Adaptivity for Fluid Structure Interaction

CEM DEGI RMENCI

KTH Computer Science and Communication

Master of Science Thesis
Stockholm, Sweden 2010
Adaptivity for Fluid Structure Interaction

CEM DEĞIRMENCI

Master’s Thesis in Numerical Analysis (30 ECTS credits) at the Scientific Computing International Master Program
Royal Institute of Technology year 2010
Supervisor at CSC was Johan Jansson
Examiner was Michael Hanke

TRITA-CSC-E 2010:032
ISRN-KTH/CSC/E--10/032--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden
URL: www.kth.se/csc
Abstract

The implementation and testing of mesh adaptation using refinement, coarsening and swap operations with the use of MAdLib libraries for the existing fluid structure interaction code in UNICORN is described. For testing purposes a new solver which solves momentum and continuity equations in a combined system is implemented using Newton Iteration to resolve the non-linearity. Finally, the formulation of dual problem in order to bound the error of a certain quantity of interest with optimal mesh size is presented.
Adaptivitet för Fluid Struktur Interaktion

Vi beskriver implementenonen och experiment av mesh-adaptering baserat på förfinning, förgrovnning och swap-operationer med hjälp av MAdLib-biblioteket för en existerande fluid-struktur-lösare i UNICORN. Vi undersöker en variant av löseren som löser rörelsemängd och kontinuitetsekvationer i ett system och använder Newtons metod för att hantera ickelinjäriteten. Slutligen presenteras en formulering av ett dualproblem som del i feluppskattning i en given kvantitet för optimal nätstorlek.
A.1 MADLib Interface ........................................ 32
  A.1.1 Converting MDBmesh to Dolfin Mesh Format .......... 32
  A.1.2 Converting Dolfin Mesh to MDBmesh Format .......... 33
A.2 Interpolation ............................................. 34
A.3 Solver .................................................... 35
  A.3.1 Optimized Form File .................................... 35
  A.3.2 Form File with All Terms ............................... 36
  A.3.3 Main Loop .............................................. 39

Bibliography .................................................. 41
Acknowledgments

First, I would like to thank my supervisor, Doctor Johan Jansson for his extraordinary patience and help, and Professor Johan Hoffman for invaluable comments in the progress of this project.

Finally, I would like to thank my family for supporting and guiding me through the oscillations of life.
Chapter 1

Introduction

Fluid Structure Interaction (FSI) is the interaction of fluid flow and elastic structure where the fluid flow causes deformation in the the elastic body and the state of the elastic body effects the fluid flow simultaneously. FSI results in problems often too complex to be solved analytically and these problems are analyzed with expensive numerical simulations. Examples of such problems are for example investigations on the oscillations of airplane wings or bridges, sedimentation of fibers in paper manufacturing or flow of blood inside elastic blood vessels. The images below show visualization of computations for such a project to investigate flapping bird flight using the unified continuum mechanics solver developed at KTH.

Table 1.1. FSI project for simulating flapping bird flight. Computational Technology Lab, KTH.

In order to solve the FSI problems, an approach is using separate solvers with different equations for fluid and structure (coupled methods) where fluid structure interface forms part of boundary. The two solvers can be coupled by first assuming some displacement on the interface and iterated by using stress/force obtained from fluid solver as a boundary condition for the structure solver to calculate the displacement again for that time step. This fixed point iteration is looped until the assumed and calculated displacements converge[1]. For this method stability and convergence analysis as well as error estimation is difficult. Another method uses
construction of an algebraic system to couple the degrees of freedom on interface and uses a Newton-type iteration\cite{2, 3}. Again, there is no unified discretization for global error estimation and stability analysis.

Besides having two coupled solvers for fluid and structure, another paradigm is formulating fluid and structure equations in one continuum model. Having a single continuum model allows formulation of a single residual function in whole system with ease, enabling global error estimation and stability analysis. The paper\cite{6} by Hron and Turek uses an Arbitrary Lagrangian Eulerian (ALE) formulation with a very complex discretization for different conservation equations for fluid and solid. Dunne and Rannacher use an Initial Position (IP) set with a fixed computational mesh in\cite{5} to prevent smearing of interface.

In this work an ALE formulation is used which needs a mesh changing at each time step. As it will be explained in detail later, the mesh movement is needed to capture the moving interface between fluid and structure without diffusion. In order to control the quality and size of mesh while it is changing, methods for mesh adaptivity which consists of local coarsening, refinement and swapping operations are used. This way a previously implemented monolithic FSI solver is also extended by replacing its method to adjust quality by mesh smoothing using the solution of a partial differential equation such as Laplace equation. Two main contributions are presented:

1. Investigation of replacing mesh smoothing by local coarsening, refinement and swapping operations to maintain mesh size/quality.
2. Derivation of an error estimate for the Unified Continuum model as a base for adaptive error control.

The second chapter presents Unified Continuum equations in Arbitrary Lagrangian Eulerian sense \cite{4} and variational formulations.

The third chapter explains a solver implementation which solves momentum and continuity equations in single system and uses Newton Iteration to resolve the nonlinearity.

The fourth chapter consists of theoretical work on obtaining an optimal mesh for some error tolerance in a functional of interest using the dual problem. It contains information about the dual problem and a formulation of discrete residual.

In fifth chapter some results in 3D for a sedimenting particle problem and for 2D the results of the benchmark problem from [5] and [6] is provided.

In sixth chapter discussion and future work is given and implementation details are presented in appendix.
Chapter 2

Unified Continuum model

The Unified Continuum model in Eulerian coordinates described in [4] is a framework where conservation laws for mass and momentum is used and stress \( \sigma \) and phase variable \( \theta \) is used as data for discriminating property differences in continuum.

The motivation behind the equations are given below. The complete derivations for the differential equations for fluid flow can be found in [20] while the derivation of elasticity equations are stated in [21].

1. Conservation of momentum : This law states that the rate of change of momentum of a moving volume equals the total force (surface stresses plus body forces) acting on it. It can be reduced to:

\[
\rho (\dot{u} + (u \cdot \nabla)u) = \nabla \cdot \sigma + f
\]

or equivalently

\[
\rho \left( \frac{\partial u_i}{\partial t} + \sum_j u_j \frac{\partial u_i}{\partial x_j} \right) = \sum_j \frac{\partial \sigma_{ij}}{\partial x_j} + f_i
\]  

(2.1)

where \( u \) represents velocity, \( \sigma \) stress and \( f \) represents external force such as gravity. The density \( \rho \) consists of a fluid density \( \rho_f \) and solid density \( \rho_s \) as in

\[
\rho = \theta \rho_f + (1 - \theta) \rho_s
\]

Where \( \theta \) is the function that marks a particle as fluid or solid.

The \( \sigma \) in (2.1) is decomposed further into a part resulting from Pressure \( P \) that is orthogonal to surface of the volume examined, the solid stress \( \sigma_s \) and the contribution from viscous forces of fluid. Assuming incompressible Newtonian fluid with viscosity \( \nu_f \), we can write:

\[
\sigma_{ij} = \theta \nu_f \delta_{ij} \frac{\partial u_i}{\partial x_j} + (1 - \theta) \sigma_{sij} - \delta_{ij} P
\]
Here $\delta_{ij}$ is the Kronecker delta function.

2. Conservation of mass: This law states that the rate of change of mass of a volume is equal to the rate at which mass crosses boundary of that volume, since mass is neither created nor destroyed. With the incompressibility assumption for fluid and solid, it is stated as:

$$\nabla \cdot \mathbf{u} = 0$$

or equivalently

$$\sum_j \frac{\partial u_j}{\partial x_j} = 0$$ (2.2)

3. Phase Convection Equation: A phase function $\theta$ is used to mark particles as solid (if $\theta = 0$) or fluid (if $\theta = 1$). Phase of a particle is assumed not to change during simulation, so the rate of change of fluid particles in a volume is equal to rate at which fluid crosses boundary of that volume. This is stated as:

$$\dot{\theta} + \mathbf{u} \cdot \nabla \theta = 0$$

or equivalently

$$\frac{\partial \theta}{\partial t} + \sum_j u_j \frac{\partial \theta}{\partial x_j} = 0$$ (2.3)

4. Constitutive Laws for Stress: The last equation is to model evolution of the solid stress $\sigma_s$ with time. As stated in [4], many different models may be chosen to represent structures with different properties. For our purposes the following example fluid structure constitutive laws for Neo-Hookean [5, 7] object is used:

$$\frac{\partial \sigma_s}{\partial t} = 2\mu_s \left( \frac{1}{2} (\nabla \mathbf{u} + (\nabla \mathbf{u})^T) + \nabla \mathbf{u} \sigma_s + \sigma_s (\nabla \mathbf{u})^T \right)$$ (2.4)

the reader can find the derivation in [4]. Here $\mu_s$ is the shear modulus of structure, which is the ratio of the shear stress to shear strain.

# 2.1 Finite Element Method

The finite element method is a tool for solving differential equations[5,6] which can deal with complex geometries efficiently.

Assuming that $\dot{\mathbf{v}}$ is a smooth function with compact support (test function), by multiplying this test function to the differential equation and integrating in domain, it then becomes possible to move the derivatives from the solution of the equation to the test function using integration by parts. The new form is called
2.1. FINITE ELEMENT METHOD

A weak formulation and a weak solution is a solution that satisfies weak formulation for given test function.

In order to compute an approximation to the solution, we restrict ourselves to a subset of the original space for solution. This is piecewise linear continuous functions and piecewise discontinuous constant functions in our case.

