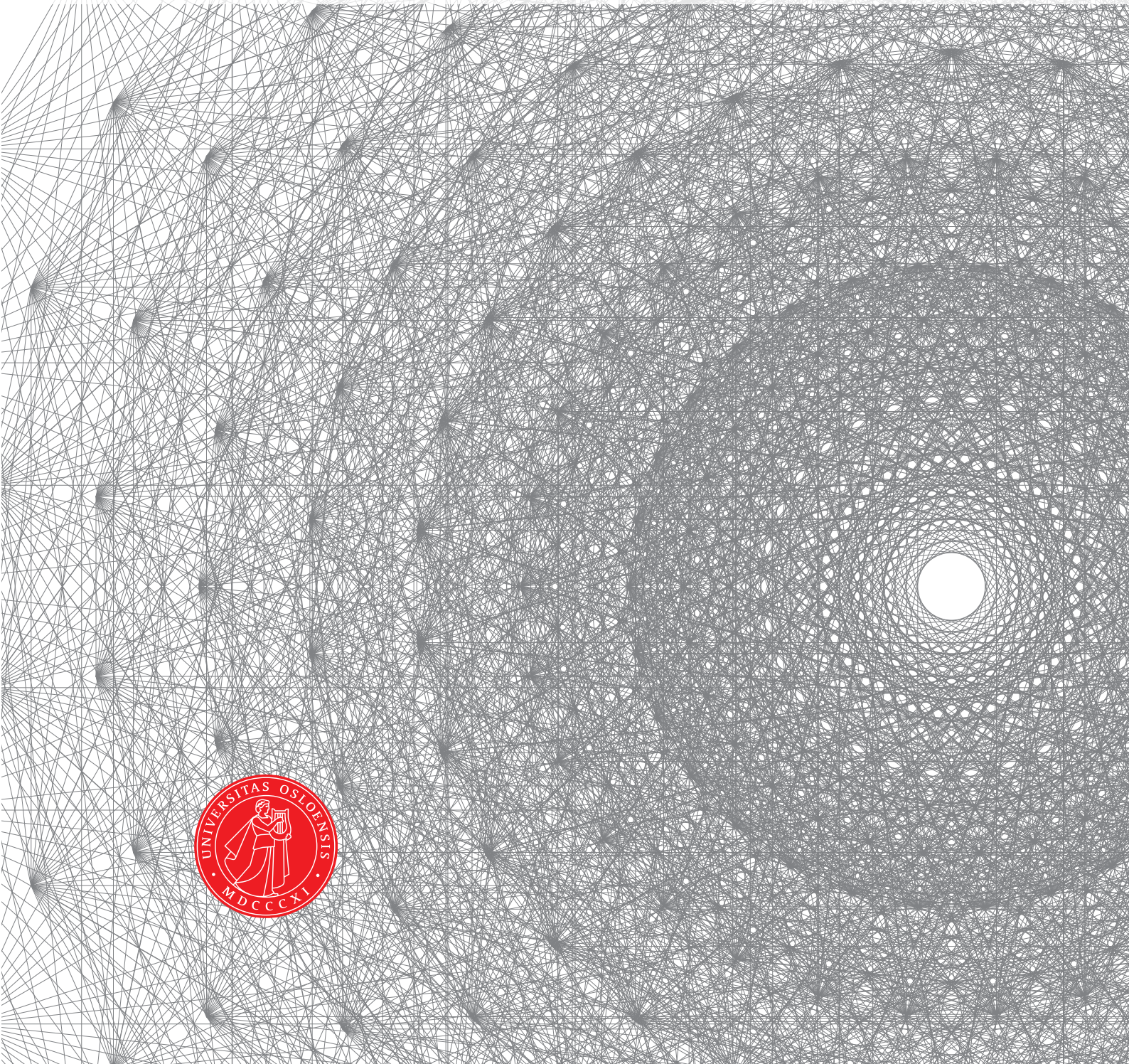
UiO : Department of Mathematics
University of Oslo

Fluid structure interaction

Andreas Strøm Slyngstad

Master's Thesis, Spring 2017



This master's thesis is submitted under the master's programme *Computational Science and Engineering*, with programme option *Mechanics*, at the Department of Mathematics, University of Oslo. The scope of the thesis is 60 credits.

