Thesis Title

Institution Name

Author Name

Day Month Year

Innhold

Kapittel 1

Implementation of Fluid Structure Interaction

We will in this section give an overview of the total Fluid-Structure interaction implementation introduced in chapter 2. A brief description will be given for the most central components and technologies used for this thesis.

1.1 FEniCS

The main component of this thesis is the FEniCS project, an open-source finite element environment for solving partial differential equations (https://fenicsproject.org/). Using a combination of high-level Python and C++ interfaces, mathematical models can be implemented compactly and efficiently. FEniCS consists of several sub-modules and we will give a brief overview of the most central components used during implementation and computation.

1.1.1 DOLFIN

DOLFIN is the computational C++ backend of the FEniCS project, and the main user interface. It unifies several FEniCs components for implementing of computational mesh, function spaces, functions and finite element assembly.

- UFL (The Unified Form Language) is a domain specific language, used for the discretization of mathematical abstractions of partial differential equations on a finite element form. Its implementation on top of Python, makes it excellent to define problems close to their mathematical notation without the use of more complex features. One uses the term *form* to define any representation of some mathematical problem defined by UFL.

- FFC (The form compiler) compiles the finite elements variation forms given by UFL, generating low-level efficient C++ code

- FIAT the finite element backend, covering a wide range of finite element basis functions used in the discretization of of the the finite-element forms. It covers a wide range of finite element basis functions for lines, triangles and tetrahedras.

DOLFIN also incorporate the necessary interfaces to external linear algebra solvers and data structures. Within FEniCS terminology these are called linear algebra backends. PETSc is the default setting in FEniCS, a powerful linear algebra library with a wide range of parallel linear and nonlinear solvers and efficient as matrix and vector operations for applications written in C, C++, Fortran and Python.

1.2 Implementation

As implementation of mathematics differ from the choices of programming languages and external libraries, a deep dive within the implementation in FEniCS will not be covered in this thesis. Only variational forms and solvers will be presented as to give the reader a general overview of the key concept and the interpretation of mathematics. Basic knowledge of coding is assumed of the reader.

1.2.1 Variational Form

Implementation of the code-blocks of the fluid variational form given in Chapter 3, and Newton solver will be presented. It is not the intention to give the reader a deep review of the total implementation, but rather briefly point out key ideas intended for efficient speedup of the calculation. These ideas have proven essential as for the reduction of computation time of the complex problem.

```
def F_(U):
        return Identity(len(U)) + grad(U)

def J_(U):
        return det(F_(U))

def sigma_f_u(u,d,mu_f):
    return  mu_f*(grad(u)*inv(F_(d)) + inv(F_(d)).T*grad(u).T)

def sigma_f_p(p, u):
    return -p*Identity(len(u))

def A_E(J, v, d, rho_f, mu_f, psi, dx_f):
    return rho_f*inner(J*grad(v)*inv(F_(d))*v, psi)*dx_f \
        + inner(J*sigma_f_u(v, d, mu_f)*inv(F_(d)).T, grad(psi))*dx_f


def fluid_setup(v_, p_, d_, n, psi, gamma, dx_f, ds, mu_f, rho_f, k, dt, v_deg, theta, **semimp_namespace):

        J_theta = theta*J_(d_["n"]) + (1 - theta)*J_(d_["n-1"])
        F_fluid_linear = rho_f/k*inner(J_theta*(v_["n"] - v_["n-1"]), psi)*dx_f

        F_fluid_nonlinear =  Constant(theta)*rho_f*inner(J_(d_["n"])*grad(v_["n"])*inv(F_(d_["n"]))*v_["n"], psi)*dx_f
        F_fluid_nonlinear += inner(J_(d_["n"])*sigma_f_p(p_["n"], d_["n"])*inv(F_(d_["n"])).T, grad(psi))*dx_f
        F_fluid_nonlinear += Constant(theta)*inner(J_(d_["n"])*sigma_f_u(v_["n"], d_["n"], mu_f)*inv(F_(d_["n"])).T, grad(psi))*dx_f
        F_fluid_nonlinear += Constant(1 - theta)*inner(J_(d_["n-1"])*sigma_f_u(v_["n-1"], d_["n-1"], mu_f)*inv(F_(d_["n-1"])).T, grad(psi))*dx_f
        F_fluid_nonlinear +=inner(div(J_(d_["n"])*inv(F_(d_["n"]))*v_["n"]), gamma)*dx_f
        F_fluid_nonlinear += Constant(1 - theta)*rho_f*inner(J_(d_["n-1"])*grad(v_["n-1"])*inv(F_(d_["n-1"]))*v_["n-1"], psi)*dx_f
        F_fluid_nonlinear -= rho_f*inner(J_(d_["n"])*grad(v_["n"])*inv(F_(d_["n"]))*((d_["n"]-d_["n-1"])/k), psi)*dx_f

        return dict(F_fluid_linear = F_fluid_linear, F_fluid_nonlinear = F_fluid_nonlinear)
```