To construct these functions, the domain is partitioned into a mesh and finite number of basis functions are then constructed on this subdivision. Below in figure 2.1 a basis function for piecewise continuous linear functions and a basis function for piecewise discontinuous constant functions is given on a 1D mesh.

![Figure 2.1. basis functions for piecewise discontinuous constant functions (upper) and piecewise linear continuous function](image)

Stress $\sigma$ and phase $\theta$ functions are approximated with piecewise discontinuous constant functions and others are approximated by piecewise continuous linear functions in space(trial functions). The approximation of the exact solution will be a linear combination of these basis functions and the coefficients in this combination are the unknowns.

Ritz-Galerkin method constructs a system of equations for the unknowns by using the test functions that are equal to the basis functions but take the value 0 for the points where Dirichlet Boundary conditions are given. (for example where inflow velocity is given or pressure is known for outflow). In order to construct the system of equations, weak formulation is used with the trial and test functions described.

Let $Q = \Omega \times I$ be the space-time domain and $V_h = (V^u_i, V^p, V^\theta)$ be the space consisting of linear piecewise continuous(vector), linear piecewise continuous(scalar) and piecewise constant discontinuous functions in space for given mesh respectively which satisfy given Dirichlet Boundary conditions. Our test functions are then from the space $V_0 = (V^u_0, V^p_0, V^\theta_0)$ which consist of again linear piecewise continuous(vector), linear piecewise continuous(scalar) and piecewise constant discontinuous functions in space but on Dirichlet boundaries take the value 0. The functions
of these spaces are piecewise continuous linear in time.

The Galerkin approximation for the equations 2.1, 2.2, 2.3 are then given (assuming homogeneous Neumann stress) by:

Find \((u_i, P, \theta) \in V_h\) s.t \(\forall (v^u_i, v^p, v^\theta) \in V_0\)

\[
\rho \int_Q \left( \frac{\partial u_i}{\partial t} + \sum_j u_j \frac{\partial u_i}{\partial x_j} \right) - f_i v_i^u dQ + (1 - \theta) \int_Q \sum_j (\sigma_{ij} \frac{\partial v_i^u}{\partial x_j}) dQ
\]

\[
+ \theta \nu_f \int_Q \sum_j \frac{\partial u_i}{\partial x_j} \frac{\partial v_i^u}{\partial x_j} dQ + \int_Q \frac{\partial P}{\partial x_i} v_i^u dQ = 0
\]

\[
\int_Q \left( \frac{\partial \theta}{\partial t} + \sum_j u_j \frac{\partial \theta}{\partial x_j} \right) v^\theta dQ = 0
\]

### 2.2 Stabilized Weak Formulation

The Navier Stokes equations are not stable for equal order elements caused by the violation of LBB (Ladyshenskaya, Babuska, Brezzi) condition.[9]

In order to avoid instability caused by both violation of LBB condition and spurious oscillations caused by sharp gradients in convection dominated cases, Streamline Diffusion method is used which adds least square weighted residuals to the equation resulting in good stability properties and high accuracy[10].

The weak formulation without any stabilization term can also be written as \((R(\hat{U}), \hat{v}) = 0\) where \(\hat{U} = (U_i, P, \theta), \hat{v} = (v^u_i, v^p, v^\theta)\) and \(R(\hat{U})\) is the residual with components:

\[
R(\hat{U})_i = \rho \left( \frac{\partial U_i}{\partial t} + \sum_j U_j \frac{\partial U_i}{\partial x_j} \right) - \sum_j \frac{\partial \sigma_{ij}}{\partial x_j} - f_i
\]

\[
R(\hat{U})_P = \sum_j \frac{\partial U_j}{\partial x_j}
\]

\[
R(\hat{U})_\theta = \frac{\partial \theta}{\partial t} + \sum_j u_j \frac{\partial \theta}{\partial x_j}
\]

The method changes the formulation to \((R(\hat{U}), \hat{v} + \delta R(\hat{v})) = 0\) with stabilization parameter \(\delta\). Similar to the work in [4], only relevant stabilization terms are added to the formulation. It should however be noted that this method is not fully consistent when not all terms are added.
2.3 Moving Mesh Methods

2.3.1 Local Arbitrary Lagrangian Eulerian (ALE) coordinate map

To prevent diffusion of the phase interface, local ALE coordinate map [4, 8] is introduced. The mesh is moved with velocity $\beta_h$ which is equal to $u$ at the vertices where $\theta = 0$. For the rest of the vertices, $\beta_h$ is calculated through mesh smoothing if mesh smoothing is being used to increase mesh quality. The equations for calculations on a moving mesh are then obtained using chain rule which is explained in [22] and they are:

Find $(u_i, P, \theta) \in V_h$ s.t $\forall(u_i^u, v^p, v^\theta) \in V_0$

\[
\rho \int_Q \frac{\partial u_i}{\partial t} v^u_i dQ + \rho \int_Q \left( \sum_j (u_j - \beta_h) \frac{\partial u_i}{\partial x_j} - f_i \right) (v_i^u + \delta \sum_j (u_j - \beta_h) \frac{\partial v_i}{\partial x_j} + \delta \frac{\partial v_p}{\partial x_i}) dQ
\]
\[+(1 - \theta) \int_Q \sum_j (\sigma_{sij} \frac{\partial v_i}{\partial x_j}) dQ + \theta v_f \int_Q \sum_j \frac{\partial u_i}{\partial x_j} \frac{\partial v_i}{\partial x_j} dQ
\]
\[+ \int_Q \frac{\partial P}{\partial x_i} (v_i^u + \delta \sum_j (u_j - \beta_h) \frac{\partial v_i}{\partial x_j} + \delta \frac{\partial v_p}{\partial x_i}) dQ = 0
\]
\[
\sum_j \frac{\partial u_i}{\partial x_j} v^p dQ = 0
\]
\[
\int_Q \frac{\partial \theta}{\partial t} + \sum_j (u_j - \beta_h) \frac{\partial \theta}{\partial x_j} v^\theta dQ = 0
\]
2.3.2 Mesh Smoothing

Moving the mesh generally results in an inconsistent mesh (inverted cells) or cells with low quality. Mesh smoothing is a technique to increase quality by solving a PDE (Laplace equation for example) and realigning the mesh coordinates according to the solution[4]. With Mesh smoothing the number of vertices and cells do not change which can result in low quality elements for large deformations. In this thesis instead of mesh smoothing, refinement swapping and coarsening operations through the MAdLib library are used to obtain better mesh quality.

2.3.3 Mesh Adaptness With MAdLib

MAdLib is a library that performs global node repositioning and mesh adaptation by local mesh modifications on tetrahedral or triangular meshes. It is designed to frequently adapt the mesh in transient computations, or to generate specific meshes from any initial mesh. MAdLib is written in C++.[11]

The algorithm is explained in [23] in detail. In order to summarize, it is important to define the operations on mesh:

**Edge split** operation is splitting all the tetrahedrons(3D) or triangles(2D) containing a long edge. The operation is demonstrated in table 2.1 where the red edge is split by splitting the 2 triangles that contain the red edge.

<table>
<thead>
<tr>
<th>Table 2.1. Edge Split operation on a 2D mesh</th>
</tr>
</thead>
</table>

**Edge collapse** operation is removing an edge and the bounded elements from the mesh by merging its two extremities at one location. The operation is demonstrated in table 2.2 where the red edge is collapsed.

<table>
<thead>
<tr>
<th>Table 2.2. Edge Collapse operation on a 2D mesh</th>
</tr>
</thead>
</table>

**Edge swap** operation is re meshing a cavity surrounding an edge to improve quality of elements. The operation is demonstrated in table 2.3 where the cavity surrounding red edge is re meshed with the other diagonal of the quadrangle forming the cavity.
Table 2.3. Edge Swap operation on a 2D mesh

**Face swap** operation is replacing the facet between 2 tetrahedrons with an edge to produce 3 tetrahedrons. In table 2.4 the operation is demonstrated where the red facet is replaced with the red edge.

Table 2.4. Face Swap operation on a 3D mesh

**Face collapse** operation is a compound operation consisting of an edge split with an edge collapse demonstrated in table 2.5

Table 2.5. Face Collapse operation on a 2D mesh

**Double edge split collapse** operation is a compound operation consisting of splitting two edges of a tetrahedron and joining the new edge as demonstrated in table 2.6.

Table 2.6. Face Collapse operation on a 2D mesh

A tetrahedron is said to be a sliver when it has a small volume and long edges. A sliver is said to be type 1 if two edges almost intersect and type 2 if one vertex
is very close to opposite face. In order to decide if an element is a sliver, the cubic mean ratio \( \eta^3 = \frac{V^2}{(\sum_{i=1}^{6} L_i^2)^3} \) is used \( L_i \) being the length of an edge, \( V \) volume. An element is said to be a sliver if \( \eta \) is smaller than a given threshold set by user.

In order to remove the slivers, the following local mesh modification operations are tried with given priority order which is part of global mesh modification procedure described later.

<table>
<thead>
<tr>
<th>Sliver Type</th>
<th>Priority Level</th>
<th>Local Mesh Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Split one of the two key edges</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Collapse one of the edges of tetrahedron</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Split both key edges and collapse the new interior edge</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Split one of the key edges and collapse the new vertex</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Swap one of the two key edges</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Relocate a vertex</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Collapse one of the edges of tetrahedron</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Collapse the key face</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Swap any of the edges bounding the key face</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Swap the key face</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Relocate a vertex</td>
</tr>
</tbody>
</table>

Table 2.7. Local Mesh Modification Sequence for Sliver Removal

In order to control the edge size and element quality in a mesh, the following global mesh modification procedure is used:

Do{

• Collapse edges that are shorter than the value set by user.

