Nonlinear diffusion equation

Shafa Aria

November 25, 2014

1 Introduction

In this project we will look at the various numerical aspect of the following nonlinear diffusion equation:

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

with initial conditions $u(x,0) = I(x)$ and Neumann boundary condition $\partial u/\partial n = 0$.

We will first discretize the equation using Backward Euler (BE) then derive a variational formulation and formulating Picard iteration before using FEniCS for the implementation.

2 Discretization & formulation

For our multidimensional PDE we use BE on the LHS to get:

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

We can rewrite the above time-discrete equation as

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

and introducing u for u^n and $u^{(1)}$ for u^{n-1}

$$\varrho u - \Delta t \nabla \cdot (\alpha(u)\nabla u) - \Delta t f(x,t) = \varrho u^{(1)}$$

Now we can go on finding the variational formulation as follows: find $u \in V$ such that

$$\int_\Omega (v\varrho u - \Delta t \nabla \cdot (\alpha(u)\nabla u)v + \Delta t f(x,t)v - \varrho u^{(1)} v)\mathrm{d}x$$

Using integration by parts on the second term and realising we have Neumann condition the last term arising from the integration by parts on the second term

$\int_\Omega \Delta t \nabla \cdot (\alpha(u)\nabla u) v \mathrm{d}x = \int_\Omega \Delta t \alpha(u) \nabla u \cdot \nabla v \mathrm{d}x - \int_{\partial\Omega} \frac{\partial u}{\partial n} v \mathrm{d}s$ vanishes and we are left with

$$\int_\Omega (v\varrho u - \Delta t \alpha(u)\nabla u \cdot \nabla v + \Delta t f(x,t) v - \varrho u^{(1)} v)\mathrm{d}x$$

$$\int_\Omega v\varrho u \mathrm{d}x - \int_\Omega \Delta t \alpha(u) \nabla u \cdot \nabla v \mathrm{d}x = \int_\Omega \Delta t f(x,t) v \mathrm{d}x + \int_\Omega \varrho u^{(1)} v \mathrm{d}x$$

$$\int_\Omega v u \mathrm{d}x - \frac{\Delta t}{\varrho}\int_\Omega \alpha(u)\nabla u \cdot \nabla v \mathrm{d}x = \frac{\Delta t}{\varrho}\int_\Omega f(x,t) v \mathrm{d}x + \int_\Omega u^{(1)} v \mathrm{d}x \tag{2}$$

for $\forall v \in V$

Before going further with the Picard iteration and the implementation we will introduce the nonlinear algebraic equation that follows from setting $v = \psi_i$ and the representation $u = \sum_k c_k \psi_k$ which when written gives us

$$\int_\Omega \psi_i u \mathrm{d}x - \frac{\Delta t}{\varrho}\int_\Omega \alpha(u)\nabla u \cdot \nabla \psi_i \mathrm{d}x = \frac{\Delta t}{\varrho}\int_\Omega f(x,t)\psi_i \mathrm{d}x + \int_\Omega u^{(1)}\psi_i \mathrm{d}x$$

Picard iteration needs a linearisation where we use the most recent approximation u^- to u in $\alpha(u)$

$$F_i \approx \hat{F}_i = \int_\Omega (\psi_i u - \frac{\Delta t}{\varrho}\alpha(u^-)\nabla u \cdot \nabla \psi_i - \frac{\Delta t}{\varrho} f(x,t)\psi_i - u^{(1)}\psi_i)\mathrm{d}x$$

Now we can derive a linear system for the equations $\hat{F}_i = 0$, using $\sum_{j\in\mathcal{I}_s} A_{i,j}c_j = b_i$, $i \in \mathcal{I}_s$ for the unknown coefficients $\{c_i\}_{i\in\mathcal{I}_s}$ by inserting $u = \sum_j c_j\psi_j$. We now get

$$A_{i,j} = \int_\Omega (\psi_i u - \frac{\Delta t}{\varrho}\alpha(u^-)\nabla u \cdot \nabla \psi_i)\mathrm{d}x, \qquad b_i = \int_\Omega (\frac{\Delta t}{\varrho} f(x,t)\psi_i + u^{(1)}\psi_i)\mathrm{d}x \tag{3}$$

