A Monolithic Finite Element Model for Fluid–Structure Interaction Problems at Low Reynolds Numbers with Application to Cell Mechanics

Jakob Beran

Stockholm University

Master's Thesis (45 ECTS credits) within the
Master's Programme in Computational Physics
Department of Physics
Stockholm University
Academic year 2016/2017
Supervisor at KTH: Johan Hoffman
Supervisor at SU: Eva Lindroth
Examiner: Fawad Hassan

A Monolithic Finite Element Model for Fluid–Structure Interaction Problems at Low Reynolds Numbers with Application to Cell Mechanics

Jakob Beran

April 2017

Master's Thesis in Computational Physics

Stockholm University

Abstract

Fluid–structure interaction problems can be found in various fields in science and engineering. A current problem is found in computational biology, where primary cilium mechanosensation on a single cell model is studied. The cell mechanical problem is characterised by laminar flow where the Reynolds number is low.

The open-source software and FEniCS-HPC/Unicorn is used to solve fluid–structure interaction problems, and is mainly used for fluid flows at high Reynolds numbers. In this project, the Unicorn solver has been extended to apply to fluid flows at low Reynolds numbers, and in particular to a simple cell mechanical model problem where the deflection of a primary cilium is studied.

The results show that the developed solver apply to problems where the deformation of the structure is moderate. For large mesh deformations, further development of the program code is needed.

Sammanfattning

Fluid–struktur interaktionsproblem förekommer i en mängd olika områden inom naturvetenskap och ingenjörskonst. Ett aktuellt problem finns inom beräkningsbiologi där cellmekanik hos flimmerhår studeras. I det cellmekaniska problemet är flödet runt cellen laminärt vilket innebär att Reynoldstalet är lågt.

Den öppna programvaran FEniCS-HPC/Unicorn används för att lösa fluid–struktur interaktionsproblem och då främst för turbulenta flöden, dvs. för höga Reynoldstal. I detta projekt har FSI-lösaren Unicorn utökats till att vara tillämpbar på problem där Reynoldstalet är lågt. För att testa programkoden har en enkel modell av det cellmekaniska problemet använts och böjningen av flimmerhåret på cellen har simulerats.

Resultaten visar att beräkningsmodellen fungerar för måttliga deformationer av strukturen. För att även kraftiga deformationer ska kunna hanteras krävs vidare utveckling av programkoden.

Acknowledgements

First, I would like to thank my supervisor Prof. Johan Hoffman for giving me the opportunity to work with this interesting project. It has been a pleasure for me to study the theory and to work with the implementation of the solver. I also thank Dr. Johan Jansson for valuable discussions about the code behind the Unicorn solver.

Further, I acknowledge the original cell model supplied by Dr. Hanifeh Khayyeri at Lund University.

I am very grateful for the support I have got from other people at the department. In particular, I sincerely thank Niyazi Cem Degirmenci and Jeannette Hiromi Spühler for their time to answer my questions and for valuable advises during the implementation process. I also thank Văn Đang Nguyễn for all the discussions we have had, for helping me with software installations and for your friendship.

At last, I would like to thank my wife, my children and my family for their support and encouragement during my studies.

Contents

Chapter 1

Introduction

1.1 Introduction

Partial differential equations (PDEs) are used to model phenomena in a wide rage of applications within science and engineering. Typical applications of PDEs in engineering can be found in fluid and structure mechanics, such as flow in a pipe and deformation of a solid structure. In science, we can find applications of solving PDEs in areas such as geophysics, meteorology and biology.

The finite element method (FEM) provides a general methodology for computing solutions to PDEs to simulate the phenomena being modeled. Coupled multi-physics problems pose particular challenges, where multiple PDEs are coupled to model complex phenomena. One such coupled problem is fluid–structure interaction (FSI), where models for describing the mechanics of fluids and solids are coupled.

The coupling algorithm may be *explicit* (one-way) or *implicit* (two-way), or *monolithic* where both PDEs are solved together as one single system. Explicit methods are efficient for some problems, but may experience stability problems. To get a more robust method, implicit or monolithic coupling is used.

The character of a flow problem depends on the Reynolds number that gives the relation between viscous and inertial effects in the problem, with low Reynolds numbers corresponding to highly viscous flow and high Reynolds number to turbulent flow.

In this project we develop a monolithic FEM model for FSI at low Reynolds numbers, based on previous work for high Reynolds number flow [8]. We implement the method in the open source software FEniCS/Unicorn [10], which we use to investigate mechanosensation in cell mechanics [12].

1.2 Previous work

The starting point for this master thesis project is the work of Khayyeri et al. [12], where mechanosensation in a model of a biological cell is studied through numerical computations. The authors have investigated how primary cilia mechanics affected mechanosensation in the cell, by studying the effects of a deflected primary cilium at the surface of the cell.

Primary cilia are extensions of the cell which resembles of "antennas" attached to the cell, and as it moves by the fluid flow around the cell it deflects. A deflected primary cilium at the surface induces strains on the internal cell components, giving information to the cell nucleus about the surroundings. The primary cilium is thus a flow sensor for the cell.

The computational approach of Khayyeri et al. was a finite element model of a single cell with a primary cilium in a flow chamber. A one-way decoupled method, with two different solvers, was used to calculate the forces acting on the cell. One solver was used for the fluid region and a second solver was used for the structure region with the single cell model.

In this master project, we aim to solve a similar fluid–structure interaction problem of a cell and a primary cilium with a *monolithic* model, where the entire computational domain is solved simultaneously as one single system. The monolithic solver developed in this project is based on the Unicorn solver within the FEniCS project.

The Unicorn solver is the implementation of the *Unified Continuum model* (UC model), which is a robust method to solve fluid–structure interaction problems. The Unicorn solver has been developed to study FSI problems at high Reynolds numbers, which are problems characterised by turbulent flow. The solver has been developed by J. Hoffman et al. [8, 9, 6], and is the FSI solver within the FEniCS project [13].

1.3 The aim of the project

The aim of this master thesis project has been to develop a monolithic fluid–structure interaction solver applicable to low Reynolds number flow, based on the Unicorn/FEniCS solver for high Reynolds number flow. A simplified cell mechanical problem, similar to previous work of Khayyeri et al. [12], has been used as the model problem for the project.

The main purpose of this project is that the FSI solver developed for the model problem is monolithic, i.e. two-way coupled, while the solver used in the previous study is a one-way solver. A two-way coupled FSI solver for cell mechanical problems does not exist yet, so a progress in this field is of great interest to research.

1.4 Outline

The cell mechanical problem behind the model problem in this project is described in the first section of Chapter 2. It is followed by a short introduction to the Finite Element Method, where the basic concepts in the method are described by considering a simple problem in one dimension. The chapter also gives an introduction to the Navier-Stokes equations.

In Chapter 3 the Unified Continuum model is introduced and the general theory behind the UC solver is presented. The theory used for the specific fluid–structure interaction problem for low Reynolds numbers is also presented.

In Chapter 4 the method and the procedures in the project are described. An overview of the work behind the computational meshes are given here, and the new features of the solver are briefly described.

In Chapter 5 the results to verify the model and the developed FSI solver are given. The solver is applied to the cell mechanical model problem and simulation results are presented. Finally, the results and the limits of the cell model are discussed.

Chapter 2

Background

2.1 The cell mechanical problem

A current research field in computational biology is the study of primary cilia mechanics and its effects on the mechanosensation of the cell. The model problem for this master project is a simplified version of a cell mechanical problem proposed by Khayyeri et al. [12]. We will here give a short background about primary cilia and the cell mechanical problem behind the model problem that we investigate.

2.1.1 Primary cilia

Primary cilia are non-motile extensions of the cell that protrude into the extracellular environment. They usually occur one per cell, and in the human body, singular primary cilia can be found on cells from different tissues, such as kidneys, liver and bone.

Primary cilia are known to play an important role in the life cycle of the cell, and they are essential to normal development and functionality of many human organs. Malfunctioning primary cilia are associated with diseases like polycystic kidney disease and osteoarthritis.

A primary cilium can be described as a rigid spike which acts as an antenna, gathering sensory information to the cell from the surrounding environment. As the primary cilium deflects under fluid flow it induces a large number of biomechanical signals and molecular events inside the cell body.

Several research projects have contributed to a better understanding of the mechanisms of the primary cilia mechanosensation. For example, research studies have shown that the degree of deflection regulates the strength of molecular response, e.g. the degree of calcium release. Other studies have shown that the removal of primary cilia reduces the sensitivity of cells to biophysical signals.

In a computational study by Hanifeh Khayyeri, Sara Barreto and Damien Lacroix [12], the authors investigated, by use of a finite element model, whether the mechanics of the primary cilium influences cell mechanosensation. For example, they showed how the length and stiffness properties of the primary cilium affected the transmitted mechanical signals to other cell organelles, such as the cell nucleus, as the primary cilium was subjected to flow stimulation.

2.1.2 The model problem

The computational approach of Khayyeri et al. was a finite element model of a single cell model, together with a finite element representation of a flow chamber. Their cell model was consisting of a cell body equipped with a primary cilium, where the cell body had an inner structure consisting of nucleus, cytoplasm, cortex, microtubules and actin bundles. All the solid parts in this model, the primary cilium included, was treated as compressible neo-Hookean materials.

The fluid–structure interaction problem was solved using a one-way decoupled FSI computation. The simulation starts with a CFD analysis for the fluid domain, where the forces, by means of fluid pressure and shear forces acting on the cell and the primary cilium, are determined. In the next step, these forces are used as input data to a decoupled computation of the single structure model.

The one-way decoupled model described above is an example of a *partitioned* method for the fluid–structure problem, since the flow and the displacement of the structure is solved separately, by two different solvers. Another approach is to solve the entire computational domain simultaneously as one system, with a single solver. Such an approach is called *monolithic*.

In a partitioned one-way model, the interaction can only propagate in one direction, in this case from the fluid to the structure. On the other hand, with a monolithic two-way model the interaction can propagate in both directions. The fluid can affect the structure, but the movements of the structure can also have an influence on the fluid. The advantages of a monolithic model is that the flow mechanical problem can be more realistic and accurately modelled, as compared to a partitioned model.

The model problem considered in this master project is a fluid–structure interaction problem similar to the cell mechanical problem described above, but solved with a monolithic model. The geometry used is similar to the geometry used by Khayyeri et al., but the cell model has been simplified to only consist of a homogeneous cell body and a primary cilium. The cell body consists of two materials: the cytoplasm (inner core) and the cortex (outer shell). No other inner structure has been included in the model.

The material parameters used are equivalent to values given in the previous study. The viscosity of the fluid is the same as of water. As the length scale is in the order of $10^{-4}\,\mathrm{m}$ and the fluid flows with a velocity of $1\,\mathrm{mm/s}$, the Reynolds number is low and we thus have viscous flow. The cell constituents are characterised by the stiffness of the different materials. The solid parts in the model are treated as *incompressible* neo-Hookean materials.

As the model is greatly simplified compared to the detailed model in the previous study, it can not be used to answer any questions of interest related to primary cilia mechanosensation. The simple cell model in this project has solely been used as a model problem in the development of a monolithic solver in the regime of low Reynolds number flow.

A future extension of the model would be to also include the inner structure of the cell, such as nucleus, actin bundles and microtubules. With a more realistic and detailed model of the cell, the outcome from simulations with the monolithic solver can possibly contribute to a better understanding of primary cilia mechanosensation.

2.2 The Finite Element Method

The Finite Element Method (FEM) is a general method to find numerical approximations of solutions to partial differential equations. The method is based on Galerkin's method with piecewise polynomial approximations.

The computational domain, or the *mesh*, is an approximation of the geometrical domain for the PDE problem. The mesh is a collection of non-overlapping finite *elements*. In one dimension the elements are segments on a line, in two dimensions they may be triangles in a plane, and in three dimensions the elements may be tetrahedrons. The finite elements are connected to each other in the mesh with *nodes*, *edges* and *faces*.

In FEM, we seek an approximate solution to the PDE on the mesh in a finite-dimensional space spanned by a set of basis functions. The basis functions are usually piecewise polynomial functions of low order, and a common choice is piecewise *linear* functions.

As an illustration of the method, we can consider a one-dimensional domain $\Omega = [0, 1]$ along the x-axis, subdivided into N intervals $I_i = [x_{i-1}, x_i]$ with nodes x_i where $i = 0, 1, 2, \ldots, N$. Let us now define a finite set of piecewise linear basis functions on the domain Ω as the *hat functions* $\varphi_i(x)$, where $i = 0, 1, \ldots, N$. The hat functions or the *nodal basis functions* are defined by

$$\varphi_i(x) = \begin{cases} \dfrac{x - x_{i-1}}{x_i - x_{i-1}} & , \quad x_{i-1} < x \leq x_i \quad , \quad i = 1, \ldots, N \\ \dfrac{x_{i+1} - x}{x_{i+1} - x_i} & , \quad x_i < x \leq x_{i+1} \quad , \quad i = 0, \ldots, N-1 \\ 0 & , \quad \text{elsewhere} \end{cases} \tag{2.1}$$

and the functions are depicted in Figure 2.1. Note that the basis functions φ_0 and φ_N at the endpoints are "half" hat functions. From the definition (2.1) we see that a hat function $\varphi_i(x)$ at an interior node is only non-zero on the interval $[x_{i-1}, x_{i+1}]$. This property of the basis functions will simplify integral calculations on the domain, since only neighbouring basis functions have non-zero overlap. Due to this reason, we will later see that the linear system of equations to solve becomes a sparse matrix system.

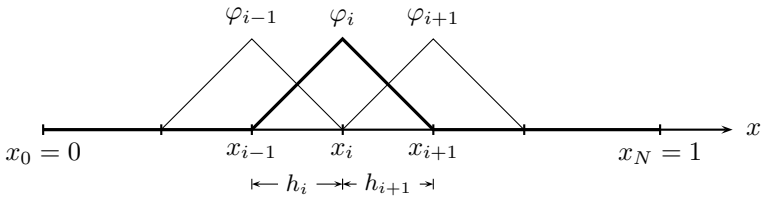


Figure 2.1: The nodal basis functions $\varphi_i(x)$ on the domain $\Omega = [0, 1]$ together with the neighbouring basis functions $\varphi_{i-1}(x)$ and $\varphi_{i+1}(x)$.

Let V_h be the finite element function space spanned by the set of hat functions $\{\varphi_i(x)\}_{i=0}^N$. Any function that belongs to this function space can be expressed as a linear combination of the basis functions. In the finite element method we seek an approximate solution $U(x) \in V_h$ to a given PDE, and the

solution can be expanded in the basis functions as

$$U(x) = \sum_{j=0}^{N} \xi_j \, \varphi_j(x) \tag{2.2}$$

where the unknown coefficients ξ_j are to be determined.

Solving a PDE with the finite element method turns out to be a problem of solving the linear system of equations $A\,\xi = b$, where A is a known matrix, ξ is a column vector of the unknown coefficients, and where b is a column vector of known data.

In the following section, we give an example of deriving the linear system of equations for a one-dimensional problem. The example is taken from the book *Computational Differential Equations* by Eriksson et al. [5].

2.2.1 FEM in 1D: heat conduction in wire

To show how the finite element method can be applied to solve a physical problem, we consider heat conduction in a thin heat-conducting wire occupying the interval $[0, 1]$. The wire is heated by a heat source $f(x)$ and we are interested in finding the stationary distribution of the temperature $u(x)$ in the wire. The situation can be described by the two-point boundary value problem

$$\begin{cases} -u''(x) = f(x) & \text{in } (0,\, 1) \\ u(0) = u(1) = 0 \end{cases} \tag{2.3}$$

where the homogeneous Dirichlet boundary conditions, $u(0) = u(1) = 0$, correspond to keeping the temperature zero at the endpoints of the wire.

We will now formulate the finite element method for (2.3) based on continuous piecewise linear approximations.

Let $\mathcal{T}_h : 0 = x_0 < x_1 < \ldots < x_{M+1} = 1$ be a triangulation of the interval $I = (0, 1)$ into sub-intervals $I_j = (x_{j-1}, x_j)$ of length $h_j = x_j - x_{j-1}$. Let V_h denote the set of continuous piecewise linear functions on $\mathcal{T}_h$ that are zero at $x = 0$ and $x = 1$. An example of such a function is shown in Figure 2.2.

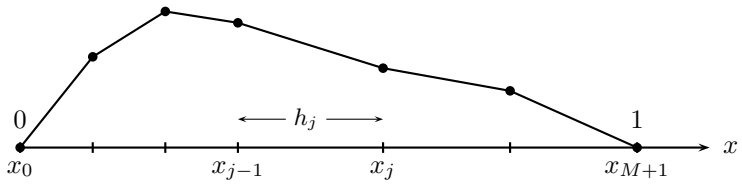


Figure 2.2: A continuous piecewise linear function in V_h.

Any function in the space V_h can be expressed with a basis consisting of the hat functions $\{\varphi_j\}_{j=1}^{M}$, i.e. a function $v \in V_h$ can be written

$$v(x) = \sum_{j=1}^{M} v(x_j) \, \varphi_j(x) \tag{2.4}$$

where $v(x_j)$ are the values at the interior nodes x_j.

Galerkin's method is based on stating the differential equation $-u'' = f$ in the form

$$\int_0^1 (-u'' - f)\, v\, dx = 0 \qquad \text{for all functions } v \tag{2.5}$$

corresponding to the residual $-u'' - f$ being orthogonal to the test functions v. Since the functions in V_h only have first order derivatives, we need to rewrite the second order derivative into an expression containing only first order derivatives. By partial integration we can write

$$\begin{aligned} -\int_0^1 u''v\, dx &= -u'(0)v(0) + u'(1)v(1) + \int_0^1 u'v'\, dx \\ &= \int_0^1 u'v'\, dx \end{aligned}$$

where the boundary terms vanishes since $v(0) = v(1) = 0$.

We are thus led to the following *variational formulation* of (2.3): Find the function u with $u(0) = u(1) = 0$ such that

$$\int_0^1 u'v'\, dx = \int_0^1 f\, v\, dx \tag{2.6}$$

for all functions v such that $v(0) = v(1) = 0$. Equation (2.6) is referred to as the *weak form* of (2.5).

The Galerkin finite element method for (2.3) is the following discrete analog of (2.6): Find $U \in V_h$ such that

$$\int_0^1 U'v'\, dx = \int_0^1 f\, v\, dx \tag{2.7}$$

for all $v \in V_h$.

We note that since the functions U and v are continuous piecewise linear functions, the derivatives U' and v' are *discontinuous* piecewise constant functions. A discontinuous piecewise constant function, as the function depicted in Figure 2.3, is not defined at the nodes x_j. However, the integral with integrand $U'v'$ is nevertheless uniquely defined as the sum of integrals over the subintervals.

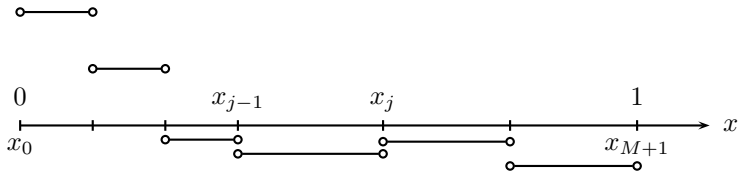


Figure 2.3: The derivative of the continuous piecewise linear function in Figure 2.2.

The finite element solution $U(x)$ can be expanded in the basis of hat functions as

$$U(x) = \sum_{j=1}^{M} \xi_j\, \varphi_j(x) \tag{2.8}$$

where the nodal values $\xi_j = U(x_j)$ are determined by the Galerkin orthogonality (2.7). Note that only the interior nodes are included in the expansion, since the solution at the boundary is given by the homogeneous boundary condition $U(0) = U(1) = 0$. Inserting the expansion above in (2.7) gives

$$\sum_{j=1}^{M} \xi_j \int_0^1 \varphi_j'(x)\, v'\, dx = \int_0^1 f\, v\, dx \quad , \qquad \forall\, v \in V_h \tag{2.9}$$

This expression is valid for *all* test functions v in V_h, so if we simply use the set of basis functions $\{\varphi_i\}_{i=1}^M$ as the test function v, we get a $M \times M$ linear system of equations

$$\sum_{j=1}^{M} \xi_j \int_0^1 \varphi_j'(x)\, \varphi_i'(x)\, dx = \int_0^1 f\, \varphi_i(x)\, dx \quad , \quad i = 1, \ldots, M \tag{2.10}$$

for the unknown coefficients $\{\xi_j\}$.

If we let $\xi = (\xi_j)$ denote the vector of unknown coefficients, we can write this equation as a matrix equation on the form $A\xi = b$ where $A = (a_{ij})$ is a $M \times M$ matrix called the *stiffness matrix*, with elements

$$a_{ij} = \int_0^1 \varphi_j'(x)\, \varphi_i'(x)\, dx \tag{2.11}$$

and where $b = (b_i)$ is called the *load vector* with

$$b_i = \int_0^1 f(x)\, \varphi_i(x)\, dx \tag{2.12}$$

Since the hat functions $\varphi_i(x)$ and the source function $f(x)$ are known, we can easily calculate the coefficients a_{ij} and b_i.

From the definition of the hat functions (2.1) and guided by Figure 2.1, we see that all the coefficients a_{ij} are zero unless $i = j-1$, $i = j$ or $i = j+1$. Thus, the stiffness matrix A becomes a sparse tridiagonal matrix. It is easy to show that the non-zero coefficients can be computed as

$$a_{i,i-1} = \int_{x_{i-1}}^{x_i} \frac{1}{h_i} \cdot \frac{-1}{h_i}\, dx = -\frac{1}{h_i} \tag{2.13}$$

$$a_{i,i} = \int_{x_{i-1}}^{x_i} \left(\frac{1}{h_i}\right)^2 dx + \int_{x_i}^{x_{i+1}} \left(\frac{-1}{h_{i+1}}\right)^2 dx = \frac{1}{h_i} + \frac{1}{h_{i+1}} \tag{2.14}$$

$$a_{i,i+1} = \int_{x_i}^{x_{i+1}} \frac{-1}{h_{i+1}} \cdot \frac{1}{h_{i+1}}\, dx = -\frac{1}{h_{i+1}} \tag{2.15}$$

and that the coefficients of b becomes

$$b_i = \int_{x_{i-1}}^{x_i} f(x)\, \frac{x - x_{i-1}}{h_i}\, dx + \int_{x_i}^{x_{i+1}} f(x)\, \frac{x_{i+1} - x}{h_{i+1}}\, dx \quad , \quad i = 1, \ldots, M \tag{2.16}$$

The linear system of equations is preferably solved on a computer using general algorithms for linear algebra. As soon as the unknown coefficients are determined, the solution $U(x)$ to the PDE is obtained by inserting the coefficients in the expansion (2.8).

2.2.2 FEM in two and three dimensions

The finite element metod with piecewise polynomials can be extended to higher dimensions. Usually, we are interested in applying the method in two or three dimensions. We first describe some basic concepts of the finite element method in two dimensions, and then extend the formulation to three dimensions.

Consider a two-dimensional domain Ω with a polygonal boundary Γ. The domain is divided into an non-overlapping set of triangles K, such that no vertex of a triangle lies on the edge of another triangle. The sub-division $\mathcal{T}_h = \{K\}$ is called a *triangulation* of Ω, which also defines the computational mesh. The mesh consists of *elements* (or cells), *vertices* (or nodes) and *edges*.

On the mesh we can have global functions that are defined piecewise on each element. These functions can be either continuous or discontinuous.

Let us measure the size of a triangle $K \in \mathcal{T}_h$ by the length h_K as the largest side of the triangle. We can then define a *mesh function* $h(x)$ on the triangulation $\mathcal{T}_h$, which is a piecewise constant function. The function is defined as

$$h(x) = h_K \quad \text{for} \quad x \in K \quad \text{for each} \quad K \in \mathcal{T}_h \tag{2.17}$$

and is and example of a discontinuous function on the mesh.

In order to use the finite element method, we need to define piecewise polynomial basis functions on the mesh. We define a finite element vector space V_h consisting of continuous piecewise linear functions on $\mathcal{T}_h$ defined by

$$V_h = \left\{ v \,:\, v \text{ is continuous on } \Omega, \quad v\Big|_K \in \mathcal{P}^1(K) \quad \text{for} \quad K \in \mathcal{T}_h \right\} \tag{2.18}$$

where $\mathcal{P}^1(K)$ denotes the set of linear functions in K. The functions in V_h can be described by their nodal values, since a linear function in two dimensions is uniquely determined by three points.

The global set of basis functions $\{\varphi_j\}_{j=1}^M$ for V_h are called *tent functions* and are defined by

$$\varphi_j(N_i) = \begin{cases} 1 & , \quad i = j \\ 0 & , \quad i \neq j \end{cases} \tag{2.19}$$

where $i, j = 1, \ldots, M$ and $N_1, N_2, \ldots, N_M$ is an enumeration of the nodes.

In Figure 2.4 a typical tent function is shown, and we can notice that φ_i is built up by triangular planes over the individual elements. The basis function φ_i is only non-zero in the set of elements that share the common node N_i. This is called the *support* of φ_i.

With the tent functions as basis functions we can express any function $v \in V_h$ as

$$v(x) = \sum_{i=1}^{M} v(N_i)\, \varphi_i(x) \tag{2.20}$$

where $v(N_i)$ are the nodal values.

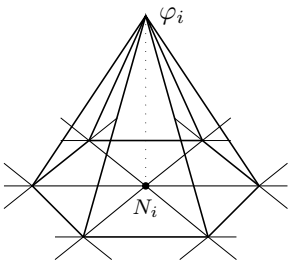


Figure 2.4: A typical global basis tent function on a two-dimensional mesh.