• Loop over all edges, compute quality of elements around the edges(Cubic mean ratio). If quality is lower than the threshold set by user, try different swap configurations to improve quality. Apply the new swap configuration if it improves quality and does not produce any long or short edge.

• Eliminate sliver tetrahedrons using the operations in table 2.7.

• Split edges that are longer than the value set by the user.

}While the mesh is modified.

Since MAdLib uses a different mesh representation ((MDBMesh class that implements 'Gmsh' of [24])) than the representation used in the implementation of solver, an interface class for data conversion was necessary. The number of vertices and elements is not constant with this approach which also the values for new elements have to be interpolated causing a source for error.
Chapter 3

Implementation of solver

In this chapter the discretization and implementation of the equations 2.4 - 2.7 is described. The implementation of a variant of Newton iteration is discussed and restrictions on time step is discussed. The phase function $\theta$ is updated at each time step and mesh is modified to capture the interface between fluid and solid exactly to prevent diffusion of structure. Moving the mesh however reduces the quality of the mesh and adapting the mesh to obtain better elements becomes necessary. The MAdLib library is introduced to the implementation to obtain mesh with better quality instead of mesh smoothing. Next the benchmark problem is given and results are discussed.

3.1 Preliminaries

FEniCS is an open source project for the automated solution of differential equations using finite element method[13].

DOLFIN is the C++/Python interface of FEniCS, providing a consistent problem solving environment for ordinary and partial differential equations[14].

The FEniCS Form Compiler FFC provides state-of-the-art automatic and efficient evaluation of general multi linear forms (variational formulations) for FEniCS. FFC functions as the form evaluation system for DOLFIN but can also be used to compile forms for other systems[15].
3.2 Solver Details

3.2.1 Equations 2.5 and 2.6

To implement equations 2.5 and 2.6, it is possible to add equations together to form and solve a combined system for $u$ and $P$ rather than solving for momentum and
continuity separately. The equation becomes:

\[ \sum_i \rho \int_Q \frac{\partial u_i}{\partial t} v_i^n dQ + \rho \int_Q \left( \sum_j (u_j - \beta_{ij}) \frac{\partial u_i}{\partial x_j} - f_i(v_i^n + \delta \sum_j (u_j - \beta_{ij}) \frac{\partial v_i^n}{\partial x_j} + \frac{\partial v^n}{\partial x_i})dQ \right) + (1 - \theta) \int_Q \sum_j \left( \sigma_{sij} \frac{\partial v_i^n}{\partial x_j} \right) dQ + \theta \nu_f \int_Q \sum_j \frac{\partial u_i}{\partial x_j} \frac{\partial v_i^n}{\partial x_j} dQ \]

\[ + \int_Q \frac{\partial P}{\partial x_i} (v_i^n + \delta \sum_j (u_j - \beta_{ij}) \frac{\partial v_i^n}{\partial x_j} + \frac{\partial v^n}{\partial x_i})dQ \] + \int_Q \sum_j \frac{\partial u_i}{\partial x_j} v_j dQ = 0 \] 

(2.8)

The method uses a Crank-Nicholson type discretization and thus nonlinearity is present. To overcome this, derivatives of the residual in weak form is calculated and a Newton iteration is performed which will give a linearized problem at each iteration. Note that use of equation 2.4 is necessary to find derivative of solid stress with respect to \( u_i \) and \( P \). The residual function for this weak form can then be written as:

\[ \sum_i \rho \int_Q \frac{\partial u_i}{\partial t} v_i^n dQ + \rho \int_Q \left( \sum_j (u_j - \beta_{ij}) \frac{\partial u_i}{\partial x_j} - f_i(v_i^n + \delta \sum_j (u_j - \beta_{ij}) \frac{\partial v_i^n}{\partial x_j} + \frac{\partial v^n}{\partial x_i})dQ \right) + (1 - \theta) \int_Q \sum_j \left( \sigma_{sij} \frac{\partial v_i^n}{\partial x_j} \right) dQ + \theta \nu_f \int_Q \sum_j \frac{\partial u_i}{\partial x_j} \frac{\partial v_i^n}{\partial x_j} dQ \]

\[ + \int_Q \frac{\partial P}{\partial x_i} (v_i^n + \delta \sum_j (u_j - \beta_{ij}) \frac{\partial v_i^n}{\partial x_j} + \frac{\partial v^n}{\partial x_i})dQ \] + \int_Q \sum_j \frac{\partial u_i}{\partial x_j} v_j dQ = -R_w(u_0, u_1, P, v_0^n, v_1^n, v^n) \] 

(2.9)

The derivative of \( R_w \) in 2.9 with respect to \( u_0 \) is given below as an example.

\[ -\frac{\partial R_w}{\partial u_0}(u) = \int_Q \frac{\partial w}{\partial t} v_0^n dQ + \rho \int_Q \left[ \frac{\partial u_0}{\partial x_0} + \sum_i (u_i - \beta_{i0}) \frac{\partial w}{\partial x_i} \right] [v_0^n + \delta \sum_i (u_i - \beta_{i0}) \frac{\partial v_0^n}{\partial x_i} + \frac{\partial P}{\partial x_0}] \]

\[ + \int_Q \sum_i (u_i - \beta_{i0}) \frac{\partial u_0}{\partial x_i} - f_i \left[ \frac{\partial v_0^n}{\partial x_i} \right] dQ \]

\[ + \int_Q \left[ \frac{\partial u_0}{\partial x_0} + \sum_i (u_i - \beta_{i0}) \frac{\partial w}{\partial x_i} \right] dQ \]

The iteration can be summarized as \( J(\hat{U}^n)(\hat{U}^{n+1}) = J(\hat{U}^n) - R(\hat{U}^n) \) where \( J \) is the Jacobian consisting of \( \frac{\partial R_w}{\partial u_0}, \frac{\partial R_w}{\partial u_1}, \frac{\partial R_w}{\partial P} \). Here \( \hat{U}^n \) is the numerical solution.
at iteration \( n \) with first components for velocity and the last for pressure. Iteration is stopped when the change (measured with infinity norm) in 2 successive iterations is lower than a given threshold.

For time discretization a Crank-Nicholson scheme is chosen where time derivative of \( w \) is expressed as
\[
\frac{\partial w}{\partial t} = \frac{w(t_N) - w(t_{N-1})}{t_N - t_{N-1}}
\]
and other terms are replaced with
\[
\bar{w} = \frac{1}{2}(w(t_N) + w(t_{N-1}))
\]

The form file can be found at last chapter. A performance optimized form file not containing all the terms as well as a slower implementation with all the terms is available.

3.2.2 Equation 2.7 & Mesh Adaptation

With the introduction of local ALE, Equation 2.7 becomes trivial to solve \( \dot{\theta} = 0 \) in solid part since \( u_j = \beta h_j \) in solid part and no form file is necessary. However the mesh movement lowers the quality of elements and in this thesis swapping, coarsening and refinement operations are used with the MAAdLib library to protect mesh quality. With MAAdLib version 1.1.0 errors in projection is detected and instead of using the default projection by MAAdLib, a new projection function has been written.
Chapter 4

Adaptivity for Error in Functionals of Interest

The huge computational cost for the fluid structure interaction (FSI) problems forces the researcher to use the existing computational power as effective as possible. Adaptive mesh refinement with respect to calculated weighted residuals and solution of dual problem as explained in [16] enables bounding the error of a functional of interest and using minimal number of elements during calculations.

In this section the formulation of dual problem and how it can be used to bound errors in functionals of interest is briefly given. For an elaborate discussion cf. [17, 18].

4.1 Definitions

Definition 1 A Cauchy sequence is a sequence $x_i$ if for each $\epsilon$ there exists $N$ such that for all $m,n > N$, $|x_m - x_n| \leq \epsilon$.

Definition 2 A normed linear space $(V, ||.||)$ is called a Banach space if it is complete with respect to the metric induced by the norm $||.||$. Completeness for a space means containing the limits of each Cauchy sequence in the space.

Definition 3 A Linear functional is a linear map from a vector space to its field of scalars. A functional is bounded (continuous) if there is a constant $C > 0$ such that $|Lx| \leq C ||x||$ for all $x \in X$ where $L : X \rightarrow \mathbb{R}$ or $\mathbb{C}$

Definition 4 If $X$ is a normed vector space, the space $L(X,Y)$ consists of bounded linear functions $L : X \rightarrow Y$. The norm for $L(X,Y)$ (operator norm) is defined as

$$||L|| = \sup_{x \in X, x \neq 0} \frac{||Lx||_Y}{||x||_X}$$

The space $L(X, \mathbb{R})$ is named as dual space of $X$ and is represented with $X^*$. 
Riesz representation theorem will be used for this section.

**Theorem 1** For every bounded linear functional $F$ on a Hilbert space $X$, there is a unique element $y$ in $X$ such that

$$F(x) = (x, y) \forall x \in X, \text{and } ||y||_X = ||F||_{X^*}.$$ 

This theorem is known as Riesz representation theorem.

### 4.2 Outline of the method

As a starting point, it should be noted that it is possible to write

$$R(u) - R(U) = \int_0^1 R'(su + (1 - s)U)(u - U)ds$$

where $R'$ is the Jacobian of the residual. Assuming $R'$ to be a continuous function. Since $u$ is not available, $u$ in this integral is approximated by $U$. Leaving us with

$$R(u) - R(U) \approx R'(U)(u - U)$$

For simplicity in notation $A = R'(U)$ is used in the rest of the text.