3 Implementation and execution using FEniCS

Now for the weak formulation that we have (2) we can introduce the following unified notation:

$$a(u,v) = L(v) \tag{4}$$

where we have

$$a(u,v) = \int_\Omega (vu - \frac{\Delta t}{\varrho}\alpha(u)\nabla u \cdot \nabla v)\mathrm{d}x$$

$$L(v) = \int_\Omega (\frac{\Delta t}{\varrho} f(x,t) v + u^{(1)} v)\mathrm{d}x$$

which are the bilinear and linear form respectively. As can be seen from the above equation we identify all terms with the unknown u and collect them in $a(u,v)$ and similarly identify and collect all known functions in $L(v)$. This will be our requisition on making a FEniCS program.

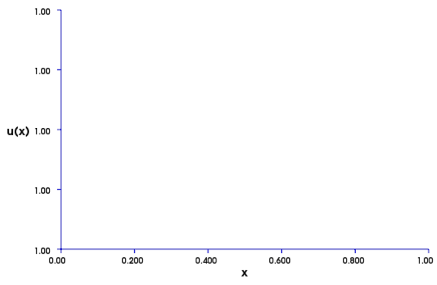
We can then implement the first step of Picard iteration, as shown analytically above, in the FeniCS program by this piece of code[1]

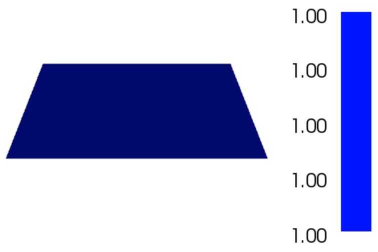
```
u = TrialFunction(V)
v = TestFunction(V)
u_k = interpolate(u0, V)  # previous (known) u
a = u*v*dx - (dt/rho)*inner(alpha(u_k)*nabla_grad(u), nabla_grad(v))*dx
f = Constant(0.0)
L = (u_k + (dt/rho)*f)*v*dx
```

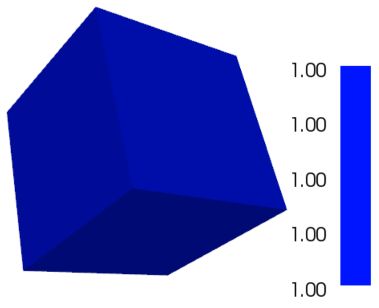
Where as in the mathematical expression we use the previously calculated u^- here denoted `u_k` in our function $\alpha(u)$

4 Tests

We have written a short code for the test of a constant solution see the program `test_constant.py`. Running our code gives us the following three plots







Which is as expected considering our choice of constant $C = 1$ in other words our initial condition `u0 = Constant(1.0)`. Using the VTK plotter included in dolfin which comes with FEniCS does not give us a more visual line since it is bounded at the axis. A remedy is to vectorise the solution in one dimension and plot using `scitools.std` by the following:

[1]For the complete code, see main.py

```
u = u.vector().array()
x = linspace(0,1,len(u))
plot(x, u, xlabel="x", ylabel="u(x)", title = "1D")
```

this will give us a clear line on at at the $u(x)$ axis. We will not be plotting the graphs for the P2 elements as there is really no point for a constant solution, instead we will find the error in P1 and P2 elements running the code[2] will print out the errors for us where we are only looking at the second element in our vectors for the solutions. If we wish to see the errors at all points it is adequate to write `u_ex - u`. The error in the second element for the P1 and P2 in all three dimensions:

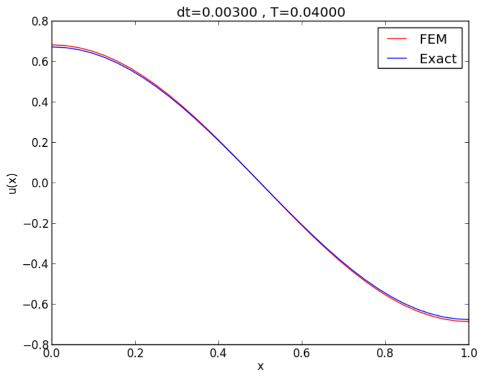
```
Error 2.042810365310288E-14 for P1 elemnt in 1D
Error 2.642330798607873E-14 for P2 elemnt in 1D
Error 3.996802888650564E-15 for P1 elemnt in 2D
Error 8.075762281123389E-13 for P2 elemnt in 2D
Error 1.421085471520200E-14 for P1 elemnt in 3D
Error 1.865174681370263E-14 for P2 elemnt in 3D
```