The front page depicts a section of the root system of the exceptional Lie group E_8, projected into the plane. Lie groups were invented by the Norwegian mathematician Sophus Lie (1842–1899) to express symmetries in differential equations and today they play a central role in various parts of mathematics.

Fluid structure interaction

Andreas Strøm Slyngstad

01 01 01

Contents

Chapter 1

Time discretization of ALE-Framework

The aim of this chapter is to present some of the main challenges regarding discretization of a general monolithic fluid-structure interaction(FSI) problem, using the ALE-framework. The general FSI problem consist of coupling the fluid and solid governing equations together with the interface conditions. Even separately, the discretization of fluid and structure problems impose rather difficult issues due to their non-linear nature. However, their long-time existence within research community makes them well known problems and a vast number of rigorous approaches and commercial software exist to solve them individually. When solving the fluid and structure simultaneously however, the overall problem gets more complex due to the overall dependency of the two sub-problems and their interaction to one another.
In this chapter we will focus on time-stepping schemes for solving the monolithic FSI problem, using the ALE-framework. One of the main challenges is the additional non-linearty intruduced by the domain-velocity term in the fluid problem.

Problem 1.1. *ALE term*

$$\hat{\mathrm{J}}(\hat{F}_W^{-1}(\hat{\mathbf{v}} - \frac{\partial \hat{\Gamma}_W}{\partial t}) \cdot \hat{\nabla})\hat{\mathbf{v}}$$

Closer inspection of the convection term reviels spatial and temporal differential operators depending non-linearly on one another. Within computational science, these operators appear separated. Therefore the discretization of a general time-stepping scheme is not directly intuitive. It has been reported in [] [], that the stability of higher-order time stepping schemes is affected by the ALE-convection term. To what extent is this unclear, but several effort by [?], based on [?] have investigated the stability of the ALE formulation with first and second-order time schemes.
Though only the fluid problem will be discussed, it must be emphasized that the discretization of the solid problem is of great importance. Several studies exists for the individual solid problem, but a deeper analysis considering a fluid-structure interaction setting is abvient from the FSI litterature CITE(on time dic, richter)

1.1 Implementation of a one-step θ scheme

Problem 1.2. *One-step θ-scheme for laplace and elastic mesh moving model. Find $\hat{\mathbf{u}}_s, \hat{\mathbf{u}}_f, \hat{\mathbf{v}}_s, \hat{\mathbf{v}}_f, \hat{p}_f$ such that*

