INF5620 Final Project
Linear elasticity - Finte elements

Nina Kristine Kylstad (ninakky@student.matnat.uio.no)

INF5620 - Numerical methods for partial differential equations

November 14, 2012

Abstract

This report investigates the time-dependent equations for linear elasticity, and solving them using finite element methods.

Contents

1 Mathematical problem

We consider the general time-dependent equations for linear elasticity,

$$\rho \mathbf{u}_{tt}(\mathbf{x} = \nabla \cdot \sigma + \rho \mathbf{b}, \qquad \mathbf{x} \in \Omega, t \in (0, T], \tag{1}$$

$$\sigma = \sigma(\mathbf{u}) = 2\mu\epsilon(\mathbf{u}) + \lambda tr(\epsilon(\mathbf{u}))I, \tag{2}$$

$$\mathbf{u}(\mathbf{x}, 0) = \mathbf{u_0}, \qquad \mathbf{x} \in \Omega, \tag{3}$$

$$\mathbf{u}_t(\mathbf{x}, 0) = \mathbf{v_0}, \qquad \mathbf{x} \in \Omega, \tag{4}$$

$$\mathbf{u}(\mathbf{x}, t) = \mathbf{g}, \qquad \mathbf{x} \in \partial\Omega_D, t \in (0, T], \tag{5}$$

$$\sigma \cdot \mathbf{n} = 0, \qquad \mathbf{x} \in \partial\Omega_N, t \in (0, T], \tag{6}$$

where $\epsilon(\mathbf{u}) = \frac{1}{2}\left(\nabla\mathbf{u} + \nabla\mathbf{u}^T\right)$ is the strain tensor, μ and λ are the Lamé parameters, and I is the identity tensor. The initial conditions are given by (3) and (4). We will consider Dirichlet and Neumann boundary conditions, as shown in (5) and (6) respectively.

2 Discretization

2.1 Finite Difference in time

We begin by discretizing the time derivative, using a centered difference approximation:

$$\frac{\partial^2 \mathbf{u}}{\partial t^2} \approx \frac{\mathbf{u^{n+1}} - 2\mathbf{u^n} + \mathbf{u^{n-1}}}{\Delta t^2} \tag{7}$$

We insert this approximation into (1) and evaluate at time $t = t_n$:

$$\rho\frac{\mathbf{u^{n+1}} - 2\mathbf{u^n} + \mathbf{u^{n-1}}}{\Delta t^2} = \nabla \cdot \sigma(\mathbf{u}^n) + \rho\mathbf{b}^n \tag{8}$$

For simplicity, we let $\sigma^n = \sigma(u^n)$. We solve for $\mathbf{u^{n+1}}$:

$$\begin{aligned}\rho(\mathbf{u^{n+1}} - 2\mathbf{u^n} + \mathbf{u^{n-1}}) &= \Delta t^2 \nabla \cdot \sigma^n + \rho\mathbf{b}^n \\ \mathbf{u^{n+1}} &= 2\mathbf{u^n} - \mathbf{u^{n-1}} + \frac{\Delta t^2}{\rho}\nabla \cdot \sigma^n + \Delta t^2\mathbf{b}^n \end{aligned} \tag{9}$$

We now have a discretization in time.

2.2 Finite Elements in space

We use a Galerkin method to find the variational form of (1), by introducing a test function $\mathbf{v} \in V$:

$$\int_\Omega \mathbf{u^{n+1}} \cdot \mathbf{v}dx = 2\int_\Omega \mathbf{u^n} \cdot \mathbf{v}dx - \int_\Omega \mathbf{u^{n-1}} \cdot \mathbf{v}dx + \frac{\Delta t^2}{\rho}\int_\Omega (\nabla \cdot \sigma^n) \cdot \mathbf{v}dx + \Delta t^2 \int_\Omega \mathbf{b}^n \cdot \mathbf{v}dx \tag{10}$$

We need to integrate the $\int_\Omega (\nabla \cdot \sigma^n) \cdot vdx$ -term by parts, as it contains second derivatives. We get

$$\int_\Omega (\nabla \cdot \sigma^n) \cdot \mathbf{v}dx = -\int_\Omega \sigma^n : \nabla\mathbf{v}dx + \int_{\partial\Omega} (\sigma^n \cdot \mathbf{n}) \cdot \mathbf{v}ds \tag{11}$$

where $\sigma^n : \nabla\mathbf{v}$ is a tensor inner product. The term $\int_{\partial\Omega} (\sigma^n \cdot \mathbf{n}) \cdot \mathbf{v}ds$ vanishes for the Neumann boundary condition $\sigma \cdot \mathbf{n} = 0$, and for constant Dirichlet boundary conditions. We will leave out the term in the rest of the calculations. If needed, it can be implemented at a later stage.