The basic idea for bounding the error in a functional $M(\hat{u} - \hat{U})$ is first making use of Riesz’ representation theorem where

$$||M(\hat{u}) - M(\hat{U})|| = |(\hat{u} - \hat{U}, \zeta)|$$

$U$ is the calculated solution in a Hilbert space $V$ whereas $u$ is the exact solution.

By constructing a dual problem such that it satisfies

$$A\phi = \zeta$$

with boundary conditions that allows for any $v, w \in V$

$$(Av, w) = (v, A^*w)$$

It is then possible to bound the error in a functional as

$$||M(\hat{u}) - M(\hat{U})|| = |(u - U, \zeta)|
= |(u - U, A^*\phi)|
= |(A(u - U), \phi)|
= \left|hR(U), h^{-1}(\phi - \pi_h\phi)\right|
\leq |hR(U)| \left|h^{-1}(\phi - \pi_h\phi)\right|
\leq |hR(U)| \left|C_i \left(\sum_i \frac{\partial^2 \phi}{\partial x_i^2}\right)^{1/2}\right|$$

Where the residual is already calculated and $\phi$ can be obtained by solving the dual problem. $C_i$ is interpolation error constant depending on the minimum angle in the mesh and order of estimate.[8]
4.3 Formulation of Linearized problem

The Jacobian of strong residual will be needed to calculate $A(\hat{U})$ which satisfies $R(\hat{u}) - R(\hat{U}) \approx A(\hat{U})(\hat{u} - \hat{U})$ For the 2D system this will be of size 4x4 since \( \hat{(U)} = (U_1, U_2, P, \theta)^T \). The system is given by:

\[
J = 
\begin{bmatrix}
\frac{\partial R_0}{\partial U_0} & \frac{\partial R_0}{\partial U_1} & \frac{\partial R_0}{\partial P} & \frac{\partial R_2}{\partial \theta} \\
\frac{\partial R_1}{\partial U_0} & \frac{\partial R_1}{\partial U_1} & \frac{\partial R_1}{\partial P} & \frac{\partial R_2}{\partial \theta} \\
\frac{\partial R_2}{\partial U_0} & \frac{\partial R_2}{\partial U_1} & \frac{\partial R_2}{\partial P} & \frac{\partial R_4}{\partial \theta} \\
\frac{\partial R_3}{\partial U_0} & \frac{\partial R_3}{\partial U_1} & \frac{\partial R_3}{\partial P} & \frac{\partial R_4}{\partial \theta}
\end{bmatrix}
\]

Using the stress identity the entries of the solid stress tensor will be replaced with

\[
\begin{bmatrix}
\sigma_{00} & \sigma_{01} \\
\sigma_{10} & \sigma_{11}
\end{bmatrix} = \begin{bmatrix}
\tilde{\sigma}_{00} & \tilde{\sigma}_{01} \\
\tilde{\sigma}_{10} & \tilde{\sigma}_{11}
\end{bmatrix} + k \mu \frac{1}{2} \begin{bmatrix}
\frac{\partial U_0}{\partial x_0} & \frac{\partial U_1}{\partial x_0} + \frac{\partial U_0}{\partial x_1} \\
\frac{\partial U_1}{\partial x_0} & \frac{\partial U_1}{\partial x_1} + \frac{\partial U_1}{\partial x_1}
\end{bmatrix} +
\]

\[
k \begin{bmatrix}
\frac{\partial U_0}{\partial x_0} & \frac{\partial U_0}{\partial x_1} \\
\frac{\partial U_1}{\partial x_0} & \frac{\partial U_1}{\partial x_1}
\end{bmatrix} \begin{bmatrix}
\sigma_{00} & \sigma_{01} \\
\sigma_{10} & \sigma_{11}
\end{bmatrix} + k \begin{bmatrix}
\sigma_{00} & \sigma_{01} \\
\sigma_{10} & \sigma_{11}
\end{bmatrix} \begin{bmatrix}
\frac{\partial U_0}{\partial x_0} & \frac{\partial U_1}{\partial x_0} \\
\frac{\partial U_0}{\partial x_1} & \frac{\partial U_1}{\partial x_1}
\end{bmatrix}
\]
where $\sigma_{ij}$ is the stress tensor from previous iteration to calculate jacobian entries as

$$
\frac{\partial R_0}{\partial U_0}(w) = \rho \frac{\partial w}{\partial t} + \rho \sum_i U_i \frac{\partial w}{\partial x_i} + \rho \frac{\partial U_0}{\partial x_0} - \theta \nu \sum_i \frac{\partial^2 w}{\partial x_i^2} - (1 - \theta) k \left[ \mu \frac{1}{2} \left( \frac{\partial^2 w}{\partial x_0^2} + \frac{\partial^2 w}{\partial x_1^2} \right) + \sigma_{00} \left( \frac{\partial^2 w}{\partial x_0^2} \right) + \sigma_{01} \left( \frac{\partial^2 w}{\partial x_0 \partial x_1} \right) + \sigma_{10} \left( \frac{\partial^2 w}{\partial x_1 \partial x_0} \right) + \sigma_{11} \left( \frac{\partial^2 w}{\partial x_1^2} \right) \right]
$$

$$
\frac{\partial R_0}{\partial U_1}(w) = \rho \frac{\partial U_0}{\partial x_1} - (1 - \theta) k \left[ \mu \frac{1}{2} \left( \frac{\partial^2 w}{\partial x_0 \partial x_1} \right) + \sigma_{10} \left( \frac{\partial^2 w}{\partial x_0 \partial x_1} \right) + \sigma_{11} \left( \frac{\partial^2 w}{\partial x_1^2} \right) \right]
$$

$$
\frac{\partial R_0}{\partial P}(w) = \frac{\partial w}{\partial x_0}
$$

$$
\frac{\partial R_0}{\partial \theta}(w) = - \nu \sum_i \frac{\partial^2 U_0}{\partial x_i^2} + \sum_i \frac{\partial \sigma_{0i}}{\partial x_i}
$$

$$
\frac{\partial R_1}{\partial U_0}(w) = \rho \frac{\partial U_1}{\partial x_0} - (1 - \theta) k \left[ \mu \frac{1}{2} \left( \frac{\partial^2 w}{\partial x_0 \partial x_1} \right) + \sigma_{10} \left( \frac{\partial^2 w}{\partial x_0 \partial x_1} \right) + \sigma_{11} \left( \frac{\partial^2 w}{\partial x_1^2} \right) \right]
$$

$$
\frac{\partial R_1}{\partial U_1}(w) = \rho \frac{\partial U_1}{\partial x_1} - \theta \nu \sum_i \frac{\partial^2 w}{\partial x_i^2} - (1 - \theta) k \left[ \mu \frac{1}{2} \left( \frac{\partial^2 w}{\partial x_1^2} + \frac{\partial^2 w}{\partial x_0^2} \right) + \sigma_{00} \left( \frac{\partial^2 w}{\partial x_0^2} \right) + \sigma_{01} \left( \frac{\partial^2 w}{\partial x_0 \partial x_1} \right) + \sigma_{10} \left( \frac{\partial^2 w}{\partial x_0 \partial x_1} \right) + \sigma_{11} \left( \frac{\partial^2 w}{\partial x_1^2} \right) \right]
$$

$$
\frac{\partial R_1}{\partial P}(w) = \frac{\partial w}{\partial x_1}
$$

$$
\frac{\partial R_1}{\partial \theta}(w) = - \nu \sum_i \frac{\partial^2 U_1}{\partial x_i^2} + \sum_i \frac{\partial \sigma_{1i}}{\partial x_i}
$$

$$
\frac{\partial R_2}{\partial U_0}(w) = \frac{\partial w}{\partial x_0}
$$

$$
\frac{\partial R_2}{\partial U_1}(w) = \frac{\partial w}{\partial x_1}
$$

$$
\frac{\partial R_2}{\partial P}(w) = 0
$$

$$
\frac{\partial R_2}{\partial \theta}(w) = 0
$$

$$
\frac{\partial R_3}{\partial U_0}(w) = \frac{\partial \theta}{\partial x_0}
$$

$$
\frac{\partial R_3}{\partial U_1}(w) = \frac{\partial \theta}{\partial x_1}
$$

$$
\frac{\partial R_3}{\partial P}(w) = 0
$$

$$
\frac{\partial R_3}{\partial \theta}(w) = \frac{\partial w}{\partial t} + \sum_i U_i \frac{\partial w}{\partial x_i}
$$
4.4 Formulation of Dual Linearized problem

