INF5620
FEniCS Project

Krister Stræte Karlsen

November 30, 2015

Abstract

In this project a non-linear diffusion equation (2.1) will be solved using the FEniCS software. The solutions produced will be compared to various known analytical solutions.

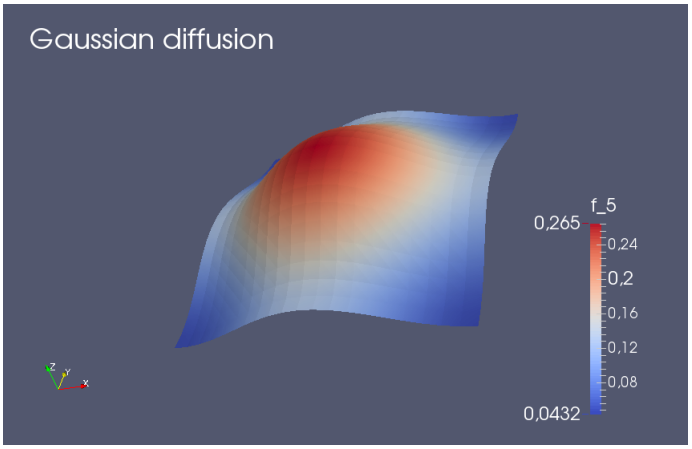

Figure 1: Simulation of diffusion with a Gaussian distribution as initial condition in FEniCS. Paraview used for editing.

The programs for this report are available at:
https://github.com/UiO-INF5620/INF5620-krikarls under *FEniCS_project*.

1 Introduction

In this project we will solve a non-linear diffusion equation using the FEniCS software.

2 The mathematical problem

The full mathematical formulation of the problem reads

$$\rho\frac{\partial u}{\partial t} = \nabla\cdot(\alpha(u)\nabla u) + f(\mathbf{x},t) \tag{2.1}$$

$$\begin{aligned} u(\mathbf{x},0) =& I(\mathbf{x}) \\ \frac{\partial u}{\partial n} =& 0, \quad on \quad \partial\Omega \end{aligned}$$

3 Discretization and variational form

To solve this equation using FEniCS we must get it on variational form, and preferably *weak form*. That is,

$$a(u,v) = L(v), \quad \forall v \in V \tag{3.1}$$

where v is a test function in some test space V and both u and v are differentiable once.

To obtain this variational formulation we start be discretizing the time derivative

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

and multiply by the a test function v

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

Before we integrate over the domain it is convenient to gather all terms from the same time level n on the same side

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

$$\int_\Omega -\rho u^{n-1} v d\mathbf{x} = \int_\Omega \Delta t \nabla \cdot (\alpha(u^n)\nabla u^n) v d\mathbf{x} + \int_\Omega \Delta t f^n(\mathbf{x}) v - \rho u^n v d\mathbf{x}.$$

Now we can apply integration by parts to the first term on the right hand side

$$\begin{aligned}\int_\Omega \nabla \cdot (\alpha(u)\nabla u) v d\mathbf{x} &= \int_{\partial\Omega} (\alpha(u)\frac{\partial u}{\partial n}) v d\mathbf{s} - \int_\Omega \alpha(u)\nabla u \nabla v d\mathbf{x} \\ &= -\int_\Omega \alpha(u)\nabla u \nabla v d\mathbf{x}\end{aligned}$$

Inserted into the original equation and rearrange we get the following

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

or

$$a(u, v) = L(v),$$

which is the weak form of (2.1).

3.1 Linearization and Picard iterations

We want to solve this system of non-linear algebraic equations as a sequence of linear systems. This can be done by letting $u \approx u^-$, where u^- is the latest approximation of u, for the non-linear function $\alpha(u)$. The linearized variational form now reads

$$\int_\Omega \Delta t \alpha(u^-)\nabla u \nabla v d\mathbf{x} + \int_\Omega \rho u v d\mathbf{x} = \int_\Omega \rho u^- v d\mathbf{x} \int_\Omega \Delta t f(\mathbf{x}, t) v d\mathbf{x},$$

or as a linear system

$$A(u^-)u = b(u^-).$$

3.2 Single Picard iteration

Using only one Picard iteration corresponds to using latest known approximation of u for $\alpha(u)$, which is u^-. An example of an implementation of this method can be found in the script `diffusion_constant_test.py`.

4 Verification

To verify that FEniCS implementation of the problem runs correctly a number of test cases with known solutions will be investigated. For all the test cases the error will be computed according to

$$E = \sqrt{\frac{1}{N}\sum_{i}^{N}(u_{e(i)} - u_i)^2} \tag{4.1}$$

which is the *discrete L2-norm*. This was typically done for chosen time level t.

