Contribution to numerical modeling of fluid-structure interaction in hydrodynamics applications

by Herry Lesmana

A thesis submitted in partial fulfilment
of the requirements for the degree of

Master of Science in Computational Mechanics

at

Ecole Centrale de Nantes

Supervisors:

Prof. Laurent Stainier

Institut de Recherche en Genie Civil et Mecanique (GeM)

Dr. Alban Leroyer

Laboratoire de Recherche en Hydrodynamique, Énergétique et Environnement Atmosphérique – Equipe Modélisation Numérique (LHEEA - EMN)

Abstract

The thesis presents uncoupled two-dimensional structure and fluid dynamics simulations using Zorglib and ISIS-CFD solver respectively, a development of computational procedures in particular for the fluid-structure interface dynamic library to couple the two solvers to solve fluid-structure interactions (FSI) problems, and eventually benchmark coupled numerical simulations. The Zorglib solver is based on the finite element discretization method whereas the ISIS-CFD solver is based on the unstructured finite volume discretization method. The coupling of the two solvers adopts the implicit strongly coupled partitioned approach to ensure stable solutions in the case where added-mass effects take place [16].

The initial phase of the work in this thesis is to ensure the mesh convergence of the solid simulation by comparing different two-dimensional solid elements to solve an analytical static problem. Following after that is a numerical test performed on a benchmark solid dynamic [13] and the results were found to agree well. For the ISIS-CFD solver a numerical test is performed on the classical case of a two-dimensional laminar incompressible flow past a fixed square cylinder based on [17] and the results also agree well. The second phase of the work is to integrate Zorglib, an object oriented based solver, in ISIS-CFD solver to enable the stable coupling of the two solvers. The computational stability is ensured by implementing an internal convergence loop with under-relaxation on the fluid forces and solid displacements as part of ISIS-CFD non-linear iterations. By using this approach the total CPU time depends only on the CPU time of ISIS-CFD. Eventually in the last phase of the work the verification of the coupling strategy is done by performing FSI simulations which show good agreements with the work done in [7] [12] [13].

Acknowledgements

I would like to thank Prof. Laurent Stainier and Dr. Alban Leroyer for giving the opportunities to do this research project and guiding the learning process with full supports. It has been my pleasure to work under their guidance and learn many new things in the computational solid and fluid area.

I would like also to thank Prof. Nicolas Chevaugeon for his support during the Master course and his recommendation so that I could pursue an internship in Nantes.

During the last two semesters of the master course I have been fortunate to have Hector, Mattia, and Cyril who share their thesis experience and in a way work alongside to achieve the same purposes.

Contents

1 Introduction

There have been numerous international publications and practical applications of numerical simulations concerning the fluid-structure interactions problems. The knowledge of the numerical procedure in these areas has been steadily growing in the past decades as the problems increasingly become more complicated and different numerical treatments are required to ensure stable and accurate solutions. The application of fluid-structure interactions encompasses a wide range of engineering and life science, to name a few:

- Structural engineering design for designing long-span bridges, high-rise buildings, and lightweight roof structures with wind flow interactions.
- Aerospace engineering analyses such as analysis of airfoil oscillations, flutter prediction, and parachute dynamics.
- Biomechanics design such as cardiovascular mechanics, cerebrospinal mechanics, and artificial heart valves design.

In most cases to model the real world problems, the numerical simulations have to be able to characterize large solid deformations as a result of interactions with viscous fluids. When the structure is very light or the flow is highly compressible, this becomes a highly non-linear problem and to model an efficient, accurate, and stable computational procedure for this kind of problem is quite challenging and it is still pursued by many researchers in this area.

There are a number of different numerical strategies to couple the fluid and solid discretized solutions to solve FSI problems. In particular [6] proposed a simultaneous solution procedure where the discretized model equations for fluid, structure and coupling conditions are unified in a single non-linear system of equations and combined with the mesh dynamics equation to be solved in every iteration loop. This solution procedure is often called the monolithic solution procedure. In general the monolithic solution procedure is stable and the solution can converge relatively fast when solving large solid deformations interacting with viscous fluid. The drawback is that it does not have the flexibility to utilize the readily established solvers with different solid and fluid discretizations.

A weak coupling partitioned solution procedure is proposed by [18] to solve transient aero elastic problems where the solution advances in time without any iteration to ensure the convergence of the coupling. Some additional correction strategies can be adapted to limit the accumulation of the errors. This procedure is relatively simple and the CPU time can be significantly low but the procedure is only first order accurate. It is conditionally stable and in some cases the time step constraint to achieve a stable solution means the CPU time advantage does not exist anymore. When the fluid density is comparable or higher than the solid density, there exists an added mass effect which cannot be solved by the reducing the time step anymore [8].

A more elaborate partitioned solution procedure is proposed in [7] with strong coupling where an iterative procedure is performed on the FSI interface computation to achieve a specified accuracy requirements. The strategy is relied on the Newton-Raphson full exact linearization procedure to solve the incremental problem. Several numerical examples which are shown in [7] have robust solutions when dealing with highly non-linear FSI problems.

Taking advantage of the well-established computational structural dynamics (CSD) solver, Zorglib, and computational fluid dynamics (CFD) solver, ISIS CFD, developed in Ecole Centrale de Nantes, a postdoctoral work commenced the development of computational procedures to couple the two solvers to perform two-dimensional fluid-rigid body interactions simulations by adopting the strongly coupled implicit partitioned procedure. The main objective of this thesis is to develop further the existing computational procedures to be able to perform two dimensional fluid-structure interactions simulations of elastic and flexible body.

The outline of the following chapters is as follows:

Chapter 2: This chapter discusses the main conservation equations which are the fundamental principle equations used in the solid and fluid computational models.

Chapter 3: The fundamental aspects of ISIS CFD solver i.e. governing equations, discretization strategy, and computation algorithm are discussed briefly in this chapter. Numerical examples are presented at the end of the chapter compared to the referenced international publication.

Chapter 4: The fundamental aspects of Zorlib solver are discussed in the same manner like in *Chapter 2* but with additional short explanations of the time integration strategy and Newton Raphson method to solve non-linear equations. A mesh sensitivity

analysis and numerical tests compared to the analytical solution and referenced international publication are presented at the end of the chapter.

Chapter 5: This chapter firstly explained different conditions that have to be satisfied on the fluid-structure interface and followed with the coupling strategy adopted in this work. Eventually the coupling interface library is explained at the end of the chapter.

Chapter 6: In this chapter several fluid-structure numerical results are presented. The simulations are based on the test cases in the international publications and the current results are compared to the ones in the publications.

Chapter 7: The conclusion of the work done in the thesis.

2 Conservation Equations

The section is synthesized from [1]. The following two mathematical expressions will be used to derive the balance principles.

Material time derivative:

$$\frac{Df}{Dt} = \frac{\partial f}{\partial t} + \vec{v} \cdot \vec{\nabla} f \tag{2.1}$$

Reynold's Transport Theorem:

$$\frac{D}{Dt}\int_{\Omega} f \, d\Omega = \int_{\Omega} \left(\frac{Df}{Dt} + f \, \vec{\nabla} \cdot \vec{v}\right) d\Omega \tag{2.2}$$

where f can be a scalar, vector, or tensor function, $\vec{v}$ is the velocity vector field, $\vec{\nabla}(\blacksquare)$ is gradient operator, $\vec{\nabla} \cdot (\blacksquare)$ is divergence operator, and Ω is the material domain.

2.1 Conservation of Mass

The mass of any material domain in the body is constant and this implies that the material time derivative of the mass has to be zero:

$$\frac{Dm}{Dt} = \frac{D}{Dt}\int_{\Omega} \rho \, d\Omega = 0 \tag{2.3}$$

Using **Eq.** (2.1) and (2.2):

$$\frac{D}{Dt}\int_{\Omega} \rho \, d\Omega = \int_{\Omega} \left(\frac{D\rho}{Dt} + \rho \, \vec{\nabla} \cdot \vec{v}\right) d\Omega = 0 \tag{2.4}$$

$$\int_{\Omega} \left(\frac{\partial \rho}{\partial t} + \vec{v} \cdot \vec{\nabla}\rho + \rho \, \vec{\nabla} \cdot \vec{v}\right) d\Omega = 0 \tag{2.5}$$

Because **Eq.** (2.5) holds for any sub-domain, the integral can be omitted:

$$\frac{\partial \rho}{\partial t} + \vec{v} \cdot \vec{\nabla}\rho + \rho \, \vec{\nabla} \cdot \vec{v} = 0 \tag{2.6}$$

$$\frac{\partial \rho}{\partial t} + \vec{\nabla} \cdot (\rho \, \vec{v}) = 0 \tag{2.7}$$

Eq. (2.7) is also known as the conservative form of the conservation of mass equation.

2.2 Conservation of Momentum

The basis of the conservation of linear momentum equation is that the rate of change of linear momentum is equal to the total applied force. The linear momentum is the product of the density, ρ, and the velocity vector, $\vec{v}$, over an arbitrary domain, Ω, and the conservation of momentum is defined as the following:

$$\frac{D}{Dt}\int_{\Omega} \rho\vec{v}\, d\Omega = \int_{\Omega} \rho\vec{b}\, d\Omega + \int_{\Gamma} \vec{t}\, d\Gamma \tag{2.8}$$

where $\rho\vec{b}$ is the body forces vector and $\vec{t}$ is the surface traction vector. The left hand side of **Eq.** (2.8) can be derived further using the *Reynold's Transport Theorem* as defined in **Eq.** (2.2) and combined with **Eq.** (2.6):

$$\frac{D}{Dt}\int_{\Omega} \rho\vec{v}\, d\Omega = \int_{\Omega} \left(\frac{D(\rho\vec{v})}{Dt} + \rho\vec{v}\,\vec{\nabla}\cdot\vec{v}\right) d\Omega$$

$$= \int_{\Omega} \left(\rho\frac{D\vec{v}}{Dt} + \vec{v}\left(\frac{D\rho}{Dt} + \rho\,\vec{\nabla}\cdot\vec{v}\right)\right) d\Omega$$

0, **Eq.** (4) of conservation of mass

$$= \int_{\Omega} \rho\frac{D\vec{v}}{Dt} d\Omega \tag{2.9}$$

Using the Gauss's divergence theorem, the boundary integral for the surface traction can be transformed into the following formulation:

$$\int_{\Gamma} \vec{t}\, d\Gamma = \int_{\Omega} \vec{\nabla}\cdot\boldsymbol{\sigma}\, d\Omega \tag{2.10}$$

where $\boldsymbol{\sigma}$ is the *Cauchy* stress tensor. By substituting **Eq.** (2.9) and (2.10) into (2.8) and omitting the volume integral because it holds for any sub-domain, the conservation of momentum equation becomes:

$$\rho\frac{D\vec{v}}{Dt} = \rho\vec{b} + \vec{\nabla}\cdot\boldsymbol{\sigma} \tag{2.11}$$

Eq. (2.11) is expressed in current configuration and includes the material time derivative. This is usually used in the solid mechanics problem and it can be expressed in terms of the spatial coordinates in the current configuration to give the following formulation:

$$\rho\frac{\partial\vec{v}}{\partial t} = \rho\vec{b} + \vec{\nabla}\cdot\boldsymbol{\sigma} \tag{2.12}$$

Eq. (2.12) is known as the *updated Lagrangian formulation* in non-linear solid mechanics problems. Another formulation which is known as the *total Lagrangian formulation* is obtained by defining the momentum equation in the reference configuration using the material coordinates:

$$\rho_0 \frac{\partial \vec{v}}{\partial t} = \rho_0 \vec{b} + \vec{\nabla}_0 \cdot \boldsymbol{P} \tag{2.13}$$

where the subscript 0 refers to the reference configuration and $\boldsymbol{P}$ is the *first Piola-Kirchhoff* stress tensor.

The *Eulerian formulation* of **Eq.** (2.11) can be obtained by deriving the velocity material time derivative using **Eq.** (2.1):

$$\rho \frac{D\vec{v}}{Dt} = \rho \left(\frac{\partial \vec{v}}{\partial t} + \vec{v} \cdot \vec{\nabla} \vec{v} \right) = \rho \vec{b} + \vec{\nabla} \cdot \boldsymbol{\sigma} \tag{2.14}$$

Eq. (2.14) is usually used in the fluid mechanics problem.

2.3 Conservation of Energy

For thermomechanical process, the conservation of energy or known as well as the first law of thermodynamics states that the power of the internal and kinetic energy is equal to the external power and the power supplied by heat sources:

$$\mathbb{P}^{int} + \mathbb{P}^{kin} = \mathbb{P}^{ext} + \mathbb{P}^{heat} \tag{2.15}$$

Each of the powers in **Eq.** (2.15) can be expanded as follows:

$$\frac{D}{Dt} \int_\Omega \rho w^{int} \, d\Omega + \frac{D}{Dt} \int_\Omega \frac{1}{2} \rho \, \vec{v} \cdot \vec{v} \, d\Omega$$

$$= \int_\Omega \vec{v} \cdot \rho \, \vec{b} \, d\Omega + \int_\Gamma \vec{v} \cdot \vec{t} \, d\Gamma + \int_\Omega \rho \, s \, d\Omega - \int_\Gamma \vec{n} \cdot \vec{q} \, d\Gamma \tag{2.16}$$

where w^{int} is the specific internal energy, s is the heat sources, and $\vec{q}$ is the heat flux. **Eq.** (2.16) can be derived further into *Eulerian* and *Lagrangian* partial differential equations and considering only mechanical process as follows:

$$\rho \frac{Dw^{int}}{Dt} = \boldsymbol{\sigma} : \boldsymbol{d} \tag{2.17}$$

$$\rho_o \dot{w}^{int} = \boldsymbol{P} : \dot{\boldsymbol{F}}^T \tag{2.18}$$

where $\boldsymbol{d}$ and $\dot{\boldsymbol{F}}$ is the rate of deformation tensor and deformation gradient tensor respectively.

3 Computational Fluid Dynamics (CFD) Solver

3.1 Introduction

The CFD solver used in this thesis is ISIS-CFD flow solver which is developed by Equipe Modélisation Numérique (EMN) in Ecole Centrale de Nantes [10]. It is based on the unsteady incompressible Reynolds-Averaged Navier Stokes Equations (RANSE) which is discretized using the finite volume method to build the spatial discretization of the transport equations. The velocity field is obtained from the momentum conservation equations and the pressure field is extracted from the mass conservation equation transformed into a pressure-equation. The face-based method is generalized to two-dimensional or three dimensional unstructured meshes where non-overlapping control volumes are bounded by an arbitrary number of constitutive faces.

In this thesis the flow regime under investigation is limited to the laminar flow but ISIS-CFD is capable to model the turbulent flows using additional transport equations which are discretized and solved in the same manner as the momentum conservation equations.

3.1 Governing Equations

The assumptions taken to model the fluid dynamics problem are:

- Incompressible flow
- Uniform density
- Viscous fluid
- Isothermal condition

Using index notation, the mass conservation equation for a moving domain is formulated as:

$$\frac{\partial}{\partial t}\int_V \rho \, dV + \int_S \rho\left(v_i - v_i^d\right) n_i \, dS = 0 \tag{3.1}$$

whereas the momentum conservation equation is formulated as the following:

$$\frac{\partial}{\partial t}\int_V \rho v_i \, dV + \int_S \rho v_j\left(v_i - v_i^d\right) n_j \, dS = \int_S \left(\tau_{ij} I_j - p I_i\right) n_j \, dS + \int_V \rho b_i \, dV \tag{3.2}$$

The fluid domain which is represented by a control volume V is bounded by a surface S which moves with a velocity $\vec{v}_d$ and an outward normal vector $\vec{n}$. The pressure and velocity of the flow field are represented by $\vec{v}$ and p respectively. The viscous component of the stress tensor and body force vector are represented by τ_{ij} and b_i. I_j is a vector whose components vanish, except for the component j which is equal to unity. τ_{ij} for a Newtonian fluid is defined as:

$$\tau_{ij} = 2\,\mu\,D_{ij} = \mu\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right) \tag{3.3}$$

where D_{ij} is the strain rate tensor.

3.2 Finite Volume Discretization

The finite volume method is used to discretize spatially the governing equations. The fundamental aspect of a finite volume method lies in the approximation of a volume integration of a function F in a domain V by the product of the volume V by the value of the function F at the center of the domain C:

$$\int_V F\,dV \approx F_C V \tag{3.4}$$

For two functions $F(\vec{x},t)$ and $G(\vec{x},t)$ in a control volume V and its center C, the finite volume approximate the following products:

$$\int_V F\,G\,dV \approx F_C G_C V \tag{3.5}$$

$$\int_V \frac{F}{G}dV \approx \frac{F_C}{G_C}V \tag{3.6}$$

A second order approximation can be achieved if the location of C is at the geometric barycenter of the domain.

The finite volume method also requires evaluations of fluxes across the cell faces which means surface integrations have to be utilized. All the variables are located at the geometric center of cells so values of functions at center of cell faces have to be built from cell-centered values of the functions from each side of the cell face.

If the density is uniform and constant, **Eq.** (2.1) becomes:

$$\frac{\partial}{\partial t}\int_V dV + \int_S \left(v_i - v_i^d\right) n_i \, dS = 0 \tag{3.7}$$

The following formula has to be imposed for a displacement of a surface bounding a control volume:

$$\frac{\partial}{\partial t}\int_V dV - \int_S v_i^d \, n_i \, dS = 0 \tag{3.8}$$

With this constraint, finally the mass conservation equation for a moving domain with uniform and constant density becomes:

$$\int_S v_i \, n_i \, dS = 0 \tag{3.9}$$

and after discretization it becomes:

$$\sum_f v_i \, S_i \; = 0 \tag{3.10}$$

For a generic variable Q for a cell $\mathbb{V}$ with its center C and bounded by an arbitrary number of faces f, the discretization of the momentum conservation equation is formulated as the following:

$$\frac{\partial}{\partial \tau}(\rho V Q)_C + \frac{\partial}{\partial t}(\rho V Q)_C + \sum_f \left(CF_f - DF_f\right) = \left(S_Q^{\mathbb{V}}\right) + \sum_f \left(S_Q^f\right) \tag{3.11}$$

$$CF_f = \dot{m}_f Q_f \quad ; \quad DF_f = \left(\Gamma_Q\right)_f \left(\vec{\nabla} Q_f \cdot \overrightarrow{\iota_k}\right)(S_k)_f \tag{3.12}$$

The terms CF_f and DF_f are the convective and diffusive fluxes through the face f. $S_Q^{\mathbb{V}}$ is the source term of the cell volume and S_Q^f is the source term of the cell face. τ is a local fictitious time variable and the vector $\overrightarrow{\iota_k}$ is a generic unit vector where $\overrightarrow{\iota_1}$ = (1, 0, 0). The present of τ is to enforce the diagonal dominance for the linearized equations that are solved successfully in a non-coupled way. Γ_Q is an isotropic or anisotropic diffusion coefficient.

The mass fluxes $\dot{m}_f$ is formulated as the following:

$$\dot{m}_f = \rho(\vec{v} - \vec{v}_d)_f \cdot \vec{S}_f \tag{3.13}$$

$$\vec{S}_f = S_f \, \vec{n}_f \tag{3.14}$$

where $\vec{S}_f$ is the oriented surface vector.

The temporal derivatives are evaluated by the upwind second-order discretization using the following formulation:

$$\frac{\partial A}{\partial t} \cong e^c A^c + e^p A^p + e^q A^q \tag{3.15}$$

where the superscript c refers to the current time t^c, p is one time step before t^c, and q two time steps before t^c. The coefficients (e^c, e^p, and e^q) are obtained from the Taylor series expansion from t^c and depend on a possibly prescribed variable time step law $\Delta t(t)$.

The fictious local time derivative is needed to stabilize the solution procedure for steady flows and evaluated as the following:

$$\frac{\partial A}{\partial \tau} \cong \frac{A^c - A^{c0}}{\Delta \tau} \tag{3.16}$$

where A^{c0} is the previous estimation of A^c in the non-linear loop.

The final form of the generic discrete transport equation is the following:

$$\left(e^c + \frac{1}{\Delta \tau}\right)(\rho V Q)_C^c + \sum_f \left(CF_f - DF_f\right) = \left(S_Q^{\mathbb{V}}\right) + \sum_f \left(S_Q^f\right) - (e\rho VQ)_C^p - (e\rho VQ)_C^q + \frac{(\rho VQ)_C^{c0}}{\Delta \tau} \tag{3.17}$$

In ISIS-CFD, the quantities on the face center can be built using either the centered face reconstruction or the upwinded face reconstruction. The upwinded face reconstruction numerically more stable and prevents unphysical oscillations with the order of accuracy between 1 and 2.

The computation of the gradient in a cell uses either the *Weighted Least-Square* and *Gauss* method in ISIS-CFD. For the pressure equation, the pseudo-physical Rhie & Chow mass flux reconstruction is used to avoid the chequerboard oscillations.

The computation algorithm used in ISIS-CFD is similar to the *Semi-Implicit Method for Pressure-Linked Equations* (SIMPLE) algorithm which solved a segregated decoupled momentum and pressure equations. The linear systems resulting from the momentum and pressure equations are solved with the help of iterative linear solvers.

The computation algorithm can be summarized as the following:

1. Initialization of quantities A^0 at $t = t^0$

2. New time step $t = t + \Delta t$
3. Start the non-linear loop
4. If needed, compute turbulence fields from A^0
5. Solve the discretized momentum equations to obtain a new prediction of the velocities
6. Solve the pressure equation to obtain a new pressure field
7. Update the velocity fluxes and correct the velocity components with new pressure field
8. If the non-linear residuals are still above the tolerance, update the non-linear fields and return to step 4
9. Go to step 2 and update the time

3.3 Numerical Example

This section presents a CFD simulation example of an unsteady two dimensional flow pass a square cylinder based on the work of [17] for the unsteady case with a blockage ratio of 1/8 and several *Reynolds'* number, *Re*, of 75, 100, 150, and 200. The simulation set up is depicted in **Figure 3.1** with the fluid dynamic viscosity of 1.0 Pa s, the time step of 0.01 s, and the fluid density varied from 75, 100, 150, and 200 kg/m^3 to obtain the corresponding Reynolds' number where it is based on the cylinder diameter, *D* of 1 m, and the maximum flow velocity of the parabolic inflow profile, U_{max} of 1 m/s.

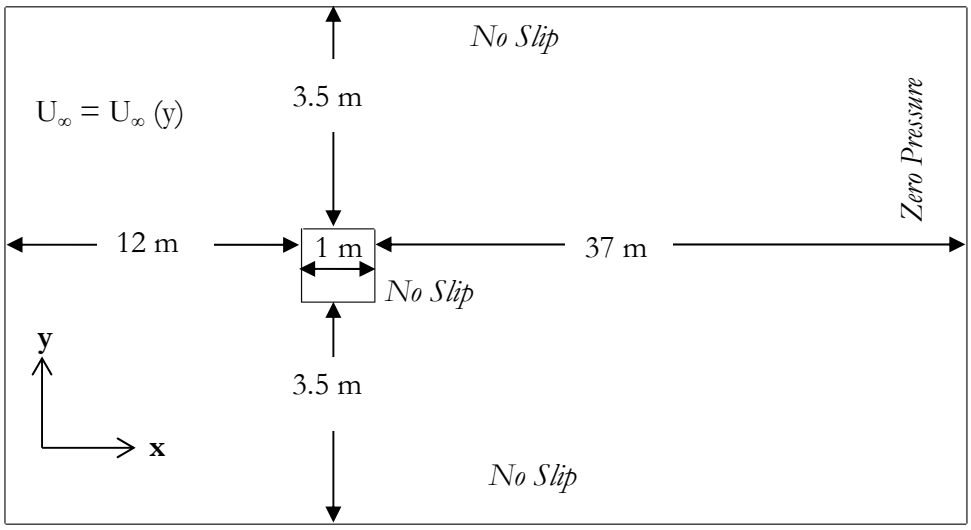


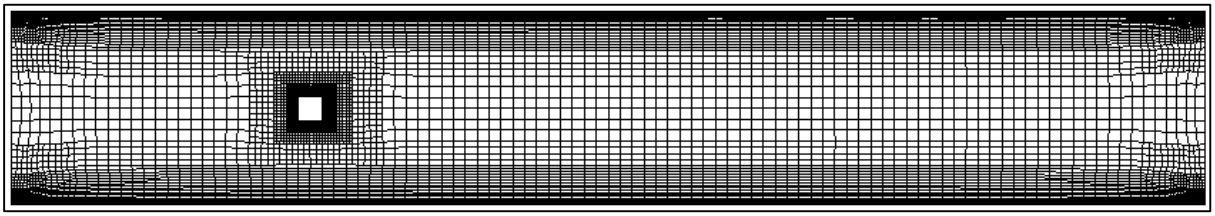
Figure 3.1 Simulation set up for the two dimensional flow pass a square cylinder

The parabolic velocity profile is defined with the following equation:

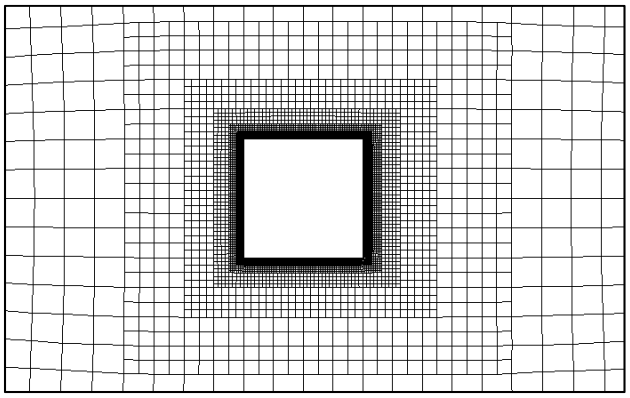
$$U_{\infty}(y) = U_{max}\frac{y(H-y)}{(H/2)^2} \tag{3.18}$$

where H is the channel height.

The fluid mesh is shown in **Figure 3.2** where it has 17,380 cells and 36,426 nodes. Specific refinements are applied to regions close to the cylinder body and on the top and bottom wall to capture the boundary layer phenomena as the boundary conditions are 'no slip' for those regions.



(**a**)



(**b**)

Figure 3.2 (**a**) Full view of the fluid mesh (**b**) Zoomed view on the square cylinder body

Figure 3.3 and **3.4** show the streamlines and pressure for *Reynolds* number of 100 and 200 respectively at three different times when the lift force reaches the minimum, maximum, and then minimum amplitude again. **Figure 3.5** compares the lift coefficient, C_L, of each *Reynolds* number and it shows that the oscillation of the lift occurs earlier with higher amplitude when the *Reynolds* number increased. An important parameter to be analyzed is the *Strouhal* number defined as follows:

$$St = \frac{f\ D}{U_{max}} \tag{3.19}$$

where f is the measured frequency of the vortex shedding which is determined by using the Fast Fourier Transform (FFT) analysis of the time series of the lift coefficient. **Figure 3.6** shows the results of *Strouhal* number and average drag coefficient, C_D, against the *Reynolds* number which agrees well with the result in [17] as shown in **Figure 3.7**. As explained in [17] the increase and decrease of the *Strouhal* number is due to the fact that when the *Reynolds* number increases the initial separation point moves from the trailing edge to the leading edge and this in turn change the frequency of the vortex shedding.