Since \((Ae, φ) = (e, A^*φ)\), entries of \(A^*\) can be constructed with integration by parts. After doing so, the entries of the matrix \(A^* = a^*_{ij}\) is found as:

\[-a_{00}^*(v) = -\frac{∂ρ}{∂t} - \nabla.(ρ(U)v) + \frac{∂U_0}{∂x_0}v - ν\nabla(θv) - k[μ \frac{1}{2}(2 \frac{∂^2(1-θ)v}{∂x_0^2} + \frac{∂^2(1-θ)v}{∂x_1^2}) + (2 \frac{∂^2(1-θ)σ_{00}v}{∂x_0^2}) + (2 \frac{∂^2(1-θ)σ_{01}v}{∂x_0∂x_1})]
\]

\[-a_{10}^*(v) = \frac{∂U_0}{∂x_1}v - k[μ \frac{1}{2}(2 \frac{∂^2(1-θ)v}{∂x_1∂x_0}) + (2 \frac{∂^2σ_{10}(1-θ)v}{∂x_0^2}) + (2 \frac{∂^2σ_{11}(1-θ)v}{∂x_1∂x_0})]
\]

\[-a_{20}^*(v) = -\frac{∂v}{∂x_0}
\]

\[-a_{30}^*(v) = -ν\nabla U_0v + \sum_i \frac{∂σ_{0i}}{∂x_i}v
\]

\[-a_{01}^*(v) = \frac{∂U_1}{∂x_0}v - k[μ \frac{1}{2}(2 \frac{∂^2(1-θ)v}{∂x_0∂x_1}) + (2 \frac{∂^2σ_{10}(1-θ)v}{∂x_1∂x_0}) + (2 \frac{∂^2σ_{11}(1-θ)v}{∂x_0∂x_1})]
\]

\[-a_{11}^*(v) = -\frac{∂ρ}{∂t} - \nabla.ρ(U)v + \frac{∂U_1}{∂x_1}v - ν\nabla(θv) - k[μ \frac{1}{2}(2 \frac{∂^2(1-θ)v}{∂x_1^2} + \frac{∂^2(1-θ)v}{∂x_0^2}) + (2 \frac{∂^2σ_{00}(1-θ)v}{∂x_0^2}) + (2 \frac{∂^2σ_{01}(1-θ)v}{∂x_0∂x_1}) + (2 \frac{∂^2σ_{10}(1-θ)v}{∂x_0∂x_1}) + (2 \frac{∂^2σ_{11}(1-θ)v}{∂x_1^2})]
\]
\[-a_{21}^*(v) = - \frac{\partial v}{\partial x_1}\]
\[-a_{31}^*(v) = - \nu \Delta U_1 v + \sum_i \frac{\partial \sigma_{1i}}{\partial x_i} v\]
\[-a_{02}^*(v) = - \frac{\partial v}{\partial x_0}\]
\[-a_{12}^*(v) = - \frac{\partial v}{\partial x_1}\]
\[-a_{22}^*(v) = 0\]
\[-a_{32}^*(v) = 0\]
\[-a_{03}^*(v) = \frac{\partial \theta}{\partial x_0} v\]
\[-a_{13}^*(v) = \frac{\partial \theta}{\partial x_1} v\]
\[-a_{23}^*(v) = 0\]
\[-a_{33}^*(v) = - \frac{\partial v}{\partial t} - \nabla.(U) v\]

Now for \(\zeta = (\zeta_0, \zeta_1, \zeta_2, \zeta_3)^T\) we will have the following equations for dual problem:

\[-\psi_0 = - \frac{\partial \rho \zeta_0}{\partial t} - \nabla.(\rho(U) \zeta_0) + \rho \frac{\partial U_0}{\partial x_0} \zeta_0 - \nu \Delta (\theta \zeta_0) - \]
\[k \left[ \frac{1}{2} \left( \frac{\partial^2 (1 - \theta) \zeta_0}{\partial x_0^2} + \frac{\partial^2 (1 - \theta) \zeta_0}{\partial x_1^2} \right) + \left( \frac{\partial^2 (1 - \theta) \sigma_{00} \zeta_0}{\partial x_0^2} \right) + \left( \frac{\partial^2 (1 - \theta) \sigma_{01} \zeta_0}{\partial x_0 \partial x_1} \right) + \right.\]
\[\left. \frac{\partial^2 (1 - \theta) \sigma_{10} \zeta_0}{\partial x_0 \partial x_1} + \frac{\partial^2 (1 - \theta) \sigma_{11} \zeta_0}{\partial x_1^2} \right]
\[+ \rho \frac{\partial U_1}{\partial x_0} \zeta_1 - k \left[ \frac{1}{2} \left( \frac{\partial^2 (1 - \theta) \zeta_1}{\partial x_0^2} \right) + \left( \frac{\partial^2 \sigma_{10} (1 - \theta) \zeta_1}{\partial x_0^2} \right) + \left( \frac{\partial^2 \sigma_{11} (1 - \theta) \zeta_1}{\partial x_0 \partial x_1} \right) \right]\]
\[+ \frac{\partial \zeta_2}{\partial x_0} + \frac{\partial \theta}{\partial x_0} \zeta_3\]
4.5 Boundary conditions for Dual Linearized Problem

The dual problem will be constructed such that the boundary terms in integration by parts will vanish. It will be necessary to solve the dual problem backwards in time $A^*\zeta = \psi$ to find $\zeta$ such that the equality $(\tilde{e}, \psi) = (\tilde{e}, A^*\zeta) = (A\tilde{e}, \zeta) = (-R(\tilde{U}), \zeta) = (h(-R(\tilde{U})), h^{-1}(\zeta - \Pi h\zeta)) + (-R(\tilde{U}), \Pi h\zeta)$ can be used to bound the error in drag.

Doing integration by parts on each term we see that the bilinear identity $0 = (Au, v) - (u, A^*v)$ holds only if the dual problem is set up this way:

\[
A^*\zeta(x, t) = \psi(x, t) \quad (x, t) \in \Omega \times (0, T) \\
\zeta_0(., T) = 0 \\
\zeta_1(., T) = 0 \\
\zeta_0(x, .) = 0 \quad x \in \partial\Omega \\
\zeta_1(x, .) = 0 \quad x \in \partial\Omega \\
\zeta_3(., T) = 0 \\
\zeta_3(x, .) = 0 \quad x \in \partial\Omega
\]
4.6 Formulation of Residual

For the 2D system the strong residual is:

\[-R(U, f, \rho, \theta, \sigma)_0 = -f_0 + \rho \frac{\partial U_0}{\partial t} + \rho(U) \nabla U_0 - \theta \nu \triangle U_0 - \sum_i (1 - \theta) \frac{\partial \sigma_{0i}}{\partial x_i}\]

\[-R(U, f, \rho, \theta, \sigma)_1 = -f_1 + \rho \frac{\partial U_1}{\partial t} + \rho(U) \nabla U_1 - \theta \nu \triangle U_1 - \sum_i (1 - \theta) \frac{\partial \sigma_{1i}}{\partial x_i}\]

\[-R(U, f, \rho, \theta, \sigma)_2 = \nabla U\]

\[-R(U, f, \rho, \theta, \sigma)_3 = \frac{\partial \theta}{\partial t} + (U) \nabla \theta\]

The calculated residual will be a piecewise discontinuous vector function in space and continuous in time. We know that \(U\) is a piecewise continuous linear function in space, while calculating \(\triangle U_i\) we will need an integration by parts over the cells which will introduce jumps at the facets, the same is also valid for stress which is a piecewise discontinuous constant function. If \(C\) is the cell size, \(\Omega\) is the domain of cell, \(\partial \Omega\) is the cell boundary, for the cell that contains \(x\), the residual for a cell at a certain time will be computed as:

\[-C \ast R(x, t)_0 = -\frac{1}{2} \int_{\Omega} f_0 dx + \frac{1}{\Delta t} \int_{\Omega} U_0^{n+1} - U_0^n dx + \rho \int_{\Omega} (\bar{U} - \bar{W}) \nabla \bar{U}_0 dx +\]

\[\int_{\partial \Omega} \frac{\partial P}{\partial x_0} dx + \frac{1}{2} \int_{\partial \Omega} \theta \nu [\nabla \bar{U}_0 \cdot n] dS + \frac{1}{2} \int_{\partial \Omega} (1 - \theta) [\sum_i \sigma_{0i} n_i] dS\]

\[-C \ast R(x, t)_1 = -\int_{\Omega} f_1 dx + \frac{1}{\Delta t} \rho \int_{\Omega} U_1^{n+1} - U_1^n dx + \rho \int_{\Omega} (\bar{U} - \bar{W}) \nabla \bar{U}_1 dx +\]

\[\int_{\partial \Omega} \frac{\partial P}{\partial x_1} dx + \frac{1}{2} \int_{\partial \Omega} \theta \nu [\nabla \bar{U}_1 \cdot n] dS + \frac{1}{2} \int_{\partial \Omega} (1 - \theta) [\sum_i \sigma_{1i} n_i] dS\]

\[-C \ast R(x, t)_2 = \int_{\Omega} \nabla \bar{U} dx\]

\[-C \ast R(x, t)_3 = \frac{1}{\Delta t} \int_{\Omega} \theta^{n+1} - \theta^n dx + \int_{\Omega} (\bar{U}) \nabla \theta dx\]

Where \([y]\) is difference in limit values \(y(\cdot^+) - y(\cdot^-)\) of \(y\) between 2 cells sharing the same facet. The 1/2 in front of jump integrals exist since each jump is calculated in each 2 cells sharing same facet. \(y^n\) represents the value of \(y(t_n)\) and \(\bar{y}\) represents the value \(\frac{1}{2}(y(t_{n+1}) + y(t_n))\). Again since the mesh changes at each time step, projection is necessary and errors from projection is neglected for this work.
Chapter 5

Numerical Results

5.1 Mesh Adaptivity Results in 3D

In order to test MAdLib integration in 3D, a problem similar to [12] where 2 ellipsoid particles sediment in a container is set up. Viscosity of the fluid is chosen to be $v_f = 0.01$, density of solid is chosen as $\rho_s = 2$, density of fluid is $\rho_f = 1$, and shear modulus is taken as $\mu_s = 1.0 \times 10^6$. When the gravitational force is chosen to be $f = (0, -1.0 \times 10^3, 0)^T$ a Reynolds number around $Re = 850$ is experienced which is of course much higher than the Reynolds number in original setup. The results however show that it is possible to obtain high quality meshes with this method even with very large displacements, which is not possible with the mesh smoothing method. The mesh size has been kept smaller in a region which contains sedimenting ellipsoids and bigger in the remaining part of the domain.
Figure 5.1. 3D benchmark initial mesh state for slice in \( +z \) direction from origin

Figure 5.2. 3D benchmark mesh state after large displacement for slice in \( +z \) direction from origin
5.2 Benchmark Problem in 2D

The FSI3 benchmark from [6] is implemented and results are compared to the results in papers [6, 5]. The geometry is a 2D channel of 0.41 m height and 2.5 m length with a circular obstacle of radius 0.05 with its center placed at (0.2, 0.2) and a rectangular elastic structure fixed to the obstacle at (0.2, 0.25) which has length 0.35 and height 0.02 as shown in the following figure. The Reynolds number in the experiment is 100.