The convergence rate can then be computed as

$$r_i = \frac{log(E_{i-1}/E_i)}{log(h_{i-1}/h_i)}$$

for a number of different numerical simulations, $.., i-1, i, ..$ using different $h_i = \Delta t = \Delta x^2$.

4.1 Constant solution

As a first verification of our implementation we will use a constant solution, $u(\mathbf{x}, t) = C$. Inserting this into the PDE (2.1) we find that $f = 0$, $I = C$, while ρ and α can remain arbitrary.

A program confirming that the constant solution is reproduced can be found under the name `diffusion_constant_test.py`. The program takes user-given arguments for the number of spatial dimensions and choice of elements.

4.2 Simple analytical solution

The second verification of the FEniCS implementation is a simple analytical solution

$$u_e(\mathbf{x}, t) = e^{-\pi^2 t} cos(\pi x).$$

To obtain this results the parameters in the FEniCS solver was set to:

$$\begin{aligned} I(\mathbf{x}) &= cos(\pi x) \\ f &= 0 \\ \alpha(u) &= 1 \\ \rho &= 1 \end{aligned}$$

Simulations were done using the input above and the program
`diffusion_simple_analytical_solution.py`

The results from comparing numerical and exact solution can be found in table 1.

Table 1: Data from numerical experiments. Computing error according to (4.1) while reducing h. All measurements done at T=1.0.

E	**h**	**K=E/h**	**r**
3.04713699452e-05	0.015625	0.00195016767649	1.43456667265
6.3143838985e-06	0.00390625	0.00161648227802	1.13537024109
1.49266429934e-06	0.0009765625	0.00152848824253	1.04037611806
3.66513686511e-07	0.000244140625	0.00150124005995	1.01297538059

Looking at the data in table 1 one can observe that $E/h = K$ approches a constant as the mesh is refined and that the expected convergence rate is recovered.

4.3 Manufactured solution(1)

The next verification is of the form

$$u_e(x,t) = tx^2\left(\frac{1}{2} - \frac{x}{3}\right), \quad and$$
$$\alpha(u) = 1 + u^2$$

with a rather nasty source term f which was computed using Python's *sympy* module:

```
...
print simplify(f)
print f.simplify() -rho*x**3/3 + rho*x**2/2 + 8*t**3*x**7/9
- 28*t**3*x**6/9 + 7*t**3*x**5/2 - 5*t**3*x**4/4 + 2*t*x -
t
```

This time the solution will be studied graphically by simply plotting analytical and numerical solution together.

One can see that the numerical solution follows the exact very close, but for longer simulations, larger T, the difference between the two increases.

These results were obtained by running the program
`diffusion_manufactured_solution1.py`

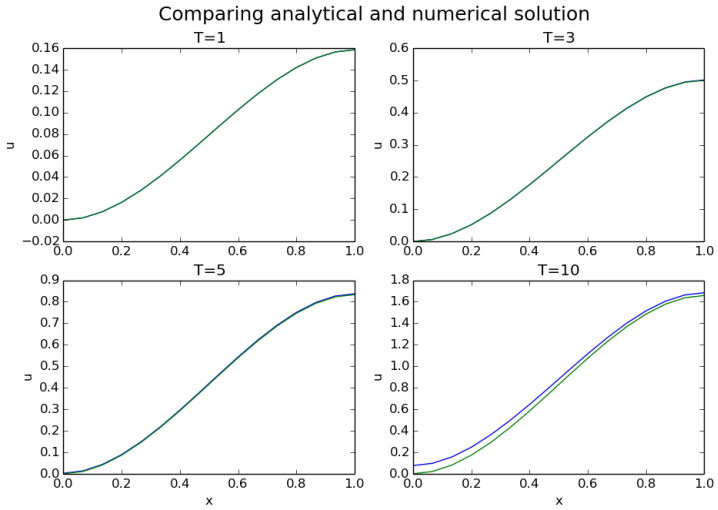

Figure 2: The numerical solution is very good for short simulations, but for longer(larger T) one can see the difference between u_e and u increase .

4.4 Manufactured solution(2)

In the last verification a manufactured solution fulfilling the PDE (2.1) for $\alpha(u^-)$, where u^- is the solution from the previous time level, will be used. This is useful in order to eliminate the error from using only a single Picard iteration. The solution is found by again adjusting the source term f accordingly using sympy:

```
...
print simplify(f)
rho*x**2*(-2*x + 3)/6 - (-12*t*x + 3*t*(-2*x +
3))*(x**4*(-dt + t)**2*(-2*x + 3)**2 + 36)/324 - (-6*t*x**2
+ 6*t*x*(-2*x + 3))*(36*x**4*(-dt + t)**2*(2*x - 3) +
36*x**3*(-dt + t)**2*(-2*x + 3)**2)/5832
```

u and $\alpha(u)$ remains the same as the previous case

$$u_e(x,t) = tx^2\left(\frac{1}{2} - \frac{x}{3}\right), \quad and$$
$$\alpha(u) = 1 + u^2.$$

Running numerical simulations in the same manner as done under section (2.1) the following error and convergence data were obtained:

Table 2: Data from numerical experiments. Computing maximum error according to (4.1) while reducing h. All measurements done at T=1.0.