We now have

$$\int_\Omega \mathbf{u^{n+1}} \cdot \mathbf{v}dx = \int_\Omega (2\mathbf{u^n} - \mathbf{u^{n-1}} + \Delta t^2\mathbf{b}^n) \cdot \mathbf{v}dx - \frac{\Delta t^2}{\rho}\int_\Omega \sigma^n : \nabla\mathbf{v}dx \tag{12}$$

This is the general scheme. We need to implement the initial conditions in order to get a special scheme for the first time step:

$$\begin{aligned}
&\mathbf{u}^0 = \mathbf{u_0} \\
&\mathbf{u}_t(\mathbf{x}, 0) = \mathbf{v_0} \Rightarrow \frac{\mathbf{u^{n+1}} - \mathbf{u^{n-1}}}{2\Delta t} \approx \mathbf{v_0} \\
&\Rightarrow \mathbf{u}^{-1} = \mathbf{u}^1 - 2\Delta t \mathbf{v_0}
\end{aligned}$$

We put this expression for $\mathbf{u}^{-1}$ into (12):

$$\begin{aligned}
2\int_\Omega \mathbf{u}^1 \cdot \mathbf{v} dx &= \int_\Omega (2\mathbf{u}^0 + 2\Delta t \mathbf{v_0} + \Delta t^2 \mathbf{b}^0) \cdot \mathbf{v} dx - \frac{\Delta t^2}{\rho} \int_\Omega \sigma^0 : \nabla \mathbf{v} dx \\
\int_\Omega \mathbf{u}^1 \cdot \mathbf{v} dx &= \int_\Omega (\mathbf{u_0} + \Delta t \mathbf{v_0} + \frac{\Delta t^2}{2} \mathbf{b}^0) \cdot \mathbf{v} dx - \frac{\Delta t^2}{2\rho} \int_\Omega \sigma(\mathbf{u}_0) : \nabla \mathbf{v} dx
\end{aligned} \tag{13}$$

We are now ready to express (1) as a variational problem. We let $u = \mathbf{u^{n+1}}$ represent the unknown at the next time level:

$$a_0(\mathbf{u}, \mathbf{v}) = \int_\Omega \mathbf{u} \cdot \mathbf{v} dx \tag{14}$$

$$L_0(\mathbf{v}) = \int_\Omega \mathbf{u_0} \cdot \mathbf{v} dx \tag{15}$$

$$a_1(\mathbf{u}, \mathbf{v}) = \int_\Omega \mathbf{u} \cdot \mathbf{v} dx \tag{16}$$

$$L_1(\mathbf{v}) = \int_\Omega (\mathbf{u_0} + \Delta t \mathbf{v}_0 + \frac{\Delta t^2}{2} \mathbf{b}^0) \cdot \mathbf{v} dx - \frac{\Delta t^2}{2\rho} \int_\Omega \sigma(\mathbf{u_0}) : \nabla \mathbf{v} dx \tag{17}$$

$$a(\mathbf{u}, \mathbf{v}) = \int_\Omega \mathbf{u} \cdot \mathbf{v} dx \tag{18}$$

$$L(\mathbf{v}) = \int_\Omega (2\mathbf{u^n} - \mathbf{u^{n-1}} + \Delta t^2 \mathbf{b}^n) \cdot \mathbf{v} dx - \frac{\Delta t^2}{\rho} \int_\Omega \sigma^n : \nabla \mathbf{v} dx \tag{19}$$

$$L(\mathbf{v}) = \int_\Omega (2\mathbf{u^n} - \mathbf{u^{n-1}} + \Delta t^2 \mathbf{b}^n) \cdot \mathbf{v} dx - \frac{\Delta t^2}{\rho} \int_\Omega (2\mu\epsilon(\mathbf{u}) + \lambda tr(\epsilon(\mathbf{u}))I) : \nabla \mathbf{v} dx \tag{20}$$

With this formulation, we could implement the problem in FEniCS.

However, it is possible to go further, in order to make the implementation more efficient. We introduce the approximations

$$\mathbf{u^{n+1}} \approx \sum_j^N \mathbf{c}_j^{n+1}\phi_j \tag{21}$$

$$\mathbf{u^{n}} \approx \sum_j^N \mathbf{c}_j^{n}\phi_j \tag{22}$$

$$\mathbf{u^{n-1}} \approx \sum_j^N \mathbf{c}j^{n-1}\phi_j \tag{23}$$

$$\mathbf{b}^n \approx \sum_j^N \mathbf{b}_j^n\phi_j \tag{24}$$

where ϕ_j are prescribed basis functions, and c_j^n and b_j^n are coefficient vectors to be determined. Let $\mathbf{v} = \phi_i$.

$$\int_\Omega \mathbf{u^{n+1}} \cdot \mathbf{v}dx = \int_\Omega \left(\sum_j^N \mathbf{c}_j^{n+1}\phi_j\right)\phi_i dx = \sum_j^N \left(\int_\Omega \phi_i\phi_j dx\right)\mathbf{c}_j^{n+1} \tag{25}$$

$$\int_\Omega (2\mathbf{u^n} - \mathbf{u^{n-1}} + \Delta t^2\mathbf{b}^n)\cdot\mathbf{v}dx = \int_\Omega \left(2\sum_j^N \mathbf{c}_j^n\phi_j - \sum_j^N \mathbf{c}_j^{n-1}\phi_j + \Delta t^2\sum_j^N \mathbf{b}_j^n\phi_j\right)\phi_i dx \tag{26}$$

$$= 2\sum_j^N \left(\int_\Omega \phi_i\phi_j dx\right)\mathbf{c}_j^n - \sum_j^N \left(\int_\Omega \phi_i\phi_j dx\right)\mathbf{c}_j^{n-1} + \Delta t^2\sum_j^N \left(\int_\Omega \phi_i\phi_j dx\right)\mathbf{b}_j^n \tag{27}$$

$$\begin{aligned}\int_\Omega \sigma^n : \nabla\mathbf{v}dx &= \int_\Omega \sigma(\sum_j^N \mathbf{c}_j^n\phi_j) : \nabla\phi_i dx \\ &= \sum_j^N \left(\int_\Omega \sigma(\phi_j) : \nabla\phi_i dx\right)\mathbf{c}_j^n\end{aligned}$$