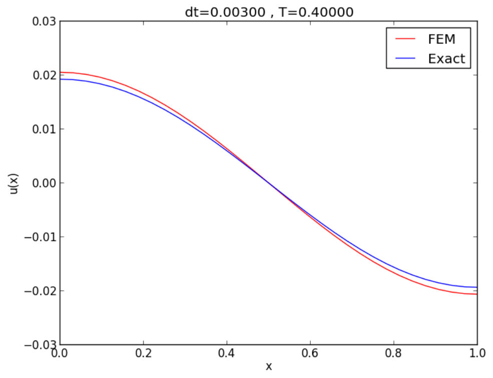
Now let us take a look at a bit more exciting expression than just a constant where we have our initial condition as $I(x, y) = \cos(\pi x)$ and an exact solution as $u(x, y, t) = e^{-\pi^2 t}\cos(\pi x)$, we will have our $\alpha(u)$ and f be equal to 1 and 0 respectively. We will look at the solution in a unit square space $\Omega = [0, 1]\times[0, 1]$.

FEniCS implementation[3] of the initial and exact solution will be as:

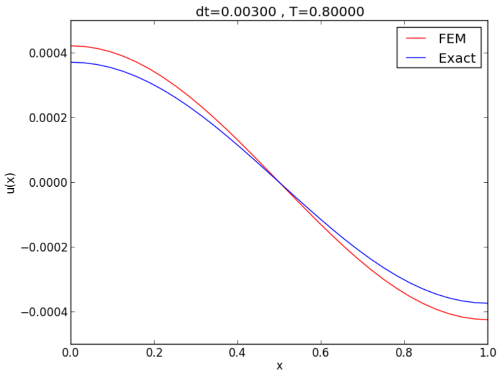
```
u0 =  Expression('cos(pi*x[0])',t=0)
def alpha(u):
return 1.0
f = Constant(0.0)
u_exact = Expression('exp(-pi*pi*t)*cos(pi*x[0])', t=T)
```

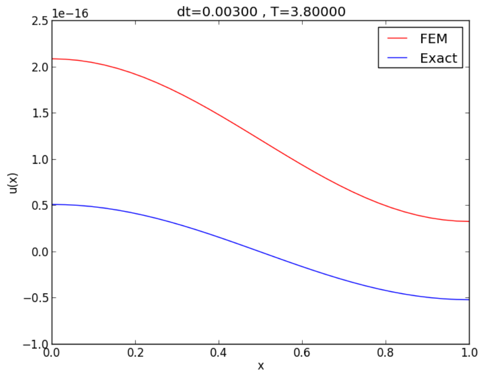
We will adjust in the following our T in 1D



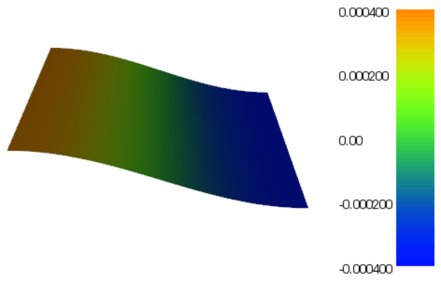


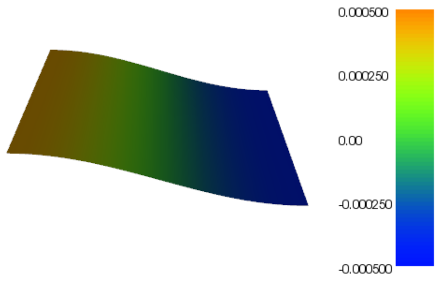
[2]test_constant.py
[3]test_exact.py





We see that as we increase T and keep our Δt constant, our approximation with the FEM mismatches with the exact solution. The same can be seen in 2D, here we have only included for the case $N = 32$, $\Delta t = 0.003$ and $T = 0.8$