The formulation above is valid also for the three-dimensional case, with the triangle elements replaced by tetrahedron elements. A tetrahedron element is defined by four nodes in the mesh, and *faces* between elements are new entities in the mesh, not present in the two-dimensional case. By increasing the number of nodes in a three-dimensional mesh, the number of elements increases more rapidly compared to a two-dimensional mesh.

The nodal basis functions in three dimensions are not easy to visualise, but they are similar to the tent functions in two dimensions. The support for the nodal basis function φ_i are all the tetrahedron elements with common node N_i.

2.3 The Navier-Stokes equations

A fundamental basis in fluid dynamics are the Navier-Stokes equations which describe the motion of viscous fluids. The Navier-Stokes equations are useful since they can describe the physics of many phenomena of interest to science and engineering. They may be used to model ocean currents, blood flow in a vein, or the air flow around a wing of an airplane.

The incompressible Navier-Stokes equations are

$$\rho\left(\partial_t u + (u \cdot \nabla)u\right) + \nabla p - \mu \triangle u - f = 0 \tag{2.21}$$

$$\nabla \cdot u = 0 \tag{2.22}$$

where ρ is the density of the fluid, u is the velocity, p is the pressure, μ is the dynamic viscosity and f is an external volume force. ∂_t is the time derivative, $\nabla = \partial/\partial x$ is the nabla operator, where x denote a generalized coordinate in $\mathbb{R}^3$, and $\triangle = \nabla \cdot \nabla$ is the Laplace operator.

Before we get into the theory to solve the fluid–structure problem, we consider a pure fluid mechanical problem. We are interested in a case where the viscosity is high compared to the velocity of the fluid, which means that the Reynolds number is low. The *Reynolds number* is defined as

$$Re = \frac{\bar{u}\,\ell}{\nu} = \frac{\bar{u}\,\ell}{\mu/\rho}$$

where $\bar{u}$ is a characteristic velocity, ℓ is a characteristic length and ν is the kinematic viscosity.

2.3.1 Rescaling of Navier-Stokes equations

When applying a mathematical equation to a physical problem, it is convenient to use scaled units rather than using SI units for the quantities. With scaled units all quantities are dimensionless parameters, but they are related to the physical quantities through scale parameters.

To find the proper scaled units for the quantities involved in equation (2.21), let us introduce the dimensionless parameters, denoted by a tilde ($\sim$), as

$$\tilde{u} = \frac{u}{U}, \quad \tilde{x} = \frac{x}{L}, \quad \tilde{t} = \frac{t}{\tau}, \quad \tilde{p} = \frac{p}{P}, \quad \tilde{f} = \frac{f}{F} \tag{2.23}$$

where U, L, τ, P and F are parameters to define the scale of the velocity, length, time, pressure and volume force, respectively.

The time derivative ∂_t and the spatial derivatives ∇ and $\triangle$ can then be written in terms of dimensionless derivatives as

$$\partial_t = \frac{\partial}{\partial t} = \frac{1}{\tau}\frac{\partial}{\partial \tilde{t}} \equiv \frac{1}{\tau}\partial_{\tilde{t}} \tag{2.24}$$

$$\nabla = \frac{\partial}{\partial x} = \frac{1}{L}\frac{\partial}{\partial \tilde{x}} \equiv \frac{1}{L}\tilde{\nabla} \tag{2.25}$$

$$\triangle = \nabla \cdot \nabla = \frac{1}{L^2}\tilde{\nabla}^2 \equiv \frac{1}{L^2}\tilde{\triangle} \tag{2.26}$$

which gives the equation

$$\rho\left(\frac{U}{\tau}\partial_{\tilde{t}}\,\tilde{u} + \frac{U^2}{L}(\tilde{u}\cdot\tilde{\nabla})\tilde{u}\right) + \frac{P}{L}\tilde{\nabla}\tilde{p} - \mu\frac{U}{L^2}\tilde{\triangle}\tilde{u} - F\tilde{f} = 0 \tag{2.27}$$

and can be rewritten as

$$\left(\partial_{\tilde{t}}\,\tilde{u} + \frac{U\tau}{L}(\tilde{u}\cdot\tilde{\nabla})\tilde{u}\right) + \frac{P\tau}{\rho U L}\tilde{\nabla}\tilde{p} - \frac{\mu\,\tau}{\rho L^2}\tilde{\triangle}\tilde{u} - \frac{F\tau}{\rho U}\tilde{f} = 0 \tag{2.28}$$

The time scale is given by the scales of the length and velocity as

$$\tau = \frac{L}{U}$$

and by eliminating the time scale τ in the equation above, the resulting equation is

$$\left(\partial_{\tilde{t}}\,\tilde{u} + (\tilde{u}\cdot\tilde{\nabla})\tilde{u}\right) + \frac{P}{\rho U^2}\tilde{\nabla}\tilde{p} - \frac{\mu}{\rho U L}\tilde{\triangle}\tilde{u} - \frac{FL}{\rho U^2}\tilde{f} = 0 \tag{2.29}$$

We can now choose the scales such that Reynolds number is defined by the scales, i.e.

$$Re = \frac{\rho U L}{\mu} \tag{2.30}$$

By choosing the scales U and L arbitrary, the unit scales for the pressure and the volume force are then given as

$$P = \rho U^2 \qquad \text{and} \qquad F = \frac{\rho U^2}{L} \tag{2.31}$$

respectively.

The dimensionless momentum equation, with the tilde notation omitted, finally becomes

$$\Big(\partial_t\, u + (u\cdot\nabla)u\Big) + \nabla p - Re^{-1}\,\triangle u - f = 0 \tag{2.32}$$

where Reynolds number is given by equation (2.30) and the scales for pressure and the volume force are given by (2.31).

2.3.2 Low Reynolds number flow

The rescaled momentum equation (2.32) is applicable for high Reynolds number flow. In order to fulfil the continuity equation in its weak formulation, the coefficients of the pressure term ∇p and the Laplace term $\triangle u$ must be balanced. For low Reynolds numbers, the Laplace term in equation (2.32) blows up and the balance between the terms is broken. Thus, we also need to derive a dimensionless formula for *low* Reynolds number flow.

By multiplying equation (2.29) with $Re = \frac{\rho UL}{\mu}$ we obtain

$$Re\Big(\partial_{\tilde{t}}\,\tilde{u} + (\tilde{u}\cdot\tilde{\nabla})\tilde{u}\Big) + \frac{PL}{\mu\,U}\tilde{\nabla}\tilde{p} - \tilde{\triangle}\tilde{u} - \frac{FL^2}{\mu\,U}\tilde{f} = 0 \tag{2.33}$$

and we can now define the unit scale for pressure and volume force respectively as

$$P = \frac{\mu\,U}{L} \qquad \text{and} \qquad F = \frac{\mu\,U}{L^2} \tag{2.34}$$

The dimensionless momentum equation for low Reynolds number becomes

$$Re\Big(\partial_t\, u + (u\cdot\nabla)u\Big) + \nabla p - \triangle u - f = 0 \tag{2.35}$$

where Reynolds number is given by equation (2.30) and the scales for pressure and the volume force are given by (2.34).

As a conclusion, we can introduce a general dimensionless equation as

$$\alpha\Big(\partial_t\, u + (u\cdot\nabla)u\Big) + \nabla p - \beta\triangle u - f = 0 \tag{2.36}$$

where the parameters α and β are set according to the Reynolds number for the current problem, that is

- For high Re: set $\alpha = 1$ and $\beta = Re^{-1} = \frac{\mu}{\rho UL}$
  pressure scale: $P = \rho\,U^2$, force scale: $F = \frac{\rho U^2}{L}$
- For low Re: set $\alpha = Re = \frac{\rho UL}{\mu}$ and $\beta = 1$
  pressure scale: $P = \frac{\mu U}{L}$, force scale: $F = \frac{\mu U}{L^2}$

2.3.3 Reynolds number limits

We can notice that in the high Reynolds number limit, $Re \to \infty$, equation (2.32) together with (2.22) reduces to the *Euler equations*

$$\Big(\partial_t u + (u \cdot \nabla)u\Big) + \nabla p = f \tag{2.37}$$

$$\nabla \cdot u = 0 \tag{2.38}$$

whereas in the low Reynolds number limit, $Re \to 0$, equation (2.35) together with (2.22) reduces to the *Stokes equations*

$$-\triangle u + \nabla p = f \tag{2.39}$$

$$\nabla \cdot u = 0 \tag{2.40}$$

which is independent of time. We see that the non-linearity of the Navier-Stokes equations disappear as $Re \to 0$.

However, none of these limits are of interest for the problem we are about to solve. Both of the limits represent idealised cases. In the cell mechanical problem the Reynolds number is low, but it would be a far too great simplification to just set it to zero.

Chapter 3

Theory

3.1 The Unified Continuum model

The Unified Continuum model (UC model) is a method used to solve fluid–structure interaction problems. In the UC model the two phases, the fluid and the solid (or the structure), are treated as one single continuum. The two phases are thus described by the same equations, but the material properties for the fluid and the solid are different.

The UC model can be derived from fundamental equations for conservation of mass and momentum, together with an equation describing the phase convection. Let $m = \rho\, u$ be the momentum, where ρ is the density and u is the velocity, and let θ be the phase function ($\theta = 1$ for fluid and $\theta = 0$ for solid). In component form, the equations are

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x_j}(u_j \rho) = 0 \qquad \textit{(conservation of mass)} \tag{3.1}$$

$$\frac{\partial m_i}{\partial t} + \frac{\partial}{\partial x_j}(u_j m_i) = \frac{\partial}{\partial x_j}\sigma_{ij} + f_i \qquad \textit{(conservation of momentum)} \tag{3.2}$$

$$\frac{\partial \theta}{\partial t} + \frac{\partial}{\partial x_j}(u_j \theta) = 0 \qquad \textit{(phase convection equation)} \tag{3.3}$$

where σ_{ij} is a stress tensor and f_i is an external force with $i = 1, 2, 3$, and where a repeated index is shorthand notation for summation over components, e.g.

$$\frac{\partial}{\partial x_j}(u_j \rho) \equiv \sum_{j=1}^{3} \frac{\partial}{\partial x_j}(u_j \rho) = \nabla \cdot (u\rho)$$

If we assume that the continuum is incompressible, i.e. a divergence free continuum, we have that $\frac{\partial u_j}{\partial x_j} = 0$, and equation (3.1) reduces to

$$\frac{\partial \rho}{\partial t} + u_j \frac{\partial \rho}{\partial x_j} = 0 \tag{3.4}$$

Combining the above equation with equation (3.2), the incompressible UC equa-

tions can be expressed as

$$\rho\left(\frac{\partial u_i}{\partial t}+u_j\frac{\partial u_i}{\partial x_j}\right)=\frac{\partial}{\partial x_j}\sigma_{ij}+f_i \quad , \quad i=1,2,3 \tag{3.5}$$

$$\frac{\partial u_j}{\partial x_j}=0 \tag{3.6}$$

$$\frac{\partial \theta}{\partial t}+u_j\frac{\partial \theta}{\partial x_j}=0 \tag{3.7}$$

To model a fluid–structure interaction problem, the total stress σ_{ij} is decomposed into a Cauchy stress $\bar{\sigma}_{ij}$ and a mechanical pressure p as

$$\sigma_{ij}=\bar{\sigma}_{ij}-p\,\delta_{ij} \tag{3.8}$$

where δ_{ij} is the Kronecker delta function. Further, the Cauchy stress $\bar{\sigma}_{ij}$ can be written as a linear combination of fluid and solid stress with use of the phase function θ,

$$\bar{\sigma}_{ij}=\theta\bar{\sigma}^f_{ij}+(1-\theta)\bar{\sigma}^s_{ij}, \tag{3.9}$$

where the superscripts f and s denote fluid and solid respectively. Thus, within the Unified Continuum model, the phase function θ is used to determine which constitutive equation and material parameter to use.

The stress tensors $\bar{\sigma}^f$ and $\bar{\sigma}^s$ are given by separate constitutive laws for the two phases. The fluid is here modeled as a Newtonian fluid and the solid as an incompressible Neo-Hookean solid. The constitutive laws are

$$\bar{\sigma}^f_{ij}=2\mu_f\epsilon(u)_{ij} \qquad (\textit{Newtonian fluid}) \tag{3.10}$$

$$\frac{\partial\bar{\sigma}^s_{ij}}{\partial t}=2\mu_s\epsilon(u)_{ij}+\frac{\partial u_i}{\partial x_k}\bar{\sigma}^s_{kj}+\bar{\sigma}^s_{ik}\frac{\partial u_k}{\partial x_j} \qquad (\textit{incomp. Neo-Hookean solid}) \tag{3.11}$$

where μ_f is the dynamic viscosity, μ_s is the shear modulus, and

$$\epsilon(u)_{ij}=\frac{1}{2}\left(\nabla u+\nabla u^\top\right)_{ij}=\frac{1}{2}\left(\frac{\partial u_i}{\partial x_j}+\frac{\partial u_j}{\partial x_i}\right) \tag{3.12}$$

is the strain rate tensor.

3.2 Finite element approximation

3.2.1 The discrete system

We are interested in finding approximate solutions to the equations (3.5), (3.6) and (3.7) for the velocity, the pressure and the phase. To this end, we here formulate the discrete system to solve with a finite element method.

Let $w=(u,p,\theta)$ denote the exact solution and let $W=(U,P,\Theta)$ denote the discrete solution. Let $v=(v^u,v^p,v^\theta)$ be a test function and let $R(W)=(R_u(W),R_p(W),R_\theta(W))$ be the residual defined by

$$R_u(W)=\rho\left(D_tU_i+U_jD_{x_j}U_i\right)-D_{x_j}\left(\Sigma_{ij}-P\delta_{ij}\right)-f_i \tag{3.13}$$

$$R_p(W)=D_{x_j}U_j \tag{3.14}$$

$$R_\theta(W)=D_t\Theta+u_jD_{x_j}\Theta \tag{3.15}$$

where $R(w) = 0$ and Σ is a piecewise constant stress corresponding to the Cauchy stress given in equation (3.9). In the above equations a more compact notation for the derivatives have been used, where $D_t \equiv \frac{\partial}{\partial t}$ and $D_{x_j} \equiv \frac{\partial}{\partial x_j}$.

The solution is computed by enforcing the Galerkin orthogonality

$$(R(W), v) = 0 \quad , \quad \forall\, v \in V_h \tag{3.16}$$

where the test space V_h consists of piecewise polynomial continuous functions in space and piecewise constant discontinuous functions in time. In the above equation, $(\cdot, \cdot)$ denotes the L_2 inner product in space and time. Recall that the L_2 inner product of some vector valued functions v and w over a space-time domain $\Omega \times I$, with $\Omega \in \mathbb{R}^3$ a spatial domain and $I = [0, T]$ a time domain, is defined as

$$(v, w) = \int_0^T \int_\Omega v \cdot w \;\; dx\, dt = \int_0^T \int_\Omega \sum_{i=1}^{3} v_i\, w_i \;\; dx\, dt$$

and the corresponding inner product for some scalar valued functions p and q reads

$$(p, q) = \int_0^T \int_\Omega p\, q \;\; dx\, dt$$

The weak formulation can then be written

$$\begin{aligned}(R_u(W), v^u) &= \left(\rho \left(D_t U_i + U_j D_{x_j} U_i\right) - f_i, v_i^u\right) \\ &\quad + \left(\Sigma_{ij}, D_{x_j} v_i^u\right) - \left(P, D_{x_i} v_i^u\right) = 0 \qquad (3.17)\\ (R_p(W), v^p) &= \left(D_{x_j} U_j, v^p\right) = 0 \qquad (3.18)\\ \left(R_\theta(W), v^\theta\right) &= \left(D_t \Theta + u_j D_{x_j} \Theta, v^\theta\right) = 0 \qquad (3.19)\end{aligned}$$

where the boundary integrals that arises from partial integration in (3.17) are omitted since we assume homogeneous Dirichlet boundary conditions.

By introducing a local ALE coordinate map on each space-time slab, we will later see that we can eliminate the equation for the phase function Θ, equation (3.19), from the discrete system.

3.2.2 Local ALE and mesh velocity

The phase interface is the boundary between the fluid and the solid, and we want this interface to be distinct throughout the computation. If the phase function Θ has different values on the same cell, it will lead to a diffusion of the phase interface. We would then end up with some elements having both fluid and solid properties, which is something we want to avoid.

By introducing a local ALE (Arbitrary Lagrangian–Eulerian) coordinate map[1] and a moving mesh, oriented along the characteristics of the convection of the phase interface, we can define the face interface at cell facets, which allows the interface to stay discontinuous throughout the computation.

In the Unified Continuum model, a local ALE coordinate map is used on each space-time slab, and an arbitrary *mesh velocity* β_h is introduced. The

[1] The ALE method is briefly described in Appendix A.1. For details, see [3].

resulting discrete phase convection equation is

$$D_t\Theta + (U - \beta_h) \cdot \nabla\Theta = 0 \tag{3.20}$$

where β_h is the mesh velocity. By choosing $\beta_h = U$ in the solid part of the mesh gives a trivial solution of the phase convection equation. Thus, with this choice we can remove the phase convection equation from the system.

What this means is that all cells that initially are defined as solid cells will remain solid cells during the entire computation. The solid cells will move according to the velocity of the continuum, and the remaining cells will adjust to this movement according to a mesh smoothing algorithm.

3.2.3 Mesh smoothing

In the Unicorn solver, the mesh moves as the solid part of the mesh is deformed, and the mesh moves with velocity β_h. In the solid domain, the mesh is deformed according to the velocity of the continuum, i.e. $\beta_h = U$ in the solid. In the fluid part, the mesh velocity is determined by a mesh smoothing algorithm. The mesh smoothing algorithm used in the Unicorn solver is based on an elastic analogy, see [8].

3.2.4 Stabilised Galerkin cG(1) cG(1) FEM

The UC model is mainly developed to treat problems involving turbulent flows, which is the case when the Reynolds number is high. For such problems, which are usually convection-dominated problems, the standard Galerkin FEM with piecewise linear functions in space and time, is unstable.

If we want to find the velocity and pressure solutions in the same function space of piecewise linear functions in space and time, we need to use a stabilised method, due to the so called *inf-sup condition* for mixed finite element methods.[2] One such method is the weighted least-squares stabilisation method, which will be considered here.

Another possibility is to search for the velocity solution in a function space of second-order polynomials, for which stabilisation of the method is not needed. However, by using a function space of higher order will also increase the number of unknowns in the equations.[3] Let us therefore remain in the space of first order polynomials and consider the stabilised Galerkin FEM.

Let $Q = \Omega \times I$ define a space-time domain, where Ω is an open domain in $\mathbb{R}^3$ with boundary Γ and $I = [0, T]$ is a given time interval.

The time interval I is divided into subintervals $I_n = (t_{n-1}, t_n)$, with associated space-time slabs $Q_n = \Omega \times I_n$. For each space-time slab we define space-time finite element spaces over a spatial mesh $\mathcal{T}_n$.

In a cG(1) cG(1) method we seek an approximate solution $\hat{U} = (U, P)$ which is continuous piecewise linear in space and time. The function spaces are based

[2] The inf-sup condition is stated in Appendix A.2, see also [1].

[3] For example, a second-order element, e.g. a Taylor-Hood element, uses additional quadrature points at the mid-points of the edges between the vertices. For a tetrahedron element with four vertices, the number of nodes will increase from four to ten when the linear basis functions are replaced by second-order polynomials.

on a spatial finite element space W^n of continuous piecewise linear functions, i.e.

$$W^n = \left\{ w \in \mathcal{C}(\Omega) : \quad w\Big|_K \in \mathcal{P}_1 \quad \forall\, K \in \mathcal{T}_n \right\} \tag{3.21}$$

where K is an element in $\mathcal{T}_n$. Let the function space W_0^n define all functions in W^n which are zero on the boundary Γ,

$$W_0^n = \left\{ w \in W^n : \quad w\Big|_\Gamma = 0 \right\} \tag{3.22}$$

Since we have homogeneous Dirichlet boundary conditions for the velocity, we introduce a function space for the velocity as $V_0^n \equiv [W_0^n]^3$.

Let $\hat{v} = (v, q) \in V_0^n \times W^n$ denote the test functions. For convenience, let us refer to the mixed function space that $\hat{U}$ and $\hat{v}$ belong to as $\hat{V}^n \equiv V_0^n \times W^n$.

With a moving mesh and a local ALE, the residuals corresponding to equations (3.13) and (3.14) are

$$R_u(\hat{U}) = \rho \left(D_t U_i + (U_j - \beta_j^h) D_{x_j} U_i \right) - D_{x_j} \left(\Sigma_{ij} - P\delta_{ij} \right) - f_i \tag{3.23}$$

$$R_p(\hat{U}) = D_{x_j} U_j \tag{3.24}$$

where β^h is the mesh velocity.

In the standard Galerkin FEM we enforce the Galerkin orthogonality on the residuals, i.e. $\big(R(\hat{U}), \hat{v}\big) = 0$. In a weighted least-squares method, we instead enforce

$$\Big(R(\hat{U}), \hat{v} + \delta R(\hat{v}) \Big) = 0 \qquad \forall\, \hat{v} \in V_0^n \times W^n \tag{3.25}$$

where $\delta > 0$ is a stabilisation parameter. The stabilisation term $\big(R(\hat{U}), \delta R(\hat{v})\big)$ can be chosen in different ways, but for simplicity, here we only use the terms contributing to stabilisation. For that reason, time derivatives and higher order terms are not present in the stabilisation term.

With the cG(1) cG(1) FEM, the stabilised UC model can finally be written

$$\begin{aligned} \Big(R_u(\hat{U}), v_i \Big) + \Big(R(\hat{U}), \delta R(v_i, 0) \Big) = {} & \big(\rho \left(D_t U_i + (U_j - \beta_j^h) D_{x_j} U_i \right) - f_i, v_i \big) \\ & + \big(\Sigma_{ij} - P\delta_{ij}, D_{x_j} v_i \big) + S D_u(\hat{U}, v_i) = 0 \end{aligned} \tag{3.26}$$

$$\Big(R_p(\hat{U}), q \Big) + \Big(R(\hat{U}), \delta R(0, q) \Big) = \big(D_{x_j} U_j, q \big) + S D_p(\hat{U}, q) = 0 \tag{3.27}$$

for all $\hat{v} \in V_0^n \times W^n$, where the stabilisation terms are

$$\begin{aligned} S D_u(\hat{U}, v_i) = {} & \Big(R(U_i, P), \delta R(v_i, 0) \Big) = \\ & \big(\delta_1 \, \rho (U_j - \beta_j^h) D_{x_j} U_i + D_{x_j} P \delta_{ij} - f_i \, , \, \rho (U_j - \beta_j^h) D_{x_j} v_i \big) \\ & + \big(\delta_2 \, D_{x_j} U_j \, , \, D_{x_i} v_i \big) \end{aligned} \tag{3.28}$$

$$\begin{aligned} S D_p(\hat{U}, q) = {} & \Big(R(U_i, P), \delta R(0, q) \Big) = \\ & \big(\delta_1 \, \rho (U_j - \beta_j^h) D_{x_j} U_i + D_{x_i} P - f_i \, , \, D_{x_i} q \big) \end{aligned} \tag{3.29}$$

with $\delta_1, \delta_2 > 0$ stabilisation parameters.

By adding the above equations together and discretize the resulting equation in time, we obtain a method for incompressible flow and fluid–structure interaction. The method reads:

Find $(U^n, P^n) \equiv (U(t_n), P(t_n))$ with $U^n \in V_0^n \equiv [W_0^n]^3$ and $P^n \in W^n$, such that

$$\begin{aligned}\big(\rho((U^n - U^{n-1})k_n^{-1} + ((\bar{U} - \beta_h)^n \cdot \nabla)\bar{U}^n), v\big) \\ + \theta\left(2\mu_f\epsilon(\bar{U}^n), \epsilon(v)\right) + (1-\theta)(S_s, \nabla v) \\ - (P^n, \nabla \cdot v) + (\nabla \cdot \bar{U}^n, q) \\ + SD_\delta^n(\bar{U}^n, P^n; v, q) = (f, v) \\ \forall\, \hat{v} = (v, q) \in V_0^n \times W^n \end{aligned} \tag{3.30}$$

where $\bar{U}^n = \frac{1}{2}(U^n + U^{n-1})$ is piecewise constant in time over I_n, and where S_s is a piecewise constant solid stress. The stabilisation term is

$$\begin{aligned}SD_\delta^n(\bar{U}^n, P^n; v, q) \equiv \\ \big(\delta_1\, \rho(\bar{U} - \beta_h)^n \cdot \nabla\bar{U}^n + \nabla P^n - f\,,\, \rho(\bar{U} - \beta_h)^n \cdot \nabla v + \nabla q\big) \\ + \big(\delta_2\, \nabla \cdot \bar{U}^n\,,\, \nabla \cdot v\big)\end{aligned} \tag{3.31}$$

with inner product

$$(v, w) = \sum_{K \in \mathcal{T}_n} \int_K v \cdot w\, dx \tag{3.32}$$

and

$$(\epsilon(v), \epsilon(w)) = \sum_{i,j=1}^{3} (\epsilon_{ij}(v), \epsilon_{ij}(w)) \tag{3.33}$$

3.2.5 Rescaling of the method

In Section 2.3.1 and 2.3.2 we derived dimensionless expressions for Navier-Stokes equations valid for high and low Reynolds numbers respectively. Now, we want to obtain similar expressions for the Unified Continuum model to solve the FSI problem at low Reynolds number.