$$\begin{aligned}
\big(\hat{\mathrm{J}}\frac{\partial \hat{\mathbf{v}}}{\partial t},\ \hat{\boldsymbol{\psi}}^u\big)_{\hat{\Omega}_f} + \big(\hat{\mathrm{J}}(\hat{F}_W^{-1}(\hat{\mathbf{v}} - \frac{\partial \hat{\Gamma}_W}{\partial t})\cdot\hat{\nabla})\hat{\mathbf{v}},\ \hat{\boldsymbol{\psi}}^u\big)_{\hat{\Omega}_f} + \big(\hat{\mathrm{J}}_W\hat{\sigma}\hat{F}_W^{-T}\hat{\mathbf{n}}_f,\ \hat{\boldsymbol{\psi}}^u\big)_{\hat{\Gamma}_i} \\
-\big(\hat{\mathrm{J}}_W\hat{\sigma}\hat{F}_W^{-T},\ \hat{\nabla}\hat{\boldsymbol{\psi}}^u\big)_{\hat{\Omega}_f} - \big(\rho_f\hat{\mathrm{J}}\mathbf{f}_f,\ \hat{\boldsymbol{\psi}}^u\big)_{\hat{\Omega}_f} = 0 \\
\big(\rho_s\frac{\partial \hat{\mathbf{v}}_s}{\partial t},\ \hat{\boldsymbol{\psi}}^u\big)_{\hat{\Omega}_s} + \big(\hat{\mathbf{F}}\hat{\mathbf{S}}\hat{\mathbf{n}}_f,\ \hat{\boldsymbol{\psi}}^u\big)_{\hat{\Gamma}_i} - \big(\hat{\mathbf{F}}\hat{\mathbf{S}},\ \nabla\hat{\boldsymbol{\psi}}^u\big)_{\hat{\Omega}_s} - \big(\rho_s\hat{\mathbf{f}}_s,\ \hat{\boldsymbol{\psi}}^u\big)_{\hat{\Omega}_s} = 0 \\
\big(\frac{\partial \hat{\mathbf{v}}_s - \hat{\mathbf{u}}_s}{\partial t},\ \hat{\boldsymbol{\psi}}^v\big)_{\hat{\Omega}_s} = 0 \\
\big(\nabla\cdot(\hat{\mathrm{J}}\hat{F}_W^{-1}\hat{\mathbf{v}}),\ \hat{\boldsymbol{\psi}}^p\big)_{\hat{\Omega}_f} = 0 \\
\big(\hat{\sigma}_{\mathrm{mesh}},\ \hat{\nabla}\hat{\boldsymbol{\psi}}^u\big)_{\hat{\Omega}_f} = 0
\end{aligned}$$

A brief description will be given for the most central components and technologies used for this thesis.

1.2 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 submodules and we will give a brief overview of the most central components used during implementation and computation.

1.2.1 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.

1.3 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.

1.3.1 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.

```
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

1.4 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 to less computational efficient operations. Mention order of iterations?

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.

The main effort of speed-up were explored around the Jacobian assembly, as this was within our control.

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, involving smarter use of properties regarding the linear system to be solved. The non-consistent method presented involves altering the equation to be solved by some simplification of the system. As these simplifications will alter the expected convergence of the solver, one must take account for additional Newton iterations against cheaper Jacobi assembly. Therefore one also risk breakdown of the solver as the Newton iterations may not converge.

1.5 Consistent methods

1.5.1 Jacobi buffering

By inspection of the Jacobi matrix, some terms of the total residue is linear terms, and remain constant within each time step. By assembling these terms only in the first Newton iteration will save some assembly time for the additional iterations needed each time step. As consequence the convergence of the Newton method should be unaffected as we do not alter the system.

1.6 Non-consisten methods

1.6.1 Reuse of Jacobian

As the assembly of the Jacobian at each iteration is costly, one approach of reusing the Jacobian for the linear system was proposed. In other words, the LU-factorization of the system is reused until the Jacobi is re-assembled. This method greatly reduced the computational time for each time step. By a user defined parameter, the number of iterations before a new assembly of the Jacobian matrix can be controlled.

1.6.2 Quadrature reduce

The assemble time of the Jacobian greatly depends on the degree of polynomials used in the discretisation of the total residual. Within FEniCS this parameter can be controlled, and as such we can specify the order of polynomials representing the Jacobian. The use of lower order polynomials reduces assemble time of the matrix at each newton-iteration, however it leads to an inexact Jacobian which may results to additional iterations.

Bibliography

[1] M Razzaq, Stefan Turek, Jaroslav Hron, J F Acker, F Weichert, I Grunwald, C Roth, M Wagner, and B Romeike. Numerical simulation and benchmarking of fluid-structure interaction with application to Hemodynamics. *Fundamental Trends in Fluid-Structure Interaction*, 1:171–199, 2010.

[2] T. Richter and T. Wick. Finite elements for fluid-structure interaction in ALE and fully Eulerian coordinates. *Computer Methods in Applied Mechanics and Engineering*, 199(41-44):2633–2642, 2010.

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

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

[5] P. Wriggers. *Computational contact mechanics, second ed., Springer*. 2006.