IN5270-project2-ejhusom

November 13, 2019

1 IN5270 - Project 2: Nonlinear diffusion equation

By Erik Johannes Husom, November 2019.

1.1 Problem description

The aim of this project is to discuss various numerical aspects of a nonlinear diffusion model:

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

where $u_t = \partial u / \partial t$. We have

- Initial condition: $u(\vec{x}, 0) = I(\vec{x})$,
- Boundary condition $\partial u / \partial n = 0$,
- Constant coefficient: ϱ,
- Known function of u: $\alpha(u)$.

1.2 a) Backward Euler discretization

We start by using the Backward Euler to construct an implicit scheme in the time direction:

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

which we rewrite as

$$u^n - \frac{\Delta t}{\varrho} \left[\nabla \cdot (\alpha(u^n)\nabla u^n) + f(\vec{x}_n, t_n)\right] = u^{n-1}.$$

The residual R becomes

$$R = u^n - \frac{\Delta t}{\varrho} \left[\nabla \cdot (\alpha(u^n)\nabla u^n) + f(\vec{x}_n, t_n)\right] - u^{n-1},$$

and we want to have $(R, v) = 0, \forall v \in V$. By moving all the terms to the left, multiplying with a test function $v \in V$ and taking the integral we get

$$\int_\Omega \left(u^n v - \frac{\Delta t}{\varrho} \left[\nabla(\alpha(u^n)\nabla u^n)v + f(x_n, t_n)v \right] - u^{n-1}v \right) d\vec{x}, \forall v \in V.$$

We use integration by parts to reduce the second order derivative:

$$\int_\Omega u^n v d\vec{x} + \frac{\Delta t}{\varrho} \int_\Omega \alpha(u^n)\nabla u \nabla v d\vec{x} - \int_{\partial\Omega} \frac{\partial u}{\partial n} v ds - \frac{\Delta t}{\varrho} \int_\Omega f(\vec{x}_n, t_n) v d\vec{x} - \int_\Omega u^{n-1} v d\vec{x}.$$

The term $\int_{\partial\Omega} \frac{\partial u}{\partial n} v ds$ is zero because of our boundary condition. Then we have the following as our variational form:

$$\int_\Omega \left(u^n v + \frac{\Delta t}{\varrho} \left[\alpha(u^n)\nabla u \nabla v - f(\vec{x}_n, t_n)v \right] - u^{n-1}v \right) d\vec{x} = 0, \forall v \in V.$$

We can write this with the following notation:

$$\varrho(u^n, v) + \Delta t(\alpha(u^n)\nabla u^n, \nabla v) = \varrho(u^{n-1}, v) + \Delta t(f(\vec{x}_n, t_n), v)$$

We call the left hand of this equation $a(u, v)$, and the right hand side we call $L(v)$.

1.3 b) Picard iteration

Based on the first equation in task a) we need to solve the following PDE in the Picard iterations, where k is the iteration counter:

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

In the first iteration we have $u^{n,0} = u^{n-1}$. By setting $u = u^{n,k+1}$, $u^- = u^{n,k}$ and $u^{(1)} = u^{n-1}$, we get the following equation for a Picard iteration:

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

Each iteration starts with setting the value from the previous time level $u^- = u^{(1)}$, and at the end of the iteration, we assign $u^- \leftarrow u$. When solving the above equation for u we get:

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

The bilinear/linear form becomes

$$\varrho(u, v) + \Delta t(\alpha(u^-)\nabla u^n, \nabla v) = \varrho(u^1, v) + \Delta t(f(\vec{x}_n, t_n), v)$$

1.4 c) A single Picard iteration

We will now restrict the Picard iteration to a single iteration, which means to use a u value from the previous time in the $\alpha(u)$ coefficient:

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

