Nonlinear diffusion equation with finite elements

Fredrik E Pettersen
fredriep@student.matnat.uio.no

December 11, 2012

Contents

1 About the problem

In this project we will solve a nonlinear diffusion equation using the finite element method and the FEniCS software package. More precicely we will be looking at the equation

$$\rho u_t = \nabla \cdot (\alpha(u)\nabla u) + f(\mathbf{x}, t) \tag{1}$$

where ρ is a constant, $\alpha(u)$ is a known function of u, and where we have the initial condition $u(\mathbf{x}, 0) = I(\mathbf{x})$, and Neuman boundary conditions $\frac{\partial u}{\partial n} = 0$.

2 Discretization

In order to solve equation (1) we will need to discretize it. The standard approach is then to pick a nummerical approximation to the derivatives. We will use a finite difference discertization in time, more specifically the Backward Euler (BE) discretization, and a finite element approximation in space. The BE discertization gives us

$$\rho u^n = \Delta t \nabla \cdot (\alpha(u)\nabla u) + \Delta t f(\mathbf{x}, t) + \rho u^{n-1}$$
$$\rho u^n - \Delta t \nabla \cdot (\alpha(u)\nabla u) - \Delta t f(\mathbf{x}, t) - \rho u^{n-1} = R$$

we now approximate R on a functionspace $V = span\{\phi_i\}$, where ϕ_i denotes the P1 elements, and minimize the error by demanding

$$\big(R, v\big) = 0 \quad \forall\, v \in V$$

which gives us

$$\rho(u^n, v) - \Delta t(f(\mathbf{x}, t_n), v) - \rho(u^{n-1}, v) - \Delta t(\nabla \cdot (\alpha(u)\nabla u^n), v) = 0$$

The first three terms are ok for now, but the last term has a double derivative in u, which is something we dont want seeing as we are trying to approximate u by P1 elements as $u \simeq \sum_k u_k\phi_k$, and the double derivative of a linear function is simply zero. To get around this problem we try to integrate by parts

$$\Delta t(\nabla \cdot (\alpha(u)\nabla u^n), v) = \Delta t \left(\int_\Omega \nabla \cdot (\alpha(u)\nabla u^n) v dx \right)$$
$$= \Delta t \left([\alpha(u)\nabla u v]_{\partial\Omega} - \int_\Omega \alpha(u)\nabla u^n \nabla v dx \right) = -\Delta t \int_\Omega \alpha(u)\nabla u^n \nabla v dx$$

where we have inserted for the boundary conditions $\frac{\partial u}{\partial n} = \mathbf{n} \cdot \nabla u = 0$ on $\partial\Omega$. We have now arrived on a linear variational problem to be solved at each timestep

$$\rho(u^n, v) - \Delta t(f(\mathbf{x}, t_n), v) - \rho(u^{n-1}, v) + \Delta t(\alpha(u)\nabla u^n, \nabla v) = 0$$
$$\underbrace{\rho(u^n, v) + \Delta t(\alpha(u)\nabla u^n, \nabla v)}_{a(u,v)} = \underbrace{\Delta t(f(\mathbf{x}, t_n), v) + \rho(u^{n-1}, v)}_{L(u,v)}$$

Notice that we have a nonlinearity in $\alpha(u)$. There are several ways to get around this nonlinearity, we will use Picard iteration, meaning that we start out with $\alpha(u^{n-1})$ and solve the linear system to get a solution $\tilde{u}^n$. Then we update $\alpha(\tilde{u}^n)$ and solve the system again. We keep dionig this untill $\|\tilde{u}^n - \tilde{u}^{n-1}\| \le \epsilon$ where ϵ is a predefined tolerance. Hopefully this converges to the correct solution, and we can set our $\tilde{u}^n = u^n$. Picard iteration leads us to the following linear system for each iteration

$$\rho \int_\Omega u^n v dx + \Delta t \int_\Omega \alpha(u^{n-1})\nabla u^n \nabla v dx = \rho \int_\Omega u^{n-1} v dx + \Delta t \int_\Omega f(x, t_n) v dx$$

we will now insert for $u \simeq \sum_k u_k\phi_k$ and because we are working with an undefined number of cells of potentially variable size we will transform all the integrals to a reference cell where $-1 \le X \le 1$. We will also do the calculations in 1 dimension, but they easily generalize to multiple dimensions by simply setting each element in say the vector $\mathbf{u}^n$ equal to a vector. The same approach gives us a block matrix for M and K.