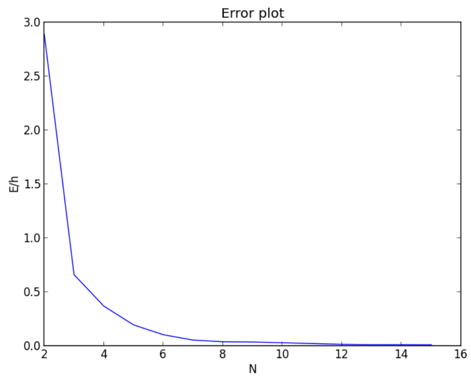
Noting the colour bar difference between the two. The error as was given by

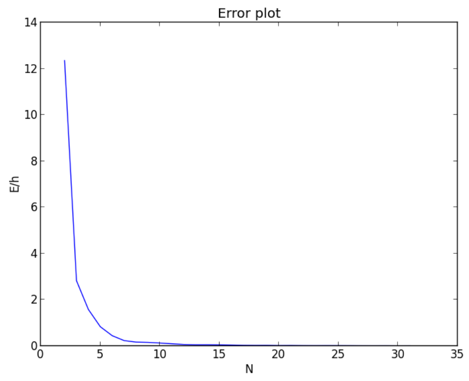
$$E = K_t \Delta t^p + K_x \Delta x^2 + K_y \Delta y^2$$

where in our case since we have BE, $p = 1$. The numerical form being

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

The error plot we get on a mesh `N=range(2,16)` and `N=range(2,32)`





We see that $K = E/h$ fades to a constant value as we increase our meshpoints.

Now let us include a RHS function f with an initial condition constant to zero and an exact solution as

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

we will have $\alpha(u) = 1 - u^2$ and only see in 1D i.e $\Omega = [0,1]$. Running the following `sympy` code will give us a simplified f that we can use in a FEniCS program[4]

```
>>> from sympy import *
>>> x, t, rho, dt = symbols('x t rho dt')
>>>
>>> def a(u):
...     return 1 + u**2
...
>>> def u_simple(x, t):
...     return x**2*(Rational(1,2) - x/3)*t
.
.
.
>>>
>>> # MMS: full nonlinear problem
>>> u = u_simple(x, t)
>>> f = rho*diff(u, t) - diff(a(u)*diff(u, x), x)
>>> 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
```

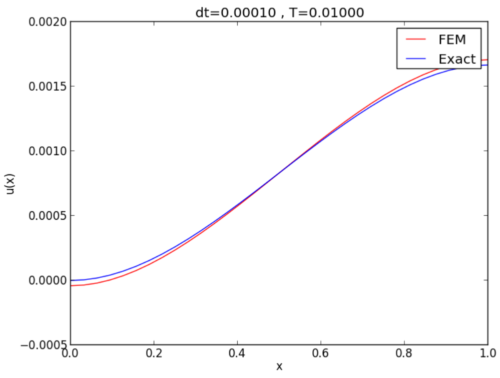
the key line here is `-rho*x**3/3 + rho*x**2/2 + 8*t**3*x**7/9 - 28*t**3*x**6/9 + 7*t**3*x**5/2 - 5*t**3*x**4/4 + 2*t*x - t` which is the simplified f we will need.

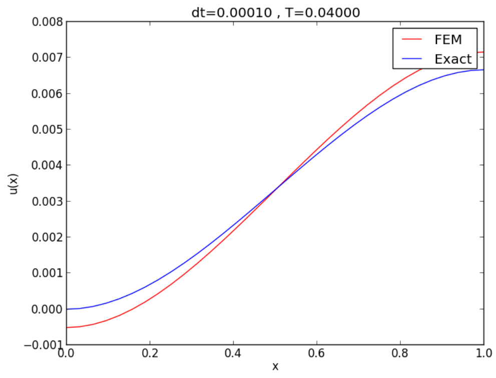
[4]test_manufactured.py

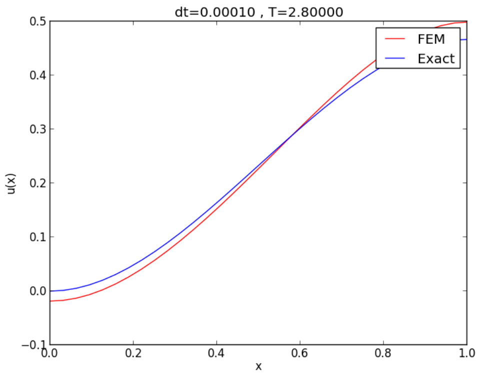
Now unfortunately we cannot use it as it is, referring to FEniCS documentation the library consists of C++ code in which `a**b` is an illegal expression and thus instead have to use `pow(a,b)` noting this and the fact that the last division is done on the entire term, e.g: `rho*x**2/2` $= \frac{\varrho x^2}{2}$ we get the following for our f[5]