We define the matrices M and K, with elements $M_{i,j} = \int_\Omega \phi_i\phi_j dx$ and $K_{i,j} = \int_\Omega \sigma(\phi_j) : \nabla\phi_i dx$. Since the coefficient vectors $\mathbf{c}^n$ are the nodal values of $\mathbf{u}$ at time $t = t^n$, we get the linear system

$$M\mathbf{u}^{n+1} = 2M\mathbf{u}^n - M\mathbf{u}^{n-1} - \frac{\Delta t^2}{\rho}K\mathbf{u}^n + \Delta t^2 M\mathbf{b}^n \tag{28}$$

$$= \left(2M - \frac{\Delta t^2}{\rho}K\right)\mathbf{u}^n - M\mathbf{u}^{n-1} + \Delta t^2 M\mathbf{b}^n \tag{29}$$

where $\mathbf{u}^n$ and $\mathbf{u}^{n-1}$ are known.

3 Implementation

3.1 Verification

3.1.1 Stationary case

We would like to make sure that the implementation of the scheme above is correct. After the first implementation of the scheme, the solution seemed to increase exponentially for each time step, which is not something we want. We therefore take a step back and look at the stationary case, where we know we have an implementation that works quite well. We can then try to come up with the correct implementation of the boundary conditions, and then apply this to the time-dependent case.

We begin by looking at a 3 dimensional (square) elastic rod of length L, and apply the following boundary conditions:

- $u_x = 0$ for $x = 0$,
- $u_x = \alpha L$ for $x = L$,
- $\sigma \cdot \mathbf{n} = 0$ for all other surfaces.

We choose the displacement vector given by

$$u = \begin{pmatrix} u_x(x) \\ u_y(y) \\ u_z(z) \end{pmatrix} = \begin{pmatrix} \alpha x \\ \gamma y \\ \gamma z \end{pmatrix} \tag{30}$$

This means that we have a rod that has a displacement α (an extension) in the x-direction, and displacement γ (a contraction) in the y- and z-directions. The chosen displacement vector satisfies the boundary condition requirements, and we therefore let the exact solution $\mathbf{u}$ be the boundary conditions in the implementation. However, the boundary conditions will then not be constant, but dependent on space. Because of this, We must now include the $\int_{\partial\Omega} (\sigma^n \cdot \mathbf{n}) \cdot \mathbf{v} ds$ - term. Although the derivation of the stationary scheme is not shown here, the calculations are the same (excluding the derivatives in time). We can therefore simply add the new term to the right-hand-side, or $L(v)$, as done in the implementation of the problem in A.1.

The solution we are looking for with these boundary conditions gives σ on the form

$$\sigma = \begin{pmatrix} \beta & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{31}$$

where β is some linear function of α. That is, if we double α, we should also see that β doubles.

We can calculate what σ will be from (30):

$$\begin{aligned}
\nabla u &= \begin{pmatrix} u_{1,x} & u_{1,y} & u_{1,x} \\ u_{2,x} & u_{2,y} & u_{2,x} \\ u_{3,x} & u_{3,y} & u_{3,x} \end{pmatrix} = \begin{pmatrix} \alpha & 0 & 0 \\ 0 & \gamma & 0 \\ 0 & 0 & \gamma \end{pmatrix} \\
(\nabla u)^T &= \nabla u \\
&\Rightarrow \epsilon = \nabla u \\
\sigma(u) &= \begin{pmatrix} 2\mu\alpha & 0 & 0 \\ 0 & 2\mu\gamma & 0 \\ 0 & 0 & 2\mu\gamma \end{pmatrix} + \begin{pmatrix} \lambda(\alpha+2\gamma) & 0 & 0 \\ 0 & \lambda(\alpha+2\gamma) & 0 \\ 0 & 0 & \lambda(\alpha+2\gamma) \end{pmatrix} \\
&= \begin{pmatrix} 2\gamma\lambda + \alpha(\lambda+\mu) & 0 & 0 \\ 0 & \alpha\lambda + 2\gamma(\lambda+\mu) & 0 \\ 0 & 0 & \alpha\lambda + 2\gamma(\lambda+\mu) \end{pmatrix}
\end{aligned}$$

As we are looking for a σ on the form (31), we must have

$$\alpha\lambda + 2\gamma(\lambda+\mu) = 0 \tag{32}$$

$$\Rightarrow \gamma = -\frac{\alpha\lambda}{2(\lambda+\mu)} \tag{33}$$

The implementation of this stationary problem is given in A.1.

The output of this program gives that σ has the size (number of data points) 6561. This is exactly what we would expect from P1 elements on `UnitCube(8,8,8)`, as we have $9 \times 9 \times 9$ points on the cube, and each point creates a σ-tensor of size 3×3.