$$\rho \sum_{i=1}^{N_x} \int_{-1}^{1} (\phi_i\phi_j) u_i^n dx + \Delta t \sum_{i=1}^{N_x} \int_{-1}^{1} \alpha(u_i^{n-1})(\phi_i'\phi_j') u^n dx = \rho \sum_{i=1}^{N_x} \int_{-1}^{1} (\phi_i\phi_j) u_i^{n-1} dx + \Delta t \sum_{i=1}^{N_x} \int_{-1}^{1} f_i(x, t_n)\phi_j dx$$

If we now set $M_{ij} = \rho \int_0^L (\phi_j \phi_i) dx$ and $K_{ij} = \Delta t \int_0^L \alpha(u^{n-1})(\phi_j' \phi_i') dx$ we get

$$M_{ij} = \rho \int_0^L (\phi_j \phi_i) dx \implies M_{00} = \rho \int_{-1}^1 (\phi_0 \phi_0) dx$$

We are using P1 elements, and so the only nonzero integrals we will get are when $i = j$ and $i \pm 1 = j$, meaning M and K will become tridiagonal. Notice that $\alpha(u^{n-1})$ is known since $\alpha(u)$ is a known function, and u^{n-1} is simply the previous timestep which is known.

$$M_{ii} = \rho \int_{-1}^1 (\phi_i \phi_i) dx, \quad M_{ij} = M_{ji} = \rho \int_{-1}^1 (\phi_j \phi_i) dx$$

$$K_{ii} = \alpha(u_{i,i}^{n-1}) \int_{-1}^1 (\phi_i' \phi_i') dx, \quad K_{ij} = K_{ji} = \alpha(u_{i,j}^{n-1}) \int_{-1}^1 (\phi_j' \phi_i') dx$$

Remember that for every M_{ii} and K_{ii} entry there are two possible function combinations because within each element there are defined two functions say $\phi_0 = \frac{1}{2}(1-x)$ and $\phi_1 = \frac{1}{2}(1+x)$. This means that for every diagonal entry we need to calculate the sum of two integrals (or just multiply by 2 since they are equal). The integrals become

$$M_{ii} = \int_{-1}^1 (\phi_i \phi_i) dX = 2\frac{h}{8} \int_{-1}^1 (1-X)^2 dX = 2\frac{h}{8} \cdot \frac{8}{3} = \frac{2h}{3}$$

$$M_{ij} = \int_{-1}^1 (\phi_i \phi_j) dX = \frac{h}{8} \int_{-1}^1 (1-X)(1+X) dX = \frac{h}{8} \cdot \frac{4}{3} = \frac{h}{6}$$

$$K_{ii} = 2\alpha(u_{i,i}^{n-1}) \int_{-1}^1 \phi_i' \phi_i' dX = 2\alpha(u_{i,i}^{n-1}) \int_{-1}^1 (\frac{2}{h}\phi_i' \frac{2}{h}\phi_i') \frac{h}{2} dX$$

$$= 2\alpha(u_{i,i}^{n-1}) \frac{2}{h} \int_{-1}^1 \frac{-1}{2}\frac{-1}{2} dX = 2\alpha(u_{i,i}^{n-1}) \frac{1}{h}$$

$$K_{ij} = \alpha(u_{i,j}^{n-1}) \int_{-1}^1 \phi_i' \phi_j' dX = \alpha(u_{i,j}^{n-1}) \frac{2}{h} \int_{-1}^1 \frac{-1}{2}\frac{1}{2} dX = \alpha(u_{i,j}^{n-1}) \frac{-1}{h}$$