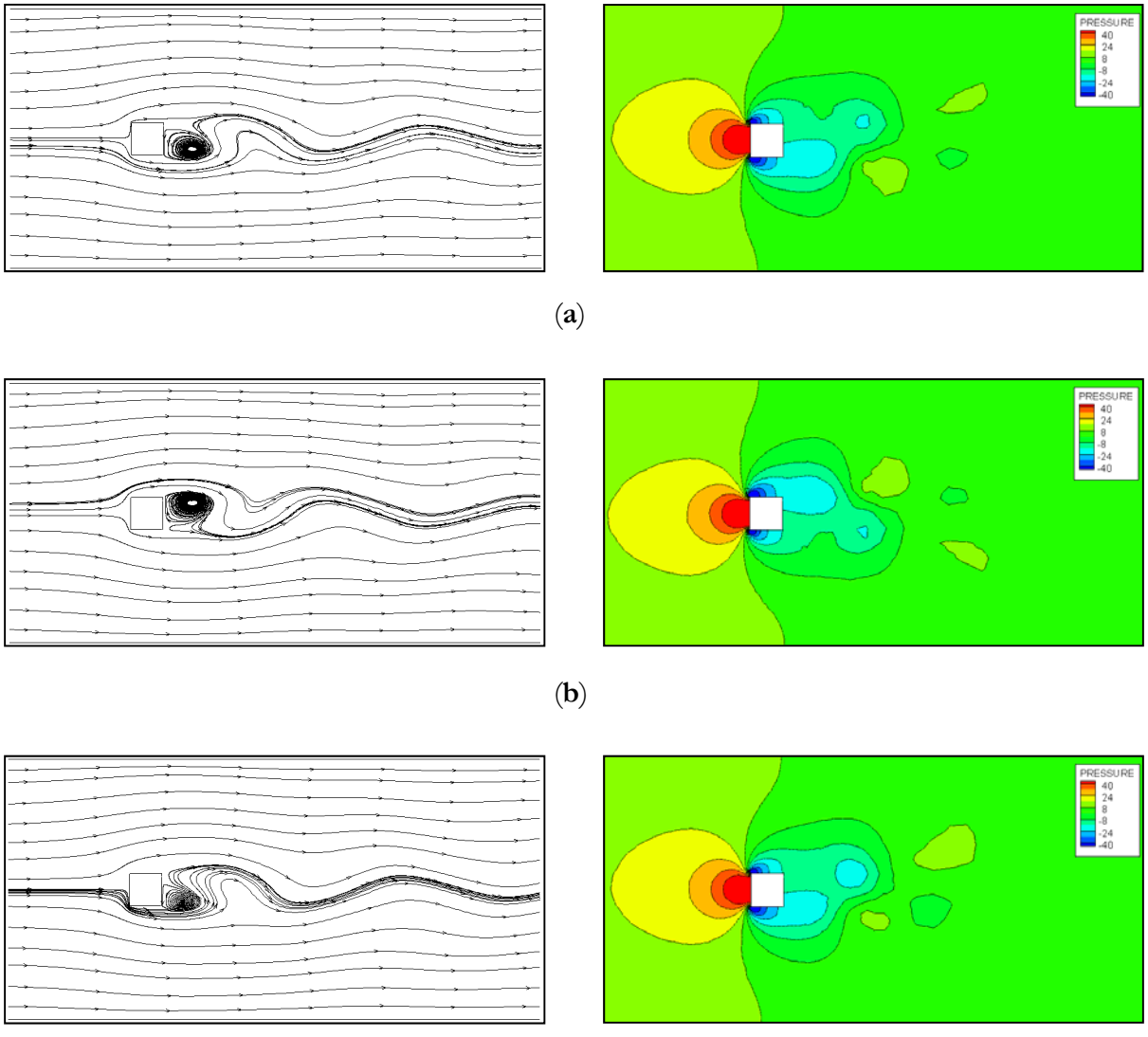


Figure 3.3 Streamlines and pressure contour plots for *Re* = 100 at (**a**) t = 401 s (**b**) t = 405 s (**c**) t = 408 s

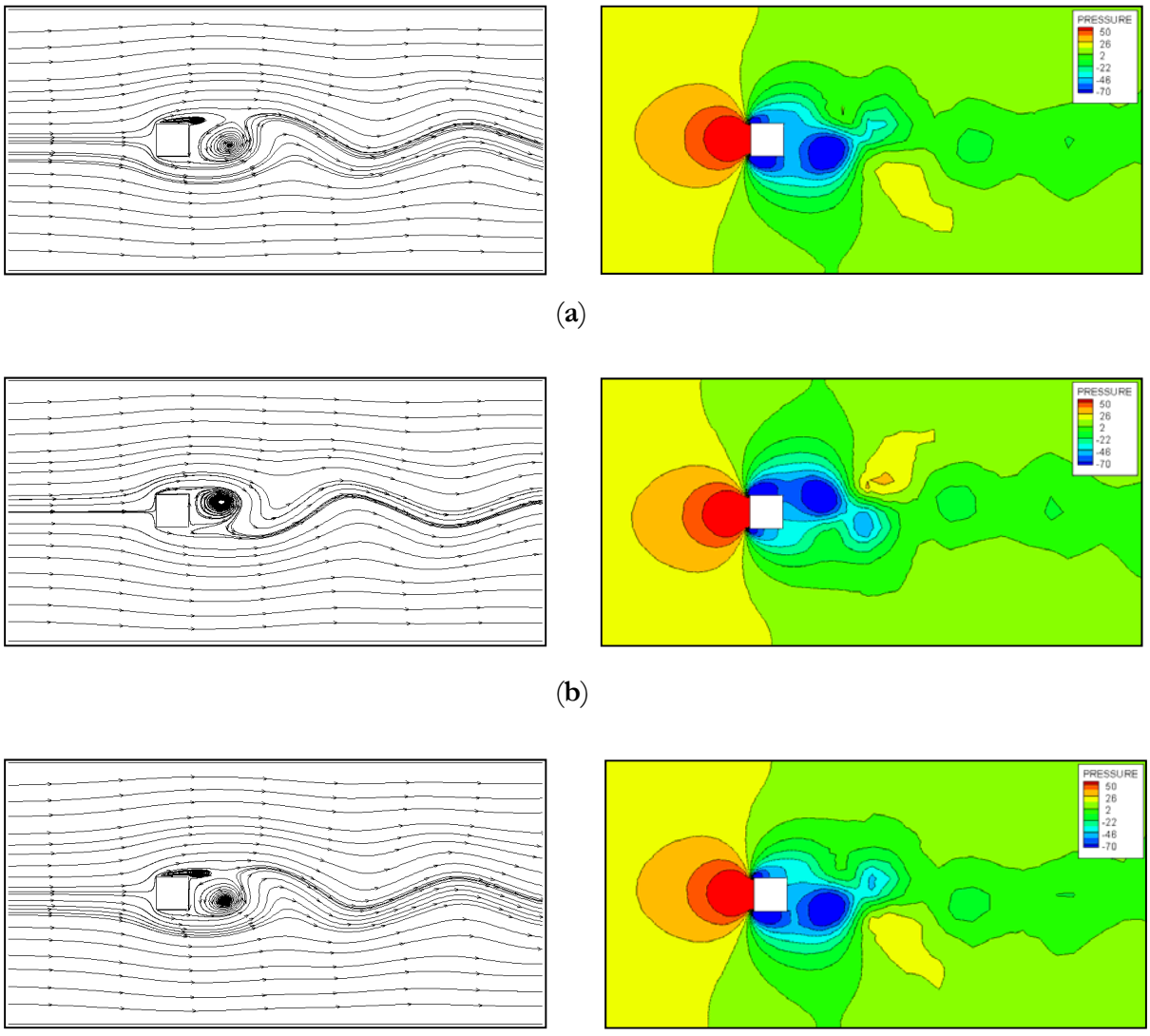


Figure 3.4 Streamlines and pressure contour plots for *Re* = 200 at (**a**) t = 284 s (**b**) t = 287 s (**c**) t = 291 s

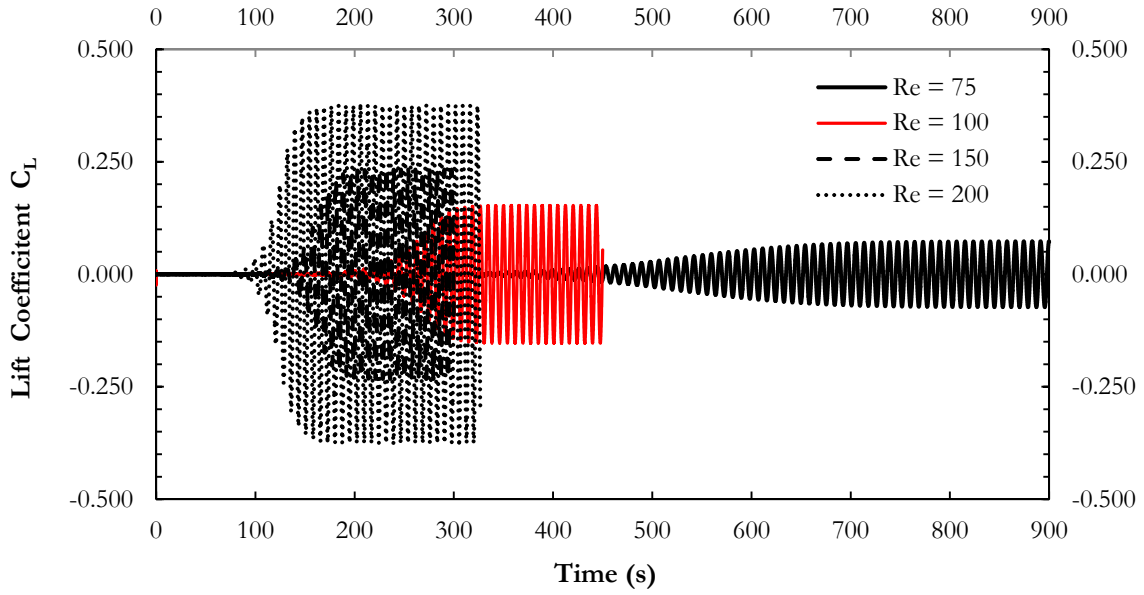


Figure 3.5 Lift coefficient from different *Reynolds* number

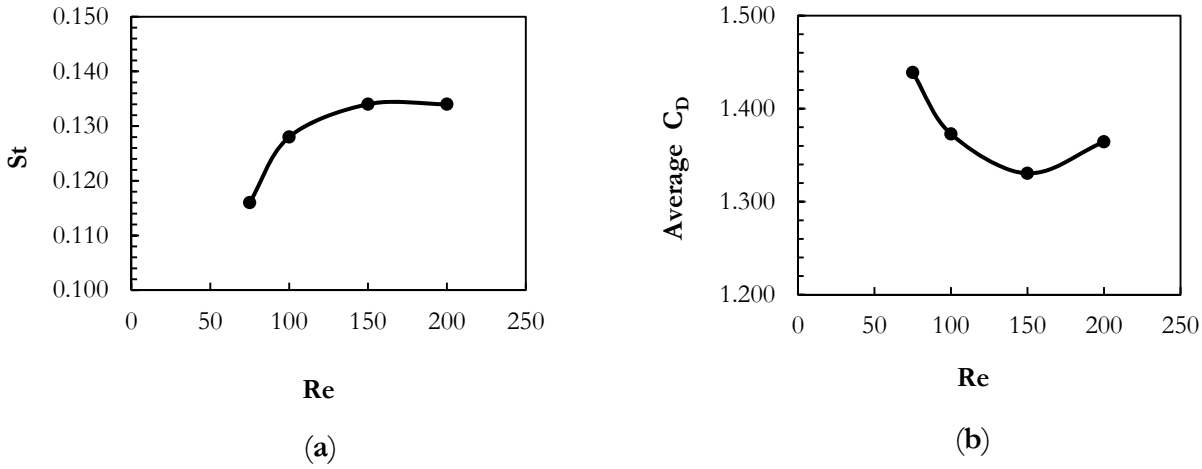


Figure 3.6 (**a**) *Strouhal* number vs *Reynolds* number (**b**) Average C_D vs *Reynolds* number

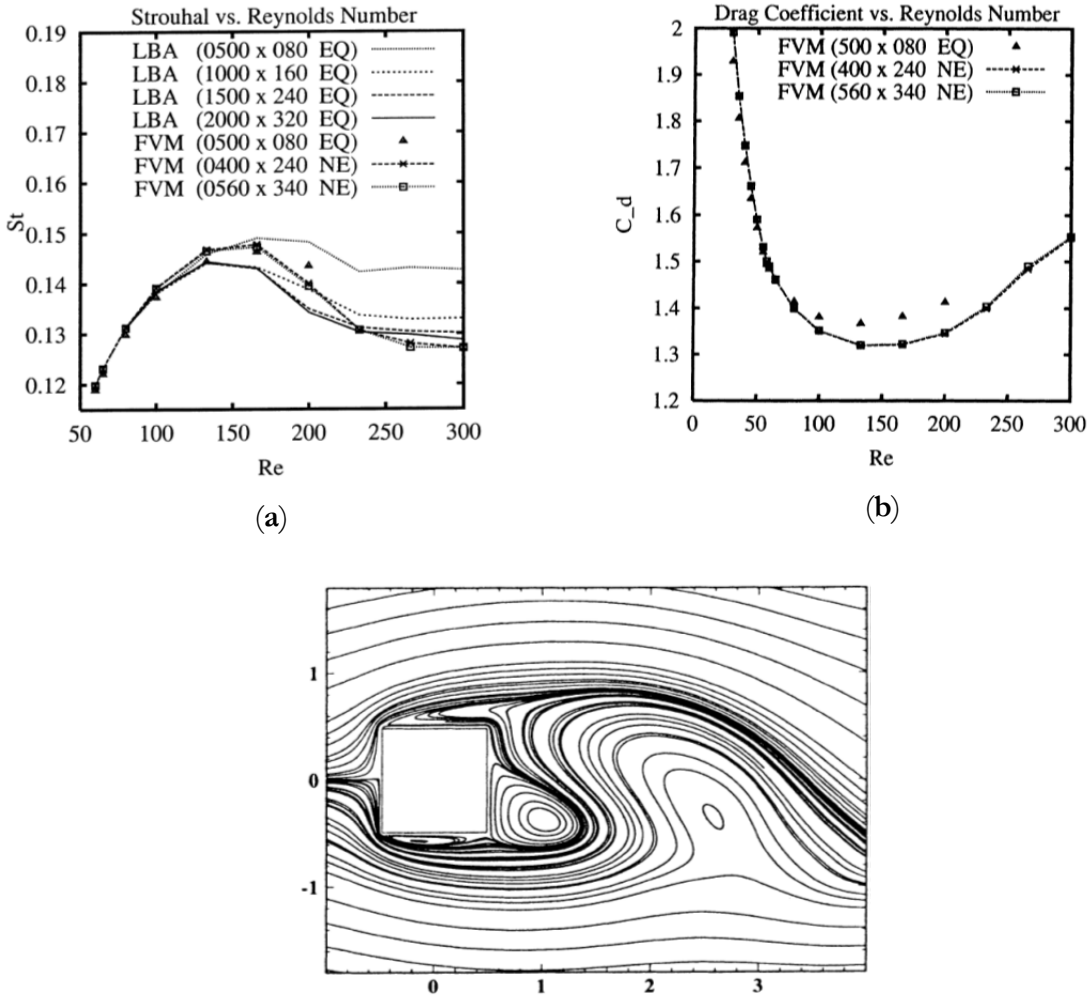


Figure 3.7 Results from [17] (**a**) *Strouhal* number vs *Reynolds* number (**b**) Average C_D vs *Reynolds* number (**c**) Streamlines for Re = 200

4 Computational Structure Dynamics (CSD) Solver

4.1 Introduction

The CSD solver used in this thesis is Zorglib CSD solver developed by Prof. Laurent Stainier in Institut de Recherche en Genie Civil et Mecanique (GeM), Ecole Centrale de Nantes. Zorglib is currently used to develop and test new constitutive models and algorithms in computational solid mechanics. It contains a library of various constitutive models, discrete formulations, and time-integration algorithms for coupled problems in general. It is based on object oriented modular software architecture to provide flexibility and compatibility with external software where for instance a constitutive model library can be plugged into other solvers. The algorithms available in Zorglib can be used for discrete systems of implicit or explicit dynamics and non-linear quasi-stationary problems. The finite element formulations are mostly for volume elements but also for linear shells and mixed boundary conditions. The constitutive models available in Zorglib are:

- Small and finite strain elasticity
- Visco-elasticity
- Visco-plasticity
- Damage model
- Thermo-mechanical model

4.2 Governing Equations

In the case of pure mechanical problem, the Zorglib solver uses finite element method to discretize the following linear momentum equation which if expressed in the current configuration using the material coordinates:

$$\rho \frac{\partial^2 u_i}{\partial t} = \frac{\partial \sigma_{ij}}{\partial x_j} + \rho f_i^b \tag{4.1}$$

where u_i is the displacement vector, σ_{ij} is the Cauchy stress tensor, ρ is the density, and f_i^b is the body force vector. For stationary problems where the inertial effect can be neglected, the

acceleration term $\frac{\partial^2 u_i}{\partial t}$ can be neglected. A proper constitutive equation, which relates the stress and strain, needs to be defined to complete the problem definition. The constitutive equation depends on the material behavior under consideration.

4.3 Linear Elasticity

In linear elasticity model, the assumption taken is that materials undergo small deformation when subjected to applied forces and when the forces are removed the materials will return to their initial shapes. This implies that the current stress at a point depends only of the current strain at the point and not the past history of strain rates at the point [1]. The strain is assumed to be infinitesimal and related to the stress by the following generalized Hooke's law formulation:

$$\sigma_{ij} = \mathbb{C}_{ijkl}\, \varepsilon_{ij} \tag{4.2}$$

where σ_{ij} is the *Cauchy* stress tensor, ε_{ij} is the infinitesimal strain tensor, and $\mathbb{C}_{ijkl}$ is well known to be the fourth order Hooke's tensor and due to symmetries can be represented by 21 scalar components. This results in the representation of Hooke's tensor by 6 x 6 symmetric tensor and often called the stiffness tensor. The expanded form of **Eq.** (4.2) is the following:

$$\begin{Bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{13} \\ \sigma_{12} \end{Bmatrix} = \begin{bmatrix} C_{11} & C_{12} & C_{13} & C_{14} & C_{15} & C_{16} \\ & C_{22} & C_{23} & C_{24} & C_{25} & C_{26} \\ & & C_{33} & C_{34} & C_{35} & C_{36} \\ & sym. & & C_{44} & C_{45} & C_{46} \\ & & & & C_{55} & C_{56} \\ & & & & & C_{66} \end{bmatrix} \begin{Bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{33} \\ 2\varepsilon_{23} \\ 2\varepsilon_{13} \\ 2\varepsilon_{12} \end{Bmatrix} \tag{4.3}$$

The above formulation takes into account the anisotropy property of the material where its properties are not the same in all the three principal directions. In the case of isotropic material the properties are assumed to be the same in all directions so $\mathbb{C}_{ijkl}$ can be simplified as:

$$\mathbb{C}_{ijkl} = \lambda \delta_{ij}\delta_{kl} + \mu\left(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}\right) \tag{4.4}$$

where λ and μ are called the *Lamé constants* and in practice are represented by the following constants that can be measured physically:

$$\mu = \frac{E}{2(1+\upsilon)} \tag{4.5}$$

$$\lambda = \frac{\upsilon\, E}{(1+\upsilon)(1-2\upsilon)} \tag{4.6}$$

where E is the Young's modulus and υ is the Poisson's ratio. Using these two constants, the expanded form of **Eq.** (4.2) for isotropic materials can be formulated as:

$$\begin{Bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{13} \\ \sigma_{12} \end{Bmatrix} = \frac{E}{(1+\upsilon)(1-2\upsilon)} \begin{bmatrix} 1-\upsilon & \upsilon & \upsilon & 0 & 0 & 0 \\ & 1-\upsilon & \upsilon & 0 & 0 & 0 \\ & & 1-\upsilon & 0 & 0 & 0 \\ & sym. & & 1-2\upsilon & 0 & 0 \\ & & & & 1-2\upsilon & 0 \\ & & & & & 1-2\upsilon \end{bmatrix} \begin{Bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{33} \\ \varepsilon_{23} \\ \varepsilon_{13} \\ \varepsilon_{12} \end{Bmatrix} \tag{4.7}$$

4.4 Hyperelasticity

The knowledge of hyperelasticity in this section is synthesized from [2]. When the material deformation is considered to be large but still able to return to its initial shape after the applied forces are removed, the linear elasticity model cannot be used anymore. Instead a hyperelastic material behavior has to be enforced. In the context of hyperelasticity, the strain is a finite strain and in *Lagrangian* description can defined as the *Green-Lagrange* strain tensor:

$$\boldsymbol{E} = \frac{1}{2}(\boldsymbol{F}^T \cdot \boldsymbol{F} - \boldsymbol{I}) \tag{4.8}$$

$$\boldsymbol{F} = \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{X}} \tag{4.9}$$

where $\boldsymbol{F}$ is the deformation gradient tensor which transform the vector in the reference configuration $\boldsymbol{X}$ to its corresponding current configuration $\boldsymbol{x}$. It is also necessary to introduce the following definitions:

$$\boldsymbol{C} = \boldsymbol{F}^T \boldsymbol{F} \tag{4.10}$$

$$\boldsymbol{b} = \boldsymbol{F}\,\boldsymbol{F}^T \tag{4.11}$$

$$J = det\,\boldsymbol{F} = (det\,\boldsymbol{C})^{\boldsymbol{1/2}} \tag{4.12}$$

$$\boldsymbol{P} = J\,\boldsymbol{F}^{-1} \cdot \boldsymbol{\sigma} \tag{4.13}$$

$$\boldsymbol{S} = \boldsymbol{P} \cdot \boldsymbol{F}^{-T} = J\,\boldsymbol{F}^{-1} \cdot \boldsymbol{\sigma} \cdot \boldsymbol{F}^{-T} \tag{4.14}$$

$$\boldsymbol{\sigma} = J^{-1}\boldsymbol{F} \cdot \boldsymbol{P} = J^{-1}\boldsymbol{F} \cdot \boldsymbol{S} \cdot \boldsymbol{F}^T \tag{4.15}$$

where $\boldsymbol{C}$ is the *right Cauchy-Green* deformation tensor, $\boldsymbol{b}$ is the *left Cauchy-Green* deformation tensor, J is the *Jacobian*, $\boldsymbol{P}$ is the *first Piola-Kirchhoff* stress tensor, $\boldsymbol{S}$ is the *second Piola-Kirchhoff* stress tensor, and $\boldsymbol{\sigma}$ is the *Cauchy* stress tensor. To derive the constitutive strain-stress relation of hyperelastic materials, firstly it is defined that a strain energy function of hyperelastic materials, $\boldsymbol{\Psi}$, is related to the *Piola-Kirchhoff* stresses using the following formulas:

$$\boldsymbol{P}(\boldsymbol{F}(\boldsymbol{X}),\boldsymbol{X}) = \frac{\partial \boldsymbol{\Psi}(\boldsymbol{F}(\boldsymbol{X}),\boldsymbol{X})}{\partial \boldsymbol{F}} \tag{4.16}$$

$$\boldsymbol{S}(\boldsymbol{C}(\boldsymbol{X}),\boldsymbol{X}) = 2\frac{\partial \boldsymbol{\Psi}(\boldsymbol{C}(\boldsymbol{X}),\boldsymbol{X})}{\partial \boldsymbol{C}} = \frac{\partial \boldsymbol{\Psi}(\boldsymbol{C}(\boldsymbol{X}),\boldsymbol{X})}{\partial \boldsymbol{E}} \tag{4.17}$$

For isotropic hyperelasticity, $\boldsymbol{\Psi}$ is only a function of the invariants of $\boldsymbol{C}$:

$$\boldsymbol{\Psi}(\boldsymbol{C}) = \boldsymbol{\Psi}(I_1, I_2, I_3) \tag{4.18}$$

$$I_1 = \operatorname{tr} \boldsymbol{C} \tag{4.19}$$

$$I_2 = (tr\ \boldsymbol{C})\ \boldsymbol{C} \tag{4.20}$$

$$I_3 = \det \boldsymbol{C} = J^2 \tag{4.21}$$

and using this constraint, the *second Piola-Kirchhoff* stress tensor now can be formulated as:

$$\boldsymbol{S} = 2\frac{\partial \boldsymbol{\Psi}}{\partial \boldsymbol{C}} = 2\frac{\partial \boldsymbol{\Psi}}{\partial I_1}\,\boldsymbol{I} + 4\frac{\partial \boldsymbol{\Psi}}{\partial I_2}\,\boldsymbol{C} + 2J^2\frac{\partial \boldsymbol{\Psi}}{\partial I_3}\boldsymbol{C}^{-1} \tag{4.22}$$

A particular hyperelastic material model that will be used in the current fluid-structure interaction simulation is the compressible *Neo-Hookean* material model. The strain energy function for this material model is defined as:

$$\begin{aligned}\boldsymbol{\Psi} &= \frac{\mu}{2}(I_1 - 3) - \mu \ln J + \frac{\lambda}{2}(\ln J)^2 \\ &= \frac{\mu}{2}(I_1 - 3) - \mu \ln I_3^{1/2} + \frac{\lambda}{2}\left(\ln I_3^{1/2}\right)^2\end{aligned} \tag{4.23}$$

where λ and μ are *Lamé constants* as defined the linear elasticity model. Using **Eq.** (4.22) the *second Piola-Kirchhoff* stress tensor for *Neo-Hookean* material model is defined as:

$$\begin{aligned}\boldsymbol{S} &= 2\frac{\partial \boldsymbol{\Psi}}{\partial I_1}\,\boldsymbol{I} + 4\frac{\partial \boldsymbol{\Psi}}{\partial I_2}\,\boldsymbol{C} + 2J^2\frac{\partial \boldsymbol{\Psi}}{\partial I_3}\boldsymbol{C}^{-1} \\ &= \mu\,\boldsymbol{I} + 2\,I_3\,\boldsymbol{C}^{-1}\left(\frac{\lambda \ln I_3^{1/2}}{2I_3} - \frac{\mu}{2I_3}\right) \\ &= \mu(\boldsymbol{I} - \boldsymbol{C}^{-1}) + \lambda \ln I_3^{1/2}\,\boldsymbol{C}^{-1} = \mu(\boldsymbol{I} - \boldsymbol{C}^{-1}) + \lambda\,(\ln J)\,\boldsymbol{C}^{-1}\end{aligned} \tag{4.24}$$

The *Cauchy* stress tensor for the *Neo-Hookean* material model can be obtained by combining **Eq.** (4.24) and (4.15):

$$\boldsymbol{\sigma} = \frac{\mu}{J}(\boldsymbol{b} - \boldsymbol{I}) + \frac{\lambda}{J}(\ln J)\,\boldsymbol{I} \tag{4.25}$$

4.5 Finite Element Discretization

This section is synthesized from [3] and [4]. The finite element formulation is derived by developing the weak form of **Eq.** (4.1). from the principle of virtual work which states that the virtual kinetic work plus the external virtual work is equal to the internal virtual work:

$$\delta W_{kinetic} + \delta W_{internal} = \delta W_{external} \tag{4.26}$$

For infinitesimal strain, **Eq.** (4.26) is expanded as follows:

$$\int_V \delta \boldsymbol{u}^T \rho \frac{\partial^2 \boldsymbol{u}}{\partial t} dV + \int_V \boldsymbol{\sigma} : \delta \boldsymbol{\varepsilon}\, dV = \int_V \delta \boldsymbol{u}^T \rho \boldsymbol{f}^b dV + \int_S \delta \boldsymbol{u}^T \boldsymbol{t}^s dS \tag{4.27}$$

where $\delta \boldsymbol{u}$ is the virtual displacement vector, $\delta \boldsymbol{\varepsilon}$ is the virtual strain tensor, $\boldsymbol{f}^b$ is the body force vector, and $\boldsymbol{t}^s$ is the surface traction vector. By imposing the equilibrium equations of the principal of virtual work to each element and assemble it for the all the finite elements, the following formula is obtained for linear elasticity finite element formulation:

$$\left[\sum_e \int_{V^e} \rho^e \boldsymbol{N}^{eT} \boldsymbol{N}^e \, dV^e\right] \hat{\ddot{\boldsymbol{u}}} + \left[\sum_e \int_{V^e} \boldsymbol{B}^{eT} \mathbb{C}^e \boldsymbol{B}^e \, dV^e\right] \hat{\boldsymbol{u}}$$
$$= \sum_e \int_{V^e} \rho^e \boldsymbol{N}^{eT} \boldsymbol{f}^{be} \, dV^e + \sum_e \int_{S^e} \boldsymbol{N}^{eT} \boldsymbol{t}^{se} \, dS^e \tag{4.28}$$

where $\boldsymbol{N}^e$ is the element displacement interpolation matrix, $\boldsymbol{B}^e$ is the element strain-displacement matrix, $\hat{\boldsymbol{u}}$ is the nodal displacement vector, and $\hat{\ddot{\boldsymbol{u}}}$ is the nodal acceleration vector.