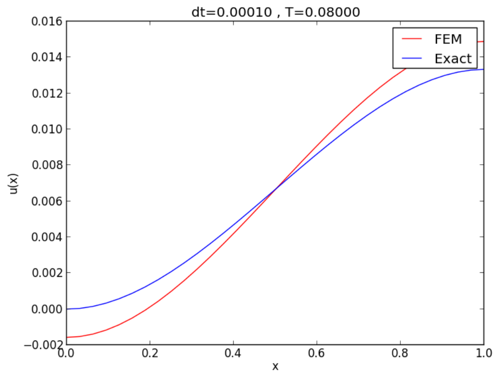
```
f = Expression('-(rho*pow(x[0],3))/3 + ... + 2.0*t*x[0] - t', t=T,rho=rho)
```

Now let us try plotting for a few different T









The corresponding error in the first element for the above plots:

```
The error: [1.5551769501E-03] for T=0.0800 and dt=0.0001
The error: [4.0560519652E-05] for T=0.0100 and dt=0.0001
The error: [4.9381654857E-04] for T=0.0400 and dt=0.0001
The error: [3.2051868842E-02] for T=2.8000 and dt=0.0001
```

In the constant solution we see that P2 elements give a slightly larger error than P1 elements. As we have seen there are several sources of numerical errors in our computation among: temporal and spatial discretization, and the first step of our iteration in which we insert u^{-}. Referring to FEniCS documentation

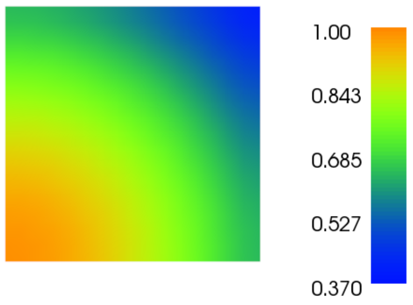
[5]please refer to the code for the complete expression as it is lengthy

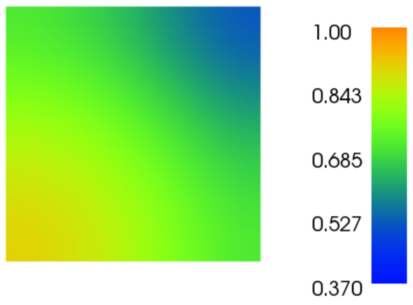
among some of the ways to tackle some of the issues is to introduce a tolerance in our computation and thus stop the computation once the iteration has reached the tolerance.

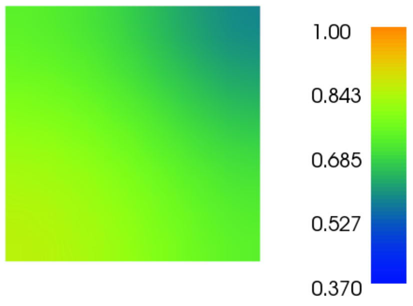
Let us take at another test where we have our initial value as a Gaussian function

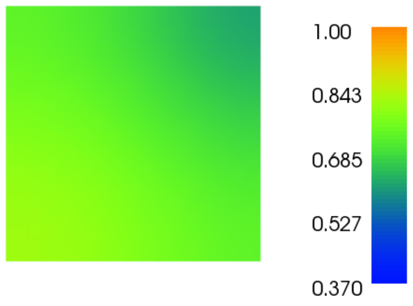
$$I(x,y) = \exp\left(-\frac{1}{2\sigma^2}\left(x^2+y^2\right)\right), \quad (x,y) \in \Omega = [0,1] \times [0,1]$$

and our $\alpha(u) = 1 + \beta u^2$ where the σ is the width of the function and β is just some constant we can choose. We will try to animate this function in 2D[6]. As for the plots we get a nice gradient on the edges and it slowly fades away as expected









once the program is run one can then see the Gaussian function as it wears off, now our computation for the above plots was for $\sigma = 1.0$ and $\beta = 2.0$.

[6]test_gaussian.py