To derive proper dimensions of the parameters we can consider the equation

$$\begin{aligned}(\rho(\dot{u} + (u \cdot \nabla)u), v) + \theta\,(2\mu_f\epsilon(u), \epsilon(v)) + (1-\theta)(\sigma_s, \nabla v) \\ - (p, \nabla \cdot v) = (f, v)\end{aligned} \tag{3.34}$$

where $\epsilon(u) = \frac{1}{2}(\nabla u + \nabla u^\top)$ is a symmetric tensor. By introducing the dimensionless parameters $\tilde{u} = u/U$, $\tilde{v} = v/U$, $\tilde{x} = x/L$, $\tilde{t} = t/\tau$, $\tilde{\nabla} = L\nabla$, $\tilde{\sigma}_s = \sigma_s/S$, $\tilde{p} = p/P$ and $\tilde{f} = f/F$, we get

$$\begin{aligned}\left(\rho\left(\tfrac{U}{\tau}\dot{\tilde{u}} + \tfrac{U^2}{L}(\tilde{u} \cdot \tilde{\nabla})\tilde{u}\right), U\tilde{v}\right) + \theta\tfrac{U^2}{L^2}\,(2\mu_f\tilde{\epsilon}(\tilde{u}), \tilde{\epsilon}(\tilde{v})) + (1-\theta)\tfrac{SU}{L}(\tilde{\sigma}_s, \tilde{\nabla}\tilde{v}) \\ - \tfrac{PU}{L}(\tilde{p}, \tilde{\nabla} \cdot \tilde{v}) = FU(\tilde{f}, \tilde{v})\end{aligned}$$

With the time scale given as $\tau = L/U$, the above expression can be rewritten as

$$\begin{aligned}\left(\left(\dot{\tilde{u}} + (\tilde{u} \cdot \tilde{\nabla})\tilde{u}\right), \tilde{v}\right) + \theta\left(2\tfrac{\mu_f}{\rho UL}\tilde{\epsilon}(\tilde{u}), \tilde{\epsilon}(\tilde{v})\right) + (1-\theta)\tfrac{S}{\rho U^2}(\tilde{\sigma}_s, \tilde{\nabla}\tilde{v}) \\ - \tfrac{P}{\rho U^2}(\tilde{p}, \tilde{\nabla} \cdot \tilde{v}) = \tfrac{FL}{\rho U^2}(\tilde{f}, \tilde{v})\end{aligned} \tag{3.35}$$

where $\frac{\mu_f}{\rho\, U L}$ can be identified as the inverse Reynolds number, Re^{-1}. Note that ρ is the density of the continuum which is equal to ρ_f in the fluid and equal to ρ_s in the solid. The unit scales for solid stress, pressure and volume force are then

$$S = \rho_s U^2 \qquad , \qquad P = \rho\, U^2 \qquad , \qquad F = \frac{\rho\, U^2}{L} \tag{3.36}$$

Equation (3.35) is valid for problems in the regime of high Reynolds numbers, i.e. for convection-dominated problems. To obtain an expression for low Reynolds numbers, we can multiply equation (3.35) with Re which gives

$$\begin{aligned}\left(Re\left(\dot{\tilde{u}} + (\tilde{u}\cdot\tilde{\nabla})\tilde{u}\right), \tilde{v}\right) + \theta\left(2\tilde{\epsilon}(\tilde{u}), \tilde{\epsilon}(\tilde{v})\right) + (1-\theta)\tfrac{\rho_f L\, S}{\mu_f \rho\, U}(\tilde{\sigma}_s, \tilde{\nabla}\tilde{v}) \\ - \tfrac{\rho_f L\, P}{\mu_f \rho\, U}(\tilde{p}, \tilde{\nabla}\cdot\tilde{v}) = \tfrac{\rho_f L^2 F}{\mu_f \rho\, U}(\tilde{f}, \tilde{v})\end{aligned} \tag{3.37}$$

where the unit scales for solid stress, pressure and volume force now are

$$S = \frac{\rho_s \mu_f U}{\rho_f L} \qquad , \qquad P = \frac{\rho\, \mu_f U}{\rho_f L} \qquad , \qquad F = \frac{\rho\, \mu_f U}{\rho_f L^2} \tag{3.38}$$

Thus, to get dimensionless expressions of the equations (3.30) and (3.31) valid for low Reynolds number, we can simply set μ_f to 1 in equation (3.34) and replace ρ with the Reynolds number Re calculated from

$$Re = \frac{\rho_f\, U\, L}{\mu_f} \tag{3.39}$$

where U and L are the scales of velocity and length respectively. The unit scales of pressure and volume force are then given by expression (3.38).

To obtain a dimensionless expression for high Reynolds number, we instead replace μ_f with Re^{-1} and set ρ to 1 in the equations (3.30) and (3.31). The unit scales of pressure and volume force are given in (3.36).

As a conclusion, we can consider a general expression of equation (3.30) similar to equation (2.36), where ρ is replaced by α and μ_f is replaced by β. The parameters α and β are set according to the Reynolds number of the current problem as

- For high Re: set $\rho = \alpha = 1$ and $\mu_f = \beta = Re^{-1} = \frac{\mu_f}{\rho_f\, U\, L}$
  pressure scale: $P = \rho\, U^2$, force scale: $F = \frac{\rho\, U^2}{L}$
- For low Re: set $\rho = \alpha = Re = \frac{\rho_f\, U\, L}{\mu_f}$ and $\mu_f = \beta = 1$
  pressure scale: $P = \frac{\rho\, \mu_f U}{\rho_f L}$, force scale: $F = \frac{\rho\, \mu_f U}{\rho_f L^2}$

3.3 Solving the non-linear algebraic system

The time discretization of Navier-Stokes equations, as the stabilised method in equation (3.30), give rise to a non-linear system of equations. With Newton's iterative method, the non-linear algebraic system is solved in each time step.

3.3.1 Newton's method

To solve the non-linear system of equations

$$F(U) = 0$$

we can use Newton's iterative method. In Newton's iterative method we calculate the Jacobian matrix

$$J = \frac{\partial F}{\partial U}$$

and solve the linear system of equations

$$JU_{i+1} = JU_i - F(U_i)$$

where i is the iteration index, U_{i+1} is a column vector containing the unknowns and U_i is a vector containing known solution coefficients. The iterative method starts with an initial guess of the solution, and in the next iteration step, the previously calculated solution is used as a new guess of the solution. The iteration continues until a convergence criterion is fulfilled. The Jacobian matrix J is a linear transformation matrix and is an approximation of the non-linear system of equations that we are about to solve. Note that the matrix J can either be calculated exactly or be chosen as an approximation of the exact Jacobian.

3.3.2 Segregation method

Let us now return to equation (3.30) and see how we can solve it with Newton's method. By simply applying Newton's method on the whole system we get

$$J = D_{\hat{U}_i^n} F(\hat{U}_i^n; \hat{v})$$
$$J(\hat{U}_{i+1}^n; \hat{v}) = J(\hat{U}_i^n; \hat{v}) - F(\hat{U}_i^n; \hat{v})$$

where $\hat{U}_i^n = (U_i^n, P_i^n)$ and $\hat{v} = (v, q)$. However, it turns out to be hard to find a preconditioner for the resulting linear system of equations in order to solve it with a Krylov method. Instead, the iterative method is segregated into velocity and pressure components,

$$J_U = D_{U^n} F((U_i^n, P_i^n), (v, 0))$$
$$J_U(U_{i+1}^n; v) = J_U(U_i^n; v) - F((U_i^n, P_i^n), (v, 0))$$

$$J_P = D_{P^n} F((U_i^n, P_i^n), (0, q))$$
$$J_P(P_{i+1}^n; q) = J_P(P_i^n; q) - F((U_i^n, P_i^n), (0, q))$$

In the next two sections, we will see how these methods are applied to solve the momentum equation and the continuity equation respectively.

3.3.3 Solving the momentum equation

The weak residual for the velocity component can be written

$$r\left((U^n, P^n), (v, 0)\right) = \left(\rho(U^n - U^{n-1})k_n^{-1}, v\right) + f_1(U^n; v) \tag{3.40}$$

where the time-independent part of the residual is

$$\begin{aligned} f_1(U^n; v) = &\left(\rho(U^n_{\text{ALE}} \cdot \nabla)\bar{U}^n, v\right) - (f, v) \\ &+ \theta(2\mu_f \epsilon(\bar{U}^n), \nabla v) + (1-\theta)(S_s, \nabla v) - (P^n, \nabla \cdot v) \\ &+ \left(\delta_1(\rho(U^n_{\text{ALE}} \cdot \nabla)\bar{U}^n + \nabla P^n), \rho(U^n_{\text{ALE}} \cdot \nabla)v\right) \\ &+ \left(\delta_2 \nabla \cdot \bar{U}^n, \nabla \cdot v\right) \end{aligned} \tag{3.41}$$

with $\bar{U}^n = \frac{1}{2}(U^n + U^{n-1})$ and $U^n_{\text{ALE}} = \bar{U}^n - \beta^n_h$.

Let us now multiply equation (3.40) with the time step k_n and define

$$F(U^n; v) \equiv (\rho U^n, v) - (\rho U^{n-1}, v) + k_n f_1(U^n; v) \tag{3.42}$$

The segregation method for the velocity component reads

$$J_U = D_{U^n} F(U^n; v) \tag{3.43}$$

$$J_U(U^n_{i+1}; v) = J_U(U^n_i; v) - F(U^n_i; v) \tag{3.44}$$

where the notation $J_U(U^n_i; v)$ is the result of acting the operator J_U on U^n_i, that is

$$\begin{aligned} J_U(U^n_i; v) &\equiv \left[D_{U^n} F(U^n; v)\right] U^n_i \\ &= (\rho U^n_i, v) + k_n \left[D_{U^n} f_1(U^n; v)\right] U^n_i \end{aligned} \tag{3.45}$$

We thus need to calculate the derivative $D_{U^n} f_1(U^n; v)$ and act it on U^n_i and U^n_{i+1} respectively. The involved derivatives are

$$\begin{aligned} \left[D_{U^n} \bar{U}^n\right] U^n &= \tfrac{1}{2} U^n \\ \left[D_{U^n} \epsilon(\bar{U}^n)\right] U^n &= \epsilon(\tfrac{1}{2} U^n) \\ \left[D_{U^n} S_s\right] U^n &= k_n \cdot 2\mu_s \epsilon(\tfrac{1}{2} U^n) \end{aligned}$$

where in the last equation the approximation

$$\frac{\partial S_s}{\partial t} = 2\mu_s \epsilon(U) + \nabla U S_s + S_s \nabla U^\top \approx 2\mu_s \epsilon(U)$$

has been used to write $S_s \approx \Delta t \cdot 2\mu_s \epsilon(U)$, where Δt is the time step. In total we get

$$\begin{aligned} \left[D_{U^n} f_1(U^n; v)\right] U^n_i = &\left(\rho(U^n_{\text{ALE}} \cdot \nabla)\tfrac{1}{2} U^n_i, v\right) \\ &+ \theta(2\mu_f \epsilon(\tfrac{1}{2} U^n_i), \nabla v) \\ &+ (1-\theta)(k_n \cdot 2\mu_s \epsilon(\tfrac{1}{2} U^n_i), \nabla v) \\ &+ \left(\delta_1 \rho(U^n_{\text{ALE}} \cdot \nabla)\tfrac{1}{2} U^n_i, \rho(U^n_{\text{ALE}} \cdot \nabla)v\right) \\ &+ \left(\delta_2 \nabla \cdot (\tfrac{1}{2} U^n_i), \nabla \cdot v\right) \end{aligned} \tag{3.46}$$

The bilinear and linear forms for the momentum equation, $a_{\text{M}}(U^n_{i+1}, v)$ and $L_{\text{M}}(v)$, are given by equation (3.44) as the left and right hand side respectively.

3.3.4 Solving the continuity equation

The weak residual for the pressure component can be written

$$\begin{aligned} r\left((U^n, P^n), (0, q)\right) &= \left(\nabla \cdot \bar{U}^n, q\right) \\ &\quad + \left(\delta_1 \left(\rho(U^n_{\text{ALE}} \cdot \nabla)\bar{U}^n + \nabla P^n - f\right), \nabla q\right) \\ &\equiv F(P^n; q) \end{aligned} \tag{3.47}$$

For the pressure component, the segregation method is

$$J_P = D_{P^n} F(P^n; q) \tag{3.48}$$

$$J_P(P^n_{i+1}; q) = J_P(P^n_i; q) - F(P^n_i; q) \tag{3.49}$$

where

$$\begin{aligned} J_P(P^n_i; q) &\equiv \left[D_{P^n} F(P^n; q)\right] P^n_i \\ &= (\delta_1 \nabla P^n_i, \nabla q) \end{aligned} \tag{3.50}$$

However, if we use Schur preconditioning [11], the iterative method instead becomes

$$\begin{aligned} J_P(P^n_{i+1}; q) + \left(2k_n \nabla P^n_{i+1}, \nabla q\right) = J_P(P^n_i; q) + (2k_n \nabla P^n_i, \nabla q) \\ - F(P^n_i; q) \end{aligned} \tag{3.51}$$

The bilinear and linear forms for the continuity equation then becomes

$$a_{\text{C}}(P^n_{i+1}, q) = \left(\delta_1 \nabla P^n_{i+1}, \nabla q\right) + \left(2k_n \nabla P^n_{i+1}, \nabla q\right) \tag{3.52}$$

$$\begin{aligned} L_{\text{C}}(q) &= (2k_n \nabla P^n_i, \nabla q) - \left(\nabla \cdot \bar{U}^n, q\right) \\ &\quad - \left(\delta_1 \rho(U^n_{\text{ALE}} \cdot \nabla)\bar{U}^n, \nabla q\right) + (\delta_1 f, \nabla q) \end{aligned} \tag{3.53}$$

where U^n_{ALE} is taken from the previous iteration step.[4]

3.3.5 The stabilisation parameters

The stabilisation parameters in the previous solver for high Reynolds numbers is chosen as

$$\delta_1 = C_1 \cdot h_n^{\frac{1}{2}} \left(\frac{1}{k_n^2} + \frac{|U^{n-1}|^2}{h_n^2}\right)^{-\frac{1}{2}} \tag{3.54}$$

$$\delta_2 = C_2 \cdot h_n^{\frac{3}{2}} \tag{3.55}$$

where $C_1 = 4$ and $C_2 = 2$.

In the new solver, developed for low Reynolds numbers, we instead use h^2 stabilisation, i.e.

$$\delta_1 = C_1 \cdot h_n^2 \tag{3.56}$$

$$\delta_2 = C_2 \cdot h_n^2 \tag{3.57}$$

where $C_1 = 1.6$ and $C_2 = 0.8$.

[4]The non-linear convective term $\rho(u \cdot \nabla)u$ is linearized by replacing it with $\rho(u_i \cdot \nabla)u$, where u_i is taken from the previous iteration. This is called the Picard method, see [11].

3.4 Problem formulation

Let us now summarise Section 3.1 and formulate the FSI problem which we are going to solve within the Unified Continuum model.

3.4.1 Theoretical summary

Let $Q = \Omega \times I$ be a space-time domain, where $\Omega \in \mathbb{R}^3$ is a spatial domain with boundary $\Gamma = \partial\Omega$, and where $I = [0, T]$ is the time domain. Let u be the velocity and ρ the density of the continuum. Let $f = \rho\, g$ be an external volume force with acceleration g, and let σ be the stress tensor. The incompressible unified continuum fluid–structure interaction problem is then given by

$$\begin{aligned} \rho\left(\partial_t u + (u \cdot \nabla)\, u\right) - \nabla \cdot \sigma &= f \quad \text{in } Q \\ \nabla \cdot u &= 0 \quad \text{in } Q \\ \partial_t \theta + (u \cdot \nabla)\, \theta &= 0 \quad \text{in } Q \\ \hat{u}\,(\cdot, 0) &= \hat{u}^0 \quad \text{in } \Omega \end{aligned} \tag{3.58}$$

where $\hat{u}^0$ are the initial conditions, and where the phase function θ defines the solid and the fluid domains by the subdomains

$$\begin{aligned} \Omega_s(t) &= \{x : x \in \Omega, \quad \theta(x,t) = 0\} \\ \Omega_f(t) &= \{x : x \in \Omega, \quad \theta(x,t) = 1\} \end{aligned}$$

The fluid part of the continuum is modeled as a Newtonian fluid and the solid part as an incompressible Neo-Hookean solid. The stress is then defined as

$$\begin{aligned} \sigma &= \bar{\sigma} - pI \\ \bar{\sigma} &= \theta\sigma_f + (1-\theta)\sigma_s \\ \sigma_f &= 2\mu_f \epsilon(u) \\ \partial_t \sigma_s &= 2\mu_s \epsilon(u) + \nabla u\, \sigma_s + \sigma_s \nabla u^\top \end{aligned}$$

where $\bar{\sigma}$ is the deviatoric part of the stress, i.e. the Cauchy stress, and where the subscripts s and f denote solid and fluid respectively. The strain rate tensor ϵ is defined as

$$\epsilon(u) = \frac{1}{2}\left(\nabla u + \nabla u^\top\right)$$

The material parameters μ_f and μ_s are the dynamical viscosity and the shear modulus respectively.

In Section 3.2, the phase equation was eliminated from the discrete system of equations by introducing a local ALE and a mesh velocity β_h. A stabilised Galerkin cG(1) cG(1) finite element method was applied to the discrete system, and the resulting system of equations is given in (3.30).

Newton's method is used to solve the non-linear system of equations, where the solution gives the velocity and the pressure of the continuum for each time step. With the segregation method, the velocity and pressure components are solved separately.

3.4.2 The geometry

The geometry of the cell model and the flow chamber is a simplified version of the setup of Khayyeri et. al [12].

The cell body is modeled as a hemisphere with a radius of 5.0 µm. It is equipped with a primary cilium modeled as a solid cylinder of length 5.0 µm and diameter 0.2 µm. The cell body model consists of an inner hemisphere for the cell cytoplasm, and an outer shell for the cell cortex. The thickness of the shell is 0.4 µm.

The cell model is placed at the bottom of a rectangular simulation box which serves as a model of a small flow chamber. The simulation box has the dimensions 100 µm, 200 µm and 50 µm, in the x-, y- and z-directions respectively. The fluid flows through the chamber in the direction of the positive y-axis, with an inlet at the left side and an outlet at the right side of the box. The simulation box with the cell model is shown in Figure 3.1. A closer view of the cell model is shown in Figure 3.2.

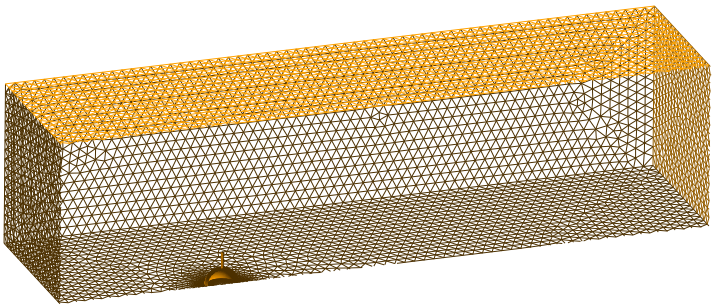


Figure 3.1: A cross section of the 3D mesh with the simulation box and the cell model.

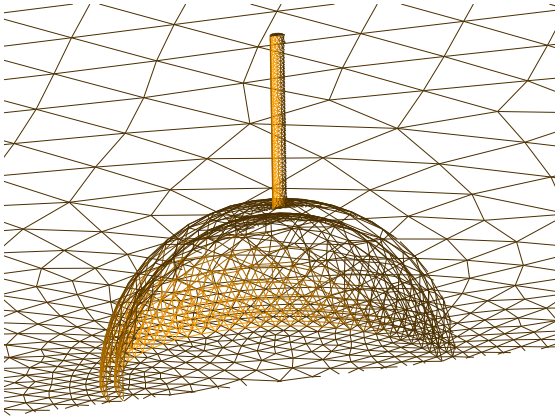


Figure 3.2: A cross section of the cell model with the primary cilium. Note that the cell body consists of two regions: an inner region (cytoplasm) and an outer shell (cortex).

3.4.3 Boundary conditions

The simulation box consists of three different boundaries, which are shown in Figure 3.3. At the left side of the box there is an inlet boundary Γ_{in}, and at the right side there is an outlet boundary Γ_{out}. The four side walls of the flow channel makes up a side wall boundary Γ_{sw}.

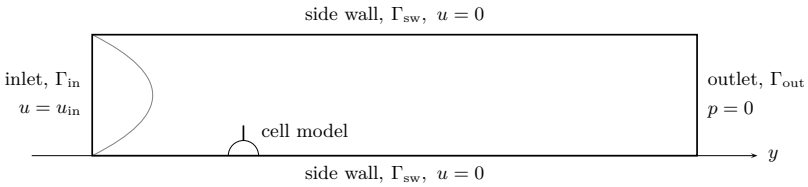


Figure 3.3: The boundaries of the simulation box and the corresponding boundary conditions. The inlet velocity profile has the shape of a parabola.

At the side walls of the box, homogeneous Dirichlet boundary conditions are applied for the velocity, i.e. $u = 0$ at Γ_{sw}. At the inlet boundary, an inlet velocity profile is applied and at the outlet boundary, the pressure is set to zero.

The fluid flows in the direction of the positive y-axis. The velocity profile at the inflow is parabolic and is given by the equation

$$u_{\text{in}}(x, z) = 16\, u_{\text{m}} \cdot \frac{(x - x_{\max})(x - x_{\min})}{(x_{\max} - x_{\min})^2} \cdot \frac{(z - z_{\max})(z - z_{\min})}{(z_{\max} - z_{\min})^2} \tag{3.59}$$

where the coordinates $(x_{\min}, z_{\min})$ and $(x_{\max}, z_{\max})$ define the rectangular inlet boundary, and where u_{m} is the maximum velocity of the fluid at the inflow in the y-direction. The inlet velocity profile is shown in Figure 3.4.

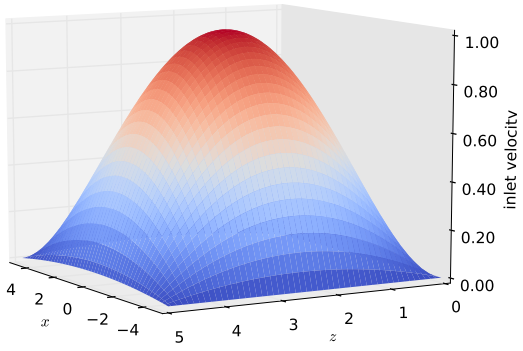


Figure 3.4: The inlet velocity profile with maximum velocity $u_{\text{m}} = 1$ in scaled units. The velocity scale is $U = 1.0\,\text{mm/s}$ and the length scale is $L = 10\ \mu\text{m}$.

3.4.4 The unit scales

The unit scales for the problem has been chosen according to the size of the cell and the maximum inlet velocity. The unit length is $L = 10\,\mu\text{m} = 10^{-5}\,\text{m}$ and the unit velocity is $U = 1.0\,\text{mm/s} = 10^{-3}\,\text{m/s}$. This gives us a time scale of

$$\tau = \frac{L}{U} = \frac{10^{-5}\,\text{m}}{10^{-3}\,\text{m/s}} = 10^{-2}\,\text{s} \tag{3.60}$$

If we let the density of the entire continuum be the same as of water, with $\rho_{\text{water}} = 10^3\,\text{kg/m}^3 = R$, we can use R as the unit scale for density, such that we have $\rho_f = \rho_s = \rho = 1$ in the calculations.

Further, we have that the dynamic viscosity of water is $\mu_f = 10^{-3}\,\text{Pa s}$. The unit scale of pressure is given by (3.38) and becomes

$$P = \frac{\rho\mu_f U}{\rho_f L} = \frac{\mu_f U}{L} = \frac{10^{-3}\,\text{Pa s} \cdot 10^{-3}\,\text{m/s}}{10^{-5}\,\text{m}} = 10^{-1}\,\text{Pa} \tag{3.61}$$

and as the unit scale for solid stress is the same, we have $S = 10^{-1}\,\text{Pa}$.

Finally, the external volume force has the unit scale of

$$F = \frac{\rho\mu_f U}{\rho_f L^2} = \frac{\mu_f U}{L^2} = \frac{10^{-3}\,\text{Pa s} \cdot 10^{-3}\,\text{m/s}}{(10^{-5}\,\text{m})^2} = 10^4\,\text{N/m}^3 \tag{3.62}$$

All the scale parameters are presented in Table 3.1.

quantity	value in SI units
time	$\tau = 10^{-2}$ s
length	$L = 10^{-5}$ m
velocity	$U = 10^{-3}\,\text{m/s}$
density	$R = 10^3$ kg/m^3
pressure	$P = 10^{-1}\,\text{Pa}$
volume force	$F = 10^4\,\text{N/m}^3$

Table 3.1: Unit scales of parameters used in the simulation.

3.4.5 Material parameters

The parameters used to model the *fluid* region in the unified continuum are the viscosity ν and the density ρ_f. For the *solid* regions, the parameters are shear modulus μ_s and density ρ_s. The densities are assumed to be the same for the fluid and the solids, i.e. $\rho_f = \rho_s = \rho = 1\,R$.

With a dynamic viscosity of $\mu_f = 10^{-3}$ Pa s $= 1\,P\,\tau$, the kinetic viscosity becomes

$$\nu = \frac{\mu_f}{\rho} = \frac{10^{-3}\text{ Pa s}}{10^3\text{ kg/m}^3} = 10^{-6}\text{ m}^2/\text{s} = 10^2\,\frac{L^2}{\tau} \tag{3.63}$$

i.e. $\nu = 100$ in scaled units.

