {
        type    fixedValue;
        value   uniform (0 0 0);
    }
    frontAndBack
    {
        type            empty;
    }

}
```

Listing A.8: `lmN` dictionary

```
FoamFile
{
```

```
    version     2.0;
    format      ascii;
    class       pointVectorField;
    location    "0";
    object      lmN;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [1 -2 -1 0 0 0 0];

internalField   uniform (0 0 0);

boundaryField
{

   interface
   {
         type    zeroGradient;
   }

   bottom
   {
        type    fixedValue;
        value   uniform (0 0 0);
   }

   frontAndBack
   {
       type            empty;
   }
}
```

Listing A.9: t_b dictionary

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    location    "0";
    object      t_b;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [1 -1 -2 0 0 0 0];

internalField   uniform (0 0 0);

boundaryField
{
    interface
    {
        type            explicitSolidTractionTraction;
        traction        uniform ( 0 0 0 );
        pressure        uniform 0;
        value           uniform (0 0 0);
    }
    bottom
    {
        type    movingTraction;
        value   uniform (0 0 0);
    }
    frontAndBack
    {
        type            empty;
    }
}
```