![Figure 5.3. 2D Geometry](image)

The boundary conditions are given as:

A parabolic velocity profile at the left channel inflow

\[ u_0(0, y) = 1.5 \bar{U} \frac{4.0}{0.1681} y(0.41 - y) \]

No Slip condition at upper and lower walls. The outflow condition choice is left to the user and in the current implementation zero pressure outflow condition is chosen. Other parameters for FSI3 experiment are

\[ \rho_s = 1000 \text{kg/m}^3, \nu_s = 0.4, \mu_s = 2 \times 10^6 \text{kg/m}^2, \rho_f = 1000 \text{kg/m}^3, \nu_f = 0.001 \text{m}^2/\text{s}, \bar{U} = 2 \text{m/s} \]

The original parameters are non dimensionalized in implementation. Parameters that have kg in dimension have also been scaled with 0.001 in the implementation.

5.3 Results for 2D Benchmark

The oscillation of the initial point at (0.2, 0.55) in y direction in paper [6] is shown below:
The same experiment is also simulated in [5] with the following results for vertical displacement in structure

The initial point at $(0.2, 0.55)$ in [6] has $y$ coordinate of $1.53 \pm 34.35 \times 10^{-3}$ for the finest mesh using 304128 degrees of freedom.

The implementation was tested on a fine(approximately 5500 vertices, 10500 cells) and coarse mesh(approximately 3500 vertices 6500 cells) resulting oscillations with very close frequency and amplitude to the former computations. The convergence to the results to former computations should also be noted with decreasing mesh size.

On current implementation the initial point at $(0.2, 0.55)$ in has been found to have $y$ coordinate of $7.31 \pm 26.37 \times 10^{-3}$ for the coarse mesh with approximately
5.3. RESULTS FOR 2D BENCHMARK

6500 cells. Using the fine mesh approximately 10500 cells $1.49 \pm 37.16 \times 10^{-3}$, a value which is quite close to the results of [6].

Figure 5.6. FSI3 Structure oscillation(y) results for a coarse mesh(approx. 3500 vert 6500 cells)

Figure 5.7. FSI3 Structure oscillation(y) results for a fine mesh(approx. 5500 vert 10500 cells)
Figure 5.8. FSI3 velocity magnitude for fine mesh at t=2

Figure 5.9. FSI3 pressure for fine mesh at t=2

Figure 5.10. FSI3 velocity magnitude for fine mesh at t=4.1

Figure 5.11. FSI3 pressure for fine mesh at t=4.1
Chapter 6

Conclusion and Future Work

Successful computational results for the mesh adaptivity through refinement, coarsening and swapping operations have been obtained. The theoretical background for mesh adaptation using linearized dual problem for error in functionals of interest is given. The implementation for the latter subject is still ongoing and is going to be the natural follow up of the current work with computational results.
Appendix A

Source Code

In this chapter necessary parts of the code is given. The reader can access Unicorn library from [19]
A.1 MAdLib Interface

A.1.1 Converting MDBmesh to Dolfin Mesh Format

```cpp
void MeshAdaptInterface::importMDBMesh()
{
  deleteIdMappings();

  int dim = M_dim(MDBMesh);
  int numVertices = M_numVertices(MDBMesh);
  int numCells = 0;

  CellType * cell_type;
  switch (dim) {
    case 3:
      cell_type = CellType::create(CellType::tetrahedron);
      numCells = M_numRegions(MDBMesh);
      break;
    case 2:
      cell_type = CellType::create(CellType::triangle);
      numCells = M_numFaces(MDBMesh);
      break;
    case 1:
      cell_type = CellType::create(CellType::interval);
      numCells = M_numEdges(MDBMesh);
      break;
    case 0:
      cell_type = CellType::create(CellType::point);
      numCells = M_numVertices(MDBMesh);
      break;
  }

  // --- create the mesh ---
  MeshEditor editor;
  editor.open(*dMesh, cell_type->cellType(), dim, dim);
  editor.initVertices(numVertices);
  editor.initCells(numCells);

  // --- add vertices ---
  int current_vertex = 0;
  if (locToMDBIds)
  {
    delete[] locToMDBIds;
    locToMDBIds = 0;
    locToMDBIds = new int[numVertices];
    VIter vit = M_vertexIter(MDBMesh);
    while (pVertex pv = VIter_next(vit))
    {
      double xyz[3];
      V_coord(pv,xyz);
      Point p(xyz[0],xyz[1],xyz[2]);
      editor.addVertex(current_vertex, p);
      pPoint pp = V_point(pv);
      cell_vertices[current_vertex] = MDBToLocIds(P_id(pp));
      current_vertex++;
    }
    VIter_delete(vit);
  }

  // --- create reverse vertex id mapping ---
  for (int i=0; i<numVertices; i++)
  {
    MDBToLocIds[locToMDBIds[i]] = i;
  }

  // --- add cells ---
  int current_cell = 0;
  if (MDBToLocCellIds)
  {
    delete[] MDBToLocCellIds;
    MDBToLocCellIds = 0;
    MDBToLocCellIds = new int[numCells];
    if (dim==3)
    {
      pPoint pp = V_point(pv);
      cell_vertices[iV] = MDBToLocIds(P_id(pp));
      editor.addCell(current_cell, cell_vertices);
      MDBToLocCellIds[current_cell] = current_cell;
      // if dolfin numbers cells in order of creation -> ?
      current_cell++;
    }
    RIter_delete(rit);
  }
  else if (dim==2)
  {
    FIter fit = M_faceIter(MDBMesh);
    while (pFace pf = FIter_next(fit))
    {
      Array<uint> cell_vertices(cell_type->numEntities(0));
      for (int iV=0; iV<F_numVertices(pf); iV++)
      {
        pVertex pv = F_vertex(pf, iV);
        pPoint pp = V_point(pv);
        cell_vertices[iV] = MDBToLocIds(P_id(pp));
      }
      editor.addCell(current_cell, cell_vertices);
      MDBToLocCellIds[current_cell] = current_cell;
      // if dolfin numbers cells in order of creation -> ?
      current_cell++;
    }
    FIter_delete(fit);
  }
  else if (dim==1)
  {
    EIter eit = M_edgeIter(MDBMesh);
    while (pEdge pe = EIter_next(eit))
    {
      Array<uint> cell_vertices(cell_type->numEntities(0));
      for (int iV=0; iV<2; iV++)
      {
        pVertex pv = E_vertex(pe, iV);
        pPoint pp = V_point(pv);
        cell_vertices[iV] = MDBToLocIds(P_id(pp));
      }
      editor.addCell(current_cell, cell_vertices);
      MDBToLocCellIds[current_cell] = current_cell;
      // if dolfin numbers cells in order of creation -> ?
      current_cell++;
    }
    EIter_delete(eit);
  }
  else
  {
    VIter vit = M_vertexIter(MDBMesh);
    while (pVertex pv = VIter_next(vit))
    {
      double xyz[3];
      V_coord(pv,xyz);
      Point p(xyz[0],xyz[1],xyz[2]);
      editor.addVertex(current_vertex, p);
      pPoint pp = V_point(pv);
      cell_vertices[current_vertex] = P_id(pp);
      current_vertex++;
    }
    VIter_delete(vit);
  }
}
```
A.1. MADLIB INTERFACE