In the case of hyperelasticity, the definition is slightly different because the state of the domain configuration has to be taken into account. Using the *Total Lagrangian Formulation*, the final form of the discretized virtual work balance is the following:

$$\left[\sum_e \int_{V_0^e} \rho_0 \boldsymbol{N}^{eT} \boldsymbol{N}^e \, dV_0^e\right] \hat{\ddot{\boldsymbol{u}}} + \sum_e \int_{V_0^e} \frac{\partial \boldsymbol{N}^{eT}}{\partial \boldsymbol{X}} \boldsymbol{P} \, dV_0^e$$
$$= \sum_e \int_{V_0^e} \rho_0 \boldsymbol{N}^{eT} \boldsymbol{f}^{be} \, dV_0^e + \sum_e \int_{S_0^e} \boldsymbol{N}^{eT} \boldsymbol{t}^{se} \, dS_0^e \tag{4.29}$$

where the subscript 0 refers to the reference configuration and $\boldsymbol{X}$ is the material coordinates. The two formulations above can be simplified into structural dynamics finite element formulation without velocity-dependent damping forces as the following:

$$\boldsymbol{M}\,\hat{\ddot{\boldsymbol{u}}} + \mathrm{F}^{\boldsymbol{int}} = \mathrm{F}^{\boldsymbol{ext}} \tag{4.30}$$

where $\boldsymbol{M}$ is the mass matrix, $\mathrm{F}^{\boldsymbol{ext}}$ is the externally applied force vector, and $\mathrm{F}^{\boldsymbol{int}}$ is the internal force vector. For linear elasticity the resulting system of equations is linear whereas in the case of hyperelasticity is non-linear.

4.5 Two-dimensional Plane Stress and Plane Strain Elements

In the subsequent two-dimensional finite element analysis, either the plane strain or plane stress states will be used. In the plane strain state, it is assumed that the solid body is long enough in the third direction so that the displacement and strain components in that direction can be omitted. In other words, in a linear elastic case:

$$\varepsilon_{33} = \varepsilon_{13} = \varepsilon_{23} = 0 \tag{4.31}$$

$$\begin{Bmatrix}\sigma_{11}\\ \sigma_{22}\\ \sigma_{12}\end{Bmatrix} = \frac{E}{(1+\upsilon)(1-2\upsilon)}\begin{bmatrix}1-\upsilon & \upsilon & 0\\ \upsilon & 1-\upsilon & 0\\ 0 & 0 & 1-2\upsilon\end{bmatrix}\begin{Bmatrix}\varepsilon_{11}\\ \varepsilon_{22}\\ \varepsilon_{12}\end{Bmatrix} \tag{4.32}$$

In the case of plane stress state, it is assumed that the strain and stress are uniform through the midplane of the body and the normal and shear stress components in the third direction can be omitted:

$$\sigma_{33} = \sigma_{13} = \sigma_{23} = 0 \tag{4.33}$$

$$\begin{Bmatrix}\sigma_{11}\\ \sigma_{22}\\ \sigma_{12}\end{Bmatrix} = \frac{E}{1-\upsilon^2}\begin{bmatrix}1 & \upsilon & 0\\ \upsilon & 1 & 0\\ 0 & 0 & 1-\upsilon\end{bmatrix}\begin{Bmatrix}\varepsilon_{11}\\ \varepsilon_{22}\\ \varepsilon_{12}\end{Bmatrix} \tag{4.34}$$

4.6 Generalized-α Time Integration Method

This section is synthesized from [5]. The time integration used for the uncoupled structural dynamics and coupled fluid-structure simulations in Zorglib is the implicit *Generalized-*α method. This method is a one-step, three-stage time integration algorithms that optimizes the high and low frequency dissipation which is controlled by a set of constants. Depending on the proper selection of constants, this method can recover the Newmark family of time integration algorithms.

It is a one-step method because the solution at time t_{n+1} depends only on the solution at time t_n and the three-stage refers to the fact that the method obtains three solution vectors i.e.

displacement, velocity, and acceleration vectors. The problem of structural dynamics can be formulated as the following:

$$\boldsymbol{M}\,\ddot{\boldsymbol{u}} + \boldsymbol{C}\dot{\boldsymbol{u}} + \boldsymbol{K}\boldsymbol{u} = \mathrm{F}^{\boldsymbol{ext}} \tag{4.35}$$

where $\boldsymbol{M}$, $\boldsymbol{C}$, and $\boldsymbol{K}$ are the mass, damping, and stiffness matrix respectively. $\ddot{\boldsymbol{u}}$, $\dot{\boldsymbol{u}}$, and $\boldsymbol{u}$ are the acceleration, velocity, and displacement vectors respectively. $\mathrm{F}^{\boldsymbol{ext}}$ is the applied external force vector which depends on time. The formulation of the *Generalized-*α method is the following:

$$\boldsymbol{u}_{n+1} = \boldsymbol{u}_n + \Delta t\,\dot{\boldsymbol{u}}_n + \Delta t^2\left(\left(\frac{1}{2}-\beta\right)\ddot{\boldsymbol{u}}_n + \beta\,\ddot{\boldsymbol{u}}_{n+1}\right) \tag{4.36}$$

$$\dot{\boldsymbol{u}}_{n+1} = \dot{\boldsymbol{u}}_n + \Delta t\,\left(\left(\frac{1}{2}-\gamma\right)\ddot{\boldsymbol{u}}_n + \gamma\,\ddot{\boldsymbol{u}}_{n+1}\right) \tag{4.37}$$

$$\boldsymbol{M}\,\ddot{\boldsymbol{u}}_{n+1-\alpha_m} + \boldsymbol{C}\,\dot{\boldsymbol{u}}_{n+1-\alpha_f} + \boldsymbol{K}\,\boldsymbol{u}_{n+1-\alpha_f} = \mathrm{F}^{\boldsymbol{ext}}\left(t_{n+1-\alpha_f}\right) \tag{4.38}$$

and for the definition of the three solution vectors and time with *alpha* parameters:

$$\boldsymbol{u}_{n+1-\alpha_f} = \left(1-\alpha_f\right)\boldsymbol{u}_{n+1} + \alpha_f\,\boldsymbol{u}_n \tag{4.39}$$

$$\dot{\boldsymbol{u}}_{n+1-\alpha_f} = \left(1-\alpha_f\right)\dot{\boldsymbol{u}}_{n+1} + \alpha_f\,\dot{\boldsymbol{u}}_n \tag{4.40}$$

$$\ddot{\boldsymbol{u}}_{n+1-\alpha_m} = \left(1-\alpha_m\right)\ddot{\boldsymbol{u}}_{n+1} + \alpha_m\,\ddot{\boldsymbol{u}}_n \tag{4.41}$$

$$t_{n+1-\alpha_f} = \left(1-\alpha_f\right)t_{n+1} + \alpha_f\,t_n \tag{4.42}$$

The initial conditions are defined as:

$$\boldsymbol{u}_0 = \boldsymbol{u}(0) \tag{4.43}$$

$$\dot{\boldsymbol{u}}_0 = \dot{\boldsymbol{u}}(0) \tag{4.44}$$

$$\ddot{\boldsymbol{u}}_0 = \boldsymbol{M}^{-1}\left(\mathrm{F}^{\boldsymbol{ext}}(0) - \boldsymbol{C}\,\dot{\boldsymbol{u}}(0) - \boldsymbol{K}\,\boldsymbol{u}(0)\right) \tag{4.45}$$

As can be seen from the above formulations if the *alpha* parameters $\alpha_f = \alpha_m = 0$, the Trapezoidal Newmark method is recovered. If $\alpha_m = 0$, the Hilbert Hughes Taylor - α (HHT-α) method is recovered, and lastly if $\alpha_f = 0$, the Wood Bossak Zienkiewiecz - α (WBZ- α) method is recovered.

The *Generalized-*α method is second-order accurate and achieves optimal high-frequency dissipation when:

$$\gamma = \frac{1}{2} - \alpha_m + \alpha_f \tag{4.46}$$

$$\beta = \frac{1}{4}\left(1 - \alpha_m + \alpha_f\right)^2 \tag{4.47}$$

An optimal low-frequency dissipation is obtained when:

$$\alpha_m = \frac{2\rho_\infty - 1}{\rho_\infty + 1} \tag{4.48}$$

$$\alpha_f = \frac{\rho_\infty}{\rho_\infty + 1} \tag{4.49}$$

where ρ_∞ is the spectral radius with a range of [0, 1].

4.7 Newton-Raphson Nonlinear Solver

This section is synthesized from [3]. To solve the hyperelasticity problem, a nonlinear system of equations is solved using Newton-Raphson iterative solver in Zorglib at time $t + \Delta t$ using the following formulation in terms of the computational residual vector:

$$\boldsymbol{R}_{n+1} = \boldsymbol{M}\,\ddot{\boldsymbol{u}}_{n+1} + \mathrm{F}_{n+1}^{int} - \mathrm{F}_{n+1}^{ext} \tag{4.50}$$

$$\boldsymbol{R}_{n+1}^{i} = \boldsymbol{R}_{n+1}^{i-1} + \left.\frac{\partial \boldsymbol{R}_{n+1}}{\partial \boldsymbol{u}_{n+1}}\right|^{i-1} \Delta \boldsymbol{u}_{n+1}^{i} = \boldsymbol{0} \tag{4.51}$$

$$\boldsymbol{K}_{n+1}^{T\ \ i-1}\ \Delta \boldsymbol{u}_{n+1}^{i} = -\,\boldsymbol{R}_{n+1}^{i-1} \tag{4.52}$$

$$\Delta \boldsymbol{u}_{n+1}^{i} = \boldsymbol{u}_{n+1}^{i} - \boldsymbol{u}_{n+1}^{i-1} \tag{4.53}$$

$$\boldsymbol{u}_{n+1}^{0} = \boldsymbol{u}_{n} \tag{4.54}$$

where $\boldsymbol{K}^T$ is the tangent stiffness matrix and the solution at iteration $i - 1$ is known. The objective is to iterate until the residual is less or equal to a specified tolerance. The formulation of the tangent stiffness matrix is a sort of linearization of the balance and constitutive equation.

4.8 Mesh Sensitivity Analysis

It is important to perform a mesh sensitivity analysis to check the convergence of the finite element analysis compared to an analytical solution in order to ensure that the element type and size chosen will produce sufficiently converged results in the FSI simulation. The sensitivity analysis is performed on four different elements i.e. linear triangle (three nodes element), quadratic triangle (six nodes element), linear quadrilateral (four nodes element), and quadratic

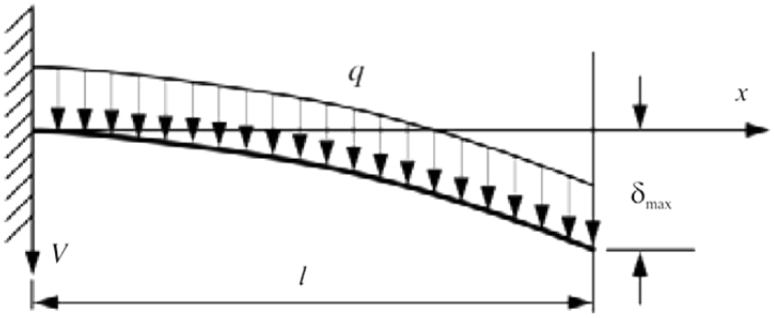
quadrilateral (nine nodes element). Three different mesh sizes are analysed for each element. The solid mesh is generated using GMSH software [11].

An analytical solution of the deflection of a cantilever beam is chosen in the following sensitivity analysis because the FSI simulations will be performed on a cantilever solid body. It is well known in many *Mechanics of Materials* textbook that an analytical solution of the deflection of a cantilever beam is formulated in the following way (see **Figure 4.1**):

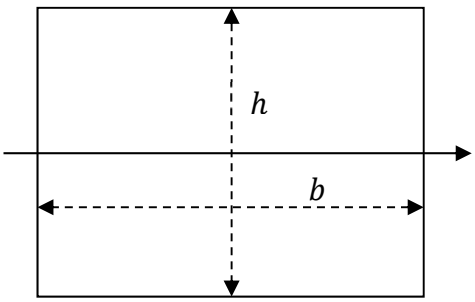
$$V(x) = \frac{qx^2}{24EI}\left[\boldsymbol{x^2} + 6l^2 - 4lx\right] \tag{4.55}$$

$$\delta_{max} = \frac{ql^4}{8EI} \tag{4.56}$$

$$I = \frac{bh^3}{12} \tag{4.57}$$



(**a**)



(**b**)

Figure 4.1 (**a**) The deflection of a cantilever beam (**b**) The cross-sectional area of the beam

where V is the deflection of the beam as a function of the coordinate in x direction, q is the uniformly distributed load, l is the length of the beam, E is the Young's modulus, and I is the area moment of inertia.

The finite element simulation setup in Zorglib and the dimension are the following:

- Two-dimensional static analysis
- Isotropic elastic material; E = 3.5 E+6 MPa; ν = 0.32
- Plane stress assumption (thickness = b = 1.0 m)
- $q = 10\ N/m$
- l = 1m; h = 0.01 m

Figure 4.2 and **4.3** compare the analytical and finite element results of the deflection between the different elements and mesh sizes. For the finite element results, the deflection values showed are the displacement in the y direction. **Table 4.1** and **4.2** summarize the comparison in details in terms of the number of nodes, relative error, and the CPU time.

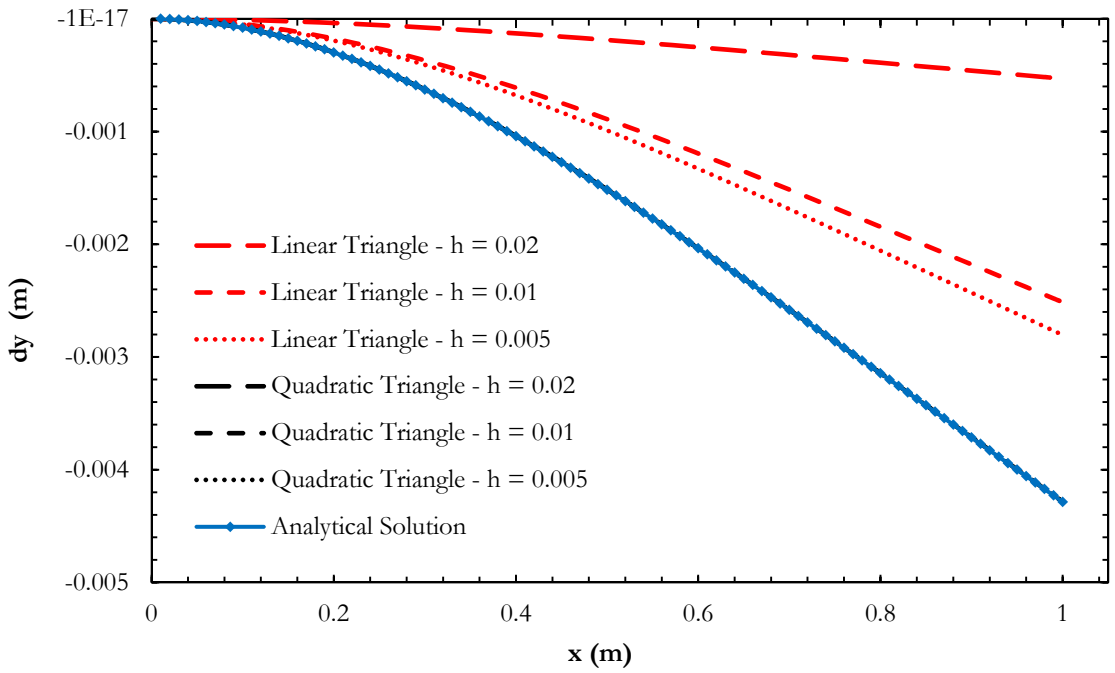


Figure 4.2 Deflection results of triangle elements for a particular mesh size and element order compared to the analytical solution of a cantilever beam

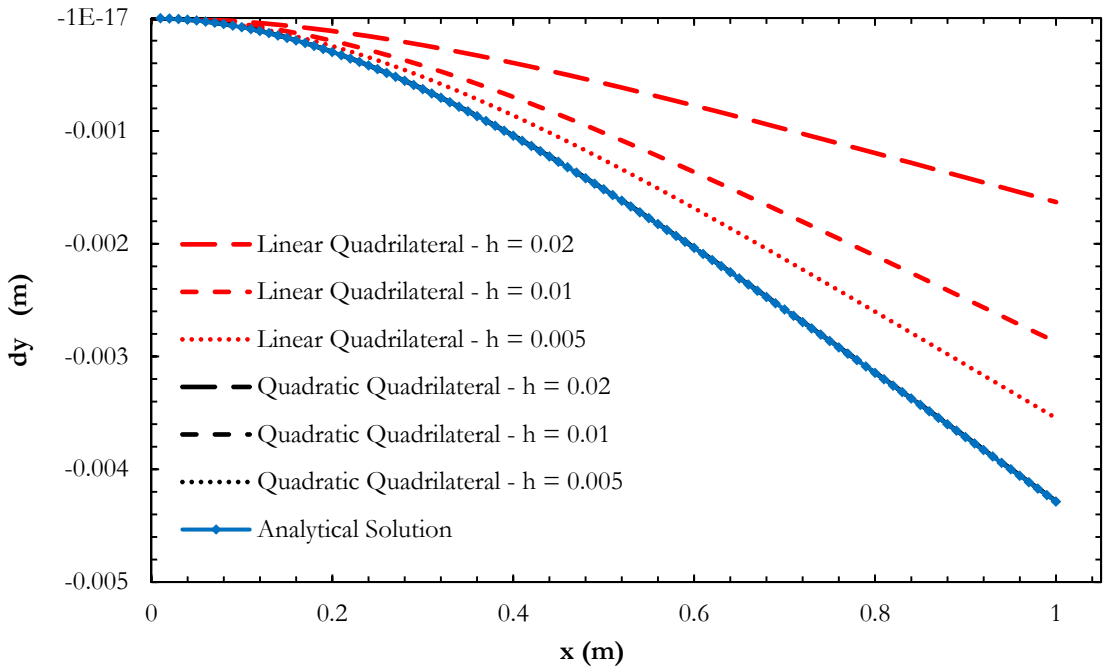


Figure 4.3 Deflection results of quadrilateral elements for a particular mesh size and element order compared to the analytical solution of a cantilever beam

	N_{nodes}	Relative Error	CPU Time (s)
Linear Triangle, h = 0.02	100	0.876	0.01
Linear Triangle, h = 0.01	299	0.413	0.01
Linear Triangle, h = 0.005	600	0.345	0.02
Quadratic Triangle, h = 0.02	297	2.19 E-3	0.06
Quadratic Triangle, h = 0.01	993	5.14 E-4	4.62
Quadratic Triangle, h = 0.005	1997	6.45 E-4	36.9

Table 4.1 Comparison of the number of nodes, relative error of the tip displacement, and CPU time between triangle elements with different mesh size and element order

	N_{nodes}	Relative Error	CPU Time (s)
Linear Quadrilateral, h = 0.02	102	0.620	0.01
Linear Quadrilateral, h = 0.01	202	0.330	0.02
Linear Quadrilateral, h = 0.005	402	0.173	0.18
Quadratic Quadrilateral, h = 0.02	303	9.88 E-4	0.13
Quadratic Quadrilateral, h = 0.01	603	4.93 E-4	0.71
Quadratic Quadrilateral, h = 0.005	1203	2.97 E-4	4.88

Table 4.2 Comparison of the number of nodes, relative error of the tip displacement, and CPU time between quadrilateral elements with different mesh size and element order

In **Figure 4.4**, the relative errors of each element are plotted against the number of nodes in a log-log scale. The quadratic elements consistently produce lower relative error compared to the linear. The triangle elements produce higher relative error compared to the quadrilateral elements of the same order. The number of nodes and CPU time of the triangle elements are considerably higher as well in particular when $h > 0.02$. **Figure 4.5** shows the displacement and Von Mises contour plots of the computation using the quadratic quadrilateral with $h = 0.01$.

In the following simulations the quadratic quadrilateral elements will be used for the CSD solver to ensure that the convergence of the CSD results can be achieved sufficiently.

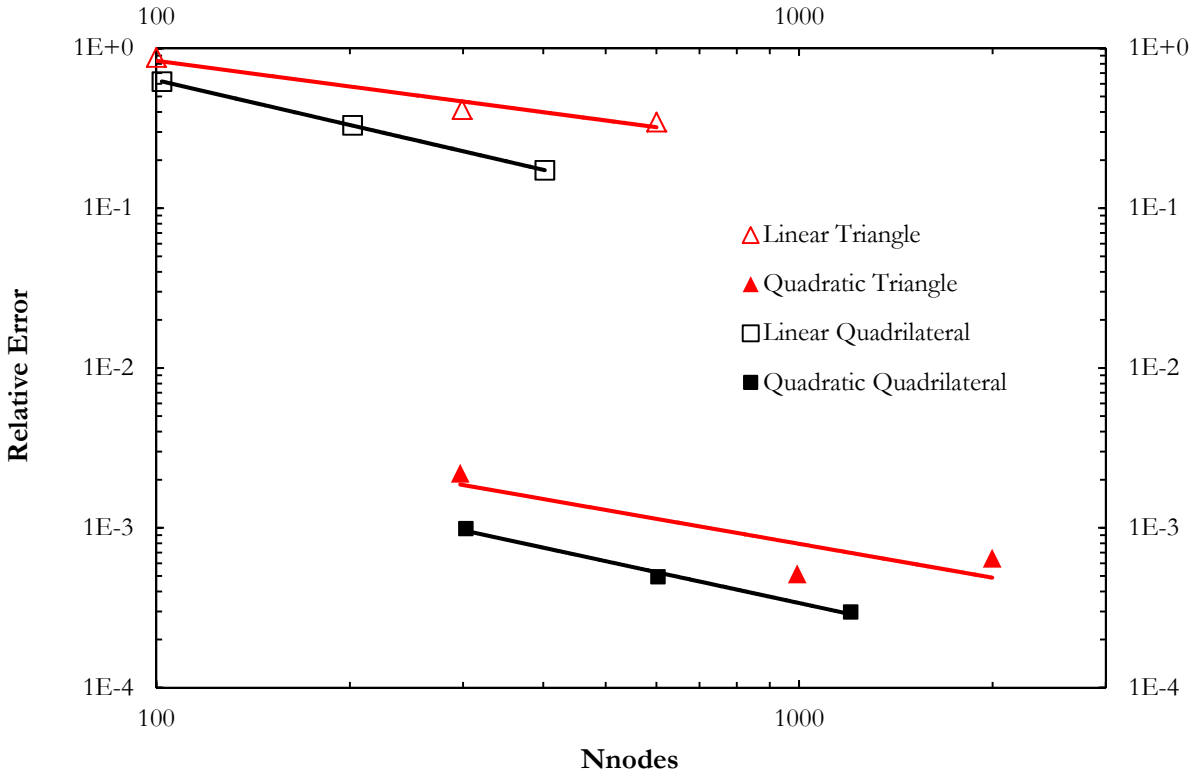


Figure 4.4 Relative error of the tip displacements for all the elements

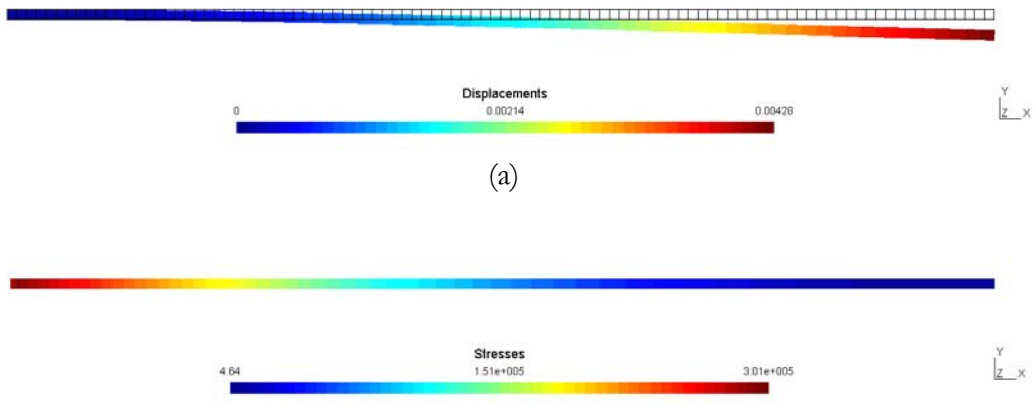


Figure 4.5 (a) Displacement (m) contour plot of the quadratic quadrilateral with $h = 0.01$ (b) Von Mises stress (Pa) contour plot of the quadratic quadrilateral with $h = 0.01$

4.9 Numerical Tests

To test the CSD solver, the CSM benchmark tests performed by [12] is referred to. The CSM tests are performed to validate the solid solver as part of the two-dimensional FSI benchmarking on incompressible laminar flow of Newtonian fluid and compressible hyperelastic solid. The solid domain is depicted in **Figure 4.6** in grey colour and it is attached to a fixed cylinder. This is similar to the cantilever beam case but now the applied load is a gravitational force per unit length distributed equally on each nodes, $\overrightarrow{f_g}$= (0, $\frac{\rho A g}{N_{nodes}}$) [N/m].