The output from printing the values of $\sigma\_$ are slightly more difficult to interpret, as it is not entirely clear how $\sigma\_$ is formatted. However, with some testing, it can be seen that the values $\sigma\_[0:729]$ are all significantly larger than 0. In fact, with $\alpha = 1.0$, these values are all very close to 1. The rest of the values, $\sigma\_[730:-1]$ are very small, in the order of magnitude of 10^{-14} and smaller. This is significant because there are $9\times9\times9 = 729$ σ-tensors, one for each point in the mesh. The results from the program suggest that when performing the print-command `sigma_.vector().array()`, we get the first element from each σ, followed by the second element and so on. If this is correct, then we get σ very close to the form we wanted. If we take $\sigma\_[i \times 729]$ for i=0,..., 8 and place those values in a matrix, we get:

$$\begin{pmatrix} 1.00000299e+00 & 1.34341531e-15 & 1.28156902e-15 \\ 1.34341531e-15 & 8.49581088e-16 & 1.12541265e-17 \\ 1.28156902e-15 & 2.27396678e-17 & 8.58702048e-16 \end{pmatrix}$$

If we double the value of α to $\alpha = 2.0$, we get

$$\begin{pmatrix} 2.00000598e+00 & 4.12999178e-15 & 2.72159629e-16 \\ 7.33314564e-15 & 1.39556359e-15 & -3.42130794e-17 \\ 2.07890357e-16 & -6.51129139e-16 & -1.87653016e-15 \end{pmatrix}$$

We see that the $[0,0]$-element has exactly doubled, while the rest of the elements are very close to 0. It would therefore appear that the stationary case works the way it should.

3.1.2 Time dependent case

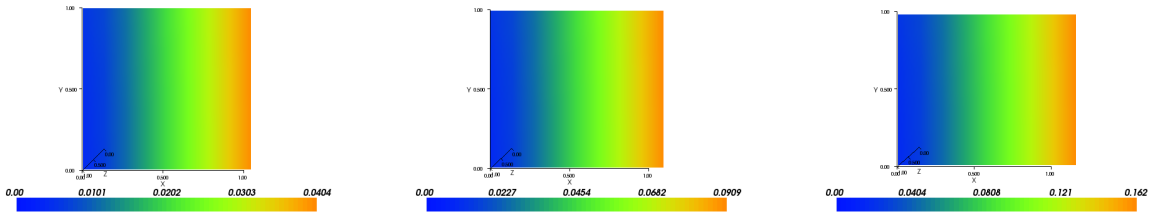
We can now extend the stationary case (which seems to be working correctly) to the time dependent case. The code for this is included in A.2.1.

The changes from the stationary case are as follows:

- Include values of the displacement for the previous two time steps as known functions.
- Implement initial conditions
- Use initial conditions to implement the special first step
- Update the displacement for every time step
- Update boundary conditions and source term for every time step

The time-dependent case implements the variational forms in (14) - (19). In the same way as in the stationary case, we must add the $\int_{\partial\Omega} (\sigma^n \cdot \mathbf{n}) \cdot \mathbf{v} ds$ - term to the variational form before implementation. Again we can simply add the term to $L(v)$ (and $L_1(v)$). However, we need to remember the factors it would have been multiplied by in the variational form - these are the same factors as for the $\int_\Omega \sigma^n : \nabla \mathbf{v} dx$ - term.

Running the program with the same type of boundary conditions as in the stationary case (i.e. a rod that is stretched in the x-direction, held at $x = 0$, and with free displacement in the y- and z-directions), we get the following plots for $\lambda = 1.0$, $\mu = 0.1$:



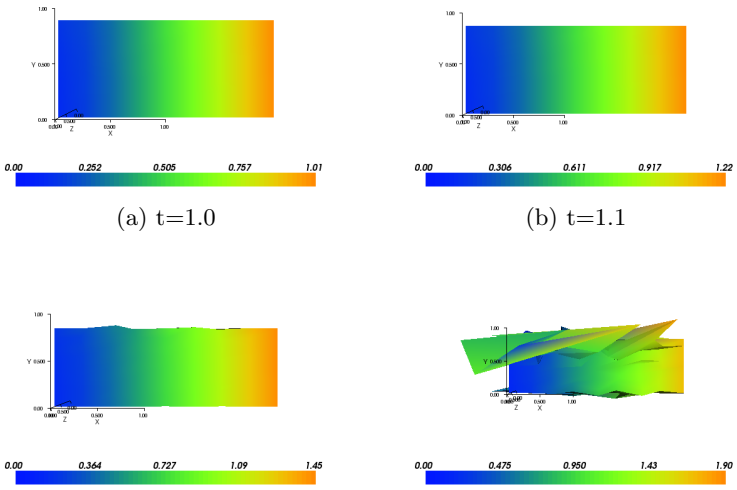
(a) t=0.2 (b) t=0.3 (c) t=0.4

(d) t=0.5 (e) t=0.6 (f) t=0.7

Figure 1: Time-dependent case for a unit cube, $t \leq 1.0$

As we can see from the plots 1a-1f, the box is stretched in the x-direction, while the y-direction (and z-direction, which can't be seen in the plots) are allowed to contract. The box appears to be behaving in the way that we would expect. The error (when comparing to an exact solution) is very small, but is increasing slightly for each time step.

The problem appears when the program runs for times longer than $t = 1.0$:



(a) t=1.0 (b) t=1.1

(c) t=1.2 (d) t=1.3

Figure 2: Time-dependent case for a unit cube, $t > 1.0$

In figures 2c and (especially) 2d we see that the solution starts to become unstable. It is possible that there are some stability criteria, involving dt, λ, μ and possibly the spacial grid. Logically, it makes sense that there should be some stability criterion involving μ and λ; these are the elasticity parameters. If we consider an elastic band, then at some point it will be stretched so far that it breaks. At that point, the equations for linear elasticity do not apply. Perhaps that is what is happening in this case. We can also look at output from the program that comes from calculating the error (the difference between the exact and numerical solution). The output is as follows:

```
Solving linear variational problem.
first step:  2.42861286637e-17
t= 0.2
Solving linear variational problem.
error:  1.35308431126e-16
t= 0.3
Solving linear variational problem.
error:  1.68615121865e-15
t= 0.4
Solving linear variational problem.
error:  2.16493489802e-14
t= 0.5
Solving linear variational problem.
error:  3.49165141245e-13
t= 0.6
Solving linear variational problem.
error:  7.0546624098e-12
t= 0.7
Solving linear variational problem.
error:  1.42625244948e-10
t= 0.8
Solving linear variational problem.
```

```
error:  3.16864245953e-09
t= 0.9
Solving linear variational problem.
error:  7.03295270355e-08
t= 1.0
Solving linear variational problem.
error:  1.56963559977e-06
```

As we can see, the error is quite small, but it is increasing by a factor 10 for every time step. Clearly, something is not working entirely as it should.