Algorithm 1.1: thetaCN.py

Alorithm 1.1 presents the implementation of the fluid residue, used in the Newton iterations. Apart from the rather lengthy form of the fluid residual, the strength of Unified Form Language preserving the abstract formulation of the problem is clear. The overall representation of the problem is by now just a form, its a representation and does not yet define vectors or matrices.

```python
def newtonsolver(F, J_nonlinear, A_pre, A, b, bcs, \
              dvp_, up_sol, dvp_res, rtol, atol, max_it, T, t, **monolithic):
    Iter      = 0
    residual   = 1
    rel_res    = residual
    lmbda = 1

    while rel_res > rtol and residual > atol and Iter < max_it:
        if Iter % 4  == 0:
            A = assemble(J_nonlinear, tensor=A, form_compiler_parameters = {"quadrature_degree": 4})
            A.axpy(1.0, A_pre, True)
            A.ident_zeros()

        b = assemble(-F, tensor=b)

        [bc.apply(A, b, dvp_["n"].vector()) for bc in bcs]
        up_sol.solve(A, dvp_res.vector(), b)
        dvp_["n"].vector().axpy(lmbda, dvp_res.vector())
        [bc.apply(dvp_["n"].vector()) for bc in bcs]
        rel_res = norm(dvp_res, 'l2')
        residual = b.norm('l2')
        if isnan(rel_res) or isnan(residual):
            print "type rel_res: ",type(rel_res)
            t = T*T
```

Algorithm 1.2: newtonsolver.py

1.3 Optimization of Newtonsolver

As for any program, the procedure of optimization involves finding the bottleneck of the implementation. Within computational science, this involves finding the area of code which is the primary consumer of computer resources.

As for many other applications, within computational science one can often assume the consummation of resources follows the *The Pareto principle*. Meaning that for different types of events, roughly 80% of the effects come from 20% of the causes. An analogy to computational sciences it that 80% of the computational demanding operations comes from 20% of the code. In our case, the bottleneck is the newtonsolver. The two main reasons for this is

- **Jacobian assembly**
  The construction of the Jacobian matrix for the total residue of the system, is the most time demanding operations within the whole computation.

- **Solver**.
  As iterative solvers are limited for the solving of fluid-structure interaction problems, direct solvers was implemented for this thesis. As such, the operation of solving a linear problem at each iteration is computational demanding, leading no less efficient operations.

Facing these problems, several attempts was made to speed-up the implementation. The FEniCS project consist of several nonlinear solver backends, were fully user-customization option are available. However one main problem which we met was the fact that FEniCS assembles the matrix of the different variables over the whole mesh, even though the variable is only defined in one to the sub-domains of the system.In our case the pressure is only defined within the fluid domain, and therefore the matrix for the total residual consisted of several zero columns within the structure region. FEniCS provides a solution for such problems, but therefore we were forced to construct our own solver and not make use of the built-in nonlinear solvers.