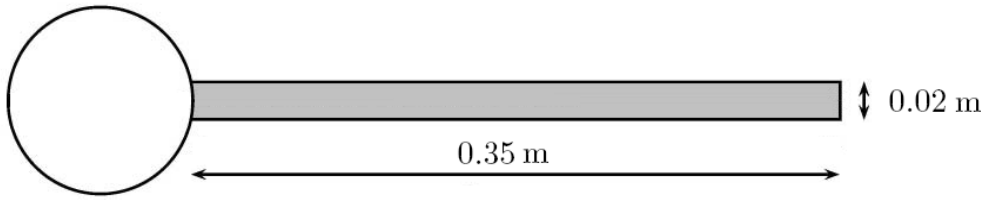


Figure 4.6 CSM test solid domain

There are three tests which are performed, namely CSM1, CSM2, and CSM3, using the following parameters:

	CSM1	**CSM2**	**CSM3**
Density, $\rho\ (kg/m^3)$	1.0 E+3	1.0 E+3	1.0 E+3
Poisson's ratio, ν	0.4	0.4	0.4
Young's Modulus, $E\ (Pa)$	1.4 E+6	5.6 E+6	1.4 E+6
Gravity, $g\ (m/s^2)$	2	2	2
Finite Element Analysis	Static	Static	Dynamic
Constitutive Model	Neohookean	Neohookean	Neohookean
2D Assumption	Plane Strain	Plane Strain	Plane Strain
Time Integrator	-	-	Trapezoidal Newmark
Time step size (s)	-	-	0.01

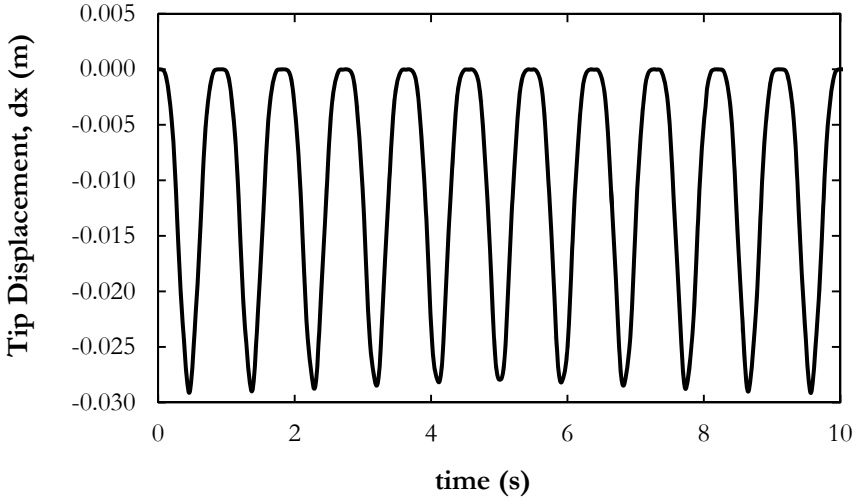
Table 4.3 Parameters for the CSM tests

The simulations use the quadratic quadrilateral element with two different meshes, *A* and *B*, where *A* has 2 x 25 elements and *B* has 2 x 50 elements. **Table 4.4** summarizes the results of the CSM tests and compared to the results obtained by [12].

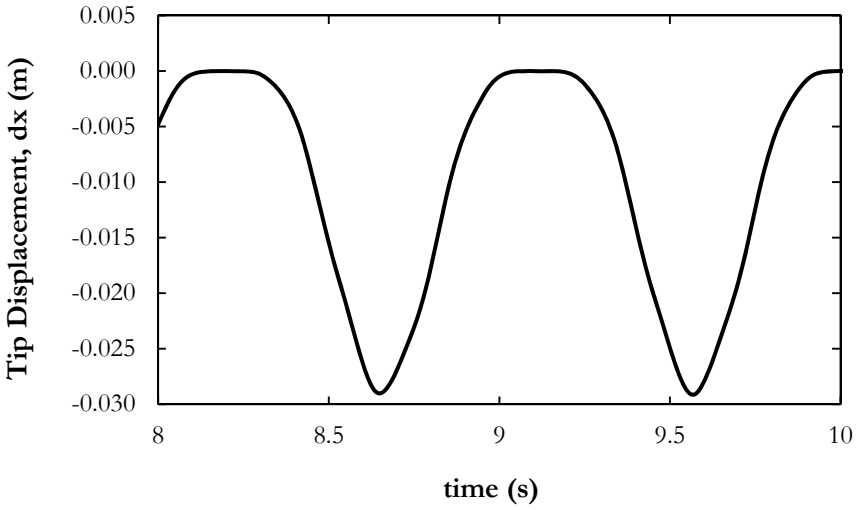
	CSM1	**CSM2**	**CSM3**
Tip Displacement, *dx* (*m*)			
A (2 x 25 elements)	-7.177 E-3	-4.684 E-4	-1.457 E-2 ± 1.457 E-2
B (2 x 50 elements)	-7.177 E-3	-4.684 E-4	-1.460 E-2 ± 1.460 E-2
Turek & Hron (2006)	-7.187 E-3	-4.690 E-4	-1.430 E-2 ± 1.430 E-2
Tip Displacement, *dy* (*m*)			
A (2 x 25 elements)	-6.610 E-2	-1.698 E-2	-6.397 E-2 ± 6.561 E-2
B (2 x 50 elements)	-6.616 E-2	-1.699 E-2	-6.400 E-2 ± 6.573 E-2
Turek & Hron (2006)	-6.610 E-2	-1.697 E-2	-6.360 E-2 ± 6.516 E-2
Frequency (Hz)			
A (2 x 25 elements)	-	-	1.0742
B (2 x 50 elements)	-	-	1.0742
Turek & Hron (2006)	-	-	1.0995

Table 4.4 Results of the CSM tests

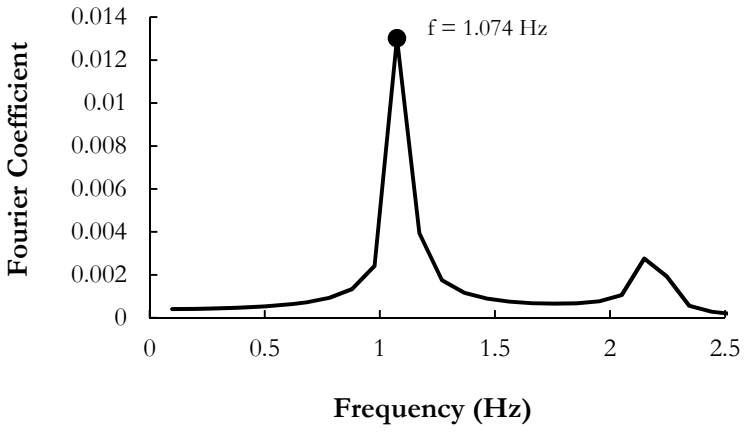
The results differences between mesh *A* and *B* are not significant and both results agree quite well with the results from [12]. The small discrepancies with [12] are most likely due to the fact that the constitutive model used is not the same where in [12] the constitutive model used is *St. Venant-Kirchhoff* model. The tip displacement and *Fast Fourier Transform* (FFT) results for CSM3 of mesh *A* are shown in **Figure 4.7** and **4.8**. **Figure 4.9** compare the tip displacements of mesh *A* and mesh *B*. **Figure 4.10** shows the displacements and stresses contour plots of mesh *A* when the solid body starts to deform until it reaches the maximum deformation.



(**a**)

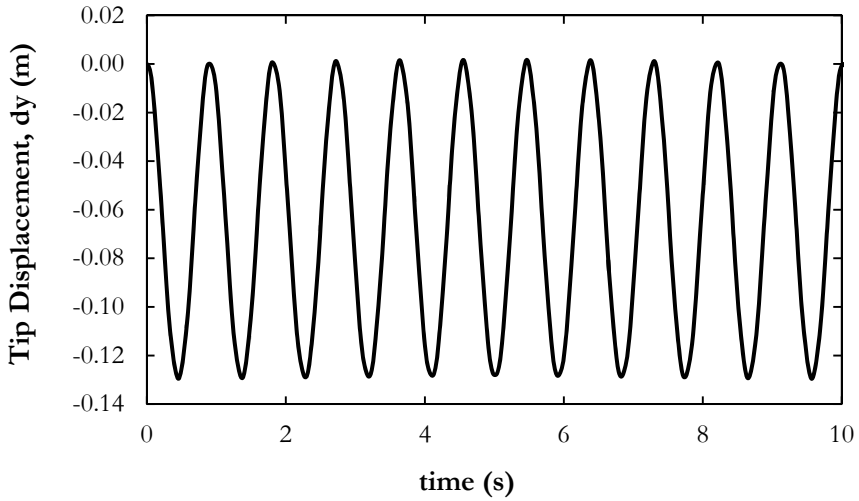


(**b**)

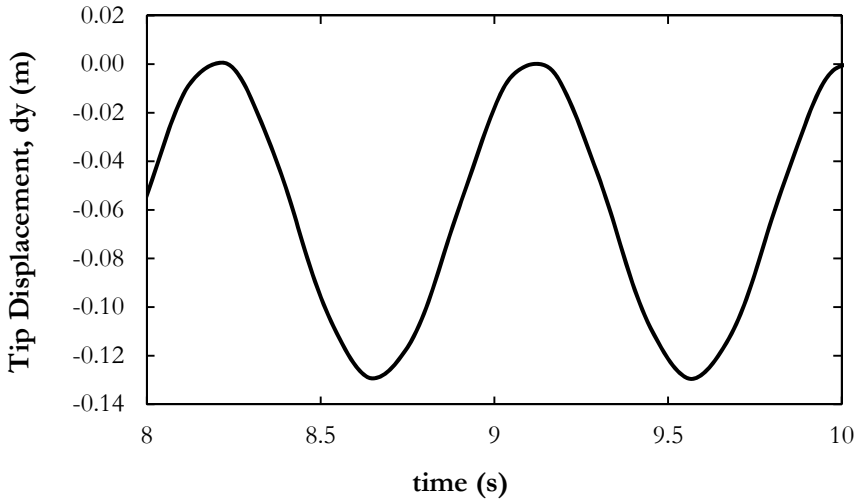


(**c**)

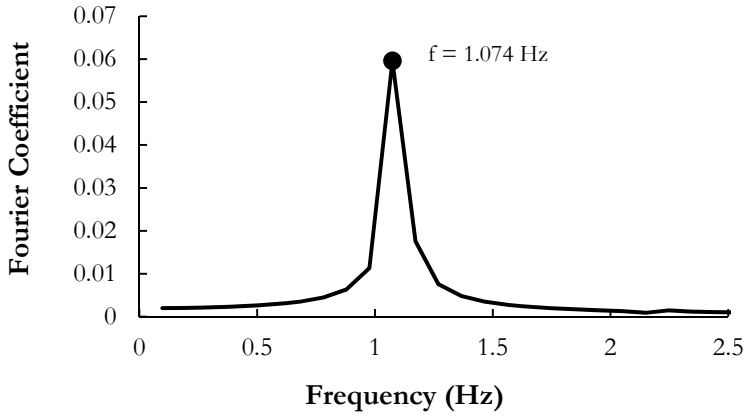
Figure 4.7 (**a**) & (**b**) Tip displacement of mesh *A* in the x direction (c) FFT result of the tip displacement evolution with the dominant frequency of 1.0742 Hz



(**a**)

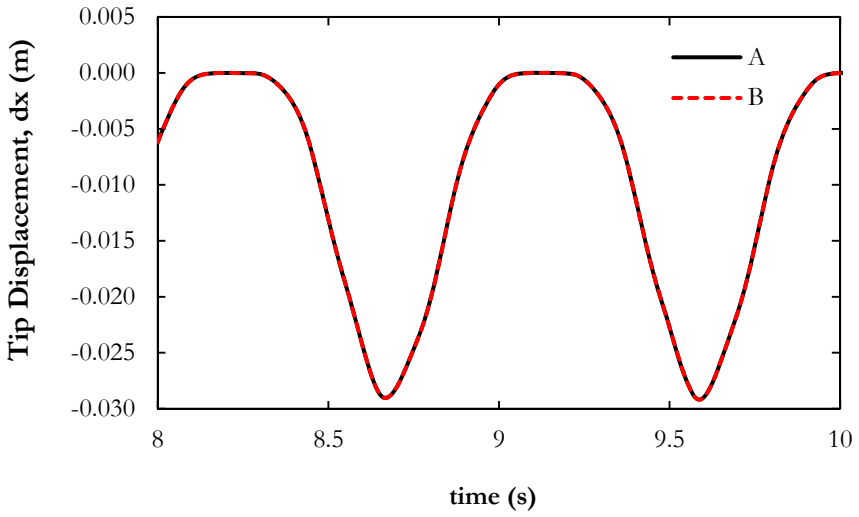


(**b**)

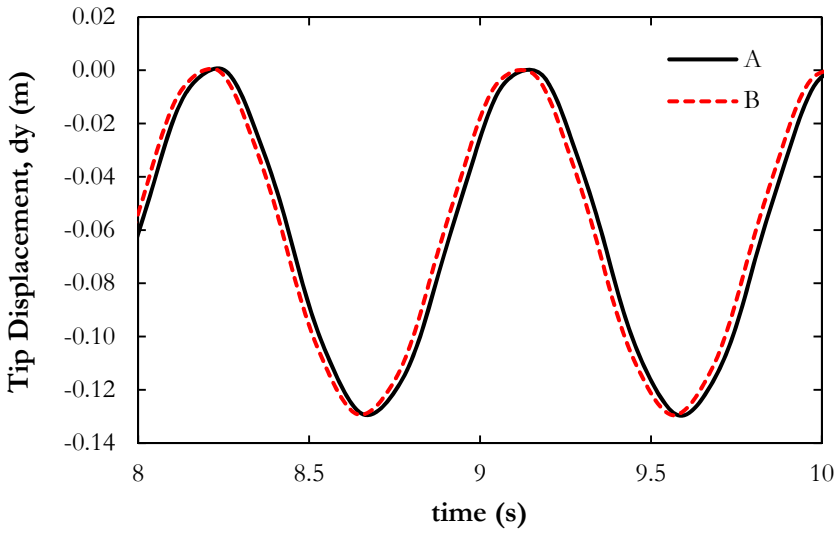


(**c**)

Figure 4.8 (**a**) & (**b**) Tip displacement of element mesh *A* in the ***y*** direction (**c**) FFT result of the tip displacement evolution with the dominant frequency of 1.0742 Hz



(**a**)



(**b**)

Figure 4.9 (**a**) Comparison of tip displacement in the x direction between mesh *A* and *B* (**b**) Comparison of tip displacement in the y direction between mesh *A* and *B*

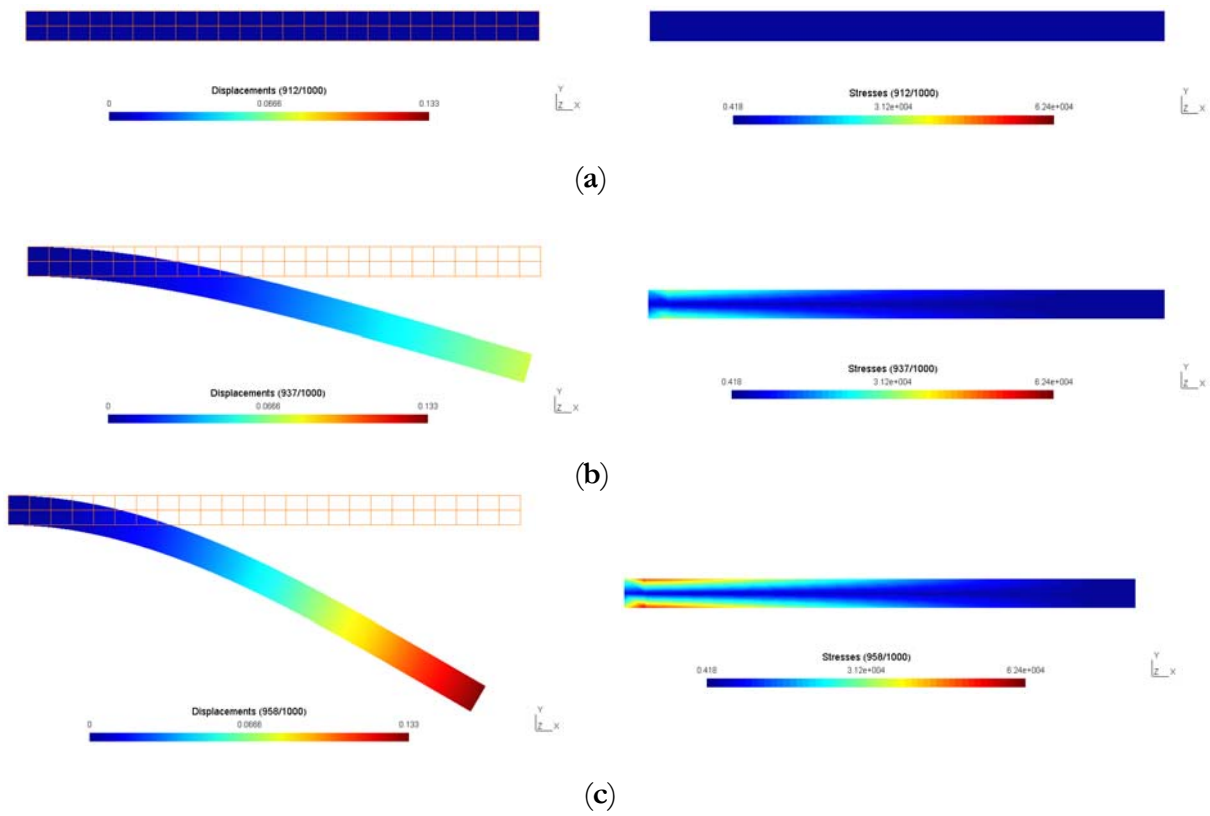


Figure 4.10 (a) Displacement (m) and Von Mises stress (Pa) contour plots of mesh A at t = 9.11 s (b) Displacement (m) and Von Mises stress (Pa) contour plots of mesh A at t = 9.36 s (c) Displacement (m) and Von Mises stress (Pa) contour plots of mesh A at t = 9.57 s

5 Fluid-Structure Coupling

5.1 Fluid-Structure Interface

On the fluid-structure interface, $\boldsymbol{\Gamma}_{fsi}$, kinematic and force equilibriums have to be satisfied:

$$\boldsymbol{u^s} = \boldsymbol{u^f} \tag{5.1}$$

$$\boldsymbol{\sigma^s \cdot n} = \boldsymbol{\sigma^f \cdot n} \tag{5.2}$$

where the superscript $\boldsymbol{s}$ and $\boldsymbol{f}$ refer to the solid and fluid domain respectively. The equilibriums are enforced to conserve the momentum and energy of the interaction where the work and energy produced by each domain are fully absorbed by the counterpart domain. To enforce this condition, boundary conditions are imposed to each domain where the interface displacements computed by the CSD solver become *Dirichlet* boundary conditions on the fluid flow domain and the fluid forces on the interface computed by the CFD solver becomes the *Neumann* boundary conditions on the solid domain.

In the case where the fluid and solid meshes match perfectly on the interface, the momentum and energy can be conserved numerically and force-displacement transfer is fairly straightforward. For a non-matching interface, an interpolation method has to be performed to fulfil the equilibrium and this can lead to non-conservative results. In this thesis the same problem is dealt with by adopting a strategy of conservation of virtual work of fluid-structure interface as given in [9] and it works well if the fluid mesh is finer than the structural mesh. To overcome this restriction, the solution proposed is to use an intermediate mesh built on the fluid-structure interface based on the intersection of the fluid and solid meshes hence the intermediate mesh is finer than the fluid and solid meshes. The force-displacement transfer is performed in this intermediate mesh and the key points of the strategy are the following:

- To conserve the force transfer from the fluid mesh to the intermediate mesh, firstly each of intermediate mesh elements is nested in a particular fluid mesh element. Then for every fluid mesh element that contains the barycentre coordinates of the intermediate mesh element, the stress of the intermediate element is equal to the stress of its counterpart fluid mesh element.
- A virtual work on the intermediate mesh, δW^I, is defined as the following:

$$\delta W^I = \sum_{n=1}^{n^I} \sum_{el=1}^{el^I} S_{el} \, (\boldsymbol{\sigma}_{el}^I \cdot \boldsymbol{n}) \, L_n \, u_n^I$$

$$= \sum_{n=1}^{n^I} \phi_n^I \, u_n^I \tag{5.3}$$

$$\phi_n^I = \sum_{el=1}^{el^I} S_{el} \, (\boldsymbol{\sigma}_{el}^I \cdot \boldsymbol{n}) \, L_n \tag{5.4}$$

Where n^I is the number of nodes on the intermediate mesh, el^I is the number of elements on the intermediate mesh, S_{el} is the surface area of the element el, L_n is the interpolation function at node n, and u_n^I is the virtual displacement at node n.

- The intermediate virtual displacement, u_n^I, is related to the structure displacement, u_i^s, as the following:

$$u_n^I = \sum_{i=1}^{i^s} \beta_{mi} \, u_i^s \tag{5.5}$$

- By substituting **Eq.** (5.5) into **Eq.** (5.3), the intermediate virtual work can be expressed in terms of the structure displacement:

$$\delta W^I = \sum_{i=1}^{i^s} \sum_{n=1}^{n^I} \phi_n^I \, \beta_{mi} \, u_i^s \tag{5.6}$$

- The conservation of virtual work between the intermediate and structure meshes can be written as the following:

$$\sum_{i=1}^{i^s} \boldsymbol{F}_i^s \, u_i^s = \sum_{i=1}^{i^s} \sum_{n=1}^{n^I} \phi_n^I \, \beta_{mi} \, u_i^s \tag{5.7}$$

Based on **Eq.** (5.7), the structure force, F_i^s, eventually can be expressed as the following:

$$\boldsymbol{F}_i^s = \sum_{n=1}^{n^I} \phi_n^I \, \beta_{mi} \tag{5.8}$$

- It can be proven that the resultant force field is conserved. Firstly, the resultant of the intermediate forces is defined as the following:

$$\sum_{e=1}^{e^I} \boldsymbol{F}_e^I = \sum_{e=1}^{e^I} S_e \, (\boldsymbol{\sigma}_e^I \cdot \boldsymbol{n}) \tag{5.9}$$

By using the fact that:

$$\sum_{n=1}^{n^I} L_n = 1 \tag{5.10}$$

Eq. (5.9) can be written as:

$$\begin{aligned}\sum_{e=1}^{e^I} \boldsymbol{F}_e^I &= \sum_{e=1}^{e^I} S_e \, (\boldsymbol{\sigma}_e^I \cdot \boldsymbol{n}) \sum_{n=1}^{n^I} L_n \\ &= \sum_{n=1}^{n^I} \sum_{el=1}^{el^I} S_e \, (\boldsymbol{\sigma}_e^I \cdot \boldsymbol{n}) \; L_n \\ &= \sum_{n=1}^{n^I} \phi_n^I \end{aligned} \tag{5.11}$$

On the other hand, using **Eq.** (5.8), the resultant of the structure forces is defined as the following:

$$\begin{aligned}\sum_{i=1}^{i^S} \boldsymbol{F}_i^S &= \sum_{i=1}^{i^S} \sum_{n=1}^{n^I} \phi_n^I \, \beta_{mi} \\ &= \sum_{i=1}^{i^S} \beta_{mi} \sum_{n=1}^{n^I} \phi_n^I \\ &= \sum_{n=1}^{n^I} \phi_n^I \end{aligned} \tag{5.12}$$

Based on **Eq.** (5.11) and (5.12), it is shown that the conservation of forces between the intermediate and structure mesh is maintained.

5.2 Coupling Strategy

As mentioned earlier, in general the coupling procedure of the fluid and solid solver can be classified into three different approaches i.e. the monolithic approach, the weakly coupled partitioned approach, and the strongly coupled partitioned approach. The work in this thesis is based on the strongly coupled partitioned approach where a convergence loop is utilized to ensure the stability of the coupled problem. The convergence loop can be either imposed externally or internally which can be explained in the following way:

- Externally Convergence Loop

  The convergence loop is coupled successively by solving the current time step of the fluid and structural problems using the previous solution of the other domain as boundary conditions. As the majority of the CPU time is spent on solving the fluid domain, the CPU time using this approach is proportional to the number of convergence iterations times the CPU time to solve the fluid domain.

- Internally Convergence Loop

  The convergence loop is integrated implicitly in the non-linear iterations of the fluid domain solution. The displacement boundary conditions imposed by the structure are updated during the fluid convergence process. This approach can be very efficient and comparable to the monolithic approach. To treat the stability issues caused by the added mass effect, an under relaxation of structural displacement has to be imposed before transferring the structural displacement to the fluid domain.

The work of this thesis is based on the strongly coupled partitioned approach using an internally convergence loop. The flow chart of this approach is shown in **Figure 5.1**.

5.3 Coupling Interface Library

As previously mentioned in **Section 4.1**, the modularity properties of Zorglib enable it to be plugged in into other solver which in this case is the ISIS CFD solver. A coupled computation is required on the FSI interface and it is managed by a C++ dynamic library. The coding of the dynamic library was first developed in the previous postdoctoral work to perform fluid rigid body interactions simulations. In this thesis it is developed further to perform two-dimensional FSI simulations for linear elastic and hyper-elastic flexible body. The dynamic library consists of three C++ external functions which are basically Fortran subroutines part of the ISIS-CFD solver. The three functions/subroutines are initGENERIC_ifs, GENERIC_ifs, and saveGENERIC_ifs. The hierarchy of the interface dynamic library is shown in **Figure 5.2** and the main tasks of the functions will be explained in the following sub sections with the complete codes listed in the **Appendix**.