3.2 Speeding up the calculations

If we use the variational forms in (14) - (19) directly in a program, as done in the program `lin_elast.py` in A.2.1, then FEniCS takes care of the assembly of the element matrices within the command `LinearVariationalProblem(a, L, u, bcs=bcs)`. This means that the program has to do a new assembly for each time step, and (especially in 3D) this can be quite time consuming. As implied at the end of section 2.2, we do not have to use the variational forms ((14) - (19)) directly.

The first thing to look at is the form of $a(\mathbf{u}, \mathbf{v})$. It is the same for each time step, including the first one. By using the `LinearVariationalProblem()` - function in `dolfin`, we assemble the element matrices for every time step. It is unnecessary to do this for every time step, as $a(\mathbf{u}, \mathbf{v})$ does not depend on time. The associated matrix A will therefore also be independent of time, and can be computed before the time loop. We then use this matrix A to solve the equation $A\mathbf{u} = \mathbf{b}$, where $\mathbf{b}$, corresponding to L, has to be computed for each new time level. The file `lin_elast_assemble.py` with source code for this is shown in A.2.2

To speed the calculations up even further, we can use the matrix equations derived in the end of section 2.2. However, we need to do something about the implementation of the boundary conditions, as we do not have this in the matrix equations yet. We then get two K-matrices, K_1 and K_2, where $K_{1i,j} = \int_\Omega \sigma(\phi_j) : \nabla\phi_i dx$ and $K_{2i,j} = \int_{\partial\Omega} (\sigma(\phi_j) \cdot \mathbf{n}) \cdot \phi_i ds$. The matrix equation then becomes

$$M\mathbf{u}^{n+1} = 2M\mathbf{u}^n - M\mathbf{u}^{n-1} - \frac{\Delta t^2}{\rho} K_1 \mathbf{u}^n + \frac{\Delta t^2}{\rho} K_2 \mathbf{u}^n + \Delta t^2 M \mathbf{b}^n \tag{34}$$

The matrices M, K_1 and K_2 are independent of time in the same way as $a(\mathbf{u}, \mathbf{v})$, and therefore only have to be computed once. Here we also have to solve a matrix equation $A\mathbf{u} = \mathbf{b}$, where $A = M$ and $\mathbf{b}$ is the right-hand-side in (34). The file `lin_elast_matrix.py` with the source code for this is shown in A.2.3

The time counter is placed around the time-loop, to see which method runs through the loop the fastest. The differences in time taken to run the simulation are shown in Table 1.

`lin_elast.py`	`lin_elast_assemble.py`	`lin_elast_matrix.py`
1.96	1.65	1.08

Table 1: Differences in time taken to run simulation

These differences are not exactly huge, but the mesh is quite coarse and might account for the fact that all the methods are relatively quick.

Another thing that could be possible to investigate further is the method used to solve the linear system. The default method used here is LU-factorization. It could be interesting to see how much faster it would be to for example use a Krylov solver, or a lumped mass matrix. However, due to time constraints, this will not be investigated in this report.

A Source code

A.1 Stationary case: `lin_elast_stationary.py`

```
from dolfin import *
import numpy as np

# Create mesh
mesh = UnitCube(8,8,8)

# Create function space
V = VectorFunctionSpace(mesh, "Lagrange", 1)

# Create test and trial functions
u = TrialFunction(V)
v = TestFunction(V)
n = FacetNormal(mesh)



# Elasticity parameters
E, nu = 1.0, 0.1
mu, lmbda = E/(2.0*(1.0 + nu)), E*nu/((1.0 + nu)*(1.0 - 2.0*nu))
rho = 1.0
alpha = 1.0
gamma = -lmbda*alpha/(2*(mu + lmbda))

# Source term
b = Expression(("0.0", "0.0", "0.0"))
def eps(u):
        return (1.0/2.0)*(nabla_grad(u) + transpose(nabla_grad(u)))

# Stress
def sigma(u):
         return  2*mu*eps(u) + lmbda*tr(eps(u))*Identity(v.cell().d)


# Governing balance equation
F = inner(sigma(u), grad(v))*dx - rho*dot(b,v)*dx - dot(dot(sigma(u),n),v)*ds


# Extract linear and bilinear forms from F
a, L = lhs(F), rhs(F)

# Boundary conditions
def left_boundary(x, on_boundary):
        tol = 1E-14
        return on_boundary and abs(x[0]) < tol

def right_boundary(x, on_boundary):
        tol = 1E-14
        return on_boundary and abs(x[0] - 1) < tol

#gamma = 0.0
c = Expression(("0.0","gamma*x[1]","gamma*x[2]"), gamma=gamma)
r = Expression(("alpha","gamma*x[1]","gamma*x[2]"), alpha=alpha, gamma=gamma)
bc_left = DirichletBC(V, c, left_boundary)
bc_right = DirichletBC(V, r, right_boundary)
bcs = [bc_left, bc_right]


u = Function(V)
problem = LinearVariationalProblem(a, L, u, bcs=bcs)
solver = LinearVariationalSolver(problem)
solver.solve()

plot(u, mode="displacement", axes=True)
```

```
plot(mesh)
interactive()


V_ = TensorFunctionSpace(mesh, "Lagrange", 1)
sigma_ = project(sigma(u), V_)
size = len(sigma_.vector().array())
print size
tol = 1E-14
counter = 0
for i in range(size-1):
        if sigma_.vector().array()[i] <= tol:
                counter += 1
print counter

test = np.zeros((3,3))
test[0,0] = sigma_.vector().array()[0]
test[0,1] = sigma_.vector().array()[729]
test[0,2] = sigma_.vector().array()[1458]
test[1,0] = sigma_.vector().array()[2187]
test[1,1] = sigma_.vector().array()[2916]
test[1,2] = sigma_.vector().array()[3655]
test[2,0] = sigma_.vector().array()[4374]
test[2,1] = sigma_.vector().array()[5103]
test[2,2] = sigma_.vector().array()[5832]

print test
print sigma_.vector().array()[728]
print sigma_.vector().array()[729]
```