The stiffness of a solid is determined by the shear modulus μ_s, which is calculated from the Young's modulus E of the solid as

$$\mu_s = \frac{E}{2(1+\nu_\text{P})} \tag{3.64}$$

where ν_P is the *Poisson's ratio* of the material. Poisson's ratio is a number between 0 and 0.5, and is a measure of the transverse expansion of the solid due to axial strain. When a piece of material is compressed in one direction it tends to expand in the other two directions. As an example, Poisson's ratio for cork is 0.0 while the corresponding value for rubber is close to 0.5. These two materials are very different in their ability of preserving volume under pressure. While cork is a highly compressible material, rubber on the other hand, is a material very close to incompressible.

However, since the solids are modeled as *incompressible* solids, the shear modulus is calculated with a Poisson ratio of 0.5, i.e. $\mu_s = \frac{1}{3}E$. The shear modulus of the different solid parts in the model are shown in Table 3.2.

solid part	E	$\mu_s = \frac{1}{3}E$
primary cilium	178 kPa	$5.93 \cdot 10^4$ Pa $= 5.93 \cdot 10^5\,S$
cell body (cortex)	2.00 kPa	667 Pa $= 6.67 \cdot 10^3\,S$
cell body (cytoplasm)	0.25 kPa	83.3 Pa $= 8.33 \cdot 10^2\,S$
stiff primary cilium	1.057 MPa	$3.52 \cdot 10^5$ Pa $= 3.52 \cdot 10^6\,S$
soft primary cilium	17 kPa	$5.67 \cdot 10^3$ Pa $= 5.67 \cdot 10^4\,S$

Table 3.2: Material parameters for the different solid parts in the cell model. The shear modulus μ_s is the parameter that determines the stiffness of the solid. The values of Young's modulus E are taken from the previous study [12].

3.4.6 The von Mises stress

To visualize the stress in the solid it is not very convenient to directly use the stress tensor, since it has nine components for each finite element. Instead, we calculate a scalar valued function called the *von Mises stress*. A definition of the von Mises stress σ_{VM} is given in [4] and reads

$$\sigma_{\text{VM}}^2 = \sum_{i,j=1}^{3} \left| \sigma_{ij} - \delta_{ij} \frac{1}{3} \text{tr}(\sigma) \right|^2 \tag{3.65}$$

where σ_{ij} are the components of the deviatoric stress tensor σ and where the trace of σ is defined as $\text{tr}(\sigma) = \sum_{i=1}^{3} \sigma_{ii}$. Note that σ_{VM} is a scalar while σ is a tensor.

Two different ways of calculating the von Mises stress has been carried out in the implemented code. One possibility is to compute the values directly from the stress tensor in the code, by iterating over all the elements in the mesh.

Another possibility is to define equation (3.65) on variational form in a form file, by defining bilinear and linear forms. The integral equation stated in the form file corresponds to the equation

$$\int_\Omega u\, v\, dx = \int_\Omega f\, v\, dx \tag{3.66}$$

where we seek the solution $u = \sigma_{\text{VM}}^2$ and where f is the right hand side of equation (3.65).

Since the function σ_{VM} is a piecewise constant function, also the test function v is piecewise constant on the finite elements. Both the left and the right hand side of equation (3.66) then correspond to diagonal matrices where the cell volume is present as a factor on the diagonal, i.e.

$$\begin{bmatrix} u_1 V_1 & & & \\ & u_2 V_2 & & \\ & & \ddots & \\ & & & u_N V_N \end{bmatrix} = \begin{bmatrix} f_1 V_1 & & & \\ & f_2 V_2 & & \\ & & \ddots & \\ & & & f_N V_N \end{bmatrix} \tag{3.67}$$

Instead of assembling both the bilinear form (the left hand side) and the linear form (right hand side) and then solve the system with a Krylov solver, we can simply multiply the equation with the inverse volume to get

$$\begin{bmatrix} u_1 & & \\ & \ddots & \\ & & u_N \end{bmatrix} = \begin{bmatrix} \frac{1}{V_1} f_1 V_1 & & \\ & \ddots & \\ & & \frac{1}{V_N} f_N V_N \end{bmatrix} \tag{3.68}$$

In this way, we obtain the von Mises stress squared simply by assembling the right hand side of equation (3.68).

Chapter 4

Method

4.1 Software and resources

All calculations have been performed on the Beskow supercomputer located at PDC Center for High Performance Computing at KTH. To utilise the parallel performance of Beskow, FEniCS-HPC has been used, which is the parallel implementation of the standard FEniCS distribution. The FEniCS-HPC/Unicorn version used for this project is called `fenics-hpc_eunison_dist`.

Other software used in this project are:

- ANSA – mesh generation software (BETA CAE Systems)
- SALOME – mesh generation software (open source)
- Gmsh – mesh converting and visualisation software
- ParaView – data visualisation software

The code for the FEniCS-HPC/Unicorn solver is written in C++, which is the main programming language used in the project. For mesh generation and for plotting, Python scripts has been used. For repeated working tasks, such as mesh converting and file transferring, small script files written in Bash has been used. Some script files are given in Appendix C.

4.2 Mesh generation

4.2.1 Mesh generation in ANSA

A three-dimensional mesh of the cell model has been constructed with the software ANSA from BETA CAE Systems [14]. The original mesh of the cell, as developed by the authors of [12], was used as input mesh to the new mesh created in ANSA. From the imported mesh, a *geometry* of the cell body and the primary cilium was created by surface wrapping. A rectangular box was built up around the cell, and the cell body was fixed at the bottom of the box. From the geometry of the cell model and the box, a *surface mesh* and a *volume mesh* was generated. The volume mesh was finally exported from ANSA in the NASTRAN file format.

The steps carried out to create a full volume mesh for the cell model are given more in detail in Appendix B.1. A surface mesh of the cell model with the primary cilium is shown in Figure 4.1.

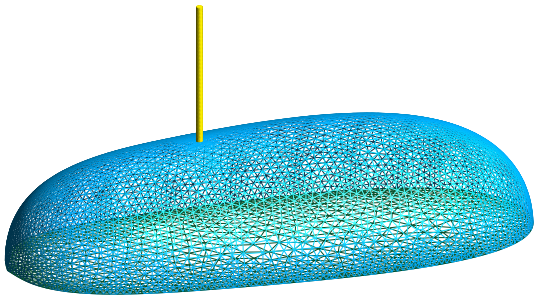


Figure 4.1: A surface mesh of the cell model generated in ANSA. The geometry is very close to the original cell model as developed by Khayyeri et al. [12].

However, it turned out that the mesh created in ANSA was not very suitable for testing purposes of the developed FSI solver. For that reason, a simpler and coarser mesh of the cell model has been created in SALOME.

4.2.2 Mesh generation in SALOME

A simplified geometry of the cell model has been created in the open source software SALOME [2]. The geometry and the mesh is generated by SALOME from a Python script, loaded by the software. The script files used are given in Appendix C.1.

The geometry of the model is built up by basic geometry objects: two spheres, one cylinder and a rectangular box. The complete geometry is then processed by a mesh generating algorithm which discretises the geometry into tetrahedron elements. Finally, the mesh is exported to UNV and XML file formats. The generated mesh of the simple cell model is shown in Figure 4.2.

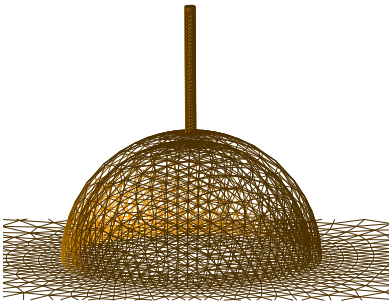


Figure 4.2: The mesh of the cell model generated in SALOME at the bottom of the flow chamber. The cell body consists of two parts: the cytoplasm (inner part) and the cortex (outer shell). The primary cilium is attached at the surface of the cortex.

4.3 Implementation

The framework for the implementation is the Unicorn solver developed for FSI problems at *high* Reynolds numbers. Since this project is about developing a FSI solver in the regime of *low* Reynolds numbers, some parts of the original implementation had to be adjusted.

Some of the new features of the Unicorn solver developed in this project are

- ability to model low Reynolds number flow
- ability to have multiple solid regions with different material parameters
- ability to measure the mass flow at the inlet and the outlet
- ability to calculate the von Mises stress in the solid
- ability to locate a vertex in the mesh and to track its position during the simulation (used to measure the deflection of the primary cilium tip)

The new features of the code will be described more in detail in the following sections.

4.3.1 The main file: `main.cpp`

Most of the specific information to define the FSI problem are declared in the main file, `main.cpp`. The subdomains, the boundary conditions and some additional mesh functions passed to the solver are all stated in the main file. All parameter values used by DOLFIN-HPC, which is the core implementation of FEniCS-HPC [9, 6], are read from the file `parameters`.

The main file also includes routines for detecting a subset of the mesh to identify different solid regions, and to set material parameters of the different solids. The solid stiffness parameters are given in the file `solid_parameters`.

Preprocessing routines that can be used for the input mesh are also given in the main file. One of these routines is used to resize the mesh according to the unit scale of length.

The main file can be found in Appendix C.4.

4.3.2 The Unified Continuum solver: `NSESolver.cpp`

The code which specifies the solving algorithm of the UC model is the class `UCSolver`, defined in the file `NSESolver.cpp`. This file contains many C++ functions, for example the functions `update` and `ComputeStabilization` where the time step k and the stabilisation parameters δ_1 and δ_2 are calculated.

The code also includes calculations of various quantities during the simulation, and some new functions have been implemented for this purpose. In the function `computeMassFlow`, the mass flow through the inlet and outlet boundaries are calculated. The von Mises stress is calculated by the function `computeVonMisesStress`. Both of these functions use form files to specify the equations to solve.

Another new feature of the code is to find a vertex and to track its position during the simulation. It has been used to measure the deflection of the primary cilium tip, but can be used to track any point in the mesh. The code works

as follows. The vertex closest to a given point is located with the function `find_closest_vertex` and the local index of this vertex is saved. A boolean flag is used to mark if the current process is owner of the vertex or not. The deflection of the cilium is calculated by the function `compute_deflection`. Only the process owner of the vertex prints the coordinates to file.

4.3.3 The form files

Form files in FEniCS are input files that defines the equations to solve. In a form file, the equation is given in its weak formulation, defined by the bilinear and linear forms. Form files are compiled by FFC, *the FEniCS form compiler*, which generates code in C++ for the solver.

Code from several form files are used by the Unicorn solver. The main form files are `NSEMomentum3D.form` and `NSEContinuity.form`, used to solve the momentum and continuity equations respectively. These two files have been modified to fit the FSI problem considered in this project.

The main purpose of this project has been to develop a solver to simulate low Reynolds number flow. In Section 2.3.2 we saw how the parameters α and β in equation (2.36) are set according to the Reynolds number. In Section 3.2.5 we came to a similar conclusion for the UC model. The parameters α and β are used as input parameters in `NSEMomentum3D.form`, and their values are determined by the function `set_alpha_beta` in the file `NSESolver.cpp`.

The modified form files are found in Appendix C.3. The bilinear and linear forms for the momentum equation are given in Section 3.3.3. The corresponding forms for the continuity equation are given in Section 3.3.4.

As mentioned above, form files are also used to calculate the mass flow and the von Mises stress. These files are also given in Appendix C.3.

Chapter 5

Results

5.1 Mesh data

Several different meshes have been used in the simulations and the data for all meshes are presented in Table 5.1. The meshes A1–A3 are generated by the script `generate_mesh_empty` while all other meshes are generated by the script `generate_mesh`. The fine and the coarse meshes have been generated by adjusting the maximum size of the elements in the mesh generation script.

The default length of the primary cilium is 5 µm. The short cilium has a length of 3 µm and the long cilium is 10 µm.

The Unicorn solver uses one mesh (called `mesh`) for the entire region, and another mesh (called `structure`) to define the structure region in the unified continuum. Additional meshes (`solid_1`, `solid_2`, `solid_3`) are used to define sub-regions of the structure.

ID	mesh	# vertices	# cells	hmin
A1	flow chamber mesh, coarse	24276	128361	0.2132
A2	flow chamber mesh, medium	53032	282232	0.2644
A3	flow chamber mesh, fine	190350	1065963	0.1464
D	FSI mesh, default	64314	346027	0.01059
D0	structure (cell model)	2693	10736	
D1	solid 1 (cilium)	806	2557	
D2	solid 2 (cortex)	1333	3954	
D3	solid 3 (cytoplasm)	1150	4225	
S	FSI mesh, short cilium 3 µm	61410	329093	0.01041
L	FSI mesh, long cilium 10 µm	71579	388402	0.00971
C	FSI mesh, coarse	35526	191386	0.01036
F	FSI mesh, fine	269512	1676707	0.00967

Table 5.1: Data for the meshes used in the calculations. The flow chamber meshes are used for pure fluid flow calculations only. The default mesh D for the FSI calculations are listed together with the sub-meshes D0–D3 that defines the different solid parts of the cell model. All other FSI meshes have corresponding sub-meshes.

5.2 Simulation results

To test and validate the developed Unicorn solver for low Reynolds numbers, the following simulations have been performed:

- Pure fluid flow with different meshes
- FSI with only the cell body as structure (no primary cilium)
- FSI with full cell model (cell body and primary cilium)
- FSI with stiff and soft primary cilium
- FSI with short and long primary cilium
- FSI at different Reynolds numbers

The simulation results are given in Table 5.2 and in Figures 5.1–5.11 in the following sections. The number of CPUs used on Beskow was in general 256 for each simulation.

		input parameters					output values		
#	simulation	mesh	Re	T	C1/C2	k	time	MF	defl.
1	pure flow	A1	0.01	1.0	1.2/0.6	1.0e-3	2m 10s	0.827	-
2	pure flow	A2	0.01	1.0	1.2/0.6	1.4e-3	2m 33s	0.893	-
3	pure flow	A3	0.01	1.0	1.2/0.6	4.3e-4	19m 45s	0.955	-
4	pure flow	D	0.01	0.2	1.2/0.6	2.2e-5	1h 37m	0.894	-
		D	0.01	0.2	1.6/0.8	4.5e-5	41m 11s	0.862	-
5	cell body only	D	0.01	0.2	1.6/0.8	4.5e-5	38m	0.862	-
6	FSI, $Re = 0.01$	D	0.01	2.0	1.6/0.8	9.0e-5	3h 28m	0.862	0.326
		D	0.01	1.0	1.6/0.8	9.0e-5	2h 19m	0.862	0.268
		D	0.01	1.0	1.2/0.6	2.2e-5	4h 50m	0.894	0.279
7	coarse mesh	C	0.01	1.0	1.6/0.8	8.6e-5	1h 37m	0.781	0.249
8	fine mesh	F	0.01	1.0	1.6/0.8	7.5e-5	7h 47m	0.936	0.291
9	long cilium	L	0.01	2.0	1.6/0.8	9.0e-5	5h 32m	0.863	0.904
10	short cilium	S	0.01	2.0	1.6/0.8	9.0e-5	3h 3m	0.861	0.080
11	stiff cilium	D	0.01	2.0	1.6/0.8	9.0e-5	4h 13m	0.862	0.305
12	soft cilium	D	0.01	2.0	1.6/0.8	4.5e-5	7h 9m	0.862	0.441
13	FSI, $Re = 0.1$	D	0.1	4.0	0.8/0.4	9.0e-4	1h 18m	0.927	0.372
14	FSI, $Re = 1$	D	1	10	0.4/0.2	9.0e-4	3h 4m	0.963	0.393
15	FSI, $Re = 10$	D	10	40	0.4/0.2	4.5e-3	1h 10m	0.995	0.068
16	FSI, $Re = 100$	D	100	40	0.4/0.2	4.5e-3	1h 13m	0.998	0.012

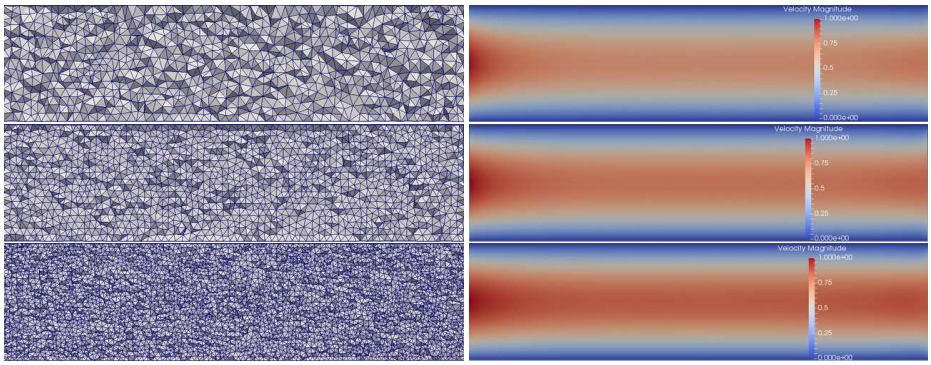
Table 5.2: Input parameters and output values for the simulations. Input parameters are: type of mesh from Table 5.1, Reynolds number Re, simulation time T in time units, stabilisation parameters C_1 and C_2, and approximate time step k in time units. The output values are: running time of calculation, ratio of the mass flow through the outlet and inlet boundaries (MF), and deflection of cilium tip in length units (distance from initial position). Additional material parameters are given in Table 3.2.

5.2.1 Pure fluid flow simulations

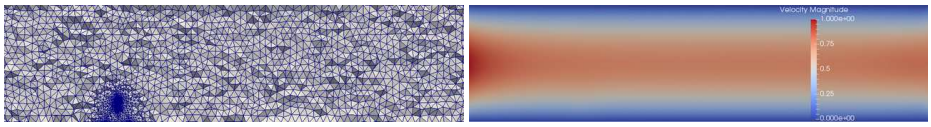
The Unicorn solver for low Reynolds numbers has been tested with four different meshes for pure fluid flow, where no solid structure is present. The simulation cases are

- Pure fluid flow with a coarse mesh (mesh A1)
- Pure fluid flow with a medium mesh (mesh A2)
- Pure fluid flow with a fine mesh (mesh A3)
- Pure fluid flow with the default mesh (mesh D)

The results of the simulations are shown in Figure 5.1. The mass conservation of the flow is measured as the ratio of the mass flow through the outlet and the inlet boundaries. The output values are presented in Table 5.2. With the coarse mesh (A1) the ratio is about 83% and with a medium mesh (A2) the ratio is 89%. The mass conservation is about 96% if the fine mesh (A3) is used.



(a) Fluid flow with coarse, medium and fine mesh. Left: cross-sections of the meshes. Right: the velocity magnitude of the fluid flow with the three meshes. The stabilisation parameters are $C_1 = 1.2$ and $C_2 = 0.6$.



(b) Fluid flow with default mesh and no structure present. The stabilisation parameters are $C_1 = 1.6$ and $C_2 = 0.8$ and the mass flow ratio is 86%.

Figure 5.1: Results of pure fluid flow through the chamber without any structure present. The meshes are from above: A1 (coarse mesh), A2 (medium mesh), A3 (fine mesh) and D (default mesh). The mass conservation of the flow is higher with a finer mesh and lower with a coarser mesh. The stabilisation parameters in the FSI simulations are chosen with respect to the mass flow ratio and a reasonable time step length.

The choice of stabilisation parameters also affect the degree of mass conservation. With smaller values of the constants C_1 and C_2, the mass flow ratio increases, but smaller time steps are also needed. Larger time steps are possible if the stabilisation parameters are increased, but with a lower mass flow ratio as a consequence. An optimal choice of the stabilisation parameters is where the mass flow ratio is sufficiently high and the time steps are large enough to get a reasonable total simulation time.

The default mesh D has a mass conservation of about 89% with the same choice of stabilisation parameters as for the three meshes A1, A2 and A3. With a slightly increased stabilisation, only half of the time is needed to run the entire simulation. With the choice $C_1 = 1.6$ and $C_2 = 0.8$, the mass flow ratio decreases to about 86% for the default mesh. These values have been used for all the other simulations at low Reynolds number.

5.2.2 FSI with cell body only

The first FSI case is a fluid–structure interaction simulation where only the cell body is present. The results of the simulation is shown in Figure 5.2.

5.2.3 FSI with default cell body model

Simulation results of the full fluid–structure interaction problem with the entire cell body model are shown in Figures 5.3 and 5.4. The simulation is run up to time $t = 2.0\,\tau$ where the primary cilium has almost reached its maximum deflection. However, running the simulation further shows that the solution blows up at time $t = 3.0\,\tau$ due to large deformations of the mesh.

The blow-up of the solution can be avoided either by improved mesh smoothing or by re-meshing, i.e. a reconstruction of the mesh to improve the quality of the elements in the mesh. In the case of re-meshing, the solution is projected from the old mesh to the new mesh and a projection error is introduced. To minimise the total error in the solution, the number of re-meshing operations in the simulation should be as few as possible.

FSI simulations with the default cell body model have also been performed with the coarse mesh C and the fine mesh F. The results are consistent with the conclusions from the pure fluid flow simulations, which showed that the mass conservation increases as the finite elements become smaller. The mass conservation ratio for the meshes C and F are shown in Table 5.2.

5.2.4 FSI with stiff primary cilium

The results of a simulation with the default cell model with a stiff primary cilium is shown in Figure 5.5. The stiffness of the cilium is $3.52 \cdot 10^5$ Pa.

5.2.5 FSI with soft primary cilium

The results of a simulation with the default cell model with a soft primary cilium is shown in Figure 5.6. The stiffness of the cilium is $5.67 \cdot 10^3$ Pa.

5.2.6 FSI with short primary cilium

The simulation results of a cell model with a short primary cilium of length 3.0 µm is shown in Figure 5.7. At $t = 2.0\,\tau$ the primary cilium has reached a stationary state with a tip deflection of 0.8 µm.

5.2.7 FSI with long primary cilium

In Figure 5.8 the results of a simulation with a long primary cilium of length 10 µm is shown at time $t = 1.9\,\tau$. The stress on the cell cortex is at this moment on its maximum, and in the next time steps the solution blows up due to inverted elements in the moving mesh. A closer view of the region of collapse is shown in Figure 5.9.

The blow-up of the solution for large deformations of the structure is a limitation of the solver. As mentioned above, this issue can be resolved by an optimisation of the mesh smoothing algorithm, or alternatively by allow for re-meshing when the mesh quality is bad.

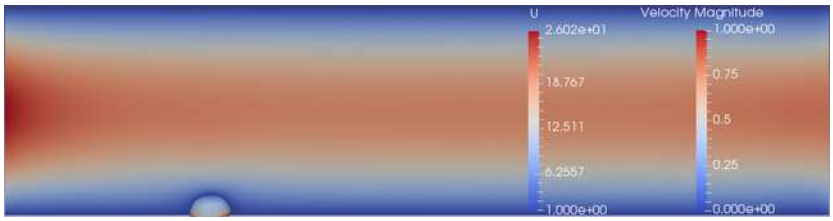
5.2.8 FSI at different Reynolds numbers

The solver has also been tested at different Reynolds number by changing the viscosity of the fluid while the inflow velocity is unchanged. The results of the simulations are shown in Figures 5.10 and 5.11.

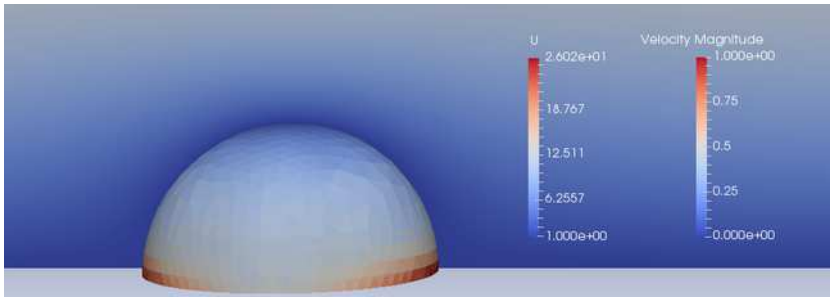
At high Reynolds numbers the forces on the cilium are smaller than at low Reynolds numbers, and larger time steps are therefore possible. However, at high Re the pressure difference between the inlet and the outlet is low and the flow needs time to propagate through the chamber, which gives a longer simulation time.

The stabilisation parameters used are different for high and low Re. However, in all the cases h^2 stabilisation has been applied, according to the equations (3.56) and (3.57). The following values of C_1 and C_2 have been used:

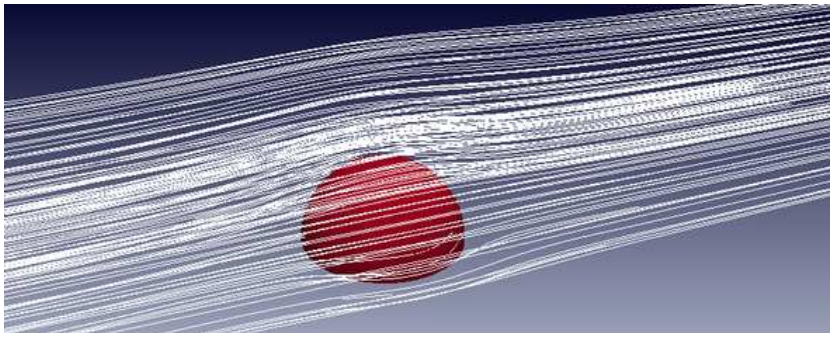
$$Re < 0.1 : \begin{cases} C_1 = 1.6 \\ C_2 = 0.8 \end{cases} \qquad 0.1 \leq Re < 1 : \begin{cases} C_1 = 0.8 \\ C_2 = 0.4 \end{cases} \qquad Re \geq 1 : \begin{cases} C_1 = 0.4 \\ C_2 = 0.2 \end{cases}$$



(a) The cell body in the flow chamber at $t = 0.2\,\tau$. The colouring on the cell body shows the von Mises stress in scaled units.