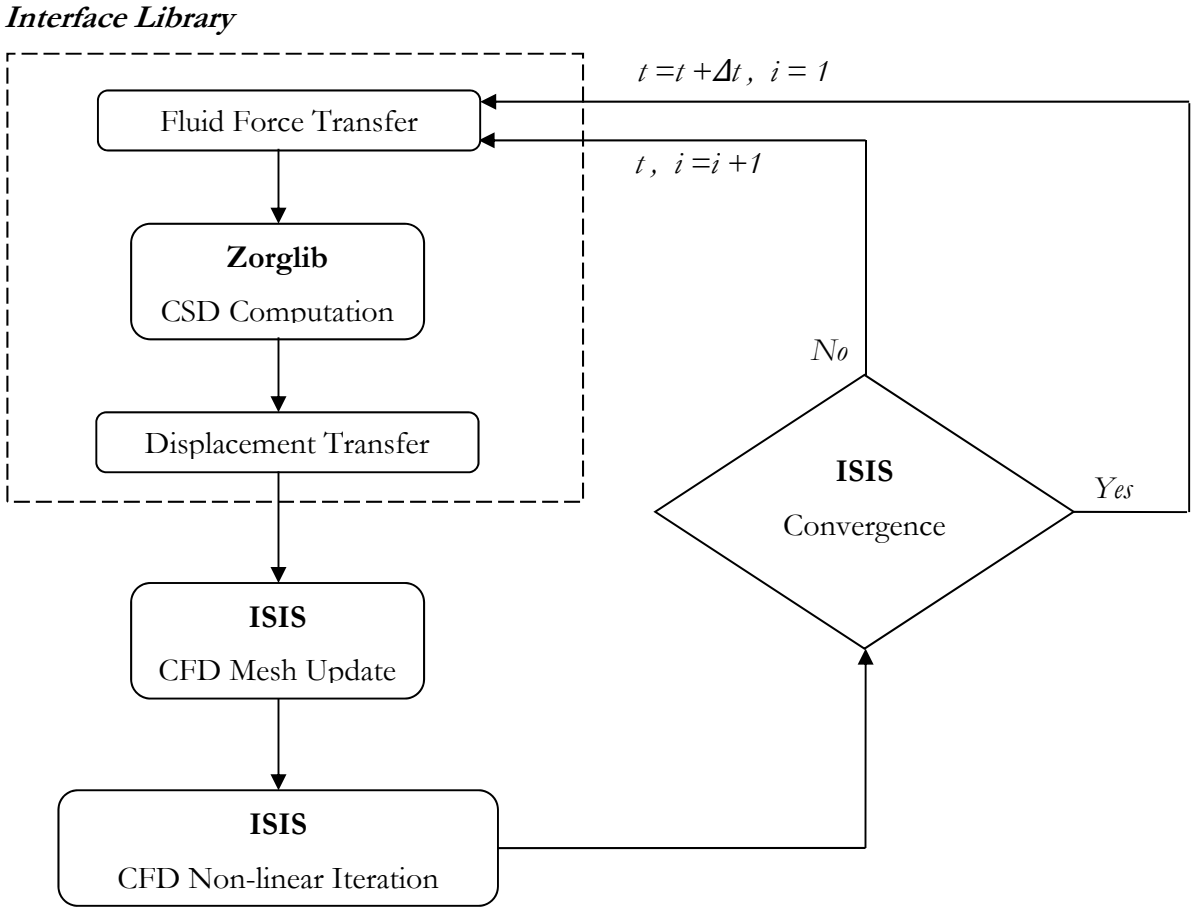


Figure 5.1 Flow chart of the fluid-structure solution procedure

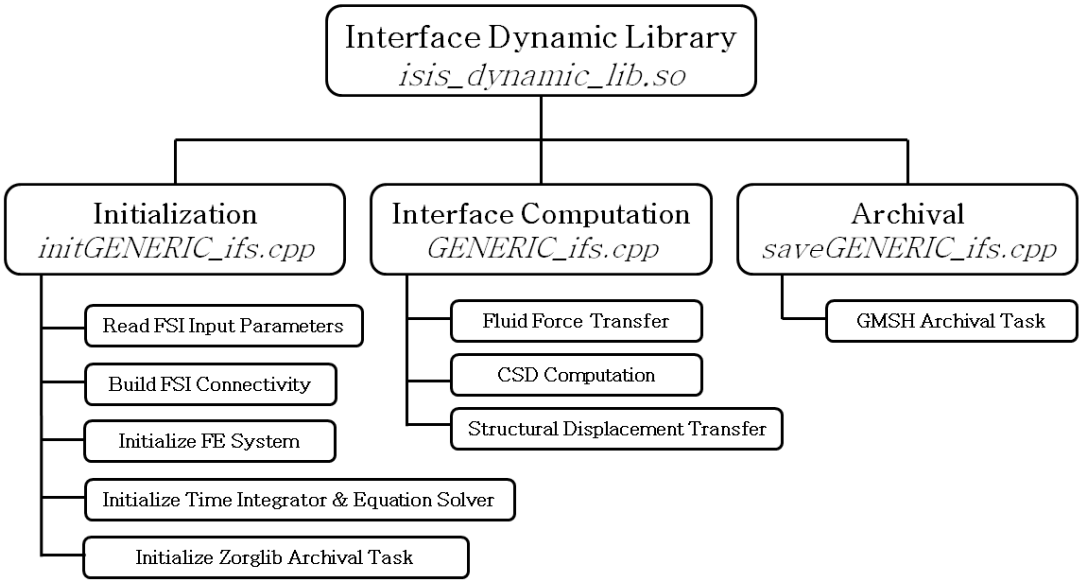


Figure 5.2 Interface dynamic library hierarchy

5.3.1 initGENERIC_ifs

This subroutine is only called at the beginning of the computation. There are several main tasks of initGENERIC_ifs subroutine and each of the main task is assigned to a C++ function:

1 Read input variables, which are used to initialize Zorglib solver, from a text file (FSI_Inputs.txt).

The input variables are:

- Problem dimension
  In this work, the dimension is 2 so the input variable is an integer of 2.
- Time step size
  The value has to be the same with the one set in ISIS solver.
- GMSH physical number
  The number is used to identify elements that are located on the FSI interface
- Material constitutive model
  There are two kind of constitutive models that can be used:
  - Linear elasticity model. For isotropic elasticity, the keyword is ISOTROPIC_ELASTICITY.
  - Hyperelasticity model. For Neohookean model the keyword is NEOHOOKEAN.
- Two dimensional assumptions
  The two dimensional assumptions can be either plane stress or plane strain. For linear elasticity models, the keywords are either LINEAR_PLANE_STRESS or LINEAR_PLANE_STRAIN. For hyperlasticity models, the keywords are either STANDARD_PLANE_STRESS or STANDARD_PLANE_STRAIN.
- Finite element solver
  The solver used depends on the analysis performed. For linear analysis the keyword is LINEAR to use LinearEquationSolver in Zorglib. For non-linear analysis the keyword is NEWTON to use NewtonSolver in Zorglib.
- Time integrator
  There are two time integrators that can be used. To use the trapezoidal/strandard Newmark method, the keyword is STANDARD_NEWMARK. To use the

Generalized-α method, the keyword is GENERAL_ALPHA and then followed by a double value between 0 and 1 for the spectral radius value (see **Eq.** (4.46) – (4.49)). For example, GENERAL_ALPHA 0.5.

- Under-relaxed parameter
  Define a value between 0 and 1. A value of 1 means there is no under-relaxation.
- Zorglib archive interval
  Define an interval to archive Zorglib results such as displacements and stresses.
- Run type
  The simulation can be started from the beginning or restarted from a previous computation. For a simulation starts from the beginning, the keyword is FULL. For a simulation restarted from a previous computation, the keyword is RESTART then followed by an integer value to define the time step where the computation starts from. For example RESTART 3000 means restart a simulation from 3000 time steps. The restart simulation must be running in the same folder as the previous simulation.

2 Collect and sort the FSI interface nodes' indexes and coordinates and FSI interface segments' connectivity.

By using objects and functions available in the Zorglib library, the FSI interface nodes' indexes and coordinates and FSI interface segments' connectivity can be collected into a set of arrays. The functions perform the collection task based on the physical number defined in GMSH mesh file for the FSI interface. Then the arrays that contain the indexes, coordinates, and connectivity have to be sorted to ensure that they correspond to consecutive nodes on the FSI interface. This is because the quadratic triangle and quadrilateral meshes created by GMESH have non-consecutive nodes' indexes in the segment connectivity and the sorting is necessary because the connectivity array will be used to build a surface mesh in the ISIS internal subroutine.

The code does not need a user input to identify whether the element is a linear or quadratic one instead it will identify it from the number of nodes in each segment. If there are only two nodes in a segment it means the element is a linear element whereas if there are three nodes the element is a quadratic one. A loop is created to sweep on all the FSI interface segments and sort and save the number of nodes and nodes' indexes in a segment.

3 Initialize the finite element system in Zorglib

Before initializing the finite element system, the following sub tasks are performed:

- Build and initialize the material model based on the constitutive model, problem dimension, and material properties
- Set the finite element context to identify if an axis symmetric or spherically symmetric or without symmetry problem is going to be solved
- Build and define the formulation of the finite element system based on the material model, context, and the two dimensional assumption.
- Impose boundary conditions using fixed displacement values or by initializing a displacement function which varies in time
- Build and initialize an external force function which varies in time

4 Initialize time integrator and solver

In this function the time integrator and solver objects are initialized based on the user input variable defined in point **1** and the finite element system object in point **3**. The displacement and velocity arrays are initialized in this task as well. The Newton Raphson solver is optimized with the line search method.

5 Initialize Zorglib archival task object

The archival tasks objects to archive the displacements, velocities, and stresses are initialized in this function. The archival tasks are called in the saveGENERIC_ifs subroutine.

5.3.2 GENERIC_ifs

This subroutine is nested in the non-linear iteration process of the ISIS-CFD solver so it is called at each ISIS non-linear iteration. In this subroutine, the transfer of the fluid force from the fluid domain to the solid domain and solid displacement transfer from the solid domain to the fluid domain are performed. An under-relaxation of the fluid force and the solid domain is also performed if the user input in point **1** is less than 1. The time integrator object is called to move forward in time but before calling it the displacement and velocity arrays are reset to the previous

time step values ($\boldsymbol{u}_{n+1}^{i} = \boldsymbol{u}_n$ and $\boldsymbol{v}_{n+1}^{i} = \boldsymbol{v}_n$). The fluid forces and a particular node displacements at every non-linear iteration are saved into a text file for checking purposes.

5.3.3 saveGENERIC_ifs

This subroutine is called at the end of every time step. The archival tasks are called at a specific interval as defined in point **1** in initGENERIC_ifs subroutine. To enable a restart computation, the displacements, velocities, and accelerations arrays are saved into a text file in this subroutine. For the purpose of analysing the solid tip displacement of the FSI simulations in this thesis, internal object and function in Zorglib are called to obtain the tip displacements and then the values are saved into a text file.

5.4 Mesh Update Technique

One important aspect of the FSI numerical model is the fluid mesh update as a result of the interface displacements. An efficient and robust technique is needed because the fluid mesh is updated at every non-linear iteration. There are several mesh update techniques adopted in ISIS-CFD which have been developed by several PhD works in Ecole Centrale de Nantes [13] [19] [20] and the technique used in this thesis is based on the work [20]. In this technique an analytic regridding method based on a weighting coefficient is further developed by propagating the rigid displacement of each faces of volume cells and diffusing it on the fluid domain so it can work properly when dealing with large deformations.

6 FSI Numerical Results

6.1 Flow Induced Excitation of Vertical Flexible Thin Plate

This test case is based on one of the FSI test cases in [13] and the simulation set up is depicted in **Figure 6.1**. A thin flexible plate is clamped at the bottom wall boundary in the downstream of a incompressible fluid flow with a uniform inflow velocity. The fluid domain is an open domain with boundary conditions described in **Figure 6.1**. A no-slip boundary condition is imposed on the plate body. The fluid and solid properties are summarized in **Table 6.1** and the resulting *Reynold's* number is 50. The fluid mesh is shown in **Figure 6.2** where it has 16,938 cells and 36,014 nodes.

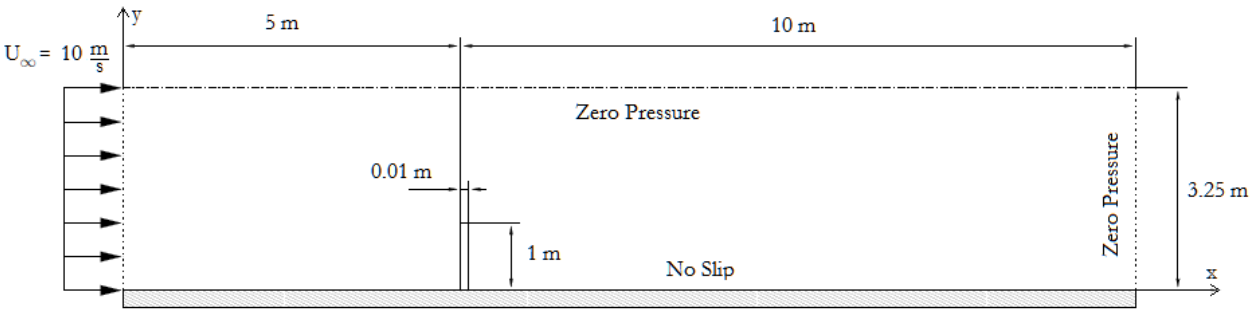
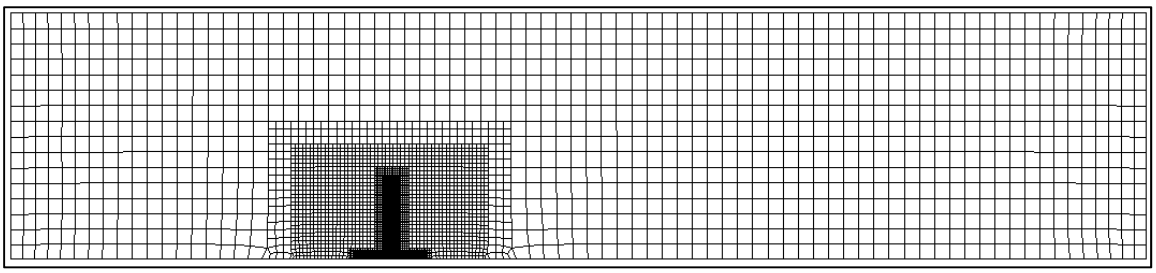


Figure 6.1 Simulation set up for the vertical flexible thin plate case

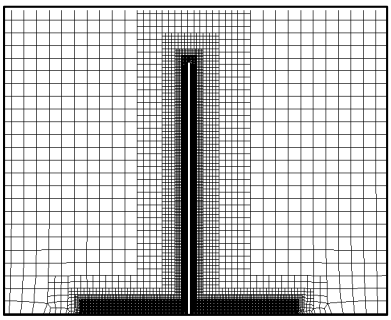
	Fluid	**Solid**
Density, ρ (kg/m^3)	1.0	1.2 E+3
Poisson's ratio, ν	-	0.32
Young's Modulus, E (Pa)	-	3.5 E+9
Dynamic Viscosity, μ $(Pa.s)$	0.2	-
Area moment inertia, I (m^4)	-	8.3 E-8

Table 6.1. Material properties of the fluid and solid for the vertical flexible thin plate case

Initially an uncoupled CFD simulation with a rigid plate runs for 10 s to obtain a stable fluid flow in the domain with a time step size of 0.01 s. **Figure 6.3** and **6.4** show the streamlines plot and pressure contour plot at time of 10 s.



(**a**)



(**b**)

Figure 6.2 (**a**) Fluid mesh of the whole domain (**b**) Fluid mesh in the surrounding zone of the plate

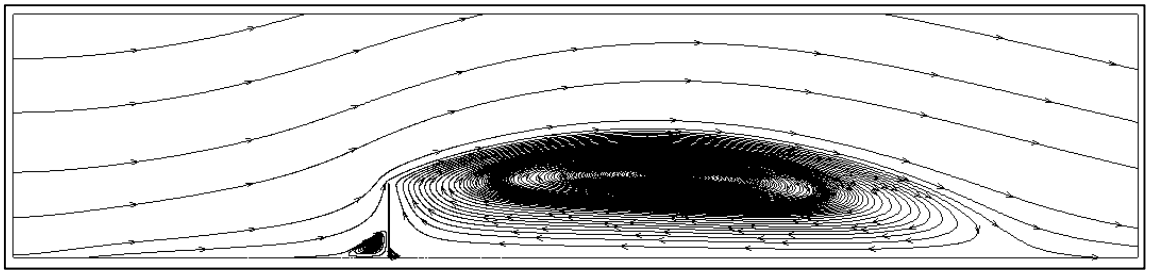


Figure 6.3 Streamlines at time of 10 s

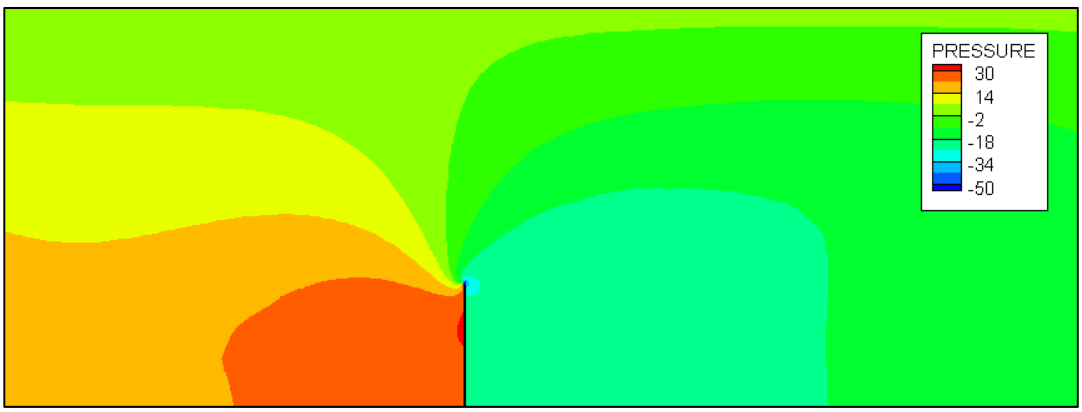


Figure 6.4 Pressure at time of 10 s

The simulation then continues as a coupled simulation runs from 10 s until 20 s with a time step size of 0.01 s. In the coupled simulation, the element type used for the solid mesh is quadratic quadrilateral with 9 nodes. One of the main analysis in this simulation is to analyse the evolution of the tip displacement in the x direction and to be able to do this a control point is set in the interface library and the library will save the displacement of the middle node on the top edge of the plate into a text file.

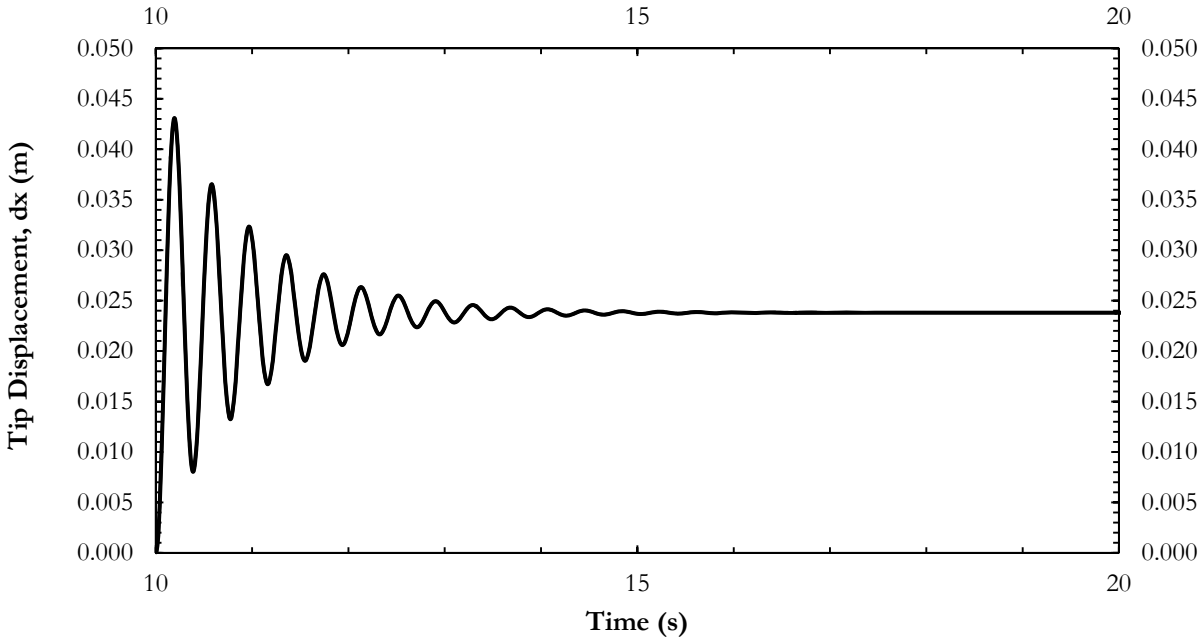
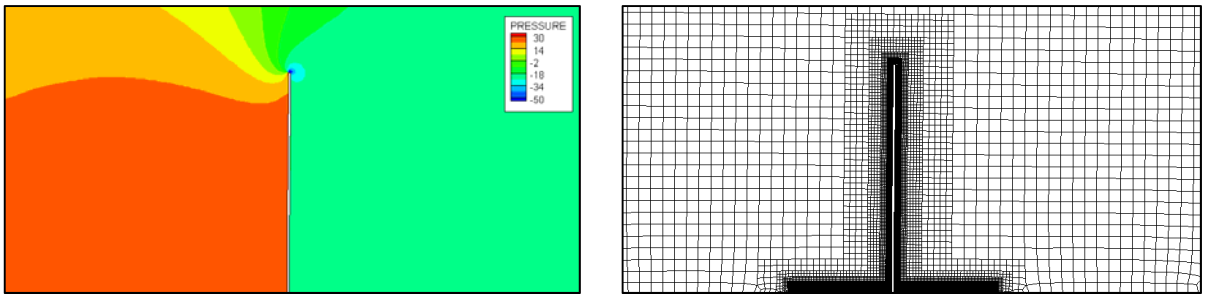


Figure 6.5 Evolution of the tip displacement in the x direction

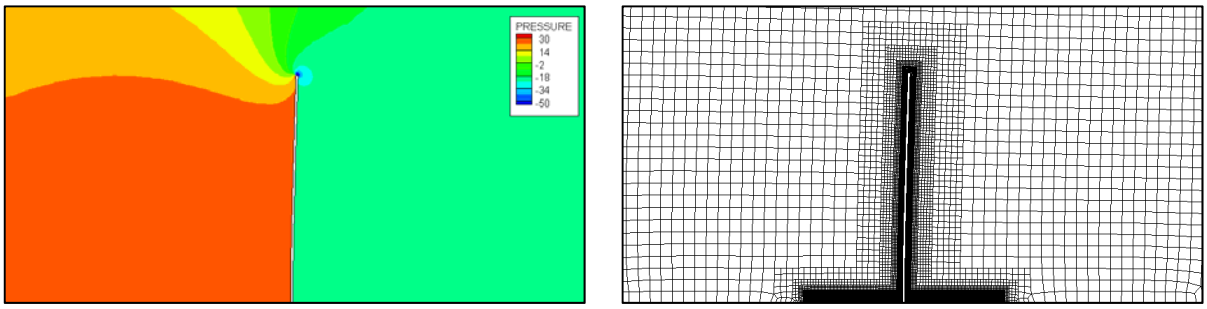
As can be observed in **Figure 6.5**, after 5 seconds the solution becomes steady with the steady tip displacement of 0.024 m and this result agrees well with [13]. **Figure 6.6** shows the evolution of the pressure and mesh between 10.05 and 10.20 s when the tip displacement reaches its maximum value. A theoretical value of the first eigenfrequency of a clamped beam can be obtained using an analytical solution [14] with the following definition:

$$f_1^{eig} = 0.5595 \frac{1}{L^2} \sqrt{\frac{E\,I}{m/L}} \tag{6.1}$$

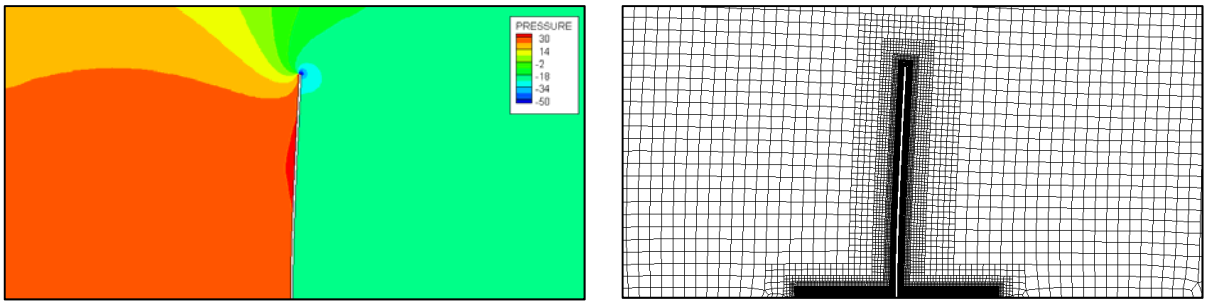
With the current solid properties and dimension, the first eigenfrequency, $f_1^{eig} = 2.758$ Hz. A *Fast Fourier Transform* (FFT) analysis is performed on the first 5 s of the tip displacement data as shown in **Figure 6.7**, which gives the dominant frequency of 2.734 Hz close to the theoretical value.



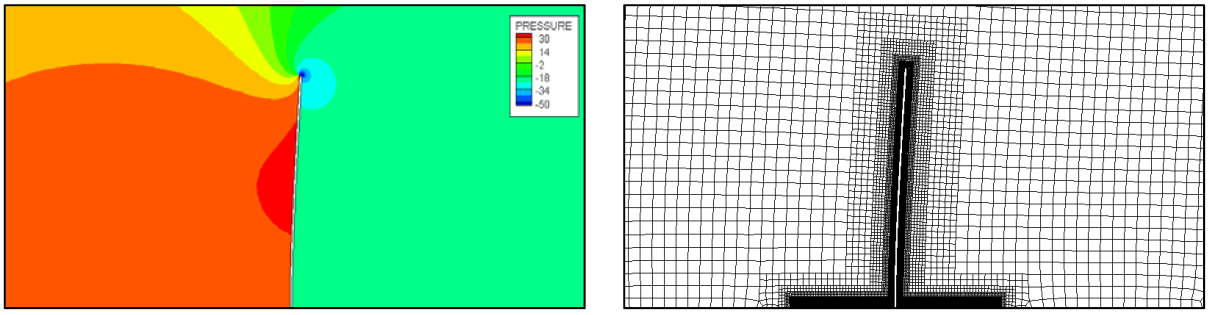
(**a**)



(**b**)



(**c**)



(**d**)

Figure 6.6 Pressure (Pa) contour plot and fluid mesh at (**a**) t = 10.05 s (**b**) t = 10.10 s (**c**) t = 10.15 s (**d**) t = 10.20 s

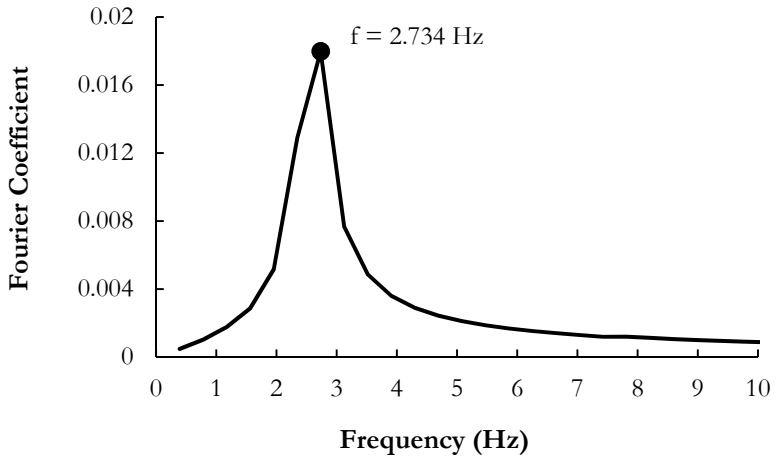


Figure 6.7 FFT analysis of the tip displacement using MATLAB FFT function on the first 5 s data

6.2 Flow Induced Excitation of Horizontal Flexible Thin Plate

In this test a flexible thin plate is clamped at the end of a square cylinder and a fluid flow with uniform velocity from the left boundary pass the structure and creates vortices which induce structural oscillations [7][15]. The simulation set up and boundary conditions are depicted in **Figure 6.8** and the solid and fluid properties are shown in **Table 6.2**. The resulting *Reynold's* number is 333.