This makes our matrices

$$M = \rho \frac{h}{6} \begin{pmatrix} 4 & 1 & 0 & \dots & & 0 \\ 1 & 4 & 1 & 0 & \dots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 & \\ 0 & \dots & 0 & 1 & 4 & 1 \\ 0 & & \dots & 0 & 1 & 4 \end{pmatrix}, \quad K = \alpha(u^{n-1}) \frac{1}{h} \begin{pmatrix} 2 & -1 & 0 & \dots & & 0 \\ -1 & 2 & -1 & 0 & \dots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 & \\ 0 & \dots & 0 & -1 & 2 & -1 \\ 0 & & \dots & 0 & -1 & 2 \end{pmatrix} \tag{2}$$

Setting the $\alpha(u^{n-1})$ is not strictly correct, since it is most likely different at each meshpoit, but it simplifies writing a bit. We are now left with

$$M\mathbf{u}^n + K(u^{n-1})\mathbf{u}^n = M\mathbf{u}^{n-1} + \mathbf{b}$$
$$\implies \mathbf{u}^n = (M + K(u^{n-1}))^{-1}(M\mathbf{u}^{n-1} - \mathbf{b})$$

to be solved for each Picard iteration. The crudest Picard iteration will be just one iteration, which is the same as solving the system using $\alpha(u^{n-1}) \simeq \alpha(u^n)$ and this is our chosen approach.

2.1 Group Finite Element method

We can also approximate the nonlinear term by the group finite element method. Looking at the variational form of our equation and inserting for $u \simeq \sum_{i=1}^{N} \phi_i u_i$ we get the following

$$\rho \int_\Omega \sum_i (\phi_i \phi_j) u_i^n d\Omega + \Delta t \int_\Omega \alpha(\sum_i \phi_i u_i^n) \sum_j (\nabla \phi_j \nabla \phi_k) u_j^n d\Omega = \rho \int_\Omega \sum_i (\phi_i \phi_j) u_i^{n-1} d\Omega + \Delta t \int_\Omega f(x,t_n) v d\Omega$$

If we now set $\alpha(\sum_i \phi_i u_i^n) \approx \sum_i \phi_i \alpha(u_i^n)$ and focus on the integral with α, we are left with (in one dimension)

$$\int_x \sum_i \phi_i \alpha(u_i^n) \sum_j (\phi_j' \phi_k') u_j^n dx$$

Which we transfer to a reference element $X \in [-1,1]$ where $\frac{d\phi}{dX} = \frac{d\phi}{dX}\frac{dX}{dx} = \frac{d\phi}{dX}\frac{2}{h}$ and $\frac{dx}{dX} = \frac{h}{2}$. Theese calculations will result in an element matrix which we can assemble so that we get a linear system. We will integrate by the trapezoidal rule since we do not know what $\alpha(u^n)$.

$$\frac{(-1)^r(-1)^s}{h} \int_{-1}^{1} \alpha(u_i)\phi_i \phi_j' \phi_k' dX$$

$$\approx \frac{(-1)^r(-1)^s}{2h} \Big(\underbrace{\sum_r \alpha(u_{i+r}) \cdot \phi_{i+r}}_{\neq 0 \text{ when } r=1} + \underbrace{\sum_s \alpha(u_{i+s}) \cdot \phi_{i+s}}_{\neq 0 \text{ when } s=0} \Big)$$

$$\implies \frac{1}{2h}(\alpha(u_{i+1}) + \alpha(u_i)) \cdot \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} = K^{(i)}$$