A.1.2 Converting Dolfin Mesh to MDBmesh Format

```cpp
void MeshAdaptInterface::exportToMDBMesh()
{
  if (!MDBMesh->model)  MDBMesh->model = new NullModel;
  std::cout << "Num pts: " << dMesh->numVertices() << ", num cells: " << dMesh->numCells() << std::endl;
  delete dMappings();
  uint* cells = dMesh->cells();
  int numCellV = dMesh->type().numEntities(0);
  MAdGEntity *geom = 0;
  if (locToMDBIds)
    delete[] locToMDBIds;
  locToMDBIds = 0;
  locToMDBIds = new int[dMesh->numVertices()];
  if (MDBToLocCellIds)
    delete[] MDBToLocCellIds;
  MDBToLocCellIds = 0;
  switch(dMesh->type().cellType()) {
    case CellType::tetrahedron:
      real* xyz = dMesh->coordinates();
      for (int iV=0; iV < dMesh->numVertices(); iV++) {
        MDBMesh->add_point(iV+1,xyz[3*iV],xyz[3*iV+1],xyz[3*iV+2]);
        locToMDBIds[iV] = iV+1;
        geom = MDBMesh->model->getRegionByTag(1);
        for (int iC=0; iC < dMesh->numCells(); iC++) {
          MDBMesh->add_tet(cells[iC*numCellV] +1,
                           cells[iC*numCellV+1]+1,
                           cells[iC*numCellV+2]+1,
                           cells[iC*numCellV+3]+1,
                           geom);
          MDBToLocCellIds[iC] = iC; // ok if the id in dolfin is iC !
        }
      }
      break;
    case CellType::triangle:
      real* xyz = dMesh->coordinates();
      for (int iV=0; iV < dMesh->numVertices(); iV++) {
        MDBMesh->add_point(iV+1,xyz[iV],xyz[iV+1],0.);
        locToMDBIds[iV] = iV+1;
        geom = MDBMesh->model->getFaceByTag(1);
        for (int iC=0; iC < dMesh->numCells(); iC++) {
          pPoint pp = MDBMesh->find_point(cells[iC*numCellV]+1);
          pp->g = geom;
          MDBToLocCellIds[iC] = iC; // ok if the id in dolfin is iC !
        }
      }
      break;
    case CellType::interval:
      real* xyz = dMesh->coordinates();
      for (int iV=0; iV < dMesh->numVertices(); iV++) {
        MDBMesh->add_point(iV+1,xyz[iV],0.,0.);
        locToMDBIds[iV] = iV+1;
        geom = MDBMesh->model->getEdgeByTag(1);
        for (int iC=0; iC < dMesh->numCells(); iC++) {
          MDBMesh->add_edge(cells[iC*numCellV] +1,
                            cells[iC*numCellV+1]+1,
                            geom);
          MDBToLocCellIds[iC] = iC; // ok if the id in dolfin is iC !
        }
      }
      break;
    case CellType::point:
      real* xyz = dMesh->coordinates();
      for (int iV=0; iV < dMesh->numVertices(); iV++) {
        MDBMesh->add_point(iV+1,xyz[iV],0.,0.);
        locToMDBIds[iV] = iV+1;
        geom = MDBMesh->model->getVertexByTag(1);
        for (int iC=0; iC < dMesh->numCells(); iC++) {
          pPoint pp = MDBMesh->find_point(cells[iC*numCellV]+1);
          pp->g = geom;
          MDBToLocCellIds[iC] = iC; // ok if the id in dolfin is iC !
        }
      }
      break;
  }
  MDBMesh->classify_unclassified_entities();
  MDBMesh->destroyStandAloneEntities();
  // --- create reverse vertex id mapping ---
  for (int i=0; i<dMesh->numVertices(); i++) {
    MDBToLocIds[locToMDBIds[i]] = i;
  }
  getBoundaryVertices();
}
```
A.2 Interpolation

```cpp
void ICNSPDE::retrieveSolution()
{
    P.vector().vec().set(pArrNew);
    Uptr->vector().vec().set(uArrNew);
    h0f->vector().vec().set(h0fArrNew);
    // phiPtr->vector().vec().set(dPhiArrNew);
    Sptr->vector().vec().set(dSArrNew);
}
/* virtual */
void ICNSPDE::projectSolution()
{
    int dim = M_dim(MDBMesh);
    int numVertices = M_numVertices(MDBMesh);
    int numCells = 0;
    if (dim == 3)
    {
        numCells = M_numRegions(MDBMesh);
    }
    else
    {
        numCells = M_numFaces(MDBMesh);
    }

    // prepare the arrays to contain functions
    double dPointCoords[3];
    real rPointCoords[3];
    real pointValues[9];
    if (uArrNew)
    delete []uArrNew;
    uArrNew = new real[numVertices * dim];
    if (h0fArrNew)
    delete []h0fArrNew;
    h0fArrNew = new real[numVertices];
    if (pArrNew)
    delete []pArrNew;
    pArrNew = new real[numVertices];

    // first the continuous functions
    VIter vit = M_vertexIter(MDBMesh);
    int count = 0;
    while (pVertex pv = VIter_next(vit))
    {
        V_coord(pv,dPointCoords);
        rPointCoords[0] = dPointCoords[0];
        rPointCoords[1] = dPointCoords[1];
        rPointCoords[2] = dPointCoords[2];
        Uptr->eval(pointValues, rPointCoords);
        for (int i = 0; i < dim; i++)
            uArrNew[i*numVertices + count] = pointValues[i];
        h0f->eval(pointValues, rPointCoords);
        h0fArrNew[count] = pointValues[0];
        P.eval(pointValues, rPointCoords);
        pArrNew[count] = pointValues[0];
        count++;
    }

    // find mid point for the cell
    double midX;
    double midY;
    double midZ;
    midX = 0;
    midY = 0;
    midZ = 0;

    for (int iV = 0; iV < R_numVertices(pr); iV++)
    {
        pVertex pv = R_vertex(pr,iV);
        V_coord(pv,dPointCoords);
        midX += dPointCoords[0];
        midY += dPointCoords[1];
        midZ += dPointCoords[2];
    }
    midX /= R_numVertices(pr);
    midY /= R_numVertices(pr);
    midZ /= R_numVertices(pr);
    rPointCoords[0] = midX;
    rPointCoords[1] = midY;
    rPointCoords[2] = midZ;

    // interpolate value for the mid point
    Sptr->eval(pointValues, rPointCoords);
    for (int i = 0; i < dim * dim; i++)
        dSArrNew[i*numCells + count] = pointValues[i];
    count++;
    RIter_delete(rit);
}
```

```cpp
void ICNSPDE::retrieveSolution()
{
    P.vector().vec().set(pArrNew);
    Uptr->vector().vec().set(uArrNew);
    h0f->vector().vec().set(h0fArrNew);
    // phiPtr->vector().vec().set(dPhiArrNew);
    Sptr->vector().vec().set(dSArrNew);
}
/* virtual */
void ICNSPDE::projectSolution()
{
    int dim = M_dim(MDBMesh);
    int numVertices = M_numVertices(MDBMesh);
    int numCells = 0;
    if (dim == 3)
    {
        numCells = M_numRegions(MDBMesh);
    }
    else
    {
        numCells = M_numFaces(MDBMesh);
    }

    // prepare the arrays to contain functions
    double dPointCoords[3];
    real rPointCoords[3];
    real pointValues[9];
    if (uArrNew)
    delete []uArrNew;
    uArrNew = new real[numVertices * dim];
    if (h0fArrNew)
    delete []h0fArrNew;
    h0fArrNew = new real[numVertices];
    if (pArrNew)
    delete []pArrNew;
    pArrNew = new real[numVertices];

    // first the continuous functions
    VIter vit = M_vertexIter(MDBMesh);
    int count = 0;
    while (pVertex pv = VIter_next(vit))
    {
        V_coord(pv,dPointCoords);
        rPointCoords[0] = dPointCoords[0];
        rPointCoords[1] = dPointCoords[1];
        rPointCoords[2] = dPointCoords[2];
        Uptr->eval(pointValues, rPointCoords);
        for (int i = 0; i < dim; i++)
            uArrNew[i*numVertices + count] = pointValues[i];
        h0f->eval(pointValues, rPointCoords);
        h0fArrNew[count] = pointValues[0];
        P.eval(pointValues, rPointCoords);
        pArrNew[count] = pointValues[0];
        count++;
    }

    // find mid point for the cell
    double midX;
    double midY;
    double midZ;
    midX = 0;
    midY = 0;
    midZ = 0;

    for (int iV = 0; iV < R_numVertices(pr); iV++)
    {
        pVertex pv = R_vertex(pr,iV);
        V_coord(pv,dPointCoords);
        midX += dPointCoords[0];
        midY += dPointCoords[1];
        midZ += dPointCoords[2];
    }
    midX /= R_numVertices(pr);
    midY /= R_numVertices(pr);
    midZ /= R_numVertices(pr);
    rPointCoords[0] = midX;
    rPointCoords[1] = midY;
    rPointCoords[2] = midZ;

    // interpolate value for the mid point
    Sptr->eval(pointValues, rPointCoords);
    for (int i = 0; i < dim * dim; i++)
        dSArrNew[i*numCells + count] = pointValues[i];
    count++;
    RIter_delete(rit);
}
```
```python
# Copyright (c) 2005 Johan Jansson
# Licensed under the GNU GPL Version 2

cell = "triangle"
K = FiniteElement("Lagrange", cell, 1)
S = FunctionSpace(K)
K2 = FiniteElement("Discontinuous Lagrange", cell, 0)
S2 = FunctionSpace(K2)

# CG1 residual version without time derivative term
def RCG1(u,p) :
    return 0.5*( mult(rho, dot(UPale, grad(u)) + q.dx(0)) +
                mult(rho, dot(UPale, grad(u)) + q.dx(0)) )

# u derivative without time derivative term
def dRCG1du1(v) :
    return 0.5*( mult(rho, dot(UPale, grad(u)) + q.dx(0)) )

# p derivative
def dRCG1dp(p) :
    return ( -dot(grad(p), v) - mult(d1, dot(grad(p), grad(v)) + q.dx(0))))
```