A.2 Time dependent case

A.2.1 lin_elast.py

```
from dolfin import *
import numpy as np
import time

def run_simulation(version):
        if version=="test-case":

                # Create mesh
                mesh = UnitCube(8,8,8)

                # Create function space
                V = VectorFunctionSpace(mesh, "Lagrange", 1)

                # Elasticity parameters
                E, nu = 1.0, 0.1
                mu, lmbda = E/(2.0*(1.0 + nu)), E*nu/((1.0 + nu)*(1.0 - 2.0*nu))
                rho = 1.0
                alpha = 1.0
                gamma = -lmbda*alpha/(2*(mu + lmbda))
                dt = 0.1

                # Source term
                b = Expression(("2.0*alpha*x[0]", "2.0*gamma*x[1]", "2.0*gamma*x[2]"),\
                 alpha=alpha, gamma=gamma)

                # Initial conditions
                u0 = Expression(("0.0", "0.0", "0.0"))
                v0 = Expression(("0.0", "0.0", "0.0"))

                exact = Expression(("t*t*alpha*x[0]", "t*t*gamma*x[1]", "t*t*gamma*x[2]"),\
                 alpha=alpha, gamma=gamma, t=0.0)
```

```
                solver(mesh, V, u0, v0, b, mu, lmbda, rho, alpha, gamma, dt, exact=exact)


def solver(mesh, V, u0, v0, f, mu, lmbda, rho, alpha, gamma, dt, version="lvp", exact=None):

        # Create test and trial functions
        u = TrialFunction(V)
        v = TestFunction(V)
        n = FacetNormal(mesh)

        def eps(u):
                return (1.0/2.0)*(nabla_grad(u) + transpose(nabla_grad(u)))

        # Stress
        def sigma(u):
                 return  2*mu*eps(u) + lmbda*tr(eps(u))*Identity(v.cell().d)


        # Boundary conditions
        def left_boundary(x, on_boundary):
                tol = 1E-14
                return on_boundary and abs(x[0]) < tol

        def right_boundary(x, on_boundary):
                tol = 1E-14
                return on_boundary and abs(x[0] - 1) < tol

        def update_boundary(t=0.0):
                r = Expression(("0.0","t*t*gamma*x[1]","t*t*gamma*x[2]"), gamma=gamma, t=t)
                l = Expression(("t*t*alpha","t*t*gamma*x[1]","t*t*gamma*x[2]"), \
                        alpha=alpha, gamma=gamma, t=t)

                bc_left = DirichletBC(V, r, left_boundary)
                bc_right = DirichletBC(V, l, right_boundary)
                bcs = [bc_left, bc_right]
                return bcs



        #------------------- Special case for first time step ----------------------#
        u2 = interpolate(u0, V)

        F = (dot(u,v) - dot(u2,v) - dt*dot(v0,v) - (dt**2/2)*dot(f,v))*dx + \
        dt**2/(2.0*rho)*inner(sigma(u2),nabla_grad(v))*dx - dt**2/(2.0*rho)*dot(dot(sigma(u2),n),v)*ds

        a, L = lhs(F), rhs(F)


        bcs = update_boundary(t=dt)

        u1 = Function(V)
        problem = LinearVariationalProblem(a, L, u1, bcs=bcs)
        solver = LinearVariationalSolver(problem)
        solver.solve()

        if exact:
                exact.t = dt
                u_e = interpolate(exact, V)
                max_error = np.max(u_e.vector().array() - u1.vector().array())
                print "first step: ",max_error
        #---------------------------------------------------------------------------#


        # Governing balance equation for the remaining time steps

        F = (dot(u,v) - 2*dot(u1,v) + dot(u2,v) - dt**2*dot(f,v))*dx + \
                (dt**2/rho)*inner(sigma(u1), grad(v))*dx - (dt**2/rho)*dot(dot(sigma(u1),n),v)*ds
```

```
        u = Function(V)
        a, L = lhs(F), rhs(F)
        T = 1.0
        t = 2*dt
        counter = 2
        t1 = time.clock()
        while t <= T:
                print "t=",t
                # Extract linear and bilinear forms from F
                bcs = update_boundary(t=t)
                problem = LinearVariationalProblem(a, L, u, bcs=bcs)
                solver = LinearVariationalSolver(problem)
                solver.solve()
                u2.assign(u1)
                u1.assign(u)
                #u = TrialFunction(V)

                if exact:
                        exact.t = t
                        u_e = interpolate(exact, V)
                        max_error = np.max(u_e.vector().array() - u1.vector().array())
                        print "error: ",max_error

                t += dt
                counter += 1

                # plot(u1, mode="displacement", axes=True, title="t=%g"%(t-dt),\
                #  label="t=%g"%(t-dt), basename="time_dependent_elast")
                # #title("t="+(t-dt))
                # # plot(mesh)
                # interactive()
        t2 = time.clock()
        print "Time taken: ", t2-t1

run_simulation("test-case")
```