Where $K^{(i)}$ denotes the i'th diagonal element in the matrix K.
The entire linear system is then

$$M\mathbf{u}^n + \Delta t K \mathbf{u}^n = M\mathbf{u}^{n-1} + \mathbf{b}$$

where M is the same matrix as in equation (2). This means that the i'th equation in the system is

$$\frac{h}{6}\big(u_{i+1}^n + 4u_i^n + u_{i-1}^n\big) - \frac{\Delta t}{2h}\big(\alpha(u_{i+1}^n) + \alpha(u_i^n)\big)\big(u_{i+1}^n - 2u_i^n + u_{i-1}^n\big) - \frac{h}{6}\big(u_{i+1}^{n-1} + 4u_i^{n-1} + u_{i-1}^{n-1}\big) = \Delta t f(x_i, t_n)$$

If we massage this expression a bit we can get it on another form.

$$\frac{u_i^n - u_i^{n-1}}{\Delta t} + \frac{u_{i+1}^n - u_{i+1}^{n-1}}{6\Delta t} - \frac{2(u_i^n - u_i^{n-1})}{6\Delta t} + \frac{u_{i-1}^n - u_{i-1}^{n-1}}{6\Delta t}$$
$$= \frac{1}{2h}\big(\alpha(u_{i+1}^n) + \alpha(u_i^n)\big)\big(u_{i+1}^n - 2u_i^n + u_{i-1}^n\big) + f(x_i, t_n)$$

which looks suspiciously like a finite difference scheme. In fact we recognize the left hand side as (if we assume $h = \Delta x$)

$$D_t u + \frac{1}{6} D_t \Delta x^2 D_x D_x u$$

Looking a bit closer at the right hand side, this looks like a normal second order derivative of u wrt x multiplied by the average of $\alpha(u_{i+1/2}^n)$ and of course the source term. Now, our equation has a right hand side on the form (in one spatial dimension)

$$\frac{\partial}{\partial x}\alpha(u)\frac{\partial}{\partial x}u$$

and so it would seem that if we (for some reason) approximate this with $\frac{\partial}{\partial x}\alpha(u)\frac{\partial}{\partial x}u \approx \alpha(\bar{u})\frac{\partial^2 u}{\partial x^2}$ we arrive at a finite difference scheme equivalent to using the group finite element method

$$\left[D_t \left(u + \frac{1}{6}\Delta x^2 D_x D_x u \right) = \alpha(\bar{u}) D_x D_x u + f \right]_i^n$$

2.2 Newtons Method

As a different approach we can use Newtons method to estimate the nonlinear term in our equation (1). Newtons method is quite simply an iterative method where we linearize our problem through a first order Taylor expansion, and can be expressed as

$$\mathbf{F}(\mathbf{u}) = 0 \simeq M(u; u^q) = F(u^q) + J(u - u^q)$$

where $J = \nabla F$ which means that $J_{i,j} = \frac{\partial F_i}{\partial u_j}$ and $\mathbf{F}(\mathbf{u})$ is the equation we want to solve only written on a general form. If we reformulate a bit we find

$$\begin{aligned} J(u^{q+1} - u^q) &= -F(u^q) \\ u^{q+1} &= u^q - J^{-1}F(u^q) \end{aligned}$$

which is the very same formulation as in 1D $x^{k+1} = x^k - \frac{f(x^k)}{f'(x^k)}$ We wish to apply this on the variational form of the equation which ends up as our F: (notice that we set $\rho = 1$)

$$\begin{aligned} F_i(u) &= \int_\Omega \big(u_i^n v + \Delta t(\alpha(u_i^n)\nabla u_i^n \nabla v) - u_i^{n-1} v - \Delta t f(\mathbf{x}, t) v\big)\, d\Omega \\ J_{i,j} &= \frac{\partial}{\partial u_j^n} F_i \\ J_{i,j} &= \int_\Omega (\phi_i\phi_j + \Delta t(\alpha'(u_i^n)\phi_i \nabla u_i^n \nabla \phi_j + \alpha(u_i^n)\nabla\phi_i\nabla\phi_j))\, d\Omega \qquad (3) \end{aligned}$$

where we have used $u_i^n = \phi_i u_i^n \implies \frac{\partial}{\partial u_j^n} u_i^n = \phi_i$, $v = \phi_j$ and $\frac{\partial}{\partial u_j^n}\alpha(u_i^n) = \alpha'(u_i^n)\phi_i$. And there we have the expressions for the entries in the jacobian matrix we can use for a Newtons method on our equation. The remainig steps are to assemble the linear problem, and iterate in the same way as we did with Picard iteration. Theese are both tasks well suited for the FEniCS software.