A.3. Solver

A.3.1 Optimized Form File
A.3.2 Form File with All Terms

```python
# 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 the incompressible Navier-Stokes equations
# Compile this form with FFC: ffc Elasticity.form.

cell = "triangle"

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, d * d)
K5 = VectorElement("Discontinuous Lagrange", cell, 0, d * d)

K = K1
(v,q) = TestFunctions(K1 + K2)
(U1,P) = TrialFunctions(K1 + K2)
UP = Function(K)
U0 = Function(K)
UPm = Function(K4)
WP = Function(K)
WPm = Function(K4)
nu = Function(K3)
d1 = Function(K3)
#d2 = Function(K3)
#d2 = 0
rho = Function(K3)
phi = Function(K3)
sigma = Function(K5)
ff = Function(K)
mu = Function(K3)
lmbda = Function(K3)
k = Function(K3)
P0 = Function(K2)
PF = Function(K2)

sigmaM = tomatrix(sigma)
S0 = E(epsil0(u0), mu, lmbda)
S2 = E(epsil0(u2), mu, lmbda)
def convStab1(w,u):
    return (mult(d1, dot(x, ugrad(u-WP,v))) + mult(d1, dot(x, grad(q)))))

def RCG1(u1,u2,P):
    return (mult(d1, dot(ff, v) + mult(d1, dot(ff, ugrad(u-WP,v)))) + mult(d1, dot(ff, grad(q))))
```

---

APPENDIX A. SOURCE CODE
A.3. SOLVER

return mult(rho * 0.125, w * (U00.dx(0))*v0 + w*(U01.dx(0))*v1)
def f5th6thConvTermDerivative_u1(w,v0,v1):
    return mult(rho * 0.125, w*(U00.dx(0))*v0 + w*(U01.dx(0))*v1)
# u1 derivative without time derivative term
def dRCG1du1(w):
    # force term
    return (mult(rho * 0.5 * d1, w*ff[0]*(v0.dx(0)) + w*ff[1]*(v1.dx(0)))) -
             mult(d1 * rho * 0.125, dot(UP - WP, grad(UP0)) * w * (v0.dx(0)) + dot(UP - WP, grad(UP1)) * w * (v1.dx(0))) -
             mult(d1, f1st2ndConvTermDerivative_u1(w, dot(UP-WP, grad(v0))+(q.dx(0)), dot(UP-WP, grad(v1))+(q.dx(1)))) -
            # second term w/o stab part
            f2nd3rdConvTermDerivative_u1(w,v0,v1) -
             # second term stab part
            mult(d1, f3rd4thConvTermDerivative_u1(w, dot(UP-WP, grad(v0))+(q.dx(0)), dot(UP-WP, grad(v1))+(q.dx(1)))) -
            # fourth term w/o stab part
            f3rd4thConvTermDerivative_u1(w,v0,v1) -
            # fourth term stab part
            mult(d1, f3rd4thConvTermDerivative_u1(w, dot(UP-WP, grad(v0))+(q.dx(0)), dot(UP-WP, grad(v1))+(q.dx(1)))) -
            # fifth term w/o stab part
            f3rd4thConvTermDerivative_u1(w,v0,v1) -
            # fifth term stab part
            mult(d1, f3rd4thConvTermDerivative_u1(w, dot(UP-WP, grad(v0))+(q.dx(0)), dot(UP-WP, grad(v1))+(q.dx(1)))) -
            # sixth term w/o stab part
            f5th6thConvTermDerivative_u1(w,v0,v1) -
            # sixth term stab part
            mult(d1, f5th6thConvTermDerivative_u1(w, dot(UP-WP, grad(v0))+(q.dx(0)), dot(UP-WP, grad(v1))+(q.dx(1)))) -
            # seventh term only adds stability
            mult(d1 * rho * 0.125, w* (v0.dx(0)) + w*(U00.dx(0))*d1)
            # eighth term adds nothing
            mult(0.5 * w*(v0.dx(0)), q) -
            mult(nu * phi, (v0.dx(0))) -
            mult(0.5 * phi, (v0.dx(1))) -
            mult(1 - phi, k * mu * (v0.dx(0)) +
             w.dx(0) + 0.5 * w.dx(1) * (v0.dx(1) + v1.dx(0)) ) +
             k * v0.dx(0) * (v0.dx(0) +
             s00 + w.dx(1) * (s10 + s01) ) +
             k * v0.dx(1) * (s01 * w.dx(0) + s11 * w.dx(1)) +
             k * v1.dx(0) * (s10 * w.dx(0) + s11 * w.dx(1)))
def f1st2ndConvTermDerivative_u2(w,v0,v1):
    return mult(rho * 0.125, w* (U00.dx(0))*v0 +
             UPA0*(w.dx(0))*v1 + UPA1*(w.dx(1))*v1 +
             w*(U11.dx(1))*v1)
def f5th6thConvTermDerivative_u2(w,v0,v1):
    return mult(rho * 0.125, dot(00 - WP, grad(v1)) +
             w*(U11.dx(1))*v1)

mu = 0.5 * w*(v0.dx(0)) +
    mult(nu * phi, (v0.dx(0))) -
    mult(0.5 * phi, (v0.dx(1))) -
    mult(1 - phi, k * mu * (v0.dx(0)) +
    w.dx(0) + 0.5 * w.dx(1) * (v0.dx(1) + v1.dx(0)) ) +
    k * v0.dx(0) * (v0.dx(0) +
    s00 + w.dx(1) * (s10 + s01) ) +
    k * v0.dx(1) * (s01 * w.dx(0) + s11 * w.dx(1)) +
    k * v1.dx(0) * (s10 * w.dx(0) + s11 * w.dx(1)))
APPENDIX A. SOURCE CODE

dot((UP-WP,grad(v0))+(q.dx(0)), dot((UP-WP,grad(v1))+
(q.dx(1)) ))) -
# sixth term w.o stab part
fifth6thConvTermDerivative_u2(w,v0,v1) -
# sixth term stab part
mult(d1, fifth6thConvTermDerivative_u2(w,
dot((U0-WP,grad(v0))+(q.dx(0)), dot((U0-WP,grad(v1))+
(q.dx(1)) ))) -
# seventh term only adds stability
mult(d1*rho*0.125, dot(U0-
WP,grad(U00))*w*(v0.dx(1)) + dot(U0=
WP,grad(U01))*w*(v1.dx(1)))) -
# eighth term adds nothing
mult( 0.5 * w.dx(1) , q ) -
mult( nu * phi, w.dx(1) * v1.dx(1)) -
mult( nu * phi * 0.5, (w.dx(0)) *
(v0.dx(1) + v1.dx(0) ) ) -
mult( 1 - phi , k * mu * (0.5 *(v0.dx(1)
+ v1.dx(0) ) *w.dx(0) + w.dx(1)*v1.dx(1)) +
k * v0.dx(1) * (s00 *
w.dx(0) + s01* w.dx(1) ) +
 k * v1.dx(0) * (s00 *
w.dx(0) + s10* w.dx(1) ) +
 k * v1.dx(1) * (s01 +
s10)* w.dx(0) + 2* s11 * w.dx(1))))

# p derivative
def dRCG1dp(w):
    return (-
mult (0.5, dot(grad(w), v)  +
convStabl(grad(w), UP)) -
mult (0.5, dot(grad(w), v) +
convStabl(grad(w), U0))

JacX00= dRCG1du1(U1[0] ) - v0* rho*1/k*U1[0]
JacX01= dRCG1du2(U1[1]) - v1* rho* 1/k*U1[1]
JacX02= dRCG1du1(P) - v0* rho*1/k*U1[0]
JacX03= dRCG1du2(P) - v1* rho* 1/k*U1[1]

a = ( JacX00 + JacX01 + JacX02) * dx
L = -( RCG1(UP,U0,PP) - v0 * rho * (UP[0] - U0[0]) * 1/k - v1 * rho * (UP[1]-U0[1]) * 1/k ) * dx + (JacX00L
+ JacX01L + JacX02L )*dx
A.3. SOLVER

A.3.3 Main Loop

real ICNSPDE::iter()
{
    dx->vec() = x->vec();
    assembler->assemble(CombM, *aComb, reset_tensor);
    assembler->assemble(Combb, *LComb, reset_tensor);

    SubSystem velocity(0);
    SubSystem pressure(1);

    std::cout << "mesh vertices:" << mesh().numVertices() << std::endl;
    std::cout << "mesh cells:" << mesh().numCells() << std::endl;
    std::cout << "U size:" << U.vector().vec().size() << std::endl;

    LUSolver solverCB;
    // -> apply boundary conditions
    // -> apply momentum
    for (uint i=0; i < bc_mom.size();i++)
        bc_mom[i]->apply(CombM, Combb, *aComb);
    // -> apply continuity
    for (uint i=0; i < bc_con.size();i++)
        bc_con[i]->apply(CombM, Combb, *aComb);

    LUSolver solver;
    solver.solve(CombM, Combx, Combb);
    // -> Px should get the pressure
    // -> x should get the velocity
    Function vp;
    Function v;
    Function p;
    vp.init(mesh(), Combx, *aComb,1);
    v = vp[0];
    p = vp[1];

    x->vec() = v.vector().vec();
    x2->vec() = p.vector().vec();

    real *x2arr = x2->vec().array();
    real *parr = p.vector().vec().array();

    for (int i = 0; i < p.vector().vec().size(); i++)
        x2arr[i] = Parr[i];

    std::cout << std::endl;
    x2->vec().restore(x2arr);

    P.vector().vec() = p.vector().vec();
    PP.vector().vec() = p.vector().vec();
    x2_p->vec() = x2->vec();
    UP.vector().vec() = x->vec();

    dx->vec() = x->vec();
    real relincr = 9999;

    if(x->norm(linf)>0.0)
        relincr = dx->norm(linf) / x->norm(linf);

    // -> return the relative difference in x
    return relincr; //dx->norm(linf);
}
Bibliography


[13] FEnics Project, University of Chicago, Argonne National Laboratory, Delft University of Technology, Royal Institute of Technology KTH, Simula Research Laboratory, Texas Tech University, and University of Cambridge, http://www.fenics.org/wiki/FEniCS_Project