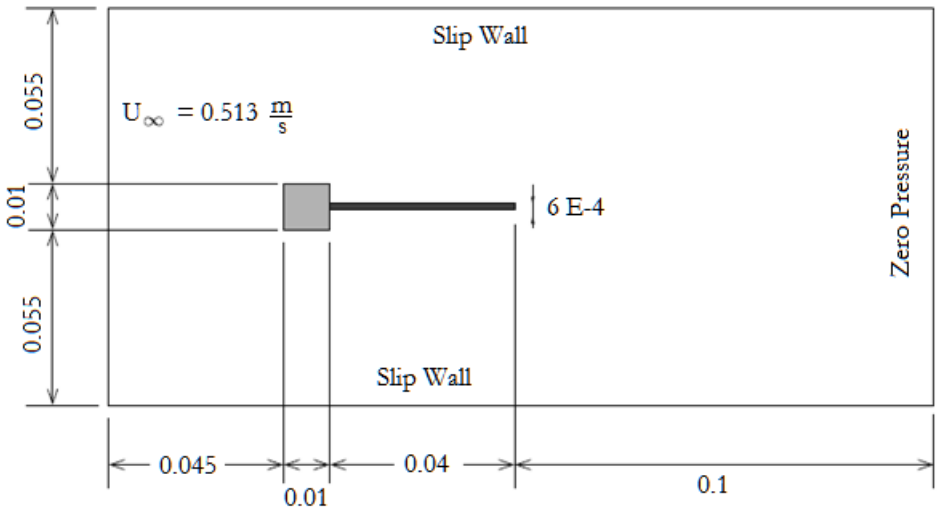
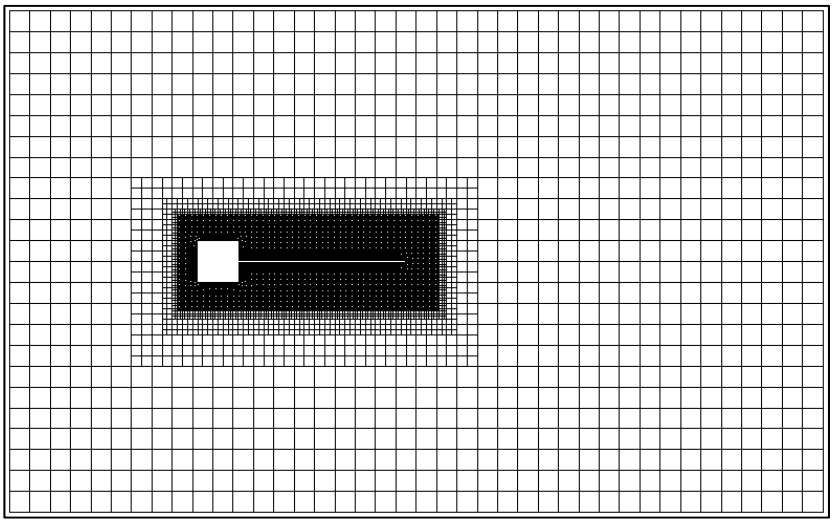


Figure 6.8 Simulation set up for the horizontal flexible thin plate case (dimension is in meter)

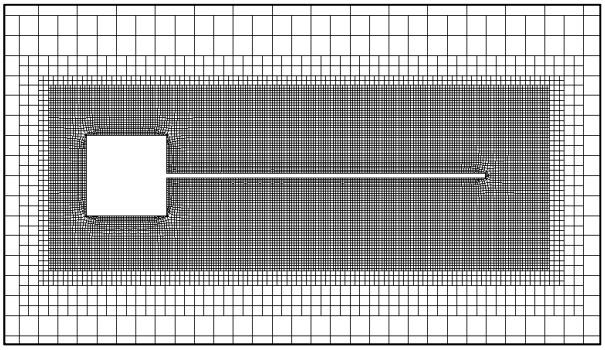
	Fluid	**Solid**
Density, $\rho\ (kg/m^3)$	1.18	100
Poisson's ratio, ν	-	0.35
Young's Modulus, $E\ (Pa)$	-	2.5 E+5
Dynamic Viscosity, $\mu\ (Pa.s)$	1.82 E-5	-
Area moment inertia, $I\ (m^4)$	-	1.8 E-11

Table 6.2 Material properties of the fluid and solid for the horizontal flexible thin plate case

The fluid mesh is shown in **Figure 6.9** where it has 16,863 cells and 34,856 nodes.



(**a**)



(**b**)

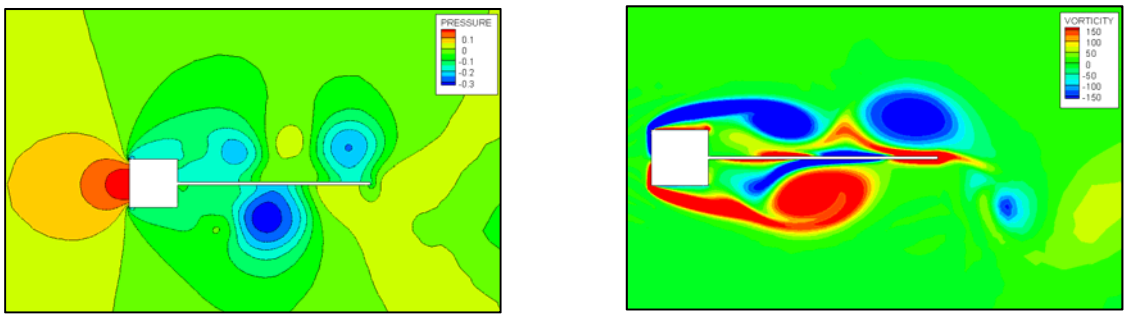
Figure 6.9 (**a**) Fluid mesh of the whole domain (**b**) Fluid mesh in the surrounding zone of the plate

Similar to the previous test case, the simulation starts with the uncoupled CFD simulation with the plate is enforced to be rigid for 5 s then it continues with the coupled simulation until 25 s when the plate becomes elastic. **Figure 6.10** shows the pressure and vorticity contour plots between 4.2 s and 4.5 s in the case of rigid plate. It can be observed that a vortex shedding has occurred at each time instance. If the whole evolution is analysed it can be seen that two small vortices on one side and a bigger one on the side appear alternatingly.

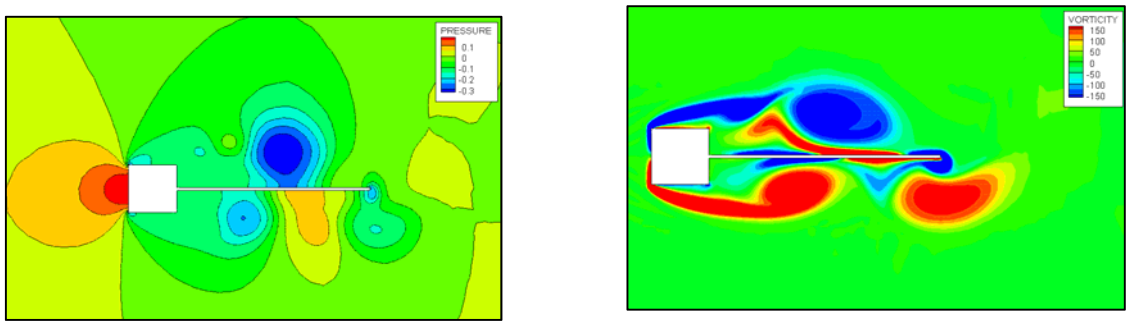
Figure 6.11 shows the evolution of the fluid mesh at a time period between 24.55 s and 24.8 s during the coupled simulation. Next in **Figure 6.12** the pressure and vorticity contour plots are shown at the same time period. As can be observed in **Figure 6.12**, the flow has a similar behaviour as in the uncoupled simulation but the vortex separation at the plate tip creates a smaller vortex when the tip displacement at its maximum and minimum, **Figure 6.12** (**b**) and (**d**) respectively.

Figure 6.13 shows the evolution of the tip displacement in the y direction between 5 s and 25 s. The steady oscillations amplitude agrees well with [7] and [15]. Using the FFT analysis for the last 16.384 s of data, the dominant frequency obtained is 2.99 Hz as shown in **Figure 6.14** and this result agrees well with the analytical value of the first eigenfrequency computed using **Eq.** (6.1) , $f_1^{eig} = 3.028$ Hz. The first eigenfrequency in [7] is reported to be in the range of 2.96 and 3.31 Hz. **Figure 6.15** shows the evolution of the lift force between 10 s to 25 s and using the FFT analysis its dominant frequency is found to be the same as the displacement in the y direction as can be seen in **Figure 6.16**.

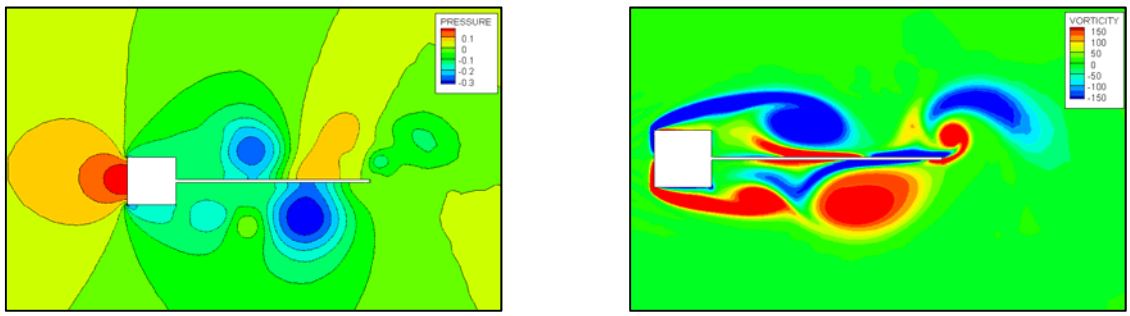
In this simulation a refined solid mesh with 2 x 80 elements is also used to compare the tip displacement results with the original solid mesh one as shown in **Figure 6.17**. There is no significant difference observed which suggests that solid mesh convergence has been attained.



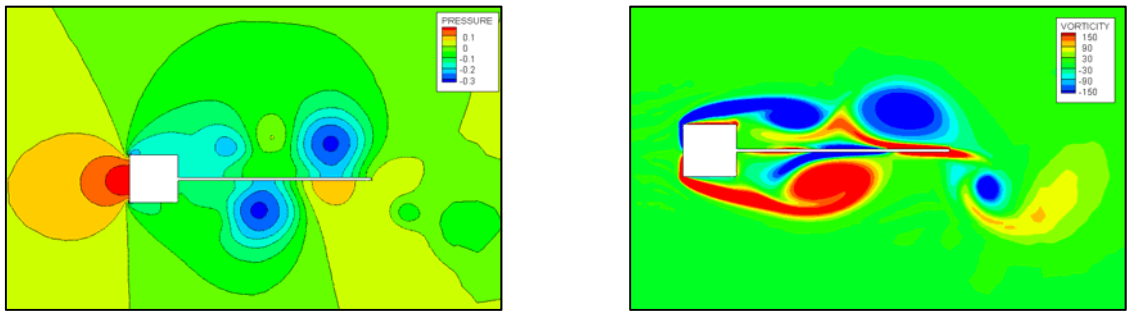
(**a**)



(**b**)

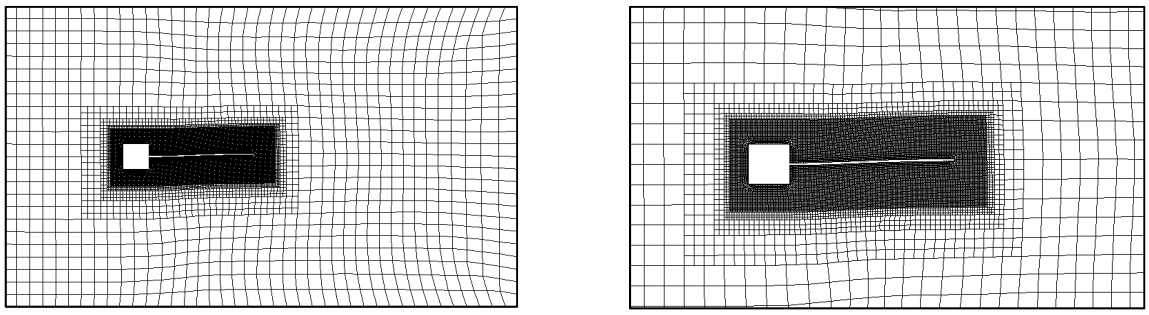


(**c**)

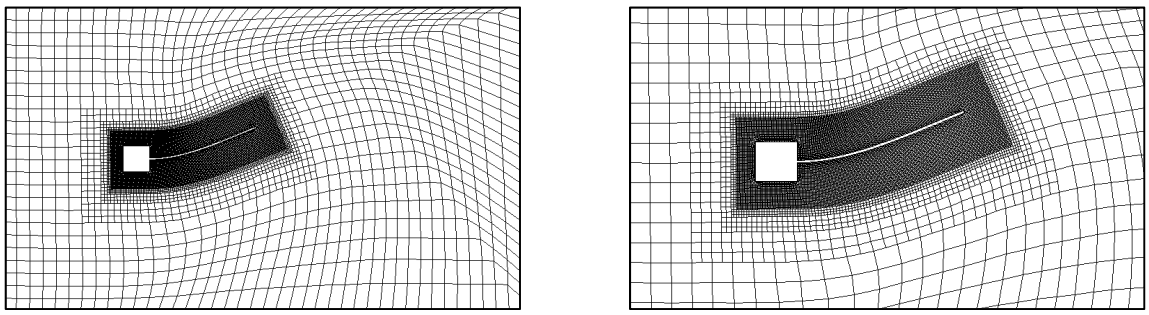


(**d**)

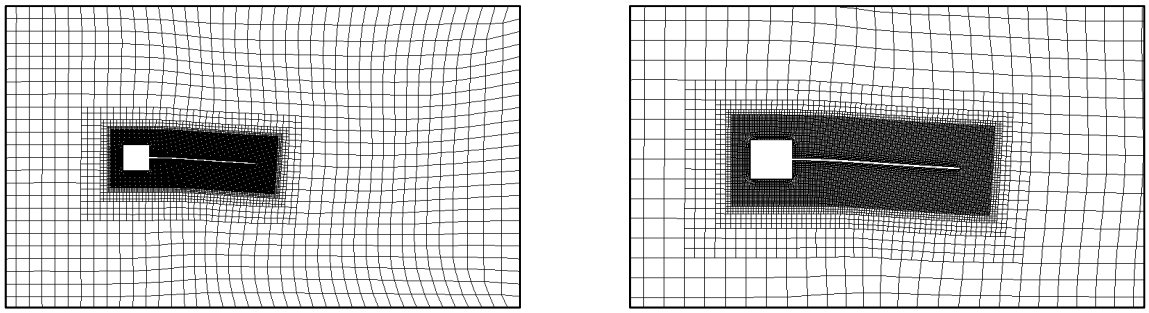
Figure 6.10 Pressure (Pa) and vorticity (1/s) contour plots at (**a**) $t = 4.2$ s (**b**) $t = 4.3$ s (**c**) $t = 4.4$ s (**d**) $t = 4.5$ s



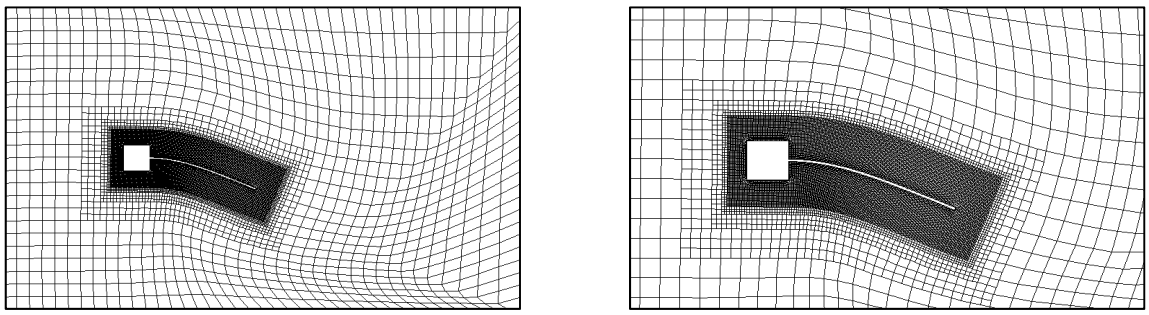
(**a**)



(**b**)

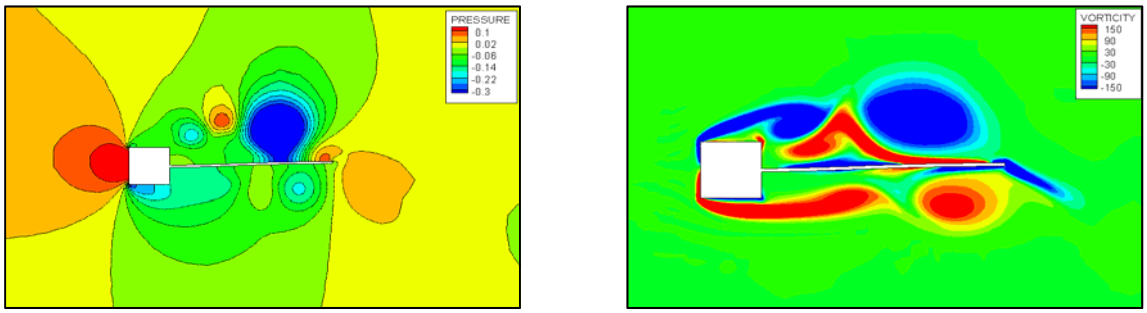


(**c**)

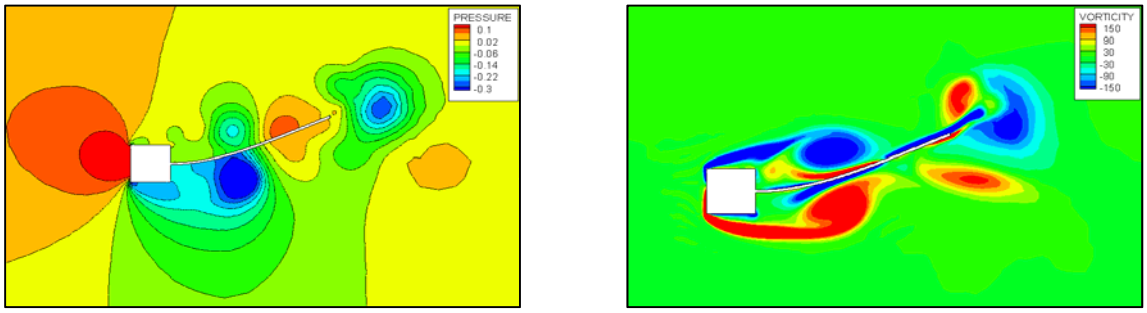


(**d**)

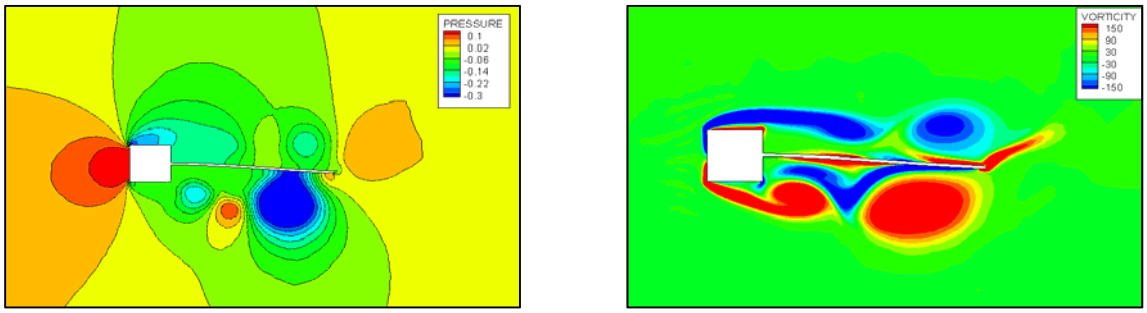
Figure 6.11 The fluid mesh at (**a**) $t = 24.55$ s (**b**) $t = 24.625$ s (**c**) $t = 24.725$ s (**d**) $t = 24.8$ s



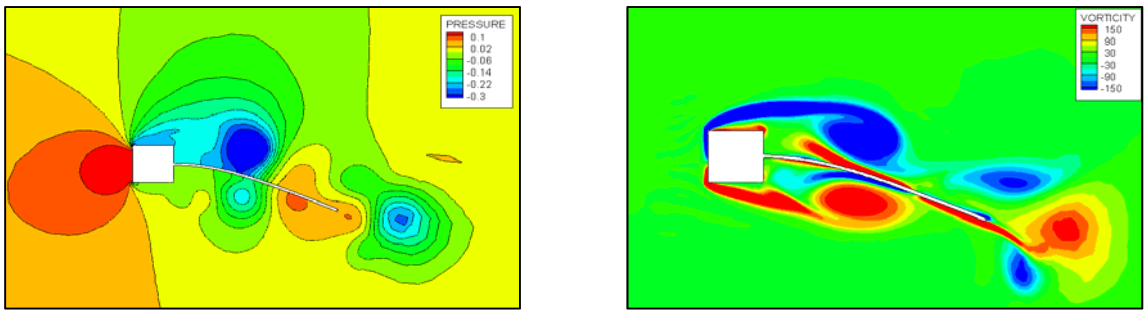
(**a**)



(**b**)



(**c**)



(**d**)

Figure 6.12 Pressure (Pa) and vorticity (1/s) contour plots at (**a**) $t = 24.55$ s (**b**) $t = 24.625$ s (**c**) $t = 24.725$ s (**d**) $t = 24.8$ s

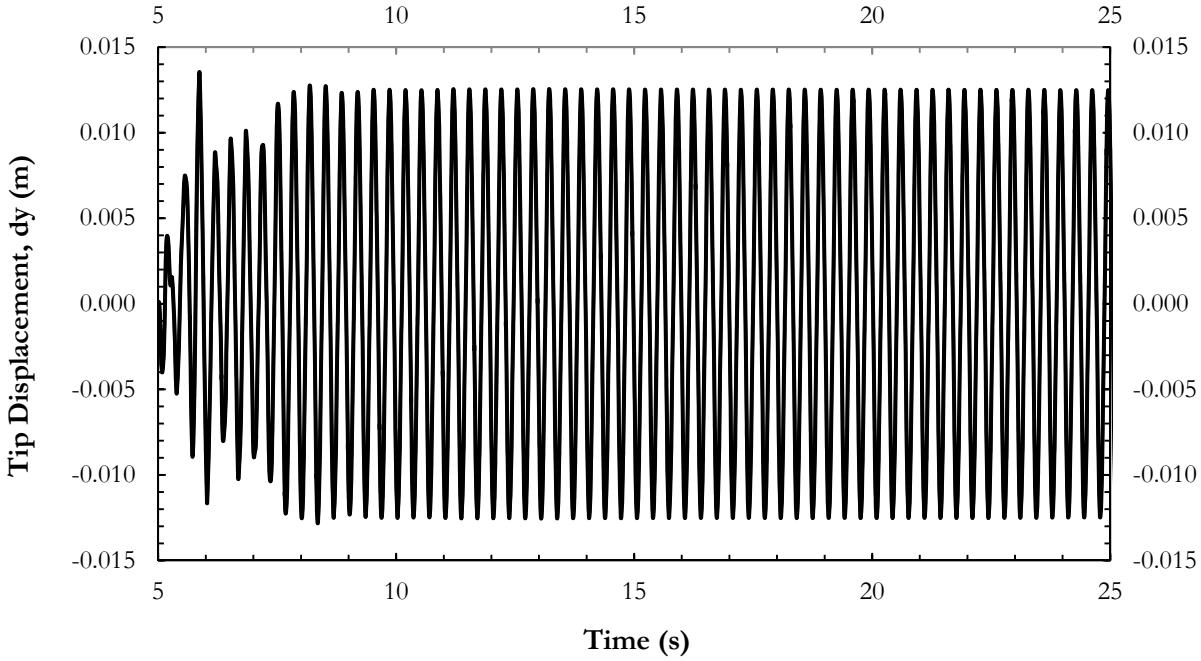


Figure 6.13 Evolution of the tip displacement (m) in the y direction

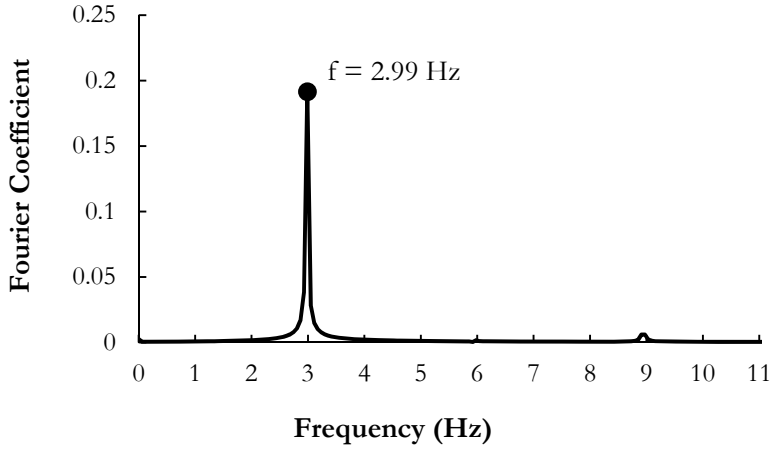


Figure 6.14 FFT analysis of the tip displacement using MATLAB FFT function on the last 16.384 s data

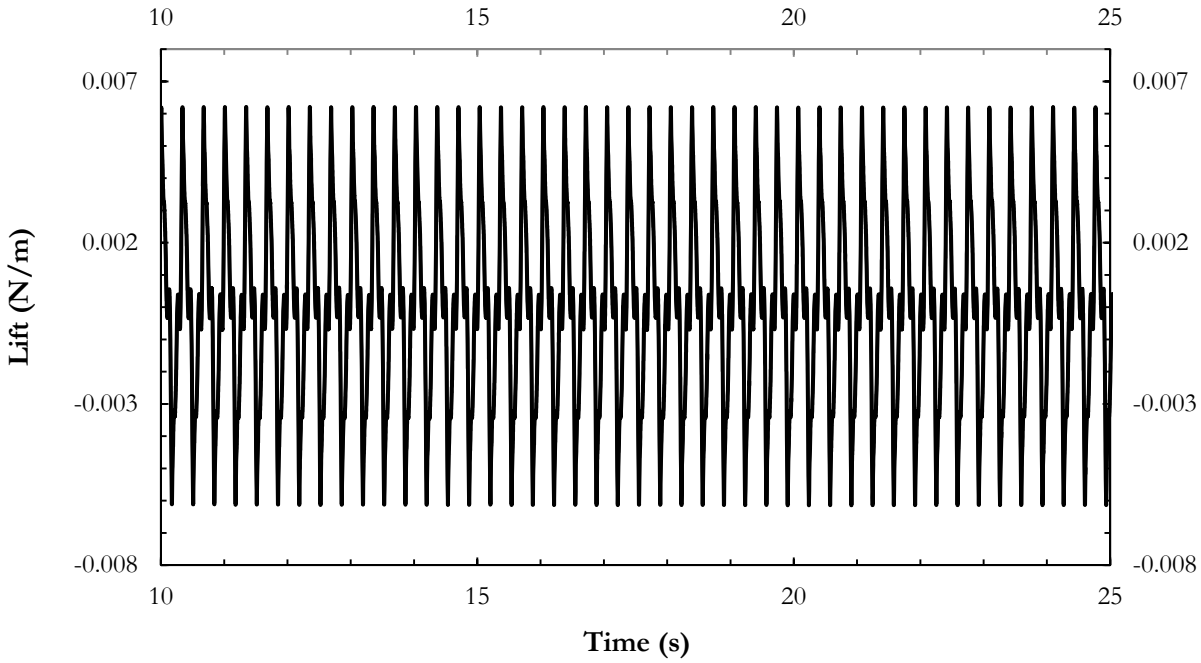


Figure 6.15 Evolution of the lift force (N/m)

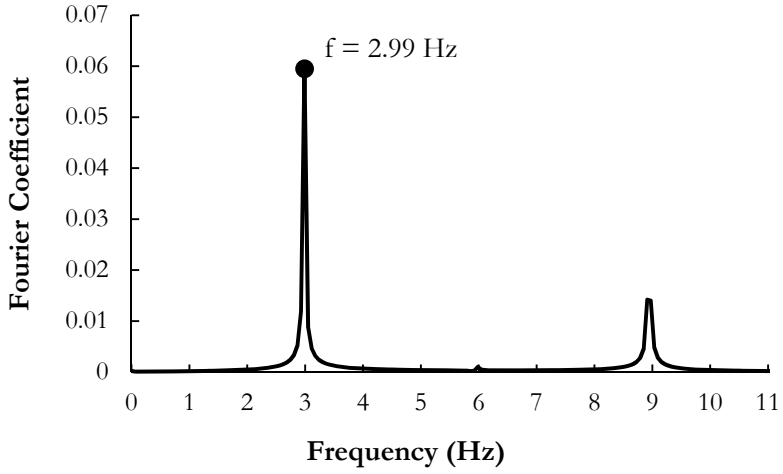


Figure 6.16 FFT analysis of the lift force using MATLAB FFT function on the last 8.192 s data

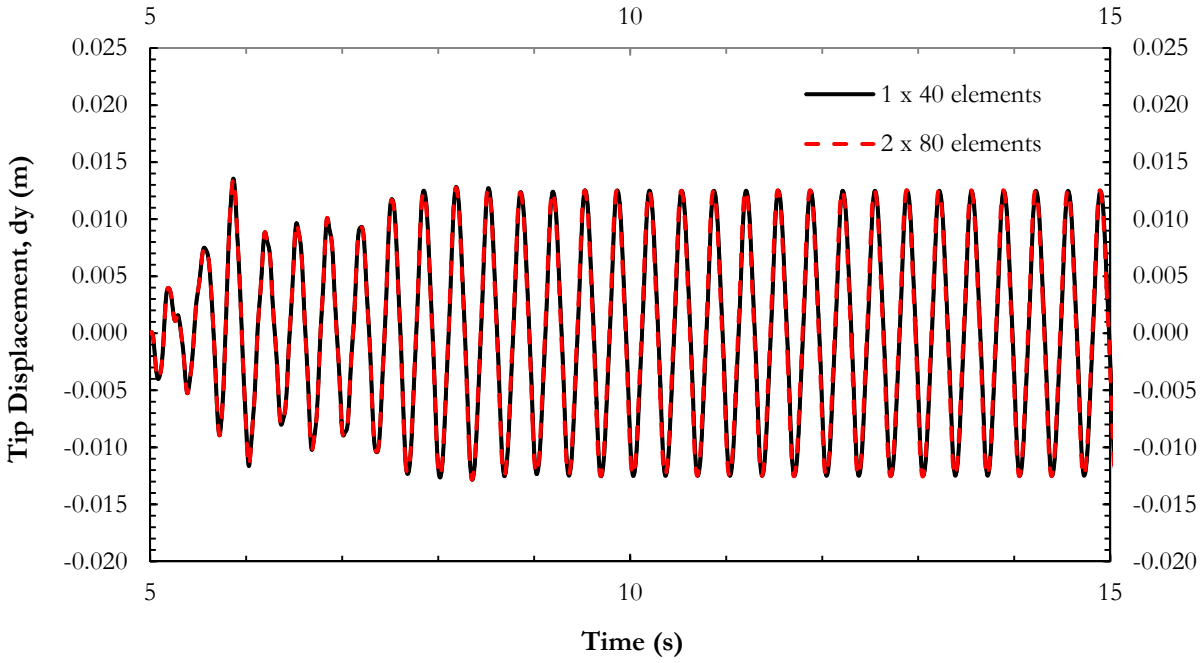


Figure 6.17 Comparison of evolution of the tip displacement (m) in the y direction between the coarse and fine solid mesh

7. Conclusions

Two dimensional numerical tests based on published numerical simulations have been performed for the CSD and CFD solvers separately. In the case of the CFD test, an unsteady flow around a square cylinder is investigated and it shows periodic vortex shedding and recirculation close to the body with the starting separation point moves from the trailing edge to the leading edge. For the CSD test, a dynamic simulation of a flexible cantilever solid with gravity load is investigated. Both of the tests show good agreements with the referenced publications. A mesh sensitivity analysis has also been performed using the CSD solver to ensure the proper element type and mesh size to be used in the FSI simulations.