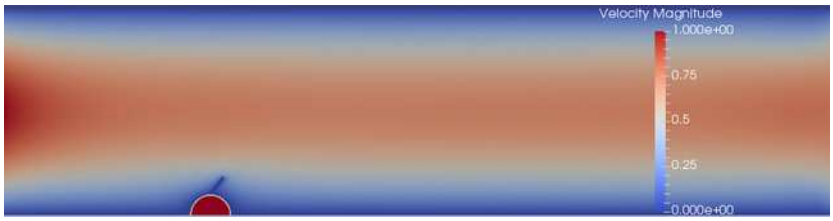


(b) The stress on the cell body is highest in the region closest to the bottom. The magnitude of the von Mises stress is however rather low, with a maximum value of $26\,S$ or 2.6 Pa.

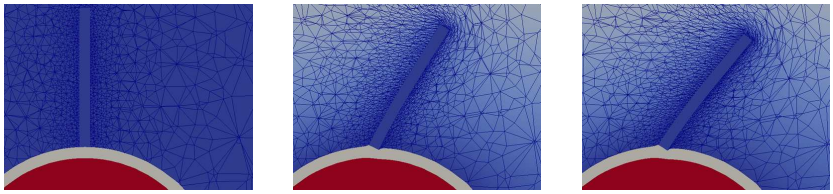


(c) Streamlines of the fluid flow around the cell body.

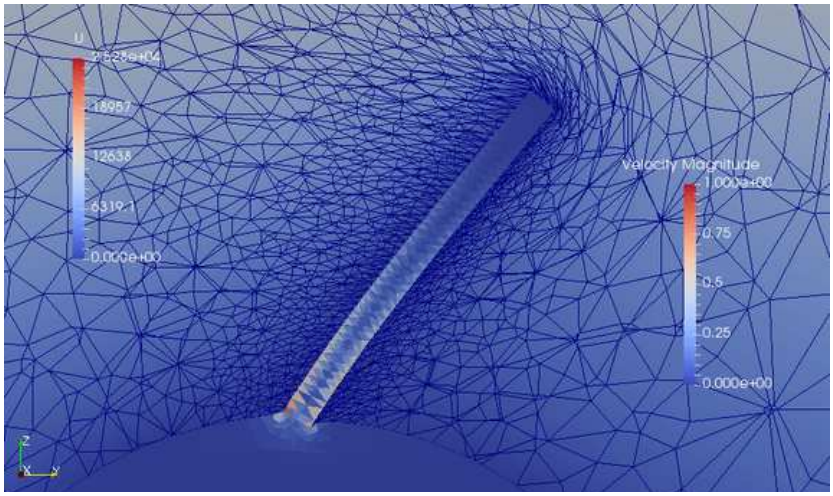
Figure 5.2: Results of a simulation with only the cell body present as the solid structure. The stress on the solid and the streamlines around the cell body indicate that the structure is interacting with the fluid.



(a) A cut through the flow chamber and the default cell body model at $t = 2.0\,\tau$. The colours of the cell model are used to mark the different parts of the cell: primary cilium (blue), cell cortex (white) and cytoplasm (red).

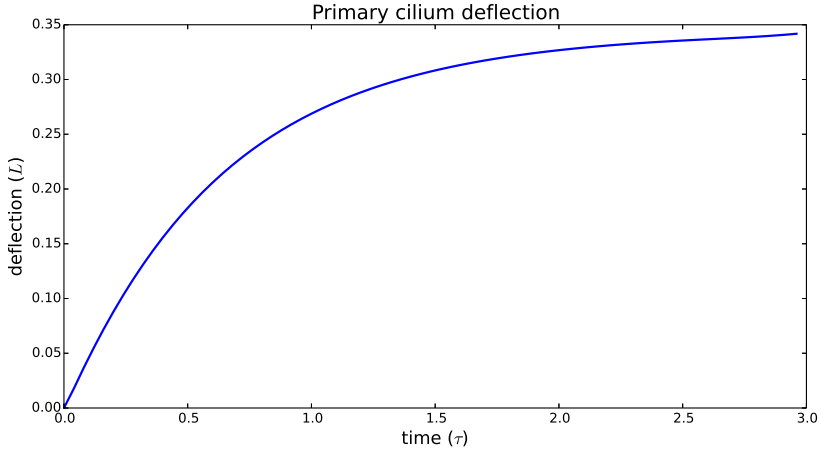


(b) A close-up view of the mesh near the cell at the initial position (left), at time $t = 1.0\,\tau$ (middle) and at time $t = 2.0\,\tau$ (right). The mesh in the fluid region follows the deformation of the structure.

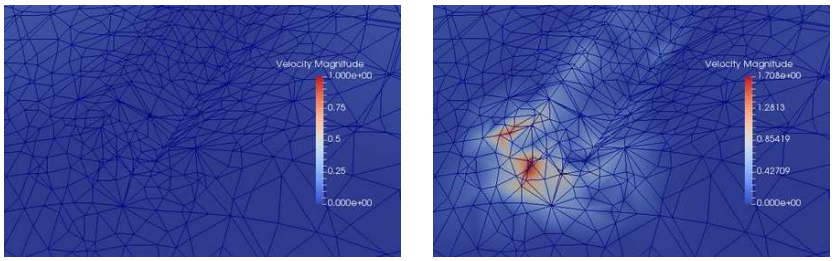


(c) A cut through the structure shows the von Mises stress in the primary cilium and in the cell cortex. The maximum stress at $t = 2.0\,\tau$ is around 2.5 kPa and is located in the cilium near the surface of the cell cortex. The deflection of the cilium tip is at this moment 3.3 µm.

Figure 5.3: Simulation results with the default cell body model. The primary cilium is deflected by the flow and the mesh moves for each time step.

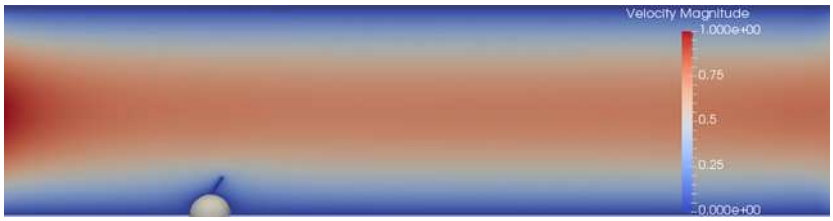


(a) The deflection of primary cilium over time. At $t = 3.0\,\tau$ the cilium reaches a maximum deflection of 3.4 µm, just before blow-up of the solution.

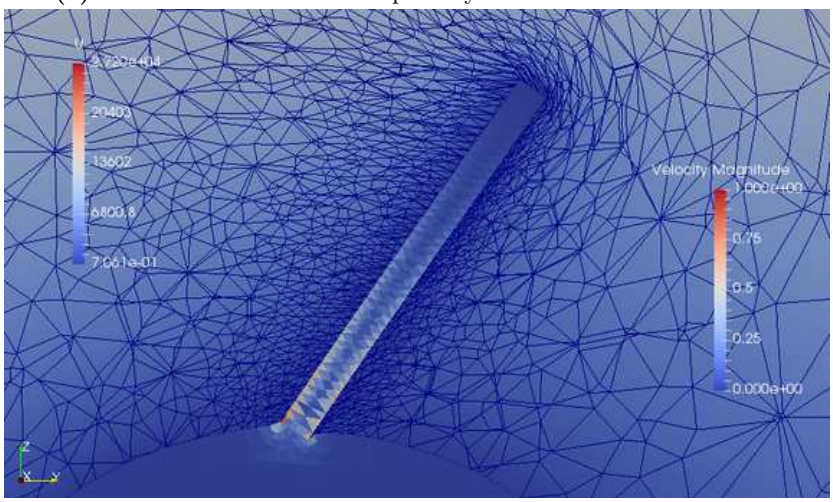


(b) A close-up view of the solution around the cell cortex at $t = 2.0\,\tau$ (left) and at $t = 3.0\,\tau$ (right). The right figure shows the collapsed elements and the blow-up of the solution due to large deformations of the mesh.

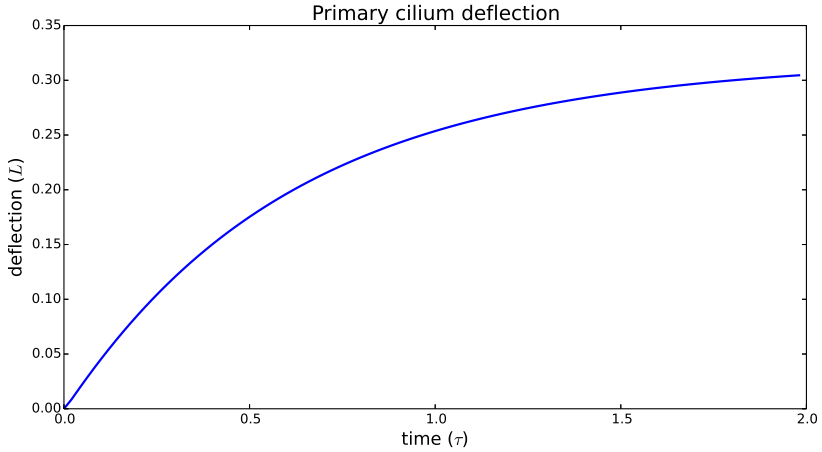
Figure 5.4: Simulation results with the default cell body model up to time $t = 3.0\,\tau$ showing the deflection of the primary cilium and the blow-up of the solution.



(a) The cell model with a stiff primary cilium in the flow chamber.

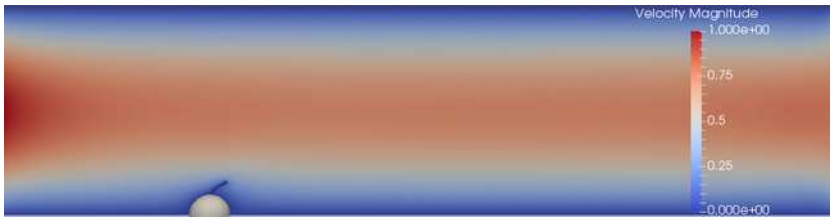


(b) The deflection of the stiff primary cilium and the stress on the structure. Notice the strong deformation of the cell cortex at the right side of the cilium.

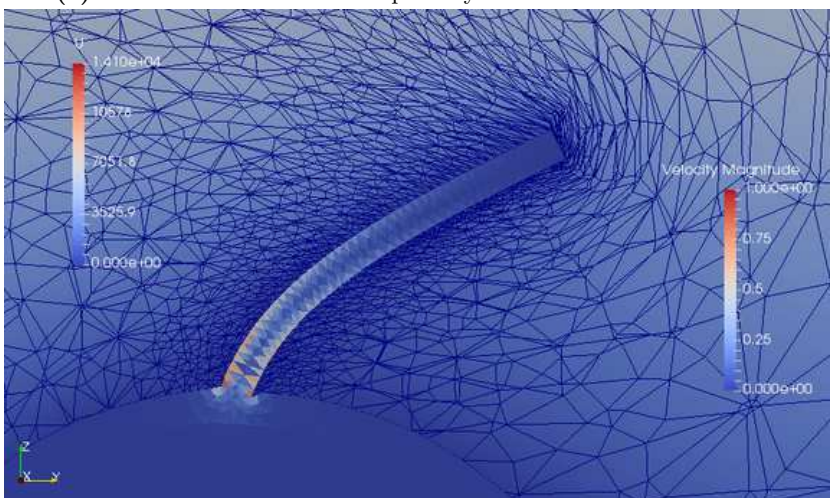


(c) The deflection of primary cilium over time. The deflection reaches an almost stationary state at $t = 2.0\,\tau$.

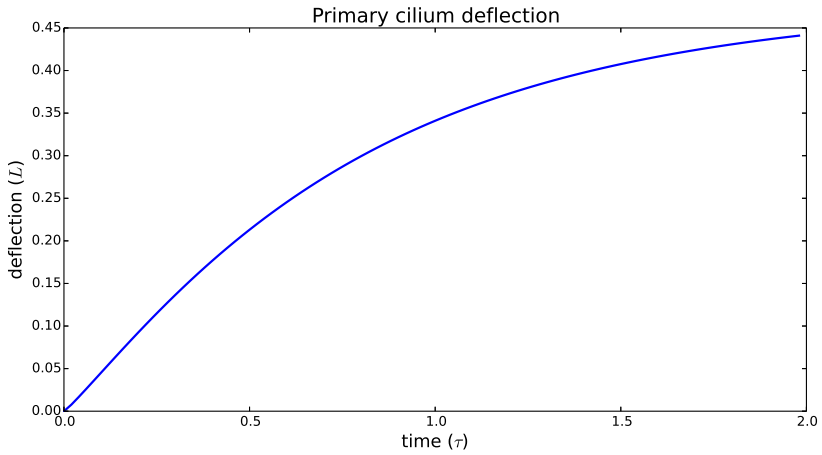
Figure 5.5: Default cell model with stiff primary cilium at $t = 2.0\,\tau$. The maximum stress is about 2.7 kPa and the deflection of the cilium tip is 3.1 µm.



(a) The cell model with a soft primary cilium in the flow chamber.

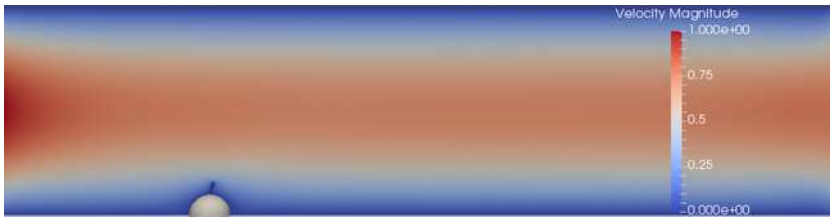


(b) The deflection of the soft primary cilium and the stress on the structure.

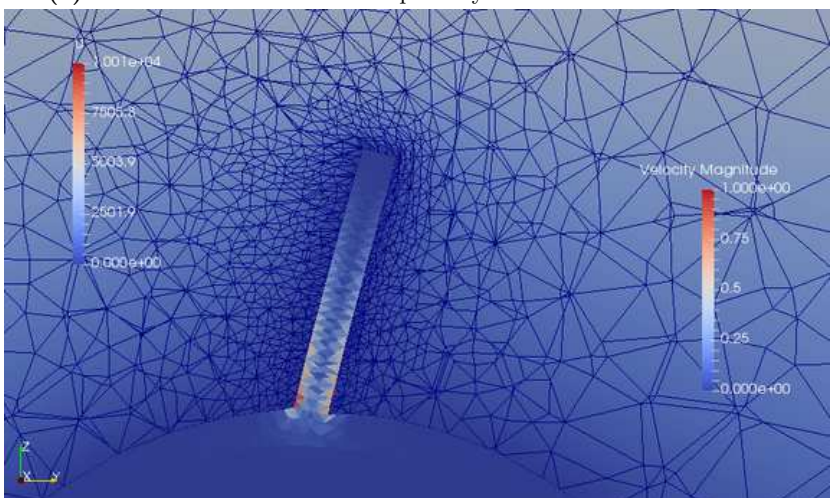


(c) The deflection of primary cilium over time. At $t = 2.0\,\tau$ the deflection of the cilium tip is close to a stationary state.

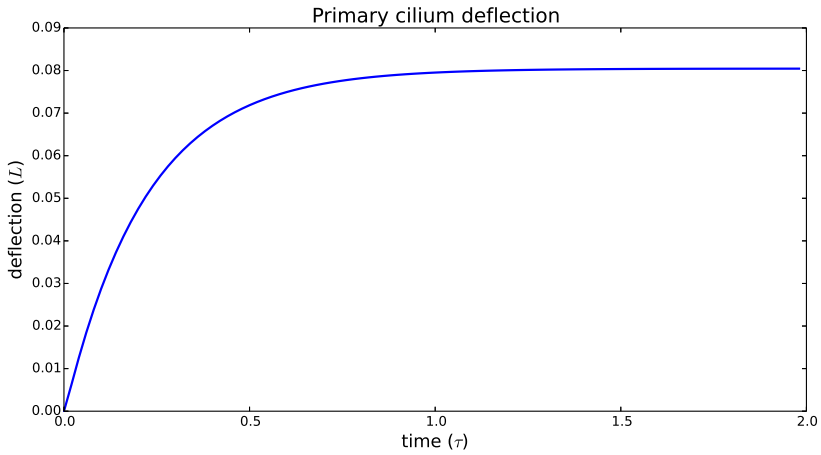
Figure 5.6: Simulation results of default cell model with soft primary cilium at $t = 2.0\,\tau$. The maximum stress is about 1.4 kPa and the deflection of the cilium tip is 4.4 µm from its initial position.



(a) The cell model with a short primary cilium in the flow chamber.

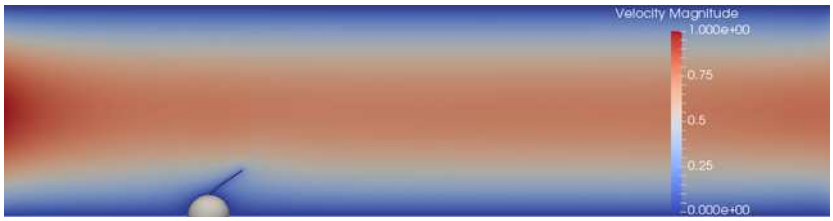


(b) The deflection of the short primary cilium and the stress on the structure.

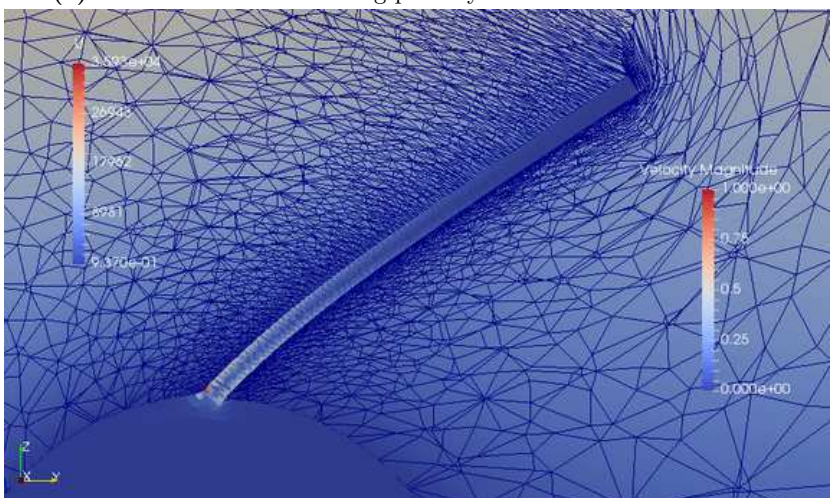


(c) The deflection of primary cilium over time. A stationary state is reached already at around $t = 1.0\,\tau$.

Figure 5.7: Results of a simulation with a cell model with short primary cilium at time $t = 2.0\,\tau$. The maximum stress is 1.0 kPa and the deflection of the cilium tip is 0.8 µm from its initial position.



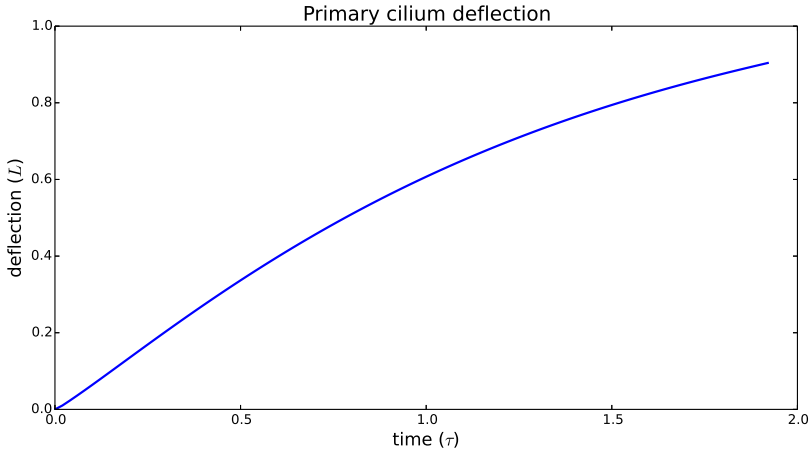
(a) The cell model with a long primary cilium in the flow chamber.



(b) The deflected long primary cilium and the stress on the structure. The maximum stress at this moment is around 3.6 kPa.

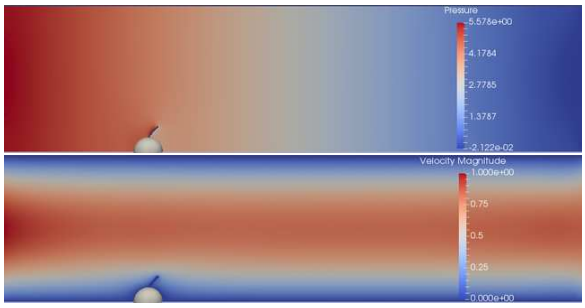
Figure 5.8: Results of a simulation with a cell model with long primary cilium at time $t = 1.9\,\tau$, just before blow-up of the solution.

(a) A close-up view of the cilium and the cell cortex at the times $t = 1.5\,\tau$ (left) and $t = 1.9\,\tau$ (right). The solution blows up due to collapse of elements in the strongly deformed region in the cortex at the right side of the deflected cilium.

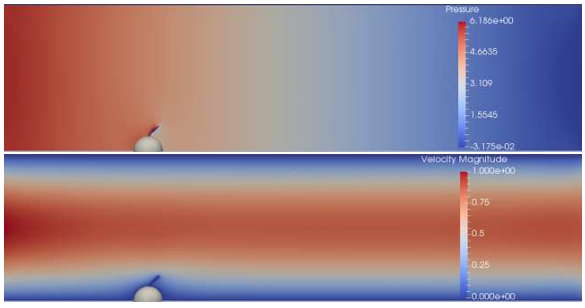


(b) The deflection of primary cilium over time. The solution blows up before a stationary state is reached.

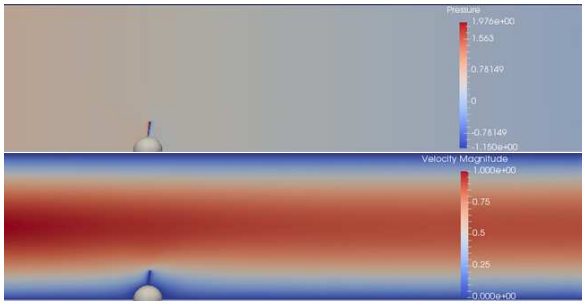
Figure 5.9: The solution with the long primary cilium blows up at $t = 1.9\,\tau$ and the region of collapse occurs in the region between the cilium and the cortex.



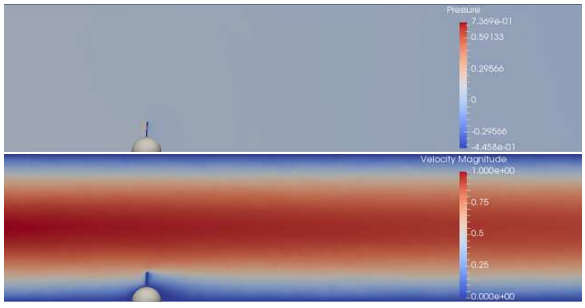
(a) The cell model in a fluid flow at $t = 2.0\,\tau$ and with $Re = 0.1$.



(b) The cell model in a fluid flow at $t = 10\,\tau$ and with $Re = 1$.

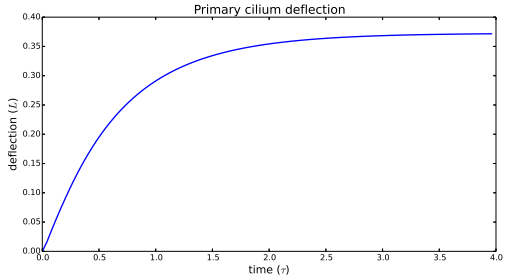


(c) The cell model in a fluid flow at $t = 40\,\tau$ and with $Re = 10$.

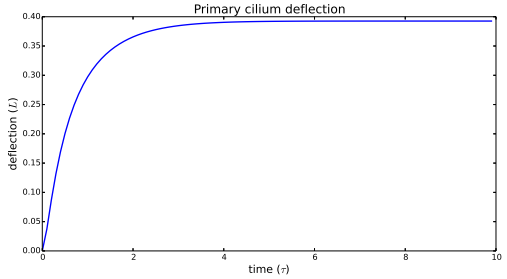


(d) The cell model in a fluid flow at $t = 40\,\tau$ and with $Re = 100$.

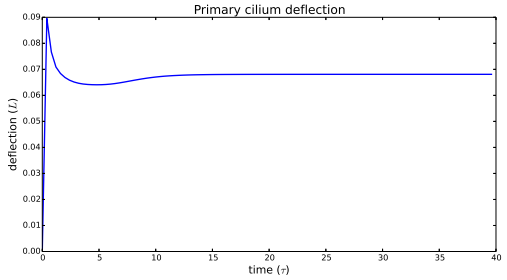
Figure 5.10: Pressure and velocity of fluid–structure interaction simulations at different Reynolds numbers. The maximum velocity of the fluid is the same for all cases, only the viscosities are different. With an increasing Reynolds number the forces on the structure decreases.



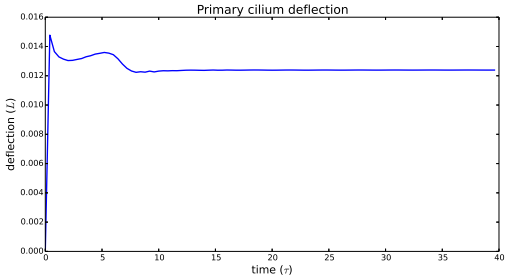
(a) $Re = 0.1$.



(b) $Re = 1$.



(c) $Re = 10$.