A.2.2 lin_elast_assemble.py

```
from dolfin import *
import numpy as np
import time


def run_simulation(version):
        if version=="test-case":

                # Create mesh
                mesh = UnitCube(8,8,8)

                # Create function space
                V = VectorFunctionSpace(mesh, "Lagrange", 1)

                # Elasticity parameters
                E, nu = 1.0, 0.1
                mu, lmbda = E/(2.0*(1.0 + nu)), E*nu/((1.0 + nu)*(1.0 - 2.0*nu))
                rho = 1.0
                alpha = 1.0
                gamma = -lmbda*alpha/(2*(mu + lmbda))
                dt = 0.1

                # Source term
                b = Expression(("2.0*alpha*x[0]", "2.0*gamma*x[1]", "2.0*gamma*x[2]"),\
                 alpha=alpha, gamma=gamma)

                # Initial conditions
                u0 = Expression(("0.0", "0.0", "0.0"))
```

```
                v0 = Expression(("0.0", "0.0", "0.0"))

                exact = Expression(("t*t*alpha*x[0]", "t*t*gamma*x[1]", "t*t*gamma*x[2]"),\
                 alpha=alpha, gamma=gamma, t=0.0)

                solver(mesh, V, u0, v0, b, mu, lmbda, rho, alpha, gamma, dt, exact=exact)


def solver(mesh, V, u0, v0, f, mu, lmbda, rho, alpha, gamma, dt, version="lvp", exact=None):

        # Create test and trial functions
        u = TrialFunction(V)
        v = TestFunction(V)
        n = FacetNormal(mesh)

        def eps(u):
                return (1.0/2.0)*(nabla_grad(u) + transpose(nabla_grad(u)))

        # Stress
        def sigma(u):
                 return  2*mu*eps(u) + lmbda*tr(eps(u))*Identity(v.cell().d)


        # Boundary conditions
        def left_boundary(x, on_boundary):
                tol = 1E-14
                return on_boundary and abs(x[0]) < tol

        def right_boundary(x, on_boundary):
                tol = 1E-14
                return on_boundary and abs(x[0] - 1) < tol

        def update_boundary(t=0.0):
                r = Expression(("0.0","t*t*gamma*x[1]","t*t*gamma*x[2]"), gamma=gamma, t=t)
                l = Expression(("t*t*alpha","t*t*gamma*x[1]","t*t*gamma*x[2]"), \
                        alpha=alpha, gamma=gamma, t=t)

                bc_left = DirichletBC(V, r, left_boundary)
                bc_right = DirichletBC(V, l, right_boundary)
                bcs = [bc_left, bc_right]
                return bcs



        #------------------- Special case for first time step ----------------------#
        u2 = interpolate(u0, V)

        F = (dot(u,v) - dot(u2,v) - dt*dot(v0,v) - (dt**2/2)*dot(f,v))*dx + \
        dt**2/(2.0*rho)*inner(sigma(u2),nabla_grad(v))*dx - dt**2/(2.0*rho)*dot(dot(sigma(u2),n),v)*ds

        a, L = lhs(F), rhs(F)

        A = assemble(a)
        b = assemble(L)

        bcs = update_boundary(t=dt)
        for bc in bcs:
                bc.apply(A,b)

        u1 = Function(V)
        solve(A, u1.vector(), b)

        if exact:
                exact.t = dt
                u_e = interpolate(exact, V)
                max_error = np.max(u_e.vector().array() - u1.vector().array())
                print "first step: ",max_error
```

```
        #-----------------------------------------------------------------------#

        # Governing balance equation for the remaining time steps

        F = (dot(u,v) - 2*dot(u1,v) + dot(u2,v) - dt**2*dot(f,v))*dx + \
                (dt**2/rho)*inner(sigma(u1), grad(v))*dx - dot(dot(sigma(u),n),v)*ds
        #a = lhs(F)
        #A = assemble(a)
        u = Function(V)
        T = 1.0
        t = 2*dt
        counter = 2
        t1 = time.clock()
        while t <= T:
                print t
                # Extract linear and bilinear forms from F
                L = rhs(F)
                bcs = update_boundary(t=t)
                b = assemble(L)
                for bc in bcs:
                        bc.apply(A,b)
                solve(A, u.vector(), b)
                u2.assign(u1)
                u1.assign(u)
                #u = TrialFunction(V)

                if exact:
                        exact.t = t
                        u_e = interpolate(exact, V)
                        max_error = np.max(u_e.vector().array() - u1.vector().array())
                        print max_error

                t += dt
                counter += 1

                # plot(u1, mode="displacement", axes=True, title="t=%g"%(t-dt),\
                #  label="t=%g"%(t-dt), basename="time_dependent_elast")
                # #title("t="+(t-dt))
                # # plot(mesh)
                # interactive()
        t2 = time.clock()
        print "Time taken: ", t2-t1


run_simulation("test-case")
```