An interface dynamic library has been developed using C++ programming language to enable the coupling of the CFD and CSD solver to solve two dimensional FSI problems of flexible and elastic body interacted with incompressible viscous fluid. The FSI computational procedure is based on the strongly coupled implicit partitioned approach with internal convergence loop as part of the CFD non-linear iteration. The modularity of Zorglib makes it possible to embed the CSD computation in the ISIS-CFD non-linear iteration. An under-relaxation method is used for the interface structure displacement and fluid force to overcome the added-mass effect.

In ISIS-CFD subroutines an intermediate mesh is built to conserve the momentum and energy on the FSI interface without any restriction of the structure or fluid mesh density. Also an efficient mesh update is performed at every non-linear iteration. These two computational procedures were developed by several PhD works at Ecole Centrale de Nantes.

The FSI simulations are based on the published works which concern the flow induced excitations of flexible thin plates. The first simulation leads to a steady solution at the end whereas in the second simulation periodical flow and plate vibrations are observed as a result of periodical vortex shedding at the edges of the square cylinder. Both simulations show good agreement when the tip displacement and dominant vibration frequency are compared with the reference results.

Appendix A FSI Simulations Procedures

The procedure to run FSI simulations using ISIS-CFD and Zorglib is as follow:

1. Compile Zorglib library

   Using the terminal, in the Zorglib library folder execute ***./configure*** then after that ***make***

2. Compile interface dynamic library

   Using the terminal, in the dynamic library folder execute ***make***. It will create ***isis_dynamic_lib.so*** file

3. Create ISIS-CFD project file

   After creating the geometry and mesh do the following:

   - In HEXPRESS under **Grid -> Boundary Conditions** set the name of the face of FSI edges so they are ended with ***_fsi***
   - Return to FINEMARINE window and rename the computation so it is ended with ***fsi***. For example it can be renamed into the following:

     - ***fsi***

     - ***new_fsi***

     but NOT with ***newfsi***.
   - Set all the required CFD parameters (fluid model, flow model, computation control , etc.)
   - Define the FSI body in the **Body Definition** option and take note of the FSI body index
   - Add the following comment in FINEMARINE **Comment** option with the value after *** FSI : SURFACES line is the FSI body index as defined previously for example the value is 100 as shown below.

     *** FSI WITH GENERIC INTERFACE ?

     *

     YES

     *

     *** FSI : SURFACES

     *

```
100
*
*** FSI : ITS FLUID GEOM : FILE
*
../_mesh/mesh.its
*----------------------------------------- BODIES ----------------------------------------
*** BODY:MVT:NON-LINEAR COUPLING LOOP ?
*
YES
*
*** BODY:DEF:MVT IMP:NUMBER
*
1
*
*** BODY:DEF:MVT IMP:1:NAME
*
body
*
*** BODY:DEF:MVT IMP:1:INDEX
*
1
*
*** BODY:DEF:MVT IMP:1:REFERENCE POINT
*
0. 0. 0.
*
*** BODY:FIXED:NUMBER
*
0
*
*** BODY:DEF:MVT IMP:1:WEIGHTING REGRID METHOD
*
3
*
```

*** DOMAIN:1:RIGID MVT GRID

*

0 0 0 0 0 0

- Activate **Adaptive Grid Refinement.** Click to **Control** and set the *Number of steps before first call to refinement procedure* to a very high value for example 1e+08
- Pre-process the computation

4. Create a 2D solid mesh using GMSH with the FSI boundary is set with a physical number and save the mesh file as ***meshZorg.msh***

5. Copy and paste the following files into the ISIS-CFD computation folder:
   - ***meshZorg.msh***
   - ***FSI_Inputs.txt***
   - ***Solid_Properties.mat***
   - ***isis_dynamic_lib.so***

6. Modify the ***FSI_Inputs.txt*** file accordingly based on the guideline in **Section 5.3.1**

7. Ensure the solid material properties in ***Solid_Properties.mat*** are defined correctly

8. Copy the ***isiscfd_FSI_2D*** binary file in the computation folder or in other folder. Run the FSI simulation using the terminal in the computation folder by executing the ***isiscfd*** binary file and followed with the computation simulation file. For example if the binary file is stored in the computation folder:

   ./isiscfd_FSI_2D<*_fsi.sim>logFSI.dat

Appendix B Interface Dynamic Library

B.1 initGENERIC_ifs.cpp

```
#include "init_ISIS.h"
#include "IFSFunction.h"

using namespace std;

extern "C" void initgeneric_ifs(int * imesg,int * mybloc,int * MAXDIM_NP_ITS_STR_IFS,int *
MAXDIM_NT_ITS_STR_IFS,int * np_its_str_ifs,int * nt_its_str_ifs,double *
X_ITS_STR_IFS,double * Y_ITS_STR_IFS,double * Z_ITS_STR_IFS,int
Con_ITS_STR_IFS[][3])
{
   cout << "********************* initGENERIC_ifs Subroutine ************************"
   << endl;

    // Read the FSI_Inputs.txt file to obtain Zorglib parameters
    Read_FSI_Inputs();

    // Build FSI segment connectivity and coordinates
    Build_FSI_Arrays(np_its_str_ifs, nt_its_str_ifs, X_ITS_STR_IFS, Y_ITS_STR_IFS,
    Z_ITS_STR_IFS, Con_ITS_STR_IFS);

    // Initialize FE system for the CSD
    Zorglib_FE_Initialization();

    // Initialize time integrator for the CSD
    Zorglib_Integrator_Initialization();

    // Initialize GMSH archival task for the CSD
    Zorglib_Archival_Initialization();
```

```
// Initialize arrays for FSI boundary displacement during non-linear iteration
iterdispX = new double[FSI_Node_Size];
iterdispY = new double[FSI_Node_Size];
iterdispZ = new double[FSI_Node_Size];
iterdispX_old = new double[FSI_Node_Size];
iterdispY_old = new double[FSI_Node_Size];
iterdispZ_old = new double[FSI_Node_Size];

// Create and initialize arrays for FSI boundaries initial coordinates
Xinit = new double [*np_its_str_ifs];
Yinit = new double [*np_its_str_ifs];
Zinit = new double [*np_its_str_ifs];

for (int i = 0; i < *np_its_str_ifs; ++i)
{
   Xinit[i] = X_ITS_STR_IFS[i];
   Yinit[i] = Y_ITS_STR_IFS[i];
   Zinit[i] = Z_ITS_STR_IFS[i];
}

// Full run or restart run
FSI_Run_Type();

// Create header for tip displacement file
TipDisp << "Node# " << "time" << " " << "dX" << " " << "dY" << " " << "Node# "
<< "time" << " " << "dX" << " " << "dY" << " " << "Node# " << "time" << " " <<
"dX" << " " << "dY" << endl;

// Create header for non-linear iteration displacement file
NL_TipDisp << "time" << " " << "dX" << " " << "dY" << endl;

// Create header for fluid force file
ForceX << "time";
for (int i = 0; i <FSI_Node_Size; ++i)
```

```
{
    ForceX << " " << "Node" << Zorglib_FSI_Nodes[i];
}
ForceX << endl;

ForceY << "time";
for (int i = 0; i <FSI_Node_Size; ++i)
{
    ForceY << " " << "Node" << Zorglib_FSI_Nodes[i];
}
ForceY << endl;

// Check the input parameters for ISIS
ofstream checkISIS("checkISIS.txt");
checkISIS << "*nt_its_str_ifs = " << *nt_its_str_ifs << endl;
checkISIS << "*np_its_str_ifs = " << *np_its_str_ifs << endl << endl;
checkISIS << "Node# " << "X_ITS_STR_IFS " << "Y_ITS_STR_IFS " <<
"Z_ITS_STR_IFS " << endl;

for (int i = 0; i < *np_its_str_ifs; ++i)
{
    checkISIS << i+1 << " " << X_ITS_STR_IFS[i] << " " << Y_ITS_STR_IFS[i] << " "
    << Z_ITS_STR_IFS[i] << endl;
}

checkISIS << endl;
checkISIS << "Con_ITS_STR_IFS" << endl;

for (int i = 0; i <*nt_its_str_ifs; ++i)
{
    checkISIS << Con_ITS_STR_IFS[i][0] << " " << Con_ITS_STR_IFS[i][1] << " " <<
    Con_ITS_STR_IFS[i][2] << endl;
}
```

```
    cout << "********************** END initGENERIC_ifs Subroutine
    ************************" << endl;

}
```

B.2 GENERIC_ifs.cpp

```
#include "init_ISIS.h"

using namespace std;

extern "C" void generic_ifs(int * imesg,int * mybloc,int * itt,int * itnl,double * tc,double * dtc,
double * dfx_ITS_STR_IFS,double * dfy_ITS_STR_IFS,double * dfz_ITS_STR_IFS,int *
np_its_str_ifs,int * nt_its_str_ifs,double * X_ITS_STR_IFS,double * Y_ITS_STR_IFS,double *
Z_ITS_STR_IFS,int Con_ITS_STR_IFS[][3],
int * nbody,int * ID_Body,char * Name_Body,double O1ref_R0[][3],double
O1tc_R0[][3][3],double Omega1tc[][2][3],double Theta1tc[][3][3] )
{
    cout << "******************** GENERIC_ifs Subroutine **********************" <<
    endl;

    // Save the current time for archival purposes
    Time_curent = *tc;

    // Stop computing if the fluid time step and solid one are different
    if(FEMtimeStep != *dtc)
    {
        cout << "FEMtimeStep != dtc" << endl;
        return;
    }

    // Reset Parameters at new time step
    if(*itnl == 1)
```

```
{
        *uSaved = *u;
        *vSaved = *v;

    for (int i = 0; i < FSI_Node_Size; ++i)
    {
    iterdispX[i] = 0.0;
    iterdispY[i] = 0.0;
        IFS_Force_X[i] = 0.0;
        IFS_Force_Y[i] = 0.0;
    }
}

// Revert back the displacement and velocity to their state before the nonlinear iteration
*u = *uSaved;
*v = *vSaved;

// Save the previous iteration displacement
*iterdispX_old = *iterdispX;
*iterdispY_old = *iterdispY;

// Under Relax the fluid forces
for (int i = 0; i <FSI_Node_Size; ++i)
{
    IFS_Force_X[i] = omegaFSI*dfx_ITS_STR_IFS[i] + (1-omegaFSI)*IFS_Force_X[i];
    IFS_Force_Y[i] = omegaFSI*dfy_ITS_STR_IFS[i] + (1-omegaFSI)*IFS_Force_Y[i];
}

// Save fluid forces
ForceX << Time_curent << " ";
for (int i = 0; i <FSI_Node_Size; ++i)
{
    ForceX << " " << IFS_Force_X[i];
}
```

```
ForceX << endl;

ForceY << Time_curent << " ";
for (int i = 0; i <FSI_Node_Size; ++i)
{
    ForceY << " " << IFS_Force_Y[i];
}
ForceY << endl;



// Run NewmarkIntegrator
FEMIntegrator->setUpdateTangent(true);
FEMIntegrator->setCurrentTime(*tc);
FEMIntegrator->setTimeStep(*dtc);
FEMIntegrator->run(*u,*v,*tc,*tc+*dtc, &std::cout);

// Obtain the displcement of the boundary nodes
for (int i = 0; i < FSI_Node_Size; ++i)
{
    Node& FSI_Node = FSI_NodeSet.node(Zorglib_Force_Con[i]);

    iterdispX[i] = FEMSystem->getNodalValue(FSI_Node,0,"DISPLACEMENTS");

    iterdispY[i] = FEMSystem->getNodalValue(FSI_Node,1,"DISPLACEMENTS");
}

// perform under-relaxation for the structural displacement
for (int i = 0; i < FSI_Node_Size; ++i)
{
    iterdispX[i] = omegaFSI*iterdispX[i] + (1-omegaFSI)*iterdispX_old[i];

    iterdispY[i] = omegaFSI*iterdispY[i] + (1-omegaFSI)*iterdispY_old[i];
}
```

```
    // Save a boundary node displacement to check stability during non-linear iterations
        NL_TipDisp << Time_curent << " " << iterdispX[(FSI_Node_Size/2)] << " " <<
        iterdispY[(FSI_Node_Size/2)] << endl;

    // Update FSI Interface Coordinates
    for( int i1 = 0; i1 < FSI_Node_Size ;i1++ )
    {
        X_ITS_STR_IFS[i1] = Xinit[i1] + iterdispX[i1];
        Y_ITS_STR_IFS[i1] = Yinit[i1] + iterdispY[i1];

    }

    cout << "******************  END GENERIC_ifs Subroutine  ********************"
    << endl;

}
```

B.3 saveGENERIC_ifs.cpp

```
#include "init_ISIS.h"

using namespace std;

extern "C" void savegeneric_ifs(int * imesg,int * mybloc,int * itt,int * itte)
{

    // Zorglib archive

    if(numberArch1%Archive_Interval == 0)
    {
        numberArch2 += 1;
        numberArch3 += 1;
        FEMTask2->process(numberArch2,Time_curent,*FEMSystem,&std::cout);
```

```
    FEMTask3->process(numberArch3,Time_curent,*FEMSystem,&std::cout);
}

// Save tip displacement - Only for the purpose of the analysis in this thesis

double dx, dy;
{
    Node& FSI_Node = FSI_NodeSet.node((FSI_Node_Size/2));
    dx = FEMSystem->getNodalValue(FSI_Node,0,"DISPLACEMENTS");
    dy = FEMSystem->getNodalValue(FSI_Node,1,"DISPLACEMENTS");
    TipDisp << FSI_Node.label() << " " << Time_curent << " " << dx << " " << dy << "
    ";
}

{
    Node& FSI_Node = FSI_NodeSet.node((FSI_Node_Size/2)+1);
    dx = FEMSystem->getNodalValue(FSI_Node,0,"DISPLACEMENTS");
    dy = FEMSystem->getNodalValue(FSI_Node,1,"DISPLACEMENTS");
    TipDisp << FSI_Node.label() << " " << Time_curent << " " << dx << " " << dy << "
    ";
}

{
    Node& FSI_Node = FSI_NodeSet.node((FSI_Node_Size/2)-2);
    dx = FEMSystem->getNodalValue(FSI_Node,0,"DISPLACEMENTS");
    dy = FEMSystem->getNodalValue(FSI_Node,1,"DISPLACEMENTS");
    TipDisp << FSI_Node.label() << " " << Time_curent << " " << dx << " " << dy <<
    endl;
}

// Save Displacement, Velocity, and Acceleration system arrays
 *output_U << *u << endl;
*output_V << *v << endl;
NewmarkIntegrator::Acceleration FEAcceleration = FEMIntegrator->acceleration();
```

```
        *output_A << FEAcceleration << endl;

        // Save interface displacement
        ofstream FSI_Interface_Disp("FSI_Interface_Disp.txt");

        FSI_Interface_Disp << Time_curent << endl;

        for (int i = 0; i < FSI_Node_Size; ++i)
        {

            FSI_Interface_Disp << iterdispX[i] << " ";
        }

        FSI_Interface_Disp << endl;

        for (int i = 0; i < FSI_Node_Size; ++i)
        {
            FSI_Interface_Disp << iterdispY[i] << " ";
        }

}
```

B.4 Read_FSI_Input

```
#include "init_ISIS.h"

using namespace std;

void Read_FSI_Inputs()
{
    // Read Input Parameters File - FSI_Inputs.txt
    ifstream FSI_Inputs("FSI_Inputs.txt");
    string inputs;
```

```
// Read problem dimension
getline(FSI_Inputs, inputs);
getline(FSI_Inputs, inputs);
istringstream input_dim(inputs);
input_dim >> dimFEM;

// Read time step
getline(FSI_Inputs, inputs);
getline(FSI_Inputs, inputs);
istringstream input_timestep(inputs);
input_timestep >> FEMtimeStep;

// Read physical number used for FSI boundary identification
getline(FSI_Inputs, inputs);
getline(FSI_Inputs, inputs);
istringstream input_FSIPhyNum(inputs);
input_FSIPhyNum >> FSIPhyNum;

// Read solid constitutive model
getline(FSI_Inputs, inputs);
getline(FSI_Inputs, inputs);
istringstream input_Constitutive_Model(inputs);
input_Constitutive_Model >>  Constitutive_Model;

// Read solid 2D assumption
getline(FSI_Inputs, inputs);
getline(FSI_Inputs, inputs);
istringstream input_TwoD_Assumption(inputs);
input_TwoD_Assumption >>  TwoD_Assumption;

// Read FE System solver
getline(FSI_Inputs, inputs);
getline(FSI_Inputs, inputs);
```

```
    istringstream input_System_Solver(inputs);
    input_System_Solver >>  System_Solver;

    // Read FE Time Integrator
    getline(FSI_Inputs, inputs);
    getline(FSI_Inputs, inputs);
    istringstream input_System_Integrator(inputs);
    input_System_Integrator >>  System_Integrator >> SPECTRAL_RADIUS;

    // Read the under-relaxation parameter
    getline(FSI_Inputs, inputs);
    getline(FSI_Inputs, inputs);
    istringstream input_omegaFSI(inputs);
    input_omegaFSI >> omegaFSI;

    // Read the Archival Interval
    getline(FSI_Inputs, inputs);
    getline(FSI_Inputs, inputs);
    istringstream input_Archive_Interval(inputs);
    input_Archive_Interval >> Archive_Interval;

    // Read the Full Run or Restart option
    getline(FSI_Inputs, inputs);
    getline(FSI_Inputs, inputs);
    istringstream input_RunType(inputs);
    input_RunType >> RunType >> RestartStep;
    cout << RunType << " " << RestartStep << endl;
}
```

B.5 Build_FSI_Arrays

```
#include "init_ISIS.h"

using namespace std;

void Build_FSI_Arrays(int * np_its_str_ifs, int * nt_its_str_ifs, double * X_ITS_STR_IFS, double * Y_ITS_STR_IFS, double * Z_ITS_STR_IFS, int Con_ITS_STR_IFS[][3])
{

    // Read GMSH mesh file and return the elements (line element -> dimFEM-1 = 1) on the FSI boundary based on the physical number (FSIPhyNum)
    themesh = GmshMeshIO::readMesh(dimFEM,"body","meshZorg.msh");
    FSI_ElementSet = themesh->getElements(dimFEM-1,FSIPhyNum);

    // Get the number of FSI elements
    int FSI_Element_Size;
    FSI_Element_Size = FSI_ElementSet.nElements();

    // Collect FSI boundary nodes
    FSI_ElementSet.collectNodes(FSI_NodeSet);

    // Get the number of FSI nodes
    FSI_Node_Size = FSI_NodeSet.nNodes();
    *np_its_str_ifs = FSI_Node_Size;

    // Check FSI boundary nodes and element size
    ofstream outpFSI_Nodes("outpFSI_Nodes.txt");
    outpFSI_Nodes << "FSI_Node_Size: " << FSI_Node_Size << endl;
    outpFSI_Nodes << "FSI_Element_Size: " << FSI_Element_Size << endl;

    // Check element connectivity in GMSH mesh file
    outpFSI_Nodes << endl << endl;
    outpFSI_Nodes << "Checking element connectivity" << endl << endl;
```

```
int Size_FSI_Node_Element;
int Size_Zorglib_Con = 0;

for (int i = 0; i < FSI_Element_Size; ++i)
{
    Element& FSI_Element = FSI_ElementSet.element(i);
    Size_FSI_Node_Element = FSI_Element.nNodes();
    Size_Zorglib_Con += Size_FSI_Node_Element;

    outpFSI_Nodes << "Element Connectivity:  ";

    for (int j = 0; j < Size_FSI_Node_Element; ++j)
    {
        Node& Element_Node = FSI_Element.node(j);
        outpFSI_Nodes << Element_Node.label() << " ";
    }

    outpFSI_Nodes << endl;
}

Size_Zorglib_Con += FSI_Element_Size;

outpFSI_Nodes << endl;
outpFSI_Nodes << "Zorglib_FSI_Con" << endl;

Zorglib_FSI_Con = new int[Size_Zorglib_Con];

{
    int iCon = 0;

    for (int i = 0; i < FSI_Element_Size; ++i)
    {
        Element& FSI_Element = FSI_ElementSet.element(i);
```

```
        Size_FSI_Node_Element = FSI_Element.nNodes();

        Zorglib_FSI_Con[iCon] = Size_FSI_Node_Element;
        outpFSI_Nodes << Zorglib_FSI_Con[iCon] << endl;
        iCon += 1;

        for (int j = 0; j < Size_FSI_Node_Element; ++j)
        {
            Node& Element_Node = FSI_Element.node(j);
            Zorglib_FSI_Con[iCon] = Element_Node.label();

            outpFSI_Nodes << Zorglib_FSI_Con[iCon] << endl;

            iCon += 1;
        }
    }
}

outpFSI_Nodes << endl;

cout << Size_Zorglib_Con << endl;

int Size_ISIS_FSI_Con = 0;

// Initialize nt_its_str_ifs
*nt_its_str_ifs = 0;

// A pointer array to obtain a consequtive nodes indexes. It is needed primarily because when triangle and quad order 2 element
// is used the middle node index is not located in the middle of the line connectivity in GMSH mesh file.
Zorglib_Force_Con = new int[FSI_Node_Size];

// 1D array of list of consecutive nodes indexes on the FSI boundary after sorting
```

```
Zorglib_FSI_Nodes = new int[FSI_Node_Size];

{
    int iZorg = 1;
    int swap;

    Node& FSI_Node = FSI_NodeSet.node(0);
    Zorglib_FSI_Nodes[0] = FSI_Node.label();
    Zorglib_Force_Con[0] = 0;

    // 2D Properties
    X_ITS_STR_IFS[0] = FSI_Node.coordinates().X(0);
    Y_ITS_STR_IFS[0] = FSI_Node.coordinates().X(1);
    Z_ITS_STR_IFS[0] = 0.0;

    outpFSI_Nodes << FSI_Node.label() << " " << FSI_Node.coordinates().X(0) << " "
    << FSI_Node.coordinates().X(1) << endl;

    for (int i = 0; i < Size_Zorglib_Con; ++i)
    {

        if (Zorglib_FSI_Con[i] == 3)
        {
            // when there are multiple bodies and the connectivity starts from the next body
            if (iZorg > 6 && Zorglib_FSI_Con[i+1] != Zorglib_FSI_Con[i-1])
            {
                Node& FSI_Node = FSI_NodeSet.node(iZorg);
                Zorglib_FSI_Nodes[iZorg] = FSI_Node.label();
                Zorglib_Force_Con[iZorg] = iZorg;

                X_ITS_STR_IFS[iZorg] = FSI_Node.coordinates().X(0);
                Y_ITS_STR_IFS[iZorg] = FSI_Node.coordinates().X(1);
                Z_ITS_STR_IFS[iZorg] = 0.0;
```

```
    outpFSI_Nodes << FSI_Node.label() << " " << FSI_Node.coordinates().X(0)
    << " " << FSI_Node.coordinates().X(1) << endl;

    iZorg += 1;
 }

 Node& FSI_Node1 = FSI_NodeSet.node(iZorg);

 swap = FSI_Node1.label();

// 2D Properties
 X_ITS_STR_IFS[iZorg + 1] = FSI_Node1.coordinates().X(0);
 Y_ITS_STR_IFS[iZorg + 1] = FSI_Node1.coordinates().X(1);
 Z_ITS_STR_IFS[iZorg + 1] = 0.0;

 outpFSI_Nodes << FSI_Node1.label() << " " << FSI_Node1.coordinates().X(0)
 << " " << FSI_Node1.coordinates().X(1) << endl;

 Node& FSI_Node2 = FSI_NodeSet.node(iZorg+1);

 // 2D Properties
 X_ITS_STR_IFS[iZorg] = FSI_Node2.coordinates().X(0);
 Y_ITS_STR_IFS[iZorg] = FSI_Node2.coordinates().X(1);
 Z_ITS_STR_IFS[iZorg] = 0.0;

 outpFSI_Nodes << FSI_Node2.label() << " " << FSI_Node2.coordinates().X(0)
 << " " << FSI_Node2.coordinates().X(1) << endl;

 Zorglib_FSI_Nodes[iZorg] = FSI_Node2.label();

 Zorglib_FSI_Nodes[iZorg+1] = swap;

int swap2;
 swap2 = Zorglib_FSI_Con[i+2];
```

```
    Zorglib_FSI_Con[i+2] = Zorglib_FSI_Con[i+3];
    Zorglib_FSI_Con[i+3] = swap2;

    Zorglib_Force_Con[iZorg] = iZorg+1;
    Zorglib_Force_Con[iZorg+1] = iZorg;

    outpFSI_Nodes << Zorglib_FSI_Con[i] << endl;
    outpFSI_Nodes << Zorglib_FSI_Con[i+1] << endl;
    outpFSI_Nodes << Zorglib_FSI_Con[i+2] << endl;
    outpFSI_Nodes << Zorglib_FSI_Con[i+3] << endl;

    i += Zorglib_FSI_Con[i];
    iZorg += 2;

    // 2D Properties
    Size_ISIS_FSI_Con += 6;

    // the number of segment for ISIS
    *nt_its_str_ifs += 2;
}

else if (Zorglib_FSI_Con[i] == 2)
{
    Node& FSI_Node1 = FSI_NodeSet.node(iZorg);

    // 2D Properties
    X_ITS_STR_IFS[iZorg] = FSI_Node1.coordinates().X(0);
    Y_ITS_STR_IFS[iZorg] = FSI_Node1.coordinates().X(1);
    Z_ITS_STR_IFS[iZorg] = 0.0;

    outpFSI_Nodes << FSI_Node1.label() << " " << FSI_Node1.coordinates().X(0)
    << " " << FSI_Node1.coordinates().X(1) << endl;

    Zorglib_FSI_Nodes[iZorg] = FSI_Node1.label();
```

```
            Zorglib_Force_Con[iZorg] = iZorg;
            Zorglib_Force_Con[iZorg + 1] = iZorg + 1;

            outpFSI_Nodes << Zorglib_FSI_Con[i] << endl;
            outpFSI_Nodes << Zorglib_FSI_Con[i+1] << endl;
            outpFSI_Nodes << Zorglib_FSI_Con[i+2] << endl;

            i += Zorglib_FSI_Con[i];
            ++iZorg;

            // 2D Properties
            Size_ISIS_FSI_Con += 3;

            // the number of segment for ISIS
            *nt_its_str_ifs += 1;
        }

    }
}

// Check Zorglib_FSI_Nodes array
outpFSI_Nodes << endl;
outpFSI_Nodes << "Zorglib_FSI_Nodes" << endl;

for (int i = 0; i < FSI_Node_Size; ++i)
{
        outpFSI_Nodes << Zorglib_FSI_Nodes[i] << endl;
}


// Build Con_ITS_STR_IFS
{
    int j = 0, k = 1, l = 0;
```

```
for (int i = 0; i < Size_Zorglib_Con; ++i)
{
    // for quadratic element
    if (Zorglib_FSI_Con[i] == 3)
    {
        // 2D Properties
        Con_ITS_STR_IFS[j][0] = k;
        Con_ITS_STR_IFS[j][1] = k+1;
        Con_ITS_STR_IFS[j][2] = 0;
        Con_ITS_STR_IFS[j+1][0] = k+1;
        Con_ITS_STR_IFS[j+1][1] = k+2;
        Con_ITS_STR_IFS[j+1][2] = 0;

        // 2D Properties
        j += 2;

        i += Zorglib_FSI_Con[i];

        if (Zorglib_FSI_Con[i+2] == Zorglib_FSI_Nodes[l+2])
        {
            k += 2;
            l += 2;
        }
        else
        {
            k += 3;
            l += 3;
        }
    }

    // for linear element
    else if (Zorglib_FSI_Con[i] == 2)
    {
```

```
            // 2D Properties
            Con_ITS_STR_IFS[j][0] = k;
            Con_ITS_STR_IFS[j][1] = k+1;
            Con_ITS_STR_IFS[j][2] = 0;

            // 2D Properties
            j += 1;

            i += Zorglib_FSI_Con[i];

            if (Zorglib_FSI_Con[i+2] == Zorglib_FSI_Nodes[l+1])
            {
                k += 1;
                l += 1;
            }
            else
            {
                k += 2;
                l += 2;
            }
        }

    }
  }

}
```