Simplfying (3) to one spatial dimension lets us compute the actual entries in the jacobian in a simple manner so that we can compare with something else. The simplified version is

$$J_{i,j} = \int_x \big(\phi_i\phi_j + \Delta t(\alpha'(u_i^n)\phi_i \cdot (u_i^n)'\phi_j' + \alpha(u_i^n)\phi_i'\phi_j')\big)\, dx$$

We notice that we have allready calculated the first and last term and so we can focus on the middle term. Again we will integrate by the trapezoidal rule.

$$\int_{-1}^{1} \alpha'(u^n)\phi_i \cdot (u^n)' \frac{2}{h}\phi_j' \frac{h}{2} dX = \frac{(-1)^r}{h}\Big(\alpha'(u_{i+s}^n)(u_i^n)'\delta_{s_0}\,\big|_{X=-1} + \alpha'(u_{i+s}^n)(u_i^n)'\delta_{s_1}\big|_{X=1}\Big)$$

we approximate the derivative of u by $(u_i^n)' \simeq \frac{1}{2}(u_i^n - u_{i+1}^n)$ and tidy up the expression a bit to get

$$\frac{1}{h}(u_i^n - u_{i+1}^n)\begin{bmatrix} \alpha'(u_i^n) & \alpha'(u_{i+1}^n) \\ -\alpha'(u_i^n) & -\alpha'(u_{i+1}^n) \end{bmatrix}$$

Combining all of the entries then gives us the Jacobian

$$\begin{aligned} J_{i,i} &= \frac{\Delta t}{h}\left((u_i^n - u_{i+1}^n)(\alpha'(u_i^n) - \alpha'(u_{i+1}^n)) + (\alpha(u_i^n) + \alpha(u_{i+1}^n)) + \frac{2}{3\Delta t}\right) \\ J_{i,i+1} &= \frac{\Delta t}{h}\left(-(u_i^n - u_{i+1}^n)\alpha'(u_{i+1}^n) + \frac{1}{2}(\alpha(u_i^n) + \alpha(u_{i+1}^n)) + \frac{1}{6\Delta t}\right) \\ J_{i,i-1} &= \frac{\Delta t}{h}\left((u_i^n - u_{i+1}^n)\alpha'(u_i^n) + \frac{1}{2}(\alpha(u_i^n) + \alpha(u_{i+1}^n)) + \frac{1}{6\Delta t}\right) \end{aligned}$$

3 Implementation

To make the program as general as possible, we will make it dimension independent by providing the number of spatial points on the commandline as n_x, n_y, n_z and choose a unit interval, square or cube accordingly. Also, even though we will only use one Picard iteration, we will implement this as a function where the maximum nuber of iterations is given as an argument. The resulting code can be found in the appendix.

4 Verification

For the first verification of the program we will assume $\rho = \alpha(u) = 1$, $f(\mathbf{x}, t) = 0$ and $I(\mathbf{x}) = \cos(\pi x)$. We will work on the domain $\Omega = [0, 1] \times [0, 1]$. This should give an analytic solution of $u(x, y, t) = e^{-\pi^2 t} \cos(\pi x)$. Since we have used the Backward Euler discertization in time we should have an error of Δt, and therefore also a first order convergence. An important quality of the error should be that the error per timestep is approximately constant per timestep. That is if $\Delta t = \Delta x^2$, and we call the error E we should have $E/\Delta t$ remains approximately constant for the same time when the mesh is refined. We calculate the error by

```
e = u_e.vector().array() - u.vector().array()
E = np.sqrt(np.sum(e**2)/u.vector().array().size)
```