E	**h**	**K=E/h**	**r**
1.9156751783e-05	0.015625	0.00122603211411	0.999424765677
4.87436601586e-06	0.00390625	0.00124783770006	0.987283233225
1.23275891758e-06	0.0009765625	0.0012623451316	0.991661948957
3.10150170479e-07	0.000244140625	0.00127037509828	0.99542593304
7.77936821335e-08	6.103515625e-05	0.00127457168807	0.997621009286

These results were obtained by running the program
`diffusion_manufactured_solution2.py`

The error decreases and the expected convergence rate is recovered. These findings are promising for the implementation.

4.5 Sources of numerical error

- The time discretization comes from truncating a Taylor series leading to an error of the order $O(\Delta t)$.

- Only a finite set of basis functions from the function space we expect the solution to be in are used. So even without any other error terms one might not be able to recover the exact solution for a "*troublesome*" function.

- The linear system is not set up entirely correct due to small errors from numerical integration.

- I am not as familiar as I should be with the solution process of the linear system, but if iterative methods are used there will be errors.

- Last, but not least, using only one Picard iteration, $(u^n)^2 \approx u^{n-1}u^n$ is not a good estimation for large Δt.

5 Simulation: 2D Gaussian distribution

A simulation of the solution of the diffusion equation (2.1) were done using the 2D Gaussian distribution as initial condition

$$I(\mathbf{x}, t) = exp\left(\frac{1}{2\sigma^2}(x^2 + y^2)\right),$$
$$\alpha(u) = 1 + \beta u^2.$$

One can from the results, fig. 3 , see that as time passes, the solution evens out. The boundaries which initially are zero, slowly increase there value, first at the middle, then the corners. This effect is due to the Neumann condition $\partial u/\partial n = 0$.

The reader is strongly encouraged to take a look at the animation
`Gaussian_diffusion.gif`

Physically this corresponds to, for instance, a substance having an initial concentration(Gaussian) in a 2D space. Say ink dropped in the middle of a a thin quadratic box filled with water. Then, as time passes, the concentration of ink evens out and the system reaches a steady state, where the concentration of ink is the same everywhere in the box. In other words, the ink cannot escape the system due to the Neumann condition at the boundary. This phenomenon is well illustrated in the FEniCS produced solution, see fig. 3 and the animation.

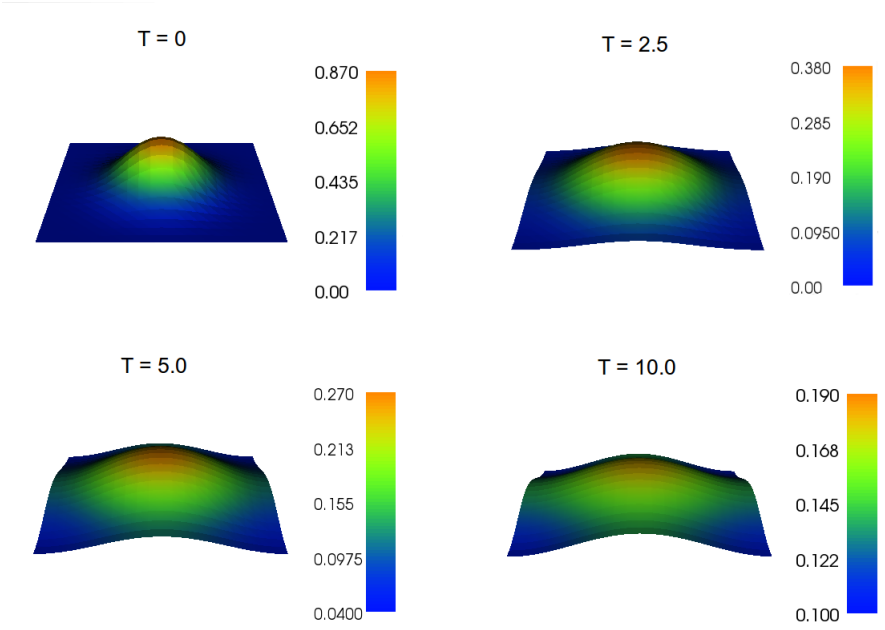

Figure 3: The solution of the 2D diffusion equation at different time levels with a Gaussian distribution as initial condition.

These results were obtained by running the program
`diffusion_gaussian.py`

6 Concluding remarks

The implementation seems to be successful, all the verification tests were passed. FEniCS is a nice environment to solve differential equations because the synax mimics the mathematical notation well, thus it is a good learning platform, especially for newcomers to the finite element method.