(d) $Re = 100$.

Figure 5.11: Deflection of primary cilium in scaled units at different Reynolds numbers. The deflection is measured as the distance from the initial position of the primary cilium tip.

5.3 Discussion

The Unicorn solver developed in this project has been extended to apply to fluid flow problems where the Reynolds number is low. In particular, the solver has been tested on a cell mechanical problem where $Re = 0.01$.

The results show that the mass conservation of the flow is high (86%) with the given choice of the stabilisation parameters C_1 and C_2. The mass conservation ratio can be increased by decreasing the values of the stabilisation parameters, or by refining the finite element mesh.

A consequence of having smaller stabilisation is that the time for running the entire simulation increases. With smaller stabilisation the algebraic system becomes more ill-conditioned and harder to solve, and smaller time steps are needed for convergence in the solution.

As an alternative to the stabilised method, mixed approximation spaces may be used for which the inf-sup condition is satisfied and for which stabilisation may be avoided. For example, with Taylor-Hood elements the function spaces are $\mathcal{P}_2/\mathcal{P}_1$, i.e. second-order polynomials are used for the velocity and linear functions are used for the pressure. With mixed approximation spaces no stabilisation is needed and the mass conservation of the flow may be exactly fulfilled. However, the number of unknowns in the equations increase which most likely will lead to longer simulation time.

A limitation of the solver shows up for large mesh deformations as the quality of the mesh becomes bad. Some of the elements in the mesh collapse by the strong deformation, and the solver reaches a point where the system of equations no longer can be solved.

A further development of the solver would be to find a strategy for treating large mesh deformations, either by optimisations of the mesh smoothing algorithm or by employing re-meshing to maintain sufficient quality of the mesh.

Another interesting topic as a further work would be to investigate the performance of the solver with different preconditioners for the algebraic system. Perhaps, with a different preconditioner it may be possible to have faster convergence of solutions at low Reynolds numbers. If the rate of convergence for each time step can be increased, larger time steps may be possible.

When it comes to the cell mechanical problem with the primary cilium, it has solely been used as a model problem to test the performance of the solver. The cell model is greatly simplified and we are not able to draw any conclusions about the primary cilium mechanosensation from this study. However, with a more sophisticated cell model, where the inner structure of the cell is represented, it may be possible to reproduce the results of Khayyeri et al. [12] even with the current solver.

The large deformations of the primary cilium in the simulations compared to the previous results indicate an incompleteness of the simple cell model. In the simple model, the primary cilium is assumed to be attached to the surface of the cell cortex only. Since the cortex is a much more flexible material than the cilium, the deformation of the structure is strongest at the cell cortex close to the primary cilium.

A more realistic model is the cell model of Khayyeri et al. where the cilium is connected both to the cell cortex and to microtubules inside the cell body. The microtubules act as strong wires holding the primary cilium in an upright position and prevents the primary cilium to damage the cell cortex by

its deflection.

With an improved cell model, where microtubules are present, the deflection of the primary cilium would most probably be smaller since the microtubules would keep the bottom of the cilium in an almost horizontal position. Since the current solver can handle small deformations, it would be possible to use it to find a stationary state of the deflected primary cilium.

A further developed FSI solver, that can handle large mesh deformations, together with a more sophisticated cell model, can probably be a good candidate for studying primary cilium mechanosensation in the future.

Bibliography

[1] Daniele Boffi, Franco Brezzi, and Michel Fortin. "Mixed finite element methods and applications". In: Springer Series in Computational Mathematics, 44. Springer, Heidelberg, 2013.

[2] Open Cascade. *SALOME. The Open Source Integration Platform for Numerical Simulation*. Version 7.0.0. 2016. URL: `www.salome-platform.org`.

[3] J. Donea et al. "Arbitrary Lagrangian-Eulerian Methods". In: *Encyclopedia of Computational Mechanics*. Ed. by Erwin Stein, René de Borst, and Thomas J.R. Hughes. Vol. 1. Chichester: John Wiley & Sons, Ltd., 2004.

[4] David Doyen, Alexandre Ern, and Serge Piperno. "A Semi-Explicit Modified Mass Method for Dynamic Frictionless Contact Problems". In: *Trends in Computational Contact Mechanics*. Vol. 58. Lecture Notes in Applied and Computational Mechanics. Springer-Verlag, 2011, pp. 157–168.

[5] K. Eriksson et al. *Computational Differential Equations*. Lund: Studentlitteratur, 1996.

[6] "FEniCS-HPC: Coupled Multiphysics in Computational Fluid Dynamics". In: Springer Lecture Notes in Computer Science, 2017.

[7] Christophe Geuzaine and Jean-Francois Remacle. "Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities". In: *International Journal for Numerical Methods in Engineering* 79.11 (2009), pp. 1309–1331.

[8] J. Hoffman, J. Jansson, and M. Stöckli. "Unified continuum modeling of fluid-structure interaction". In: *Mathematical Models and Methods in Applied Sciences* 21.3 (2011), pp. 491–513. DOI: `10.1142/S021820251100512X`.

[9] Johan Hoffman, Johan Jansson, and Niclas Jansson. "FEniCS-HPC: Automated Predictive High-Performance Finite Element Computing with Applications in Aerodynamics". In: *Parallel Processing & Applied Mathematics: 11th International Conference, PPAM 2015, Krakow, Poland, September 6-9, 2015*. Springer, 2015, pp. 356–365.

[10] J. Hoffman et al. "Unicorn: Parallel adaptive finite element simulation of turbulent flow and fluid-structure interaction for deforming domains and complex geometry". In: *Computers and Fluids* 80 (2013), pp. 310–319. DOI: `10.1016/j.compfluid.2012.02.003`.

[11] G. Houzeaux et al. "A massively parallel fractional step solver for incompressible flows". In: *Journal of Computational Physics* 228 (2009), pp. 6316–6332.

[12] H. Khayyeri, S. Barreto, and D. Lacroix. “Primary cilia machanics affects cell mechanosensation: A computational study”. In: *Theoretical Biology* (2015). DOI: 10.1016/j.tbi.2015.04.034.

[13] Anders Logg, Kent-Andre Mardal, and Garth N. Wells. *The FEniCS Book. Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[14] Beta CAE Systems. *ANSA*.

Appendix A

Theory details

A.1 ALE Method

In continuum mechanics, two classical kinematical descriptions of motion are used: the Lagrangian description and the Eulerian description. The arbitrary Lagrangian–Eulerian (ALE) description is the combination of these two algorithms.

The Lagrangian algorithm allocates grid points to the same material particle permanently, so that each grid point track the particles during motion. This algorithm is not able to follow large distortions of the computational domain without having a re-meshing operation. In the Eulerian algorithm, the computational mesh is fixed and the continuum moves with respect to the grid, the mesh is dissociated from the material particles. This algorithm can handle large distortions in the domain with expensive computations.

The ALE method brings the best features of these algorithms together. In this algorithm, the computational mesh may be moved or be held fixed in the Lagrangian and Eulerian fashion, respectively. The freedom of the computational mesh helps to deal with greater distortion in the domain.

For a detailed description of the ALE method, including the local ALE coordinate map, the reader is referred to [3].

A.2 The inf-sup condition

Consider the following saddle-point problem: Let $a : V \times V \to \mathbb{R}$ and $b : V \times Q \to \mathbb{R}$ be continuous bilinear forms, where V and Q are Hilbert spaces. Find $(u, p) \in V \times Q$ satisfying

$$a(u, v) + b(v, p) = (f, v)$$
$$b(u, q) = 0$$

for all $(v, q) \in V \times Q$, where $(\cdot, \cdot)$ denote the inner product in L_2.

There exists a $\beta > 0$ s.t. the bilinear form b satisfies the *inf-sup condition*

$$\inf_{q \in Q} \sup_{v \in V} \frac{b(v, q)}{\|v\|_V \|q\|_Q} \geq \beta \tag{A.1}$$

This is also known as the LBB (Ladyshenskaya-Babuska-Brezzi) condition.

Appendix B

Mesh generation details

B.1 Mesh generation in ANSA

The geometry and the mesh of the cell body and the primary cilium was constructed in the commercial software ANSA from BETA CAE Systems. The original mesh was used as input mesh to create a new mesh for the calculations in Unicorn.

The following steps was carried out in ANSA:

- Import the original mesh in VRML format (.wrl)
- Create a geometry out of the cilium (by using the command Elements > ToSurface)
- Wrap a coarser mesh around the cell body elements
- Create a geometry out of the coarse cell body mesh (with Elements > ToSurface)
- Add a rectangular plane as the bottom of the simulation box (Surfaces > Fit)
- Project the bottom edge of the cell body to the plane
- Replace the bottom of the cell with the new bottom in the plane
- Add five more planes to build up the simulation box (use Faces > Intersect to remove parts falling outside the box)
- Adjust the number of nodes for each Perimeter before generating a surface mesh
- Generate a surface mesh for each part of the geometry (Mesh Generation > CFD)
- Generate a volume mesh for all three regions: cilium, cell body and the fluid region (Volume > Define > Auto detect)
- Output the volume mesh to NASTRAN file format (.nas)
- Output one volume mesh for the entire domain (mesh.nas) and one volume mesh for the structure part only (structure.nas). Volume meshes for the different parts of the structure are also exported (cilium and cell body).

B.2 Mesh generation in SALOME

A simple model of the cell and the primary cilium was created in the open source software SALOME. The model is built up by basic geometry objects and has been generated by use of a Python script, loaded by SALOME. The script defines a geometry, creates a volume mesh and exports the mesh to XML file format. The script file can be found in Appendix C.1.

Appendix C

Code

C.1 Mesh generation in SALOME

generate_mesh_empty.py

```
# Generation of a 3D mesh in SALOME 6
# for fluid flow in an empty chamber
# ==================================

import os, inspect
import geompy, salome, smesh, SMESH, SALOMEDS
from MEDLoader import *
from MEDCoupling import *

# ===== THE GEOMETRY =====
xmin, xmax = -50.0, 50.0;
ymin, ymax = -50.0, 150.0;
zmin, zmax = 0.0, 50.0;

# To construct the geometry
def construct_geometry():
        print 'Constructing geometry'
        print '  xmin, xmax: ', xmin, ', ', xmax
        print '  ymin, ymax: ', ymin, ', ', ymax
        print '  zmin, zmax: ', zmin, ', ', zmax

        # the simulation box
        sim_box = geompy.MakeBox(xmin,ymin, zmin, xmax, ymax, zmax)
        geompy.addToStudy(sim_box, "simulation box")

        # define the fluid region and the computational domain
        Omega = sim_box
        geompy.addToStudy(Omega, "Omega")

        return Omega

Omega = construct_geometry()

# ===== THE MESH =====
fineness = 3
maxsize = 3.0
minsize = 1.0
growthrate = 0.2

# To construct the mesh of the total domain
def construct_total_mesh():
        print 'Constructing mesh'
        print '  fineness: ', fineness, ', maxsize: ', maxsize, ', minsize: ', minsize, ', growth rate: ',
              growthrate

        # initialize the mesh
        OmegaMesh = smesh.Mesh(Omega, "OmegaMesh")

        # define mesh algorithm
        NETGEN_1D2D3D = OmegaMesh.Tetrahedron(smesh.NETGEN_1D2D3D)
        NETGEN_3D_Parameters = NETGEN_1D2D3D.Parameters()
        NETGEN_3D_Parameters.SetSecondOrder( 0 )
        NETGEN_3D_Parameters.SetOptimize( 1 )
        NETGEN_3D_Parameters.SetFineness( fineness )
        NETGEN_3D_Parameters.SetMaxSize( maxsize )
        NETGEN_3D_Parameters.SetMinSize( minsize )
        NETGEN_3D_Parameters.SetGrowthRate( growthrate )
        #NETGEN_3D_Parameters.SetNbSegPerEdge( 19 )

        # compute the mesh
        OmegaMesh.Compute()
```

```
61
62          return OmegaMesh
63
64  OmegaMesh = construct_total_mesh()
65
66  # ===== FILE EXPORT =====
67
68  # script directory
69  directory = os.path.dirname(os.path.abspath(inspect.getfile(inspect.currentframe())))+'/'
70  subdir = directory + 'mesh_empty/'
71  if not (os.path.exists(subdir)):
72          print 'Creating subdirectory: ', subdir
73          os.mkdir(subdir)
74
75  # load script file
76  execfile(directory + 'unv2xml.py')
77
78  # To export a mesh
79  def export_mesh(theMesh, basefilename):
80          # export the mesh to *.unv and *.xml file formats
81          unvfilename = basefilename + '.unv'
82          theMesh.ExportUNV(subdir + unvfilename)
83          unv2xml(subdir, unvfilename)
84
85  # export the total mesh to UNV and XML file formats
86  print 'Exporting Omega mesh'
87  export_mesh(OmegaMesh, "mesh")
```

generate_mesh.py

```
1   # Generation of a 3D mesh in SALOME 6
2   # for fluid-structure interaction with
3   # a simple cell body model consisting of
4   # primary cilium, cell cortex and cytoplasm
5   # ========================================
6
7   import os, inspect
8   import geompy, salome, smesh, SMESH, SALOMEDS
9   from MEDLoader import *
10  from MEDCoupling import *
11
12  # ===== THE GEOMETRY =====
13  xmin, xmax = -50.0, 50.0;
14  ymin, ymax = -50.0, 150.0;
15  zmin, zmax = 0.0, 50.0;
16
17  # To construct the geometry
18  def construct_geometry():
19          # cell geometry parameters
20          radius_cell = 5.0
21          cortex_width = 0.4
22          radius_cilium = 0.2
23          height_cilium = 10.0
24          overlap_cilium = 0.5
25
26          print 'Constructing geometry'
27          print '  xmin, xmax: ', xmin, ', ', xmax
28          print '  ymin, ymax: ', ymin, ', ', ymax
29          print '  zmin, zmax: ', zmin, ', ', zmax
30          print '  radius of cell body: ', radius_cell
31          print '  cortex width: ', cortex_width
32          print '  radius of cilium: ', radius_cilium
33          print '  length of cilium: ', height_cilium
34
35          # the simulation box
36          sim_box = geompy.MakeBox(xmin,ymin, zmin, xmax, ymax, zmax)
37          geompy.addToStudy(sim_box, "simulation box")
38
39          # the cell body with cortex and cytoplasm
40          outer_shell = geompy.MakeSphere(0, 0, 0, radius_cell)
41          inner_shell = geompy.MakeSphere(0, 0, 0, radius_cell - cortex_width)
42
43          cytoplasm = geompy.MakeCommon(inner_shell, sim_box)
44          geompy.addToStudy(cytoplasm, "cytoplasm")
45
46          cortex = geompy.MakeCut(outer_shell, inner_shell)
47          cortex = geompy.MakeCommon(cortex, sim_box)
48          geompy.addToStudy(cortex, "cortex")
49
50          cell_body = geompy.MakeFuse(cytoplasm, cortex)
51          geompy.addToStudy(cell_body, "cell body")
52
53          # split cell_body into solids
54          cell_body_solids = geompy.SubShapeAll(cell_body, geompy.ShapeType["SOLID"])
55          for solid in cell_body_solids:
56                  name = geompy.SubShapeName(solid, cell_body)
57                  geompy.addToStudyInFather(cell_body, solid, name)
58
59          # the primary cilium on top of cell body
60          p1 = geompy.MakeVertex(0.0, 0.0, radius_cell - overlap_cilium)
61          p2 = geompy.MakeVertex(0.0, 0.0, radius_cell + height_cilium)
62          vc = geompy.MakeVector(p1, p2)
63          cilium = geompy.MakeCylinder(p1, vc, radius_cilium, overlap_cilium + height_cilium)
64          cilium = geompy.MakeCut(cilium, outer_shell)
65          geompy.addToStudy(cilium, "cilium")
66
```

```
67          # define the structure
68          structure = geompy.MakeFuse(cell_body, cilium)
69          geompy.addToStudy(structure, "structure")
70
71          # define the fluid region and the computational domain
72          fluid_region = geompy.MakeCut(sim_box, structure)
73          Omega = geompy.MakePartition([fluid_region, cilium, cortex, cytoplasm], \
74                  [],[],[], geompy.ShapeType["SOLID"])
75          geompy.addToStudy(fluid_region, "fluid region")
76          geompy.addToStudy(Omega, "Omega")
77
78          return Omega, structure, cell_body, cilium, cortex, cytoplasm
79
80  Omega, structure, cell_body, cilium, cortex, cytoplasm = construct_geometry()
81
82  # ===== THE MESH =====
83  fineness = 3
84  maxsize = 3.0
85  minsize = 0.1
86  growthrate = 0.2
87
88  # To construct the mesh of the total domain
89  def construct_total_mesh():
90          print 'Constructing mesh'
91          print '  fineness: ', fineness, ', maxsize: ', maxsize, ', minsize: ', minsize, ', growth rate: ',
                growthrate
92
93          # initialize the mesh
94          OmegaMesh = smesh.Mesh(Omega, "OmegaMesh")
95
96          # define mesh algorithm
97          NETGEN_1D2D3D = OmegaMesh.Tetrahedron(smesh.NETGEN_1D2D3D)
98          NETGEN_3D_Parameters = NETGEN_1D2D3D.Parameters()
99          NETGEN_3D_Parameters.SetSecondOrder( 0 )
100         NETGEN_3D_Parameters.SetOptimize( 1 )
101         NETGEN_3D_Parameters.SetFineness( fineness )
102         NETGEN_3D_Parameters.SetMaxSize( maxsize )
103         NETGEN_3D_Parameters.SetMinSize( minsize )
104         NETGEN_3D_Parameters.SetGrowthRate( growthrate )
105
106         # compute the mesh
107         OmegaMesh.Compute()
108
109         return OmegaMesh
110
111 OmegaMesh = construct_total_mesh()
112
113 # ===== FILE EXPORT =====
114
115 # script directory
116 directory = os.path.dirname(os.path.abspath(inspect.getfile(inspect.currentframe())))+'/'
117 subdir = 'mesh_cortex/'
118 if not (os.path.exists(directory + subdir)):
119         print 'Creating subdirectory: ', subdir
120         os.mkdir(directory + subdir)
121
122 # load script file
123 execfile(directory + 'unv2xml.py')
124
125 # To export a mesh
126 def export_mesh(theMesh, basefilename):
127         # export the mesh to *.unv and *.xml file formats
128         unvfilename = basefilename + '.unv'
129         theMesh.ExportUNV(directory + subdir + unvfilename)
130         unv2xml(directory + subdir, unvfilename)
131
132 # export the total mesh to UNV and XML file formats
133 print 'Exporting Omega mesh'
134 export_mesh(OmegaMesh, "mesh")
135
136 # CopyMesh parameters
137 toCopyGroups = True
138 toKeepIDs = True
139
140 # find the structure part in the mesh and create a structure mesh
141 sub_structure = geompy.GetInPlace(Omega, structure, "structure")        # extract structure
142 gr_structure = OmegaMesh.Group(sub_structure, "gr_structure")           # group structure
143 StructureMesh = smesh.CopyMesh(gr_structure, "structure", toCopyGroups, toKeepIDs)
144 print 'Exporting structure'
145 export_mesh(StructureMesh, "structure")                                                    # export structure
        mesh
146
147 # find the cell body in the mesh and create a structure mesh for the cell body
148 sub_cellbody = geompy.GetInPlace(Omega, cell_body, "cellbody")
149 gr_cellbody = OmegaMesh.Group(sub_cellbody)
150 CellbodyMesh = smesh.CopyMesh(gr_cellbody, "structure_cellbody", toCopyGroups, toKeepIDs)
151 print 'Exporting cell body structure'
152 export_mesh(CellbodyMesh, "structure_cellbody")
153
154 # find the cilium in the mesh and create a cilium mesh
155 sub_cilium = geompy.GetInPlace(Omega, cilium, "cilium")
156 gr_cilium = OmegaMesh.Group(sub_cilium)
157 CiliumMesh = smesh.CopyMesh(gr_cilium, "solid_1", toCopyGroups, toKeepIDs)
158 print 'Exporting cilium'
159 export_mesh(CiliumMesh, "solid_1")
160
161 # find the cortex in the mesh and create a cortex mesh
162 sub_cortex = geompy.GetInPlace(Omega, cortex, "cortex")
163 gr_cortex = OmegaMesh.Group(sub_cortex)
```

```
CortexMesh = smesh.CopyMesh(gr_cortex, "solid_2", toCopyGroups, toKeepIDs)
print 'Exporting cortex'
export_mesh(CortexMesh, "solid_2")

# find the cytoplasm in the mesh and create a cytoplasm mesh
sub_cytoplasm = geompy.GetInPlace(Omega, cytoplasm, "cytoplasm")
gr_cytoplasm = OmegaMesh.Group(sub_cytoplasm)
CytoplasmMesh = smesh.CopyMesh(gr_cytoplasm, "solid_3", toCopyGroups, toKeepIDs)
print 'Exporting cytoplasm'
export_mesh(CytoplasmMesh, "solid_3")
```

unv2xml.py (converting from UNV to XML)

```
# Written by Dang Van NGUYEN 2013
# Edited by Jakob Beran 2016
import os, inspect
import salome, smesh

# -*- coding: utf-8 -*-
from MEDLoader import *
from MEDCoupling import *

def unv2xml(directory, unvfilename):
        unvFile = directory + unvfilename
        unvmesh = smesh.CreateMeshesFromUNV(unvFile)    # creates a new mesh in study

        filewithoutextension = os.path.splitext(os.path.basename(unvFile))[0]

        medFile = directory + filewithoutextension + '.med'
        unvmesh.ExportMED(medFile)

        medmesh = MEDFileMesh.New(medFile)
        numvertices = medmesh.getNumberOfNodes()
        dim = medmesh.getMeshDimension()

        if (dim==2):
                numcells = unvmesh.NbTriangles()
        if (dim==3):
                numcells = unvmesh.NbVolumes()

        print "Mesh dimension: ", dim
        print "Num of vertices: ", numvertices
        print "Num of cells: ", numcells

        os.system('rm '+medFile)        # remove temporary file

        # Export .xml mesh file
        # ------------------------------------
        filename = open(directory+filewithoutextension+'.xml','w')
        filename.write('<?xml version="1.0" encoding="UTF-8"?>\n')
        filename.write('<dolfin xmlns:dolfin="http://fenicsproject.org">\n')
        if (dim==2):
                filename.write('  <mesh celltype="triangle" dim="2">\n')
        if (dim==3):
                filename.write('  <mesh celltype="tetrahedron" dim="3">\n')

        # save mesh vertices
        filename.write('    <vertices size="'+str(numvertices)+'">\n')
        for node in xrange(1,numvertices+1):
                xyz = unvmesh.GetNodeXYZ( node );
                if (dim==2):
                        filename.write('        <vertex index="'+str(node-1)+'" x="%.16e" y="%.16e" />\n' %
                                (xyz[0],xyz[1]))
                if (dim==3):
                        filename.write('        <vertex index="'+str(node-1)+'" x="%.16e" y="%.16e" z="%.16e
                                " />\n' % (xyz[0],xyz[1],xyz[2]))
        filename.write('    </vertices>\n')

        # save mesh cells
        filename.write('    <cells size="'+str(numcells)+'">\n')
        if (dim==2):
                vol = unvmesh.GetElementsByType(SMESH.FACE)
        if (dim==3):
                vol = unvmesh.GetElementsByType(SMESH.VOLUME)
        # print vol
        cellindex = 0
        for ind in vol:
                nodesid = unvmesh.GetElemNodes(ind)
                if (dim==2):
                        filename.write('        <triangle index="%d" v0="%d" v1="%d" v2="%d" />\n' % (
                                cellindex,nodesid[0]-1,nodesid[1]-1,nodesid[2]-1) )
                if (dim==3):
                        filename.write('        <tetrahedron index="'+str(cellindex)+'" v0="'+str(nodesid
                                [0]-1)+'" v1="'+str(nodesid[2]-1)+'" v2="'+str(nodesid[1]-1)+'" v3="'+str(
                                nodesid[3]-1)+'" />\n')
                cellindex = cellindex + 1

        filename.write('    </cells>\n')
        filename.write('  </mesh>\n')
        filename.write('</dolfin>\n')
        filename.close()
        # ------------------------------------

        print("Mesh converted from UNV to XML file format:")
        print('    ' + directory + filewithoutextension + '.xml')
```

C.2 Shell scripts

Converting 3D mesh from NASTRAN to XML

A shell function for converting a 3D mesh created in ANSA from NASTRAN to XML file format with use of the software `gmsh` [7]:

```
# To convert a 3D NASTRAN mesh file to *.xml format
function convert_3D_nas2xml() {
        echo "*** Converting 3D NASTRAN mesh file '$1.nas' to XML format ***"
        gmsh -3 $1.nas -o $1.msh
        dolfin-convert $1.msh $1.xml
}
```

. . . and the script for converting all the meshes used in the model:

```
convert_3D_nas2xml mesh > mesh.log
convert_3D_nas2xml structure > structure.log
convert_3D_nas2xml cilia > cilia.log
convert_3D_nas2xml cellbody > cellbody.log
```