B.6 init_ISIS.cpp

```
#include "init_ISIS.h"
#include "IFSFunction.h"

using namespace std;
```

```
#ifdef USE_ZORGLIB_NAMESPACE
USING_ZORGLIB_NAMESPACE
#endif

// copy constructor
IFSFunction::IFSFunction(const IFSFunction& src)
 : Function(src) {
    idx = src.idx;
}

// get value
double IFSFunction::value(double t) {
    unsigned int i;
    unsigned int r;
    i = idx/2;
    r = idx%2;
    if (r == 0)
        return IFS_Force_X[i];
    else
        return IFS_Force_Y[i];
}

// get derivative
double IFSFunction::slope(double t) {
    return 0.0;
}

// get value and derivative
double IFSFunction::value(double t,double& df) {
   df = 0.0e0;
    unsigned int i;
    unsigned int r;
    i = idx/2;
```

```
    r = idx%2;
    if (r == 0)
        return IFS_Force_X[i];
    else
        return IFS_Force_Y[i];
}

// print-out
std::string IFSFunction::toString() const {
    std::string s = "nothing";
    return s;
}

// Zorglib Static Parameters
double FEMtimeStep;
int dimFEM;
int dimFEMelem;
int FSIPhyNum;
double SPECTRAL_RADIUS;
string Constitutive_Model;
string TwoD_Assumption;
string System_Integrator;
string System_Solver;
int Archive_Interval;
string RunType;
int RestartStep;
MeshedBody* themesh;
vector<Element*> ElemSet;
vector<Node*> nodeSetFSI;
ElementSet FSI_ElementSet;
NodeSet FSI_NodeSet;
ConstitutiveModel* model;
MaterialProperties theproperties;
MaterialModel* theMaterial;
```

```
FEMechanicalSystem* FEMSystem;
SystemArray* u;
SystemArray* v;
SystemArray* uSaved;
SystemArray* vSaved;
SystemArray* ExtForces;
NewmarkIntegrator* FEMIntegrator;
StationaryEquation* StaticEquation;
LinearEquationSolver* StaticSolver;
FEGmshArchivalTask* FEMTask1;
FEGmshArchivalTask* FEMTask2;
FEGmshArchivalTask* FEMTask3;
FEGmshArchivalTask* FEMTask4;
int numberArch1 = 0, numberArch2 = 0, numberArch3 = 0;
int nNLIter = 1;

// Connectivity array to transfer structural displacement from Zorglib to ISIS
int* Zorglib_Force_Con;

// Arrays of displacements of the current iteration
double* iterdispX;
double* iterdispY;
double* iterdispZ;

// Arrays of displacements from the previous iteration
double* iterdispX_old;
double* iterdispY_old;
double* iterdispZ_old;

// Under-relaxation parameter
double omegaFSI;

// To store mesh data
int FSI_Node_Size;
```

```
int* Zorglib_FSI_Nodes;
int* MESH_FSI_Con;
int* Zorglib_FSI_Con;

// To store fluid force
double* IFS_Force_X;
double* IFS_Force_Y;
double* IFS_Force_Z;

// To store initial FSI coordinates
double *Xinit, *Yinit, *Zinit;

// Current step time for archival purposes
double Time_curent = 0.0;

// Output file declaration
ofstream TipDisp("TipDisp.txt");
ofstream NL_TipDisp("NL_TipDisp.txt");
ofstream ForceX("IFS_Force_X.txt");
ofstream ForceY("IFS_Force_Y.txt");
ofstream outpNodeDOF("outpNodeDOF.txt");

// Input and output streams for Restart computation
ifstream* input_U;
ifstream* input_V;
ifstream* input_A;
ofstream* output_U;
ofstream* output_V;
ofstream* output_A;
```

B.7 init_ISIS.h

```
#ifndef INIT_ISIS_H
#define INIT_ISIS_H

// std C++ library
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <math.h>

// local
#include <algo/NewmarkIntegrator.h>
#include <algo/ConstantTimeStepper.h>
#include <algo/NewtonSolver.h>
#include <algo/LinearEquationSolver.h>
#include <algo/StationaryEquation.h>
#include <zfem/FEMechanicalSystem.h>
#include <mesh/GmshMeshIO.h>
#include <mesh/GmshViewIO.h>
#include <zfem/FEGmshArchivalTask.h>
#include <zfem/FEReactionArchivalTask.h>
#include <matl/ModelDictionary.h>
#include <data/Chronometer.h>
#include <data/MixedData.h>
#include <matl/MaterialModel.h>

using namespace std;

#ifdef USE_ZORGLIB_NAMESPACE
USING_ZORGLIB_NAMESPACE
#endif
```

```
// Internal Functions
void Read_FSI_Inputs();
void FSI_Run_Type();
void Zorglib_FE_Initialization();
void Zorglib_Integrator_Initialization();
void Zorglib_Archival_Initialization();
void Build_FSI_Arrays(int *, int *, double *, double *, double *, int [][3]);

// Zorglib Static Parameters
extern int dimFEM;
extern double FEMtimeStep;
extern int FSIPhyNum;
extern double SPECTRAL_RADIUS;
extern string Constitutive_Model;
extern string TwoD_Assumption;
extern string System_Integrator;
extern string System_Solver;
extern int Archive_Interval;
extern string RunType;
extern int RestartStep;
extern MeshedBody* themesh;
extern vector<Element*> ElemSet;
extern vector<Node*> nodeSetFSI;
extern ElementSet FSI_ElementSet;
extern NodeSet FSI_NodeSet;
extern ConstitutiveModel* model;
extern MaterialProperties theproperties;
extern int dimFEMelem;
extern MaterialModel* theMaterial;
extern FEMechanicalSystem* FEMSystem;
extern FEMechanicalSystem* FEMSystemInit;
extern FEMechanicalSystem* FEMSystemStatic;
extern SystemArray* u;
```

```
extern SystemArray* v;
extern SystemArray* uSaved;
extern SystemArray* vSaved;
extern NewmarkIntegrator* FEMIntegrator;
extern SystemArray* ExtForces;
extern StationaryEquation* StaticEquation;
extern LinearEquationSolver* StaticSolver;
extern double FEMtimeStep;
extern FEGmshArchivalTask* FEMTask1;
extern FEGmshArchivalTask* FEMTask2;
extern FEGmshArchivalTask* FEMTask3;
extern FEGmshArchivalTask* FEMTask4;
extern int numberArch1, numberArch2, numberArch3;

// Connectivity array to transfer structural displacement from Zorglib to ISIS
extern int* Zorglib_Force_Con;

// Arrays of displacements of the current iteration
extern double* iterdispX;
extern double* iterdispY;
extern double* iterdispZ;

// Arrays of displacements from the previous iteration
extern double* iterdispX_old;
extern double* iterdispY_old;
extern double* iterdispZ_old;

// Under-relaxation parameter
extern double omegaFSI;

// To store mesh data
extern int FSI_Node_Size;
extern int* Zorglib_FSI_Nodes;
extern int* MESH_FSI_Con;
```

```
extern int* Zorglib_FSI_Con;

// To store initial FSI coordinates
extern double *Xinit, *Yinit, *Zinit;

// To store fluid force
extern double* IFS_Force_X;
extern double* IFS_Force_Y;
extern double* IFS_Force_Z;

// Output file declaration
extern ofstream TipDisp;
extern ofstream NL_TipDisp;
extern ofstream ForceX;
extern ofstream ForceY;
extern ofstream outpNodeDOF;

// Input and output streams for Restart computation
extern ifstream* input_U;
extern ifstream* input_V;
extern ifstream* input_A;
extern ofstream* output_U;
extern ofstream* output_V;
extern ofstream* output_A;

// Current step time for archival purposes
extern double Time_curent;

#endif
```

B.8 Zorglib_FE_Initialization.cpp

```
#include "init_ISIS.h"
#include "IFSFunction.h"

using namespace std;

void Zorglib_FE_Initialization()
{
    // Build Material Model
    model = ModelDictionary::build( Constitutive_Model,dimFEM);

    // Read and Initialize Material Properties
    theproperties.readFrom("Solid_Properties.mat");
    theMaterial = new MaterialModel(*model,theproperties);
    theMaterial->initialize();

    // Context
    Context ContextCTX(dimFEM);
    Context::Symmetry inputSymmetry = Context::NONE;
    ContextCTX.setSymmetry(inputSymmetry);

    // Create FE system
    FEMSystem = new FEMechanicalSystem(ContextCTX,*themesh);
    FEMSystem->addDeformableBody(*themesh,"DISPLACEMENTS");

    // set FE formulation
    MixedData* FEMParams = new MixedData[1];
    FEMParams[0] = 1.0;
    std::vector<std::string> FEDofs(1);
    FEDofs[0] = "DISPLACEMENTS";
    FEMSystem-
    >setFormulation("body",dimFEM,1,TwoD_Assumption,FEMParams,*theMaterial,FEDofs);
```

```
// Apply Boundary Conditions
FEMSystem->setSimpleConstraint("body",dimFEM-1,101,0.0,0,FEDofs[0]);
FEMSystem->setSimpleConstraint("body",dimFEM-1,101,0.0,1,FEDofs[0]);

// Initialize fluid force arrays
IFS_Force_X = new double[FSI_Node_Size];
IFS_Force_Y = new double[FSI_Node_Size];
IFS_Force_Z = new double[FSI_Node_Size];

for (int i = 0; i < FSI_Node_Size; ++i)
{
   IFS_Force_X[i] = 0.0;
   IFS_Force_Y[i] = 0.0;
}

// Create and fnitialize force function (IFSFunction) and Node objects
IFSFunction* fcts[FSI_Node_Size*2];

for (int i = 0; i < FSI_Node_Size*2; ++i)
{
   fcts[i] = new IFSFunction(i);
}

{
   int iF = 0;
   for(int i = 0; i < FSI_Node_Size; ++i)
   {
      Node& FSI_Node = FSI_NodeSet.node(Zorglib_Force_Con[i]);
      FEMSystem->setAppliedForce(FSI_Node,*fcts[iF],0,"DISPLACEMENTS");
      FEMSystem->setAppliedForce(FSI_Node,*fcts[iF+1],1,"DISPLACEMENTS");

      iF += 2;
   }
}
```

```
    // Initialize FE System
    FEMSystem->initialize();
}
```

B.9 Zorglib_Integrator_Initialization.cpp

```
#include "init_ISIS.h"
using namespace std;

void Zorglib_Integrator_Initialization()
{
    // Initialize System Arrays
    u = new SystemArray(*FEMSystem);
    v = new SystemArray(*FEMSystem);

    *u = 0.0;
    *v = 0.0;

    // Saved u and v variables
    uSaved = new SystemArray(*FEMSystem);
    vSaved = new SystemArray(*FEMSystem);
    *uSaved = 0.0;
    *vSaved = 0.0;

    // Initialize time integrator
    FEMIntegrator = new NewmarkIntegrator(*FEMSystem,"","FULL",false,false);

    // Set algorithmic parameters for the time integrator

    StringMap<double>::Type FEMalgParams;
    double AL_F = 0.0, AL_M = 0.0, BET = 0.0, GAM = 0.0;
```

```
if(System_Integrator == "STANDARD_NEWMARK")
{
    // Standard Newmark Paramaters

    AL_F = 0.0;
    AL_M = 0.0;
    BET = 0.25;
    GAM = 0.5;
}
else if(System_Integrator == "GENERAL_ALPHA")
{
    // General-alpha Paramaters

    AL_F = SPECTRAL_RADIUS / (1 + SPECTRAL_RADIUS);
    AL_M = (2*SPECTRAL_RADIUS - 1) / (1 + SPECTRAL_RADIUS);
    BET = 0.25 * (pow((1 - AL_M + AL_F),2));
    GAM = 0.5 - AL_M + AL_F;
}

// Set Integrator Parameters

FEMalgParams["ALPHA_F"] = AL_F;
FEMalgParams["ALPHA_M"] = AL_M;
FEMalgParams["BETA"]    = BET;
FEMalgParams["GAMMA"]   = GAM;

FEMalgParams["INITIAL_TIME_STEP"] = FEMtimeStep;
FEMalgParams["MIN_TIME_STEP"] = 0.1*FEMtimeStep;
FEMalgParams["MAX_TIME_STEP"] = FEMtimeStep;
FEMalgParams["MAX_DIVISIONS"] = 10.;

FEMIntegrator->setParameters(FEMalgParams);
```

```
    if(System_Solver == "LINEAR")
    {
        // Initialize LinearEquationSolver

        LinearEquationSolver* FEMSolver;
        FEMSolver = new LinearEquationSolver(*FEMSystem);
        FEMIntegrator->setSolver(*FEMSolver);
    }
    else if(System_Solver == "NEWTON")
    {
        // Initialize NewtonSolver

        NewtonSolver* FEMSolver;
        FEMSolver = new NewtonSolver(*FEMSystem,"","FULL");
        LineSearch* FEMlsrch;
        FEMlsrch = new LineSearch(1e-2,5,1000);
        FEMSolver->setLineSearch(*FEMlsrch);
        FEMSolver->setMaxIter(50);
        FEMSolver->setTolerance(1.e-8);
        FEMSolver->setAbsoluteTolerance(1.e-12);
        FEMSolver->setPrecision(1.e-12);
        FEMIntegrator->setSolver(*FEMSolver);
    }

    // Time integrator general log file

    FEMIntegrator->setLogFile("integ_log.plt");
}
```

B.10 Zorglib_Archival_Initialization.cpp

```
#include "init_ISIS.h"

using namespace std;

void Zorglib_Archival_Initialization()
{

    GmshMeshIO::writeMesh(*themesh,"resultsZorg.msh");
    GmshViewIO::Data outputNode = GmshViewIO::NODE;
    GmshViewIO::Data outputElement_Node = GmshViewIO::ELEMENT_NODE;
    GmshViewIO::Type outputVector = GmshViewIO::VECTOR;
    GmshViewIO::Type outputTensor = GmshViewIO::TENSOR;
    FEGmshArchivalTask::Format outputFormat = FEGmshArchivalTask::ASCII;

    // Save displacement values during NonLinear Iteration
    FEMTask1 = new
    FEGmshArchivalTask("DISPLACEMENTS","body",dimFEMelem,1,outputNode,outputVe
    ctor,"Displacements","resultsZorg-NL_Iteration",outputFormat,0);
    FEMTask1->setFormatWidth(5);
    FEMTask1->setTimeInterval(FEMtimeStep);

    // Save displacement values after a certain time interval
    FEMTask2 = new
    FEGmshArchivalTask("DISPLACEMENTS","body",dimFEMelem,1,outputNode,outputVe
    ctor,"Displacements","resultsZorg-Displ",outputFormat,0);
    FEMTask2->setFormatWidth(5);
    FEMTask2->setTimeInterval(FEMtimeStep);

    // Save stress values after a certain time interval
    FEMTask3 = new
    FEGmshArchivalTask("stress","body",dimFEMelem,1,outputElement_Node,outputTensor,"
    Stresses","resultsZorg-Stress",outputFormat,0);
```

```
    FEMTask3->setFormatWidth(5);
    FEMTask3->setTimeInterval(FEMtimeStep);

    // Save velocity values after a certain time interval
    FEMTask4 = new
    FEGmshArchivalTask("DISPLACEMENTS","body",dimFEMelem,1,outputNode,outputVe
    ctor,"Velocities","resultsZorg-Velocity",outputFormat,0);
    FEMTask4->setRateFlag(true);
    FEMTask4->setFormatWidth(5);
    FEMTask4->setTimeInterval(FEMtimeStep);
}
```

B.11 FSI_Run_Type.cpp

```
#include "init_ISIS.h"

using namespace std;

void FSI_Run_Type()
{
      if (RunType == "FULL")
      {
          output_U = new ofstream("FE_Displacements.txt");
          output_V = new ofstream("FE_Velocities.txt");
          output_A = new ofstream("FE_Accelerations.txt");

          FEMSystem->change(*u,*v);

          for (int i = 0; i < FSI_Node_Size; ++i)
          {
              iterdispX[i] = 0.0;
              iterdispY[i] = 0.0;
              iterdispZ[i] = 0.0;
```

```
    }
}

else if (RunType == "RESTART")
{
    input_U = new ifstream("FE_Displacements.txt");
    input_V = new ifstream("FE_Velocities.txt");
    input_A = new ifstream("FE_Accelerations.txt");
    string lineInput_U,lineInput_V,lineInput_A;
    Partition PFree = FEMSystem->getPartition(Partition::FREE);
    SystemArray a(*FEMSystem,PFree);
    int size_U = u->size();
    int size_V = v->size();
    int size_A = a.size();

    for(int i = 0; i < RestartStep-1; ++i)
    {
        getline(*input_U,lineInput_U);
        getline(*input_V,lineInput_V);
        getline(*input_A,lineInput_A);
    }

    getline(*input_U,lineInput_U);
    istringstream iss_U(lineInput_U);
    string inp1;
    iss_U >> inp1;

    for(int i = 0; i < size_U; ++i)
    {
        iss_U >> (*u)[i];
    }

    getline(*input_V,lineInput_V);
    istringstream iss_V(lineInput_V);
```

```
iss_V >> inp1;

for(int i = 0; i < size_V; ++i)
{
    iss_V >> (*v)[i];
}

getline(*input_A,lineInput_A);
istringstream iss_A(lineInput_A);
iss_A >> inp1;

for(int i = 0; i < size_A; ++i)
{
    iss_A >> a[i];
}

FEMSystem->change(*u,*v);
FEMIntegrator->setAcceleration(a);

output_U = new ofstream("FE_Displacements.txt");
output_V = new ofstream("FE_Velocities.txt");
output_A = new ofstream("FE_Accelerations.txt");

ifstream input_FSI_Interface_Disp("FSI_Interface_Disp.txt");
string line_FSI_Int;
getline(input_FSI_Interface_Disp,line_FSI_Int);
getline(input_FSI_Interface_Disp,line_FSI_Int);
istringstream iss_FSI_U(line_FSI_Int);

for (int i = 0; i < FSI_Node_Size; ++i)
{
    iss_FSI_U >> iterdispX[i];
}
```

```
            getline(input_FSI_Interface_Disp,line_FSI_Int);
            istringstream iss_FSI_V(line_FSI_Int);

            for (int i = 0; i < FSI_Node_Size; ++i)
            {
                iss_FSI_V >> iterdispY[i];
            }

        }
}
```

B.12 IFSFunction.h

```
#ifndef IFSFunction_H
#define IFSFunction_H

#include <data/Function.h>
#include <string>

#ifdef USE_ZORGLIB_NAMESPACE
USING_ZORGLIB_NAMESPACE
#endif

class IFSFunction : virtual public Function {

    protected:

        unsigned int idx;

    public:

        // constructor
        IFSFunction(unsigned int i, const std::string& s = "no name")
```

```
        : Function(s) {
            idx = i;
        }

        // copy constructor
        IFSFunction(const IFSFunction& );

        // destructor
        ~IFSFunction() {}

        // duplicate object
        IFSFunction* clone() const {return new IFSFunction(*this);}

        // get value
        double value(double );

        // get derivative
        double slope(double );

        // get value and derivative
        double value(double, double& );

        // print-out
        std::string toString() const;
};

#endif
```

References

1. Saracibar, C.A. (2011). *Lecture notes on continuum mechanics*, Universitat Politecnica de Catalunya, Barcelona.
2. Bonet, J., Wood, R.D. (1997). *Non linear continuum mechanics for finite element analysis*. Cambridge University Press, Cambridge
3. Bathe, K.-J. (1996). *Finite element procedures*. Prentice-Hall, New Jersey
4. Belytschko, T., Liu, W.K., Moran, B. (2000). *Nonlinear finite elements for continua and structures*. John Wiley & Sons Ltd, Chichester
5. Chung, J., Hulbert, G.M. (1993). A time integration algorithm for structural dynamics with improved numerical dissipation: the generalized-α method. *Journal of Applied Mechanics*, 60:371-375.
6. Hübner, B., Walhorn, E., Dinkler, D. (2004). A monolithic approach to fluid-structure interaction using space-time finite elements. *Computational Methods in Applied Mechanics and Engineering*, 193:2087-2104.
7. Dettmer, W.G., Perić D. (2007). A fully implicit computational strategy for strongly coupled fluid-solid interaction. *Archive of Computational Methods in Engineering*, 14:205-247
8. P. Causin, J.F. , Gerbeau, Nobile F. (2005). Added-mass effect in the design of partitioned algorithms for fluid–structure problems. *Computational Methods in Applied Mechanics and Engineering*, 194: 4506–4527
9. Farhat, C., Lesoinne, M., LeTallec, P. (1998). Load and motion transfer algorithms for fluid/structure interaction problems with non-matching discrete interfaces : momentum and energy conservation, optimal discretization and application to aeroelasticity. *Computational Methods in Applied Mechanics and Engineering*, 157:95-114.
10. EMN Ecole Centrale de Nantes (2011). Theoritical Manual for FINE™/MARINE, NUMECA International, Belgium
11. Geuzaine, C. , Remacle, J.-F. (2009). Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11): 1309-1331.
12. Turek, S. and Hron, J. (2006). Proposal for numerical benchmarking of fluid-structure interaction between an elastic object and laminar incompressible flow. *Lecture Notes in Computational Science and Engineering*, 53:371.
13. De Nayer G. (2008). Interaction fluide-structure pour les corps élancés. PhD thesis, Ecole Centrale de Nantes, France

14. Petersen C. (1996). Dynamik der Baukonstruktionen. *Vieweg*, Braunschweig/Wiesbaden
15. Wall, W.A., Ramm, E. (1998) Fluid–structure interaction based upon a stabilized (ALE) finite element method. *In: Idelsohn SR, Onate E (eds) Computational mechanics—new trends and applications*, 4thWorld Congress on Computational Mechanics, CIMNE, Barcelona, Spain, Buenos Aires, Brazil
16. Causin, P., Gerbeau, J.-F., Nobile, F. (2005). Added mass effect in the design of partitioned algorithms for fluid-structure problems. *Computer Methods in Applied Mechanics and Engineering*, 194(42-44):4506-4527.
17. Breuer, M., Bernsdorf, J., Zeiser, T., Durst, F. (2000). Accurate computations of the laminar flow past a square cylinder based on two different methods : lattice-Boltzmann and finite volume. *International Journal of Heat and Fluid Flow*, 21(186-196).
18. Piperno, S., Farhat, C., Larrouturou, B. (1995). Partitioned procedures for the transient solution of coupled aeroelastic problems. Part 1: Model problem, theory and two-dimensional application. *Computer Methods in Applied Mechanics and Engineering*, 124(79-112).
19. Leroyer, A. (2004). Etude du couplage écoulement/mouvement pour des corps solides ou à déformation imposée par résolution des équations de Navier-Stokes. Contribution à la modélisation numérique de la cavitation. PhD thesis, Ecole Centrale de Nantes, France
20. Durand, M. (2012). Interaction fluide-structure souple et légére, application aux voiliers. PhD thesis, Ecole Centrale de Nantes, France