Of the speed-ups methods explored in this thesis we will specify that some of them were *consistent* while others were *nonconsistent.* Consistent methods are methods that always will work, independent of the problem to be solved. The non-consistent method presented are problem-specific, as these methods often involves some form of simplification of the system which is not rigid and may break down for some problems.

1.4 Consistent methods

Assembly of only non-linear Jacobi (jacobi-buffering ?)

1.5 Non-consisten methods

Reuce of Jacobian
Quadrature reduce
Simplification of jacobi a possibility, but not explored in this thesis.

Bibliografi

[1] Robert T Biedron and Elizabeth M Lee-Rausch. Rotor Airloads Prediction Using Unstructured Meshes and Loose CFD/CSD Coupling.

[2] J Donea, A Huerta, J.-Ph Ponthot, and A Rodríguez-Ferran. Arbitrary Lagrangian-Eulerian methods. (1969):1–38, 2004.

[3] Th Dunne. An Eulerian approach to uid – structure interaction and goal-oriented mesh adaptation. *International Journal for Numerical Methods in Fluids*, (December 2005):1017–1039, 2006.

[4] Thomas Dunne and Rolf Rannacher. Adaptive Finite Element Approximation of Fluid-Structure Interaction Based on an Eulerian Variational Formulation. *Fluid-Structure Interaction*, 53:110–145, 2006.

[5] Richard P Dwight. Robust Mesh Deformation using the Linear Elasticity Equations.

[6] Miguel A Fernández and Jean-Frédéric Gerbeau. Algorithms for fluid-structure interaction problems. 2009.

[7] Brian T. Helenbrook. Mesh deformation using the biharmonic operator. *International Journal for Numerical Methods in Engineering*, 2003.

[8] Su-Yuen Hsu, Chau-Lyan Chang, and Jamshid Samareh. A Simplified Mesh Deformation Method Using Commercial Structural Analysis Software.

[9] Hrvoje Jasak and Željko Tuković. Automatic mesh motion for the unstructured Finite Volume Method. *Transactions of Famena*, 30(2):1–20, 2006.

[10] V V Meleshko. Bending of an Elastic Rectangular Clamped Plate: Exact Versus 'Engineering' Solutions. *Journal of Elasticity*, 48(1):1–50, 1997.

[11] Selim MM and Koomullil RP. Mesh Deformation Approaches – A Survey. *Journal of Physical Mathematics*, 7(2), 2016.

[12] Thomas Richter. Fluid Structure Interactions. 2016.

[13] K Stein, T Tezduyar, and R Benney. Mesh Moving Techniques for Fluid-Structure Interactions With Large Displacements.

[14] T E Tezduyar, M Behr, S Mittal, and A A Johnson. COMPUTATION OF UNSTEADY INCOMPRESSIBLE FLOWS WITH THE STABILIZED FINITE ELEMENT METHODS: SPACE-TIME FORMULATIONS, ITERATIVE STRATEGIES AND MASSIVELY PARALLEL IMPLEMENTATIONSt. *New Methods in Transient Analysis ASME*, 246(143), 1992.

[15] Wolfgang A. Wall, Axel , Gerstenberger, Peter , Gamnitzer, Christiane , Förster, and Ekkehard , Ramm. Large Deformation Fluid-Structure Interaction – Advances in ALE Methods and New Fixed Grid Approaches. In *Fluid-Structure Interaction: Modelling, Simulation, Optimisation*, pages 195—-232. Springer Berlin Heidelberg, 2006.

[16] Thomas Wick. *Adaptive Finite Element Simulation of Fluid-Structure Interaction with Application to Heart-Valve*. PhD thesis, Heidelberg.

[17] Thomas Wick. Solving Monolithic Fluid-Structure Interaction Problems in Arbitrary Lagrangian Eulerian Coordinates with the deal.II Library.

[18] Thomas Wick. Fully Eulerian fluid-structure interaction for time-dependent problems. *Computer Methods in Applied Mechanics and Engineering*, 255:14–26, 2013.