Where u_e is a projection of the exact solution on to the function space used for the numerical solution u. Below we have listed the output from computing $E/\Delta t$ for $t \simeq 0.005$ for different resolutions in the mesh.

```
fredriep@WorkStation:~/uio/INF5620/final_project$ python exc_cd.py 1 40 40
error:   0.175761870575   t =   0.005

fredriep@WorkStation:~/uio/INF5620/final_project$ python exc_cd.py 1 67 67
error:   0.172341502209   t =   0.00512363555358

fredriep@WorkStation:~/uio/INF5620/final_project$ python exc_cd.py 1 80 80
error:   0.167433801143   t =   0.005

fredriep@WorkStation:~/uio/INF5620/final_project$ python exc_cd.py 1 100 100
error:   0.165526711019   t =   0.005

fredriep@WorkStation:~/uio/INF5620/final_project$ python exc_cd.py 1 180 180
error:   0.16200056506   t =   0.005
```

and as expected this relation remains approximately the same for increasingly finer meshes.
We can then use the method of manifactured solutions to do another verification of the program. Restricting ourselves to one spatial dimension we choose our solution to be

$$u(x, t) = t \int_0^x q(1 - q)dq = tx^2 \left(\frac{1}{2} - \frac{x}{3}\right) \tag{4}$$

and our $\alpha(u) = 1 + u^2$. We can then calculate a function $f(x, t)$ which makes the solution fit the equation. This solution is given from an interactive sympy session as

$$f(x, t) = \rho x^2 \left(\frac{1}{2} - \frac{x}{3}\right) + x^4 t^3 \left(\frac{8.0x^3}{9} - \frac{28x^2}{9} + \frac{7x}{2} - \frac{5}{4}\right) + t(2x - 1) \tag{5}$$

Plugging this into FEniCS and comparing the nummeric and analytic solution gives the absolute max error as indicated in the output below.

```
fredriep@WorkStation:~/uio/INF5620/final_project$ python exc_cd.py 1 24
Calling FFC just-in-time (JIT) compiler, this may take some time.
error:   0.00295070292965   t =   0.2
error:   0.00443136110942   t =   0.3
error:   0.00517399129169   t =   0.4
error:   0.00554256396031   t =   0.5
error:   0.00571840378958   t =   0.6
```

Δt is here (as you can see) 0.1, and we notice that the error is of the order 10^{-3} which is significantly less than Δt. This makes sense seeing as the analytic solution is a polynomial of degree 7 meaning that for the spatial part the terms up to x^3 (assuming that gaussian quadrature is default) are represented to machine precision and the error is therefore in the higher order terms. At least the error is better than $\mathcal{O}(\Delta t)$ and that is a very good sign.

4.1 Errors

As in any (numerical) approximation there will be an error in our solution. We can distinguish two types of errors, there is the error we do in the approximation of the derivatives, and there is the nummerical error.

Our approximation using the Backward Euler scheme has an error which goes like $\mathcal{O}(\Delta t)$ for other reasons the spatial error goes like $\mathcal{O}(\Delta x^2)$. Other potential errors which will arise in the FEniCS program are errors due to loss of prescision in the nummerical integration we do upon assembly of the matrices M and K. Depending on what kind of nummerical integration we use, the error from integration could be $\mathcal{O}(\Delta x)$ for the rectangle (midpoint) rule or it could even be zero if the solution is a polynomial of degree $2n-1$ and we integrate by some form of Gaussian quadrature. n is here the order of elements used. There is allways a possibillity of loosing nummerical precision because of adding a small and a large number since we only have 16 valid decimals. In our case we also have an obvious error in the approximation of the nonlinear term.