Transfer files to Beskow

A script to transfer the edited C++ files from local computer to Beskow:

```
if [ $# -lt 1 ]; then
        echo "Usage: . transfer-to-beskow <local-directory-at-beskow>";
        echo "Local directory at Beskow must exist."
        return
fi
echo "Transfer main file to Beskow in folder: $1"
scp main.cpp jberan@beskow.pdc.kth.se:/cfs/klemming/nobackup/j/jberan/fenics-hpc_eunison_dist/Jakob/$1
echo "Updating solver files"
scp NSESolver.h jberan@beskow.pdc.kth.se:/cfs/klemming/nobackup/j/jberan/fenics-hpc_eunison_dist/Jakob/
    unicorn/
scp NSESolver.cpp jberan@beskow.pdc.kth.se:/cfs/klemming/nobackup/j/jberan/fenics-hpc_eunison_dist/Jakob/
echo "Copying solver files to Beskow in folder: $1"
scp NSESolver.h jberan@beskow.pdc.kth.se:/cfs/klemming/nobackup/j/jberan/fenics-hpc_eunison_dist/Jakob/$1
scp NSESolver.cpp jberan@beskow.pdc.kth.se:/cfs/klemming/nobackup/j/jberan/fenics-hpc_eunison_dist/Jakob/$1
```

Job script for Beskow

Example of a job script for submitting a FEniCS-HPC/Unicorn computation to Beskow at PDC:

```
#!/bin/bash -l
# The -l above is required to get the full environment with modules

# The name of the script
#SBATCH -J unicorn-cilia

# Set the time that will be given to this job
#SBATCH -t 6:00:00

# Number of nodes
#SBATCH -N 8
# Number of MPI processes per node (the following is actually the default)
#SBATCH --ntasks-per-node=32
# Number of MPI processes
#SBATCH -n 256

#SBATCH -e error_file.e
#SBATCH -o output_file.o

# Time allocation account (CAC)
#SBATCH -A 2016-10-60

# Run the executable and write the output into log files
aprun -n 256 -ss -j1 ./unicorn -m mesh.xml -s structure.xml -p parameters 1> log1 2> log2
#aprun -n 256 -ss -j1 ./unicorn -m mesh.xml -s structure_cellbody.xml -p parameters 1> log1 2> log2
#aprun -n 256 -ss -j1 ./unicorn -m mesh.xml -s nostructure.xml -p parameters 1> log1 2> log2
```

Post processing of sampled data

A shell script function `post_process` has been used to convert sampled data files in binary format to VTK format, which can be opened with Paraview. The shell function uses the post processing program `dolfin_post` that comes with FEniCS-HPC.

```
function post_process () {
        cd iter_0
        declare NUMBER_OF_FILES=$(ls solution*.bin | grep --count 'solution')
        echo "Number of files to process: $NUMBER_OF_FILES"
        ../../dolfin_post -M mesh -t vtk -s solution -n $NUMBER_OF_FILES
        ../../dolfin_post -M mesh -t vtk -s solidtype -n $NUMBER_OF_FILES
        ../../dolfin_post -M mesh -t vtk -s stressVM -n $NUMBER_OF_FILES
        cd ..
}
```

Transfer files from Beskow

A script file to transfer output files from Beskow to the local computer:

```
if [ $# -lt 1 ]; then
        echo "Usage: . transfer-solution.script <local-directory-on-beskow>";
        echo "A new directory will be created and solution files will be";
        echo "copied from local directory on Beskow to this new directory.";
        return
fi
echo "Creating a new directory: $1"
mkdir $1
cd $1
echo "Transfering files from Beskow in folder: $1"
scp jberan@beskow.pdc.kth.se:/cfs/klemming/nobackup/j/jberan/fenics-hpc_eunison_dist/Jakob/$1/iter_0/{*.pvd
      ,*.vtu,*.dat} .
scp jberan@beskow.pdc.kth.se:/cfs/klemming/nobackup/j/jberan/fenics-hpc_eunison_dist/Jakob/$1/{main.cpp,log1
      ,log2,parameters,solid_parameters,job.script} .
scp jberan@beskow.pdc.kth.se:/cfs/klemming/nobackup/j/jberan/fenics-hpc_eunison_dist/Jakob/$1/NSESolver{.cpp
      ,.h} .
scp jberan@beskow.pdc.kth.se:/cfs/klemming/nobackup/j/jberan/fenics-hpc_eunison_dist/Jakob/unicorn/{
      NSEMomentum3D,NSEContinuity3D,MassFlow,VonMisesStress}.form .
cd ..
```

Clone script on Beskow

A script for copying all essential files for a computation to a new folder on Beskow. This script is used to quickly create a new computation with a modified code or with different parameter values.

```
if [ $# -lt 1 ]; then
        echo "Usage: . clone.script <new-directory>";
        echo "clone.script: All essential files in this directory are copied to <new-directory>.";
        return
fi
echo "Creating a new test directory: $1"
mkdir ../$1
echo "Copying files to new directory"
cp -i {main.cpp,Makefile,*.xml,parameters,solid_parameters*,job.script,post_process.script,clone.script} ../
      $1
echo "Entering new directory"
cd ../$1
ls -l
```

FFC compilation at Hydra

The following script is run prior to FFC compilation at Hydra in order to add the correct modules:

```
# Important: login with the KTH.SE domain to hydra.csc.kth.se
export MODULEPATH=/afs/nada.kth.se/dept/na/ctl/pkg/@sys/modulefiles
module avail
module add dolfin-hpc
module add fiat
module add ffc
module add ufc
module add check
module list
echo "Above listed modules have been added."
echo "Now compile form file with:   ffc -l dolfin -f split_implementation FormFileName.form"
```

C.3 FEniCS-HPC/Unicorn form files

NSEMomentum3D.form

```
# Copyright (c) 2005 Johan Jansson (johanjan@math.chalmers.se)
# Licensed under the GNU GPL Version 2
#
# First added:  2005
# Last changed: 2016-12-08
#
# The bilinear form for the incompressible Navier-Stokes equations
#
# For low Reynolds number flow:  use alpha = Re = rho_f/mu_f  &  beta = 1
# For high Reynolds number flow: use alpha = 1  &  beta = Re^{-1} = mu_f/rho_f
#
# Compile this form with FFC: ffc NSEMomentum3D.form.


cell = "tetrahedron"

K1 = VectorElement("Lagrange", cell, 1)
# Dimension of domain
d = K1.cell_dimension()
K2 = FiniteElement("Lagrange", cell, 1)
K3 = FiniteElement("Discontinuous Lagrange", cell, 0)
K4 = VectorElement("Discontinuous Lagrange", cell, 0)
K5 = VectorElement("Discontinuous Lagrange", cell, 0, d * d)    # for sigma

K = K1

v = TestFunction(K)
U1 = TrialFunction(K)   # the unknown velocity solution (U_{n})
UP = Function(K)                # velocity solution in previous time step (U_{n-1})
U0 = Function(K)                # velocity solution in previous iteration (U0 converges to U1)
Uc = Function(K)                # mean velocity
Um = Function(K4)               # approximation of Uc as a piecewise constant projection of Uc

P = Function(K2)                # pressure
alpha = Function(K3)    # Reynolds number (parameter used for low Re flow with beta = 1)
beta = Function(K3)             # inverse Reynolds number (parameter used for high Re flow with alpha = 1)
d1 = Function(K3)               # stabilization parameter
d2 = Function(K3)               # stabilization parameter
ff = Function(K)                # external force
k = Function(K3)                # time step

theta = Function(K3)    # phase function
sigma = Function(K5)    # stress tensor

mu_s = Function(K3)             # solid stiffness
lmbda = Function(K3)    # additional viscosity parameter

WP = Function(K)                # mesh velocity
Wm = Function(K4)               # piecewise constant projection of WP

n = [0.0, 1.0, 0.0]             # normal direction of flow at outlet
gamma = Function(K3)    # outlet boundary marker

def tomatrix(q):
    return [ [q[d * i + j] for i in range(d)] for j in range(d) ]

sigmaM = tomatrix(sigma)

def ugradu(u, v):
    return [dot(u, grad(v[i])) for i in range(d)]

def E(e, mu_s, lmbda):
    Ee = 2.0 * mult(mu_s, e) + mult(lmbda, mult(trace(e), Identity(d)))
    return Ee

def epsilon(u):
    return 0.5 * (grad(u) + transp(grad(u)))

Sf = mult(2*beta, epsilon(Uc)) - mult(P, Identity(d))   # Sigma_f
Ss = mult(1.0, sigmaM) - mult(P, Identity(d))                   # Sigma_s
S  = mult(theta, Sf) + mult(1.0 - theta, Ss)                    # Sigma

Sfn = [dot(Sf[i], n) for i in range(d)]         # stress at outlet boundary

Uc_ALE = Uc - WP                # mean ALE velocity
Um_ALE = Um - Wm                # piecewise constant projection of Uc_ALE

def f(u, v):
    return dot(mult(alpha, ugradu(Uc_ALE, u)), v) - dot(ff, v) \
        + dot(S, grad(v)) \
                + mult(d1, dot(mult(alpha, ugradu(Um_ALE, u)), \
                                                mult(alpha, ugradu(Um_ALE, v)))) \
                + mult(d2, dot(div(u), div(v)))

def dfdu(u, k, v):
    return dot(mult(alpha, ugradu(Uc_ALE, u)), v) \
        + mult(theta, mult(2*beta, dot(epsilon(u), epsilon(v)))) + \
        + mult(1 - theta, mult(k, dot(E(epsilon(u), mu_s, lmbda), grad(v)))) \
        + mult(d1, dot(mult(alpha, ugradu(Um_ALE, u)), \
                                                mult(alpha, ugradu(Um_ALE, v)))) \
        + mult(d2, dot(div(u), div(v)))

```

```
# cG(1)
def F(u, u0, k, v):
    uc = mult(0.5, u + u0)
    return (dot(mult(alpha, u), v) - dot(mult(alpha, u0), v) + mult(k, f(uc, v)))

# the Jacobian matrix acting on u
def dFdu(u, u0, k, v):
    uc = mult(0.5, u)
    return (dot(mult(alpha, u), v) + mult(k, dfdu(uc, k, v)))

# Newton's method J U_{i+1} = J U_{i} - F(U_{i})
a = (dFdu(U1, U0, k, v)) * dx
L = (dFdu(UP, U0, k, v) - F(UP, U0, k, v))*dx + mult(gamma, mult(k, dot(Sfn, v)))*ds
```

NSEContinuity3D.form

```
# Copyright (c) 2005 Johan Hoffman
# Licensed under the GNU GPL Version 2
#
# Modified by Anders Logg 2006
#
# First added:  2005
# Last changed: 2006-03-28
#
# The contiuity equation for the incompressible
# Navier-Stokes equations using cG(1)cG(1)
#
# Compile this form with FFC: ffc NSEContinuity2D.form.

cell = "tetrahedron"

scalar = FiniteElement("Lagrange", cell, 1)
# Dimension of domain
d = scalar.cell_dimension()
vector = VectorElement("Lagrange", cell, 1)
constant_scalar = FiniteElement("Discontinuous Lagrange", cell, 0)

q  = TestFunction(scalar)   # test basis function
P  = TrialFunction(scalar)  # trial basis function
P0 = Function(scalar)       # trial basis function
uc = Function(vector)       # linearized velocity

delta1 = Function(constant_scalar) # stabilization parameter
k = Function(constant_scalar)

alpha = Function(constant_scalar)
f = Function(vector)        # external force
#fc = Function(vector)
fs = Function(scalar)       # pressure force
#n = Function(vector)
n = FacetNormal(cell)

i0 = Index() # index for tensor notation
i1 = Index() # index for tensor notation

def ugradu(u, v):
    return [dot(u, grad(v[i])) for i in range(d)]

a = ( delta1*dot(grad(P), grad(q)) + mult(2*k, dot(grad(P), grad(q))) )*dx
L = ( -div(uc)*q + mult(2*k, dot(grad(P0), grad(q))) )*dx \
    - mult(delta1, dot(mult(alpha, ugradu(uc, uc)), grad(q)))*dx \
    + mult(delta1, dot(mult(1.0, f), grad(q)))*dx \
    + fs*q*dx
```

AuxiliaryScalar.form

```
name = "AuxiliaryScalar"
scalar = FiniteElement("Lagrange", "tetrahedron", 1)

v = TestFunction(scalar)
u = Function(scalar)

L = v*u * dx
```

MassFlow.form

```
# To calculate the integrated mass flow through a boundary

cell = "tetrahedron"

K1 = VectorElement("Lagrange", cell, 1)
K2 = FiniteElement("Lagrange", cell, 1)
K3 = FiniteElement("Discontinuous Lagrange", cell, 0)
K4 = VectorElement("Discontinuous Lagrange", cell, 0)

u = Function(K1)                # velocity
n = [0.0, 1.0, 0.0]             # normal direction of flow

M = dot(u, n) * ds              # functional
```

VonMisesStress.form

```
# To calculate the von Mises Stress from the stress tensor S

cell = "tetrahedron"

K1 = VectorElement("Lagrange", cell, 1)
# Dimension of domain
d = K1.cell_dimension()
K2 = FiniteElement("Lagrange", cell, 1)
K3 = FiniteElement("Discontinuous Lagrange", cell, 0)
K4 = VectorElement("Discontinuous Lagrange", cell, 0)
K5 = VectorElement("Discontinuous Lagrange", cell, 0, d * d)

K = K3

v = TestFunction(K)
u = TrialFunction(K)
S = Function(K5)                # deviatoric stress tensor
icv = Function(K)       # inverse cell volume

def vonMises(S):
    sum = 0
    trace = 0
    for i in range(d):
        for j in range(d):
            if i==j:
                trace = S[0]+S[4]+S[8]
                sum += (S[d*i + j] - (1/3)*trace) * (S[d*i + j] - (1/3)*trace)
            else:
                sum += (S[d*i + j])*(S[d*i + j])
    return sum

a = u * v * dx
L = icv * vonMises(S) * v * dx
```

NavierStokesStress3D.form

```
# Copyright (c) 2005 Johan Jansson (johanjan@math.chalmers.se)
# Licensed under the GNU GPL Version 2
#
# First added:  2005
# Last changed: 2006-03-28
#
# The bilinear form for classical linear elasticity (Navier)
# Compile this form with FFC: ffc Elasticity.form.

cell = "tetrahedron"

K1 = VectorElement("Lagrange", cell, 1)

# Dimension of domain
d = K1.cell_dimension()

K2 = VectorElement("Lagrange", cell, 1)
K3 = VectorElement("Discontinuous Lagrange", cell, 0, d * d)
K4 = FiniteElement("Discontinuous Lagrange", cell, 0)

K = K3

v = TestFunction(K)
dotS = TrialFunction(K)
S = Function(K)
U = Function(K2)

mu = Function(K4)
lmbda = Function(K4)

icv = Function(K4)

def tomatrix(q):
    return [ [q[d * i + j] for i in range(d)] for j in range(d) ]

def epsilon(u):
    return 0.5 * (grad(u) + transp(grad(u)))

def E(e, mu, lmbda):
    Ee = 2.0 * mult(mu, e)
    return Ee

Sm = tomatrix(S)
vm = tomatrix(v)

eps = epsilon(U)
Eeps = E(eps, mu, lmbda)
#Dobj = mult(1.0, mult(Sm, grad(U)) + mult(transp(grad(U)), Sm))
Dobj = mult(1.0, mult(grad(U), Sm) + mult(Sm, transp(grad(U))))

a = (dot(dotS, v)) * dx
L = (mult(icv, dot(Eeps + Dobj, vm))) * dx
```

C.4 FEniCS-HPC/Unicorn C++ files

`main.cpp` (essential parts)

```
42  // To resize the mesh by a constant factor
43  void resize_mesh(Mesh& mesh, const float factor){
44    for (VertexIterator v(mesh); !v.end(); ++v){
45      v->x()[0]=v->x()[0]*factor;
46      v->x()[1]=v->x()[1]*factor;
47      v->x()[2]=v->x()[2]*factor;
48    }
49  }
50
51  // dimensions of the box
52  real xmin = -5.0;
53  real xmax = 5.0;
54  real ymin = -5.0;
55  real ymax = 15.0;
56  real zmin = 0.0;
57  real zmax = 5.0;


69  // Sub domain for side walls
70  class SideWallBoundary : public SubDomain
71  {
72  public:
73    bool inside(const real* x, bool on_boundary) const
74    {
75      return on_boundary && ( (x[1] > ymin + bmarg && x[1] < ymax - bmarg) ||
76                              (x[1] < ymin + bmarg &&
77                               (x[2] < zmin + bmarg || x[2] > zmax - bmarg ||
78                                x[0] < xmin + bmarg || x[0] > xmax - bmarg)) ||
79                              (x[1] > ymax - bmarg &&
80                               (x[2] < zmin + bmarg || x[2] > zmax - bmarg ||
81                                x[0] < xmin + bmarg || x[0] > xmax - bmarg)) );
82    }
83  };
84
85  // Sub domain for the inflow
86  class InflowBoundary : public SubDomain
87  {
88  public:
89    bool inside(const real* x, bool on_boundary) const
90    {
91      return on_boundary && ( x[1] < ymin + bmarg &&
92                              (x[2] > zmin + bmarg && x[2] < zmax - bmarg &&
93                               x[0] > xmin + bmarg && x[0] < xmax - bmarg) );
94    }
95  };
96
97  // Sub domain for the outflow
98  class OutflowBoundary : public SubDomain
99  {
100 public:
101   bool inside(const real* x, bool on_boundary) const
102   {
103     return on_boundary && ( x[1] > ymax - bmarg &&
104                             (x[2] > zmin + bmarg && x[2] < zmax - bmarg &&
105                              x[0] > xmin + bmarg && x[0] < xmax - bmarg) );
106   }
107 };
108
109 // No-slip boundaries (side walls)
110 class NoSlipBoundary : public SubDomain
111 {
112   bool inside(const real* x, bool on_boundary) const
113   {
114     SideWallBoundary swb;
115     return on_boundary && swb.inside(x, on_boundary);
116   }
117 };


151 // Boundary condition for momentum equation
152 class BC_Momentum_3D : public Function
153 {
154 public:
155   BC_Momentum_3D(Mesh& mesh, TimeDependent& td) : Function(mesh), td(td) {}
156   void eval(real* values, const real* x) const
157   {
158     real t = td.time();
159     real ramp = 0.0;
160     real tramp = 1.0e-4;
161     real Um = 1.0;        // maximum velocity of inlet profile
162
163     if(t < tramp)
164     {
165       ramp = (t - 0.0) / (tramp - 0.0);
166     }
167     else if(t >= tramp)
168     {
169       ramp = 1.0;
170     }
171
172     /* inlet velocity profile as a rectangular parabola along the y-axis */
```

```
    values[0] = 0;
    values[1] = ramp*16*Um*( (x[0] - xmax)*(x[0] - xmin)/((xmax-xmin)*(xmax-xmin)) *
                             (x[2] - zmax)*(x[2] - zmin)/((zmax-zmin)*(zmax-zmin)) ) ;
    values[2] = 0;
  }

  TimeDependent& td;

  uint rank() const {return 1;}

  uint dim(int i) const {int d = mesh().topology().dim(); return d;}
};

// Boundary condition for continuity equation
class BC_Continuity : public Function
{
public:
  BC_Continuity(Mesh& mesh, TimeDependent& td) : Function(mesh), td(td) {}

  void eval(real* values, const real* x) const
  {
    values[0] = 0.0;  // scalar valued function
  }

  TimeDependent& td;
  uint rank() const {return 0;}
  uint dim(int i) const {return 1;}
};


// Viscosity function
class Epsilon : public Function
{
public:
  Epsilon(Mesh& mesh, real val) : Function(mesh), val(val) {}
  void eval(real* values, const real* x) const
  {
    real epsilon = val;

    values[0] = epsilon;
  }

  uint rank() const {return 0;}
  uint dim(int i) const {return 1;}
  real val;
};


/* To detect the structure cells in the mesh (or any part of the mesh) from a structure mesh file */
void detect_structure_mesh(Mesh& mesh, Mesh*& structure_mesh, MeshFunction<bool>& structure_cells) {

  if(dolfin::MPI::processNumber() == 0)
    dolfin_set("output destination", "terminal");
  else
    dolfin_set("output destination", "silent");

  bool existRegionFile = structure_mesh;

  IntersectionDetector *idetector;        // class defined in dolfin-hpc/mesh

  if (existRegionFile) {
    cout << "Region file exists" << endl;
    idetector = new IntersectionDetector(*structure_mesh);
  }
  else {
    cout << "Region file doesn't exist" << endl;
  }

  if (existRegionFile) {
    for (CellIterator c(mesh); !c.end(); ++c) {
      Cell& cell = *c;
      Point mp = cell.midpoint();

      Array<unsigned int> overlap_cells;      // will contain the indices of overlapping cells
      overlap_cells.clear();
      idetector->overlap(mp, overlap_cells);  // gives a suggestion of possibly overlapping cells

      bool bfnd = false;
      for(int i=0; i < overlap_cells.size(); i++) {
        Cell testcell(*structure_mesh, overlap_cells[i]);     // a test cell in the structure mesh
        if (structure_mesh->type().intersects(testcell,mp)) { // check if testcell and cell are intersecting
          bfnd = true;      // structure cell found
          break;
        }
      }
      structure_cells.set(cell, bfnd);
    }
    cout << "Structure cells in mesh detected" << endl;
  }
  delete idetector;
}

/* To read solid stiffness values from file and save in array */
void read_solid_stiffness(Array<real>& mu, const int n_solids) {
  if (n_solids > 0) {
    /* read mu values from file */
    FILE* input_file = std::fopen("../solid_parameters", "r");
    for (int i=0; i < n_solids; i++) {
      float val;
```

```
310         std::fscanf(input_file, "%f %*s\n", &val);
311         mu[i+1] = (real) val;
312       }
313       std::fclose(input_file);
314
315       /* print out the mu values */
316       if(dolfin::MPI::processNumber() == 0) {
317         for (int i=0; i < mu.size(); i++) {
318           std::cout << "Solid stiffness mu[" << i << "] = " << mu[i] << std::endl;
319         }
320       }
321     }
322     else {
323       if(dolfin::MPI::processNumber() == 0) {
324         std::cout << "Only one type of solid structure is used" << std::endl;
325         std::cout << "Default stiffness of solid is " << mu[0] << std::endl;
326       }
327     }
328   }
329
330   /* To assign different solid type values to different regions of the mesh */
331   /* Look for multiple solid types (check for solid files: "solid_1.xml" etc.) */
332   void set_solid_type(Mesh& mesh, MeshFunction<unsigned int>& solid_type, const int n_solids) {
333     for (int i=0; i < n_solids; i++) {
334       int val = i+1;
335       char filename[20];
336       std::sprintf(filename, "../solid_%d.xml", val);
337       float scale_factor = 0.1;
338
339       if(dolfin::MPI::processNumber() == 0)
340         dolfin_set("output destination","terminal");
341       else
342         dolfin_set("output destination","silent");
343
344       dolfin_set("Mesh read in serial", true);
345       Mesh* solid_submesh = new Mesh(filename);
346       resize_mesh(*solid_submesh, scale_factor);    // resize solid mesh
347       message("Solid mesh %d resized", val);
348       dolfin_set("Mesh read in serial", false);
349
350       message("Number of vertices in solid %d: %d", val, solid_submesh->numVertices());
351       message("Number of cells in solid %d: %d", val, solid_submesh->numCells());
352
353       /* Detect the solid cells in the mesh file */
354       MeshFunction<bool> solid_cells_submesh(mesh, mesh.topology().dim());
355       detect_structure_mesh(mesh, solid_submesh, solid_cells_submesh);
356
357       /* Assign solid type ID value to solid cells */
358       if (solid_submesh) {
359         for (CellIterator c(mesh); !c.end(); ++c) {
360           if (solid_cells_submesh.get(*c))
361             solid_type.set(c->index(), val);
362         }
363       }
364
365       delete solid_submesh;
366     }
367   }
368
369   /* To detect the solid vertices in the mesh (for mesh smoothing) */
370   void detect_solid_vertices(Mesh& mesh, Mesh*& structure_mesh,
371                              MeshFunction<bool>& solid_cells,
372                              MeshFunction<bool>& solid_vertices) {
373
374     /* define a vertex function and a functional form */
375     Function solidvertices;
376     Form *fform = new AuxiliaryScalarLinearForm(solidvertices);
377     Vector sv_vector;
378     solidvertices.init(mesh, sv_vector, *fform, 0);
379     solidvertices.sync_ghosts();
380
381     real value = 0.0;
382     int number = 1;
383     uint gi;
384
385     /* fill vertex function with values */
386     for (CellIterator c(mesh); !c.end(); ++c) {
387       Cell& cell = *c;
388       if (solid_cells.get(c->index())) {
389         for (VertexIterator v(cell); !v.end(); ++v) {
390           gi = mesh.distdata().get_global(v->index(), 0);   // global index of vertex
391           value = 1.0;
392           solidvertices.vector().add(&value, number, &gi);  // add value to vertex function
393         }
394       }
395     }
396     solidvertices.vector().apply();
397     solidvertices.sync_ghosts();          // synchronise data between MPI processes
398
399     /* read function value and mark the solid vertices in mesh function */
400     for (VertexIterator v(mesh); !v.end(); ++v) {
401       gi = mesh.distdata().get_global(v->index(), 0);
402       solidvertices.vector().get(&value, number, &gi);  // get value of function at global index
403       if (value > 0.0) {
404         solid_vertices.set(v->index(), true);           // mark the solid vertex in mesh function
405       }
406     }
407
408     delete fform;
```

```
}

/* The solver */
void solve(Mesh& mesh, Checkpoint& chkp, long& w_limit, timeval& s_time, Mesh* structure_mesh)
{
  real T = dolfin_get("T");
  real nu = dolfin_get("nu");
  real ubar = dolfin_get("Ubar");
  real mu_0 = dolfin_get("mu");       // default value of stiffness for single solid
  real nu_s = dolfin_get("nu_s");
  real rho_s = dolfin_get("rho_s");

  dolfin_set("solution file name", "solution.bin");

  TimeDependent td;

  /* Mesh functions for marking solid cells, solid vertices and solid types */
  MeshFunction<bool> solid_cells(mesh, mesh.topology().dim());
  solid_cells = false;
  MeshFunction<bool> solid_vertices(mesh, 0);
  solid_vertices = false;
  MeshFunction<unsigned int> solid_type(mesh, mesh.topology().dim());
  solid_type = 0;               // default solid type

  const int n_solids = 3;       // number of different solids
  Array<real> mu(n_solids+1);   // [mu_0, mu_1, mu_2, ...]
  mu[0] = mu_0;                 // default stiffness (for single solid)

  bool existRegionFile = true;

  if (existRegionFile) {
    /* Detect all solid cells in the mesh using the "structure.xml" mesh file */
    detect_structure_mesh(mesh, structure_mesh, solid_cells);

    /* Detect solid vertices in mesh (for mesh smoothing) */
    detect_solid_vertices(mesh, structure_mesh, solid_cells, solid_vertices);

    /* Look for multiple solids */
    read_solid_stiffness(mu, n_solids);

    /* Look for multiple solid types (check for solid files: "solid_1.xml" etc.) */
    set_solid_type(mesh, solid_type, n_solids);
  }
  delete structure_mesh;

  /* Boundary declarations */
  InflowBoundary iboundary;
  OutflowBoundary oboundary;
  NoSlipBoundary noslipboundary;

  /* Function declarations */
  BC_Momentum_3D bcf_mom(mesh, td);
  BC_Continuity bcf_con(mesh, td);
  ForceFunction f(mesh, td);
  PressureForceFunction fs(mesh, td);
  Function f_zero(mesh, 3, 0.0);

  Alpha inflow_marker(mesh);  // inlet marker
  Beta outflow_marker(mesh);  // outlet marker
  Epsilon epsilon(mesh, nu);  // viscosity
  Kappa kappa(mesh);          // friction coefficient

  NodeNormal node_normal(mesh);   // used in Laplacian smoother

  Function U, U0;

  /* Boundary conditions */
  DirichletBC p_bcin(bcf_mom, mesh, iboundary);         // BC for inlet
  DirichletBC p_bcnoslip(f_zero, mesh, noslipboundary); // BC for side walls
  DirichletBC p_bcout(bcf_con, mesh, oboundary);        // BC for outlet

  Array <BoundaryCondition*> p_bc_momentum;
  p_bc_momentum.push_back(&p_bcin);
  p_bc_momentum.push_back(&p_bcnoslip);

  Array <BoundaryCondition*> p_bc_pressure;
  p_bc_pressure.push_back(&p_bcout);


  if(!ParameterSystem::parameters.defined("PDE CFL number"))
    dolfin_add("PDE CFL number", 0.15);
  dolfin_set("PDE CFL number", 0.15);

  UCSolver psolver(mesh, U, U0, f, fs, f, iboundary, oboundary,
                   inflow_marker, outflow_marker, epsilon, kappa,
                   p_bc_momentum, p_bc_pressure, p_bc_pressure,
                   solid_cells, solid_vertices, solid_type,
                   T, nu, mu, nu_s, rho_s, ubar, node_normal, td, "primal", chkp);

  dolfin_set("Adaptive refinement percentage", 5.0);
  dolfin_set("ODE discrete tolerance", 2.0e-3);
  dolfin_set("ODE maximum iterations", 100);
  dolfin_set("PDE number of samples", 100);

  psolver.solve(U, U0);   // UCSolver::solve() defined in TimeDependentPDE.cpp

}
```

```
int main(int argc, char* argv[])
{
  timeval s_time;
  gettimeofday(&s_time, NULL);
  Mesh mesh;
  long w_limit = 0;
  Checkpoint chkp;
  int iter = 0;
  Mesh* structure_mesh;
  bool existStructureFile = true;

  unicorn_init(argc, argv, mesh, chkp, w_limit, iter, structure_mesh);

  if(dolfin::MPI::processNumber() == 0)
    dolfin_set("output destination", "terminal");
  else
    dolfin_set("output destination", "silent");

  // mesh refinement
  if (false) {
    preprocessing_mesh(mesh);
    exit(1);
  }

  // resize the meshes to unit scale
  float scale_factor = 0.1;
  message("Resizing mesh");
  resize_mesh(mesh, scale_factor);
  message("Mesh resized");

  if (existStructureFile) {
    message("Resizing structure mesh");
    resize_mesh(*structure_mesh, scale_factor);
    message("Structure mesh resized");
  }
  else {
    message("No structure mesh to resize");
  }

  real cv_min;
  get_minimum_cell_volume(mesh, cv_min);
  message("Minimum cell volume in mesh: %g", cv_min);

  if (false) {
    calculate_mesh_volume(mesh);
  }

  unicorn_solve(mesh, chkp, w_limit, s_time, iter, 0, 0, &solve, structure_mesh);

  dolfin_finalize();
  return 0;
}
```