A.2.3 lin_elast_matrix.py

```
from dolfin import *
import numpy as np
import time


def run_simulation(version):
        if version=="test-case":

                # Create mesh
                mesh = UnitCube(8,8,8)

                # Create function space
                V = VectorFunctionSpace(mesh, "Lagrange", 1)

                # Elasticity parameters
                E, nu = 1.0, 0.1
                mu, lmbda = E/(2.0*(1.0 + nu)), E*nu/((1.0 + nu)*(1.0 - 2.0*nu))
```

```
        rho = 1.0
        alpha = 1.0
        gamma = -lmbda*alpha/(2*(mu + lmbda))
        dt = 0.1

        # Source term
        b = Expression(("2.0*alpha*x[0]", "2.0*gamma*x[1]", "2.0*gamma*x[2]"),\
         alpha=alpha, gamma=gamma)

        # Initial conditions
        u0 = Expression(("0.0", "0.0", "0.0"))
        v0 = Expression(("0.0", "0.0", "0.0"))

        exact = Expression(("t*t*alpha*x[0]", "t*t*gamma*x[1]", "t*t*gamma*x[2]"),\
         alpha=alpha, gamma=gamma, t=0.0)

        solver(mesh, V, u0, v0, b, mu, lmbda, rho, alpha, gamma, dt, exact=exact)


def solver(mesh, V, u0, v0, f, mu, lmbda, rho, alpha, gamma, dt, version="lvp", exact=None):

    # Create test and trial functions
    u = TrialFunction(V)
    v = TestFunction(V)
    n = FacetNormal(mesh)

    def eps(u):
        return (1.0/2.0)*(nabla_grad(u) + transpose(nabla_grad(u)))

    # Stress
    def sigma(u):
         return  2*mu*eps(u) + lmbda*tr(eps(u))*Identity(v.cell().d)


    # Boundary conditions
    def left_boundary(x, on_boundary):
        tol = 1E-14
        return on_boundary and abs(x[0]) < tol

    def right_boundary(x, on_boundary):
        tol = 1E-14
        return on_boundary and abs(x[0] - 1) < tol

    def update_boundary(t=0.0):
        r = Expression(("0.0","t*t*gamma*x[1]","t*t*gamma*x[2]"), gamma=gamma, t=t)
        l = Expression(("t*t*alpha","t*t*gamma*x[1]","t*t*gamma*x[2]"), \
                alpha=alpha, gamma=gamma, t=t)

        bc_left = DirichletBC(V, r, left_boundary)
        bc_right = DirichletBC(V, l, right_boundary)
        bcs = [bc_left, bc_right]
        return bcs



    #------------------- Special case for first time step ----------------------#
    u2 = interpolate(u0, V)

    F = (dot(u,v) - dot(u2,v) - dt*dot(v0,v) - (dt**2/2)*dot(f,v))*dx + \
    dt**2/(2.0*rho)*inner(sigma(u2),nabla_grad(v))*dx - dt**2/(2.0*rho)*dot(dot(sigma(u2),n),v)*ds

    a, L = lhs(F), rhs(F)

    A = assemble(a)
    b = assemble(L)

    bcs = update_boundary(t=dt)
```

```
        for bc in bcs:
                bc.apply(A,b)

        u1 = Function(V)
        solve(A, u1.vector(), b)

        if exact:
                exact.t = dt
                u_e = interpolate(exact, V)
                max_error = np.max(u_e.vector().array() - u1.vector().array())
                print "first step: ",max_error
        #-----------------------------------------------------------------------------#


        # Governing balance equation for the remaining time steps

        # F = (dot(u,v) - 2*dot(u1,v) + dot(u2,v) - dt**2*dot(f,v))*dx + \
        #         (dt**2/rho)*inner(sigma(u1), grad(v))*dx - dot(dot(sigma(u),n),v)*ds
        a_m = dot(u,v)*dx
        a_k1 = inner(sigma(u),nabla_grad(v))*dx
        a_k2 = inner(dot(sigma(u),n), v)*ds
        size = len(u1.vector().array())
        #m = Function(V)
        #m.vector().array()[:] = np.ones(size)
        #Ml = inner(Identity(v.cell().d), m.vector())
        M = assemble(a_m)
        K1 = assemble(a_k1)
        K2 = assemble(a_k2)
        A = M
        u = Function(V)
        T = 1.0
        t = 2*dt
        counter = 2
        t1 = time.clock()
        while t <= T:
                print t
                f_k = interpolate(f, V)
                bcs = update_boundary(t=t)
                b = 2*M*u1.vector() + dt**2/rho*(K2*u1.vector()- K1*u1.vector()) - M*u2.vector() + \
                dt**2*M*f_k.vector()

                for bc in bcs:
                        bc.apply(A,b)
                solve(A, u.vector(), b)
                u2.assign(u1)
                u1.assign(u)
                #u = TrialFunction(V)

                if exact:
                        exact.t = t
                        u_e = interpolate(exact, V)
                        max_error = np.max(u_e.vector().array() - u1.vector().array())
                        print max_error

                t += dt
                counter += 1

                # plot(u1, mode="displacement", axes=True, title="t=%g"%(t-dt),\
                #  label="t=%g"%(t-dt), basename="time_dependent_elast")
                # #title("t="+(t-dt))
                # # plot(mesh)
                # interactive()
        t2 = time.clock()
        print "Time taken: ", t2-t1


run_simulation("test-case")
```