5 Results

As a final experiment we shall simulate the nonlinear equation using Picard iteration with one iteration as described in section 2 and the Backward Euler scheme on the unit square. The initial contition on the system is

$$u(x,y,0) = I(x,y) = \exp\big(-\frac{1}{2\sigma^2}(x^2+y^2)\big) \tag{6}$$

and the nonlinear term is $\alpha(u) = 1 + \beta u^2$. The expected behavioure of this system (at least without the nonlinear term) is that it should simply die down. I have included some plots from this simulation in figures 1, 2 and 3.

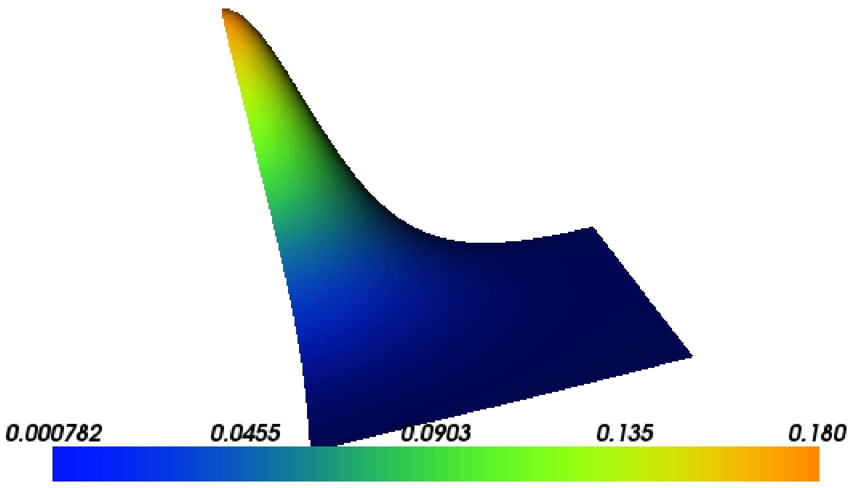

Figure 1: The initial condition on the system

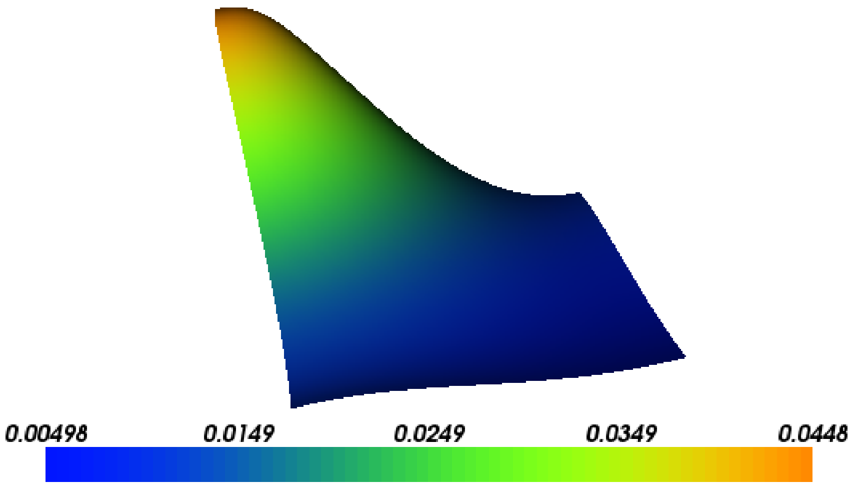

Figure 2: The system early in the simulation (t = 0.6)