`NSESolver.cpp` (essential parts)

```
using namespace dolfin;
using namespace unicorn;
//-----------------------------------------------------------------------------
UCSolver::UCSolver(Mesh& mesh, Function& U, Function& U0,
                   Function& f, Function& fs, Function& fc,
           SubDomain& inflow_boundary, SubDomain& outflow_boundary,
                   Function& inflow_marker, Function& outflow_marker,
           Function& epsilon, Function& kappa,
                   Array<BoundaryCondition*>& bc_mom,
                   Array<BoundaryCondition*>& bc_con,
                   Array<BoundaryCondition*>& bc_rho,
                   //real (*density)(Point p), bool (*collArea)(Point p),
                   MeshFunction<bool>& solid_cells,
                   MeshFunction<bool>& solid_vertices,
                   MeshFunction<unsigned int>& solid_type,
                   real T, real nu, Array<real>& mu,
           real nu_s, real rho_s, real ubar,
                   NodeNormal& node_normal, TimeDependent& td,
                   std::string solver_type, Checkpoint &chkp_i)
  : TimeDependentPDE(mesh, bc_mom, T, "UCSolver"), U(U), U0(U0),
    f(f), fs(fs), fc(fc),
    inboundary(inflow_boundary), outboundary(outflow_boundary),
    inmark(inflow_marker), outmark(outflow_marker),
    epsilon(epsilon), kappa(kappa),
    bc_mom(bc_mom), bc_rho(bc_rho), bc_con(bc_con),
    solid_cells(solid_cells), solid_vertices(solid_vertices),
    solid_type(solid_type),
    T(T), nu(nu), rho_s(rho_s), ubar(ubar),
    solver_type(solver_type), errest(0), perrest(0),
    rho_f(1.0),   // fluid density normalized to unity


  // dynamic viscosity
  mu_f = rho_f*nu;

  // define Reynolds number
  real u_scale = 1.0;               // velocity scale
  real l_scale = 1.0;               // length scale
  Re = u_scale*l_scale/nu;          // Reynolds number
```

```
219     /* The bilinear and linear forms */
220     if ( nsd == 3)    // 3D solver
221     {
222       if(solver_type == "primal")
223       {
224         // adjusted for low Reynolds number: input alpha and beta parameter
225         aM = new NSEMomentum3DBilinearForm(Uc, Um, alpha, beta, delta1, delta2, *fk,
226                                            theta, fmu_s, *lmbdaf, W, Wm);
227         LM = new NSEMomentum3DLinearForm(U, U0, Uc, Um, P, alpha, beta, delta1, delta2,
228                                          f, *fk, theta, S, fmu_s, *lmbdaf, W, Wm, outmark);
229
230         // original forms for Momentum equation
231         /*
232         aM = new NSEMomentum3DBilinearForm(Uc, Um, epsilon, delta1, delta2, *fk,
233                                            rho, theta, *muf, *lmbdaf, W, Wm, *bf);
234         LM = new NSEMomentum3DLinearForm(U, U0, Uc, Um, P, epsilon, delta1, delta2,
235                                          f, *fk, rho, theta, S, *muf, *lmbdaf, W, Wm,
236                                          *bf, *frho_d, outmark);
237         */
238         // adjusted for low Reynolds number
239         aC = new NSEContinuity3DBilinearForm(delta1, *fk);
240         LC = new NSEContinuity3DLinearForm(P0, U, delta1, *fk, alpha, f, fs);
241
242         // original forms for Continuity equation
243         /*
244         aC = new NSEContinuity3DBilinearForm(delta1, *fk);
245         LC = new NSEContinuity3DLinearForm(P0, U, delta1, *fk, rho, f, fs);
246         */
247
248         aR = new NSEDensity3DBilinearForm(U, Um, *fk, delta1);
249         LR = new NSEDensity3DLinearForm(rho0, U, Um, *fk, delta1);
250
251         aS = new NavierStokesStress3DBilinearForm;
252         LS = new NavierStokesStress3DLinearForm(S, U, fmu_s, vol_inv);
253
254         aE = new Eikonal3DBilinearForm(E0, deltaE);
255         LE = new Eikonal3DLinearForm(E0, *f_one);
256
257         aEG = new EikonalGradient3DBilinearForm;
258         LEG = new EikonalGradient3DLinearForm(EB);
259
260         // to calculate the mass flow
261         MF  = new MassFlowFunctional(U);
262
263         // to calculate the von Mises stress
264         aVMS = new VonMisesStressBilinearForm;
265         LVMS = new VonMisesStressLinearForm(S, vol_inv_vM);
266       }


752     // Initialize fmu_s
753   if(!chkp.restart())
754     {
755       set_solid_stiffness(mesh, solid_cells, solid_type, LM, mu, fmu_s, meshf_mu);
756     }
757
758     // Initialize alpha and beta
759   if(!chkp.restart())
760     {
761       set_alpha_beta(mesh, solid_cells, solid_type, LM, alpha, beta);
762     }


824     // find initial position of cilium tip (to measure the deflection)
825     is_vertex_owner = false;
826     Point p0_guess(0.0, 0.0, 1.0);  // default cilium
827   //  Point p0_guess(0.0, 0.0, 1.5);  // long cilium
828   //  Point p0_guess(0.0, 0.0, 0.8);  // short cilium
829   //  Point p0_guess(0.0, 0.0, 0.9);  // default cilium with ANSA mesh
830     find_closest_vertex(p0_guess, p0_cilium, vID_cilium);
831
832     /* end of UCSolver constructor */


841   /* To set the functions containing the stiffness parameter mu.  */
842   /* Sets meshf_mu and fmu_s to the corresponding values in mu    */
843   /* according to the cell markers solid_cells and solid_type.    */
844   /* Information about DOF is taken from the Linear Form LM.      */
845   void UCSolver::set_solid_stiffness(Mesh& mesh, const MeshFunction<bool>& solid_cells,
846                                      const MeshFunction<unsigned int>& solid_type,
847                                      Form*& LM, const Array<real>& mu,
848                                      Function& fmu_s, MeshFunction<real>& meshf_mu) {
849
850     int d = mesh.topology().dim();
851     int N = mesh.numVertices();
852     if(MPI::numProcesses() > 1)
853       N = mesh.distdata().global_numVertices();
854     int M = mesh.numCells();
855     if(MPI::numProcesses() > 1)
856       M = mesh.distdata().global_numCells();
857
858     UFC ufc(LM->form(), mesh, LM->dofMaps());
859     Cell c(mesh, 0);
860     uint local_dim = c.numEntities(0);
861     uint *idx = new uint[local_dim];
862     uint *id  = new uint[local_dim];
863     real *mu_block = new real[local_dim];
864     real mu_val;
865
```

```
866     for (CellIterator cell(mesh); !cell.end(); ++cell)
867     {
868       ufc.update(*cell, mesh.distdata());
869       (LM->dofMaps())[7].tabulate_dofs(idx, ufc.cell, cell->index());
870 
871       // Only one dof
872       uint ii = 0;
873       uint jj = 0;
874 
875       if (solid_cells.get(*cell))
876         mu_val = mu[solid_type.get(*cell)];
877       else
878         mu_val = 0.0;   // set zero stiffness to fluid region
879 
880       // set MeshFunction (used for visualization)
881       meshf_mu.set(*cell, mu_val);
882 
883       // set Function (used for the equations)
884       mu_block[jj] = mu_val;
885       id[jj++] = idx[ii];
886       fmu_s.vector().set(mu_block, jj, id);
887     }
888 
889     fmu_s.vector().apply();         // to finalize the assembly of the function
890 
891     delete[] mu_block;
892     delete[] idx;
893     delete[] id;
894   }
895 
896 
897   //-----------------------------------------------------------------------------
898   /* To set the parameters alpha and beta                     */
899   /* For low Reynolds number:  alpha = Re  &  beta = 1        */
900   /* For high Reynolds number: alpha = 1  &  beta = Re^{-1}   */
901   void UCSolver::set_alpha_beta(Mesh& mesh, const MeshFunction<bool>& solid_cells,
902                                 const MeshFunction<unsigned int>& solid_type,
903                                 Form*& LM, Function& alpha, Function& beta) {
904 
905     int d = mesh.topology().dim();
906     int N = mesh.numVertices();
907     if(MPI::numProcesses() > 1)
908       N = mesh.distdata().global_numVertices();
909     int M = mesh.numCells();
910     if(MPI::numProcesses() > 1)
911       M = mesh.distdata().global_numCells();
912 
913     UFC ufc(LM->form(), mesh, LM->dofMaps());
914     Cell c(mesh, 0);
915     uint local_dim = c.numEntities(0);
916     uint *idx = new uint[local_dim];
917     uint *id  = new uint[local_dim];
918     real *alpha_block = new real[local_dim];
919     real *beta_block  = new real[local_dim];
920     real alpha_val;
921     real beta_val;
922     real invRe = 1.0/Re;      // inverse Reynolds number
923 
924     for (CellIterator cell(mesh); !cell.end(); ++cell)
925     {
926       ufc.update(*cell, mesh.distdata());
927       (LM->dofMaps())[7].tabulate_dofs(idx, ufc.cell, cell->index());
928 
929       // Only one dof
930       uint ii = 0;
931       uint jj = 0;
932 
933       if (Re < 1.0) {
934         alpha_val = Re;
935         beta_val = 1.0;
936       }
937       else {
938         alpha_val = 1.0;
939         beta_val = invRe;
940       }
941 
942       // set Function alpha and beta
943       alpha_block[jj] = alpha_val;
944       beta_block[jj] = beta_val;
945       id[jj++] = idx[ii];
946       alpha.vector().set(alpha_block, jj, id);
947       beta.vector().set(beta_block, jj, id);
948     }
949 
950     alpha.vector().apply();
951     beta.vector().apply();
952 
953     delete[] alpha_block;
954     delete[] beta_block;
955     delete[] idx;
956     delete[] id;
957   }


1452  bool UCSolver::update(real t, bool end)
1453  {
1454    cout << "FSISolver::update: " << "t: " << t << " k: " << k << endl;
1455 
1456  // mqual->meshQuality();
```

```
1457
1458    if(t < 30 * k)
1459      mu_bar = mqual->mu_min;
1460
1461    // h^2 stabilisation
1462    real tmp = 4;
1463
1464    real cfl = 0.1 * tmp;
1465    if(t > 1.0e-4)
1466      cfl = 0.1 * tmp;
1467    if(t > 1.0e-3)
1468      cfl = 0.2 * tmp;
1469    if(t > 0.5)
1470      cfl = 0.3 * tmp;
1471  //  if(t > 1.5)
1472  //    cfl = 0.4 * tmp;
1473    k = cfl*hmin*hmin/ubar;


2106  // To compute the von Mises stress directly from the stress tensor S
2107  void UCSolver::computeVonMises()
2108  {
2109    int d = mesh().topology().dim();
2110    int N = mesh().numVertices();
2111    if(MPI::numProcesses() > 1)
2112      N = mesh().distdata().global_numVertices();
2113    int M = mesh().numCells();
2114    if(MPI::numProcesses() > 1)
2115      M = mesh().distdata().global_numCells();
2116
2117    {
2118      UFC ufc(aS->form(), mesh(), aS->dofMaps());
2119  //    Cell c(mesh(), 0);
2120  //    uint local_dim = c.numEntities(0);
2121      uint local_dim = 1;
2122      uint *idx = new uint[d * d * local_dim];
2123      uint *id  = new uint[d * d * local_dim];
2124      real *S_block = new real[d * d * local_dim];
2125      real vonMises = 0.0;
2126
2127      // calculate the von Mises stress
2128      for (CellIterator cell(mesh()); !cell.end(); ++cell)
2129      {
2130        if(solid_cells.get(*cell)) {
2131          ufc.update(*cell, mesh().distdata());
2132          (aS->dofMaps())[0].tabulate_dofs(idx, ufc.cell, cell->index());
2133          S.vector().get(S_block, d*d, idx);
2134
2135          real trace_S = S_block[0] + S_block[4] + S_block[8];
2136          vonMises = 0.0;
2137          uint jj = 0;
2138          for(int ii = 0; ii < d * d; ii++)
2139          {
2140            if(ii%4 == 0){
2141              vonMises += sqr(S_block[ii] - (1/3)*trace_S);
2142            }
2143            else
2144              vonMises += sqr(S_block[ii]);
2145          }
2146        vonMises = sqrt(vonMises);
2147        }
2148        else {
2149          vonMises = -1.0;
2150        }
2151        meshf_vonMises.set(*cell, vonMises);
2152      }
2153
2154      delete[] S_block;
2155      delete[] idx;
2156      delete[] id;
2157    }
2158  }
2159
2160  //-----------------------------------------------------------------------------
2161  // To compute the von Mises stress from the linear form LVMS
2162  void UCSolver::computeVonMisesStress()
2163  {
2164    if(MPI::processNumber() == 0) {
2165      cout << "Calculating the von Mises stress" << endl;
2166    }
2167
2168    ComputeVolInv(vol_inv_vMx);
2169
2170    // assemble the von Mises stress (squared) as a Function
2171    assembler->assemble(SvM.vector(), *LVMS);
2172
2173    // set the corresponding MeshFunction
2174    Mesh mesh = SvM.mesh();
2175    int d = mesh.topology().dim();
2176    int N = mesh.numVertices();
2177    if(MPI::numProcesses() > 1)
2178      N = mesh.distdata().global_numVertices();
2179    int M = mesh.numCells();
2180    if(MPI::numProcesses() > 1)
2181      M = mesh.distdata().global_numCells();
2182
2183    UFC ufc(aVMS->form(), mesh, aVMS->dofMaps());
2184  //  Cell c(mesh, 0);
2185  //  uint local_dim = c.numEntities(0);
```

```
    uint local_dim = 1;             // SvM is a cell based function
    uint *idx = new uint[local_dim];
    real *SvM_block = new real[local_dim];

    for (CellIterator cell(mesh); !cell.end(); ++cell)
    {
      ufc.update(*cell, mesh.distdata());
      (aVMS->dofMaps())[0].tabulate_dofs(idx, ufc.cell, cell->index());

      SvM.vector().get(SvM_block, local_dim, idx);

      // von Mises stress only defined in the solid
      if (solid_cells.get(*cell))
        meshf_SvM.set(*cell, sqrt(SvM_block[0]));
      else
        meshf_SvM.set(*cell, -1.0);
    }

    delete[] SvM_block;
    delete[] idx;
  }

  //-----------------------------------------------------------------------------
  // To compute the mass flow through the inflow and outflow boundaries
  void UCSolver::computeMassFlow(real t)
  {
    std::ofstream massflowFile;

    if(MPI::processNumber() == 0) {
      if (t == 0)
        massflowFile.open("massflow.dat", std::ios_base::out);
      else
        massflowFile.open("massflow.dat", std::ios_base::app);
      massflowFile.flush();
    }

    real mf0 = 0.0;
    real mf1 = 0.0;

    // integrate over the boundaries using the MF functional
    mf0 = assembler->assemble(*MF, inboundary);
    mf1 = assembler->assemble(*MF, outboundary);

    if(MPI::processNumber() == 0) {
      cout << "Mass flow at inlet: " << mf0 << endl;
      cout << "Mass flow at outlet: " << mf1 << endl;
      cout << "Mass flow ratio: " << mf1/mf0 << endl;
      massflowFile << t << "\t" << mf0 << "\t" << mf1 << "\t" << mf1/mf0 << "\n";
          massflowFile.flush();
      massflowFile.close();
    }
  }

  //-----------------------------------------------------------------------------
  /* To find the position and local index of the vertex closest to a given point */
  void UCSolver::find_closest_vertex(const Point& p0, Point& p1, uint& vID)
  {
    real min_dist = 1e6;
    int vertexID;

    cout << "Finding vertex closest to point: (" << p0.x() << ", " << p0.y() << ", " << p0.z() << ")" << endl;

    // find local minimum distance to given point
    uint local_index;
    uint global_index;
    for (VertexIterator v(mesh()); !v.end(); ++v) {
      Vertex& vertex = *v;
      if (vertex.point().distance(p0) < min_dist) {
        min_dist = vertex.point().distance(p0);
        local_index = vertex.index();
        global_index = mesh().distdata().get_global(vertex);
      }
    }
    real local_min_dist = min_dist;       // local minimum distance

    // find global minimum distance
    real min_dist_tmp = min_dist;
    MPI_Allreduce(&min_dist_tmp, &min_dist, 1, MPI_DOUBLE, MPI_MIN , MPI_COMM_WORLD);

    // All processes now have the same min_dist

    // reject all candidates except owner of the the closest vertex
    // ( get_owner seems to return 0 if owned by this process )
    uint vertex_owner = mesh().distdata().get_owner(local_index, 0);
    if (local_min_dist == min_dist && vertex_owner == 0) {
      is_vertex_owner = true;
      vertexID = local_index;
    }
    else {
      is_vertex_owner = false;
      vertexID = -1;
    }

    // communicate found vertex ID to all processes
    int vertexID_tmp = vertexID;
    MPI_Allreduce(&vertexID_tmp, &vertexID, 1, MPI_INT , MPI_MAX , MPI_COMM_WORLD);

    // All processes now have the same global vertex ID
    vID = vertexID;
```

```
2285
2286    if (is_vertex_owner) {
2287      p1 = mesh().geometry().point(vertexID); // the found vertex
2288      dolfin_set("output destination", "terminal");
2289      cout << "Found vertex: (" << p1.x() << ", " << p1.y() << ", " << p1.z() << ")" << endl;
2290      cout << "Vertex found by process " << MPI::processNumber() << " with local index " << vertexID << endl;
2291      dolfin_set("output destination", "silent");
2292    }
2293    else {
2294      p1 = p0;        // return guessed point for all other processes (not used anyway)
2295    }
2296  }
2297
2298  //-----------------------------------------------------------------------
2299  /* To compute and store the current deflection of the cilium tip */
2300  void UCSolver::compute_deflection(const Point& p0, const uint& local_index, real t)
2301  {
2302    if (is_vertex_owner) {
2303      std::ofstream deflectionFile;
2304
2305      if (t == 0)
2306        deflectionFile.open("deflection.dat", std::ios_base::out);
2307      else
2308        deflectionFile.open("deflection.dat", std::ios_base::app);
2309      deflectionFile.flush();
2310
2311      Point p1 = mesh().geometry().point(local_index);
2312      real deflection = p1.distance(p0);
2313
2314      deflectionFile << t << "\t" << p1.x() << "\t" << p1.y() << "\t" << p1.z() << "\t" << deflection << "\n";
2315          deflectionFile.flush();
2316      deflectionFile.close();
2317
2318      dolfin_set("output destination", "terminal");
2319      cout << "Deflection of cilium at t = " << t << " : " << deflection << endl;
2320      cout << "Position of cilium : (" << p1.x() << ", " << p1.y() << ", " << p1.z() << ")" << endl;
2321      dolfin_set("output destination", "silent");
2322    }
2323  }

2452  void UCSolver::ComputeStabilization(Mesh& mesh, Function& w, real nu, real k,
2453                                      Vector& d1vector, Vector& d2vector,
2454                                      Vector& d1invvector, Vector& dEvector,
2455                                      Form& form, real t, real Efactor)
2456  {
2457    real C1;
2458    real C2;
2459
2460    // set stabilisation parameters according to Reynolds number
2461    if (Re < 0.1) {
2462      C1 = 1.6;
2463      C2 = 0.8;
2464    }
2465    else if (Re < 1.0) {
2466      C1 = 0.8;
2467      C2 = 0.4;
2468    }
2469    else {
2470      C1 = 0.4;
2471      C2 = 0.2;
2472    }

2508    for (CellIterator cell(mesh); !cell.end(); ++cell)
2509    {
2510      normw = 0.0;
2511
2512      for(uint i =0; i < local_dim ; i++)
2513        normw += sqr( *(wp++) );
2514
2515      normw = sqrt(normw);
2516
2517      normw /= (*cell).numEntities(0);
2518
2519      real h = (*cell).diameter();
2520
2521      uint cid = 0;
2522      if ( MPI::numProcesses() > 1)
2523        cid = mesh.distdata().get_cell_global((*cell).index());
2524      else
2525        cid = (*cell).index();
2526
2527      if (true || ((normw*h/nu) > 1.0) || (nu < 1.0e-10) ){
2528        d1_block[ci] = C1 * h * h;
2529        d2_block[ci] = C2 * h * h;
2530      }
2531      else {
2532        d1_block[ci] = C1 * sqrt(h) * (1.0 / sqrt( 1.0/sqr(kk) + sqr(normw/h)));
2533        d2_block[ci] = C2 * sqrt(h) * h;
2534      }
2535      dinv_block[ci] = 0.0 / (d1_block[ci]);
2536
2537      dE_block[ci] = CE * h;
2538
2539      rows2[ci++] = cid;
2540    }
2541
2542    d1vector.set(d1_block, mesh.numCells(), rows2);
2543    d2vector.set(d2_block, mesh.numCells(), rows2);
```

Stockholm University
SE-106 91 Stockholm
Phone: 08 - 16 20 00
www.su.se

Stockholm
University