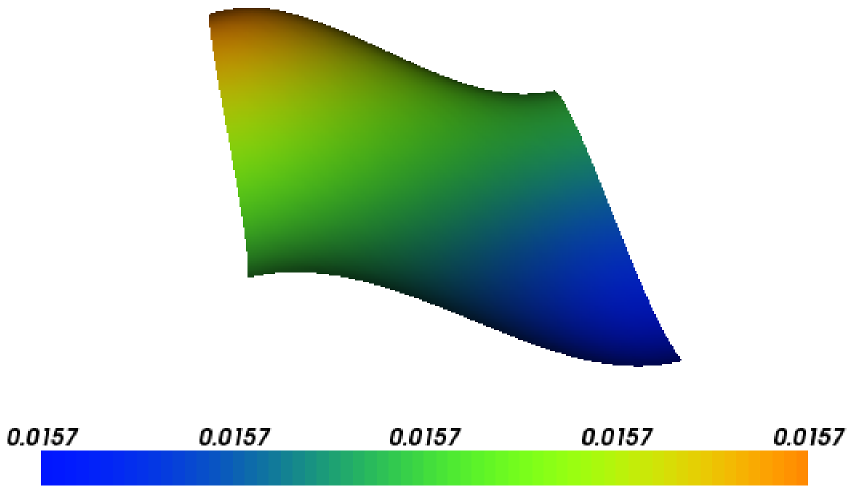

Figure 3: The system in the end of the simulation (t=1.4). Notice that the range of colors vary from 0.0157 to 0.0157.

6 Final comments

The program used to solve the numeric part of this project can be found in the appendix.

A Source code

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

degree = int(sys.argv[1])
divisions = [int(i) for i in sys.argv[2:]]
d = len(divisions) -1
domain_type = [UnitInterval,UnitSquare,UnitCube]
mesh = domain_type[d](*divisions)

def alpha(u,beta = 1.3):
        return 1.0#+beta*u**2

def picard(u,u_1,a,L,b,maxiter,tol=1e-5,order=2):
        iter = 0;eps = 1.0
        while(eps>tol and iter<maxiter):
                iter +=1
                #bc.apply(A,b)
                solve(A,u.vector(),b)
                diff = u.vector().array()-u_1.vector().array()
                eps = np.linalg.norm(diff,ord=order)
                u_1.assign(u)
        #print iter
        return None

maxiter = 1
rho = 1.0
dt = (1./divisions[0])**2
sigma = 0.1

V = FunctionSpace(mesh,'Lagrange',degree)
f = Expression('rho*x[0]*x[0]*(1.0/2-x[0]/3.0) +pow(x[0],4)*pow(t,3)*(8.0*pow(x[0],3)/9.0
                                                             7.0*x[0]/2.0  -5.0/4.0) +t*(2.0*x[0]  -1.0)',rho=rho,t=0.0)#
#f = Constant(0.0)
u = TrialFunction(V)
v = TestFunction(V)

#u0 = Expression('exp(-1/(2*sigma*sigma)*(x[0]*x[0]+x[1]*x[1]))',sigma=sigma)
#u0 = Expression('cos(pi*x[0])',pi=pi)
u0 = Constant(0.0)
u_1 = interpolate(u0,V)

def u0_boundary(x,on_boundary):
        return on_boundary

#bc = DirichletBC(V,u0,u0_boundary)

a = u*v*dx + dt*inner(alpha(u_1)*nabla_grad(u),nabla_grad(v))*dx
L = (u_1 +dt*f)*v*dx

A = assemble(a)
u = Function(V)
T = 0.1
t = dt
b = None
exact = Expression('t*x[0]*x[0]*(1.0/2- x[0]/3.0)',t=0.0)
#exact = Expression('exp(-pi*pi*t)*cos(pi*x[0])',pi=pi,t=0)
plt_lst = [0.05,0.1,0.25,0.4]
```

```
counter=0
while t<=T:
        b = assemble(L, tensor=b)
        exact.t = t
        f.t=t
        picard(u,u_1,a,L,b,maxiter)

        u_1.assign(u)
        u_e = interpolate(exact, V)
        #plot(u)
        #interactive()
        #maxdiff = np.abs(u_e.vector().array()-u.vector().array()).max()
        #print 'Max error, t=%.2f: %-10.17f' % (t, maxdiff)
        e = u_e.vector().array() - u.vector().array()
        E = np.sqrt(np.sum(e**2)/u.vector().array().size)
        counter +=1
        if counter%10==0:
                print "error: ",E," t = ",t
        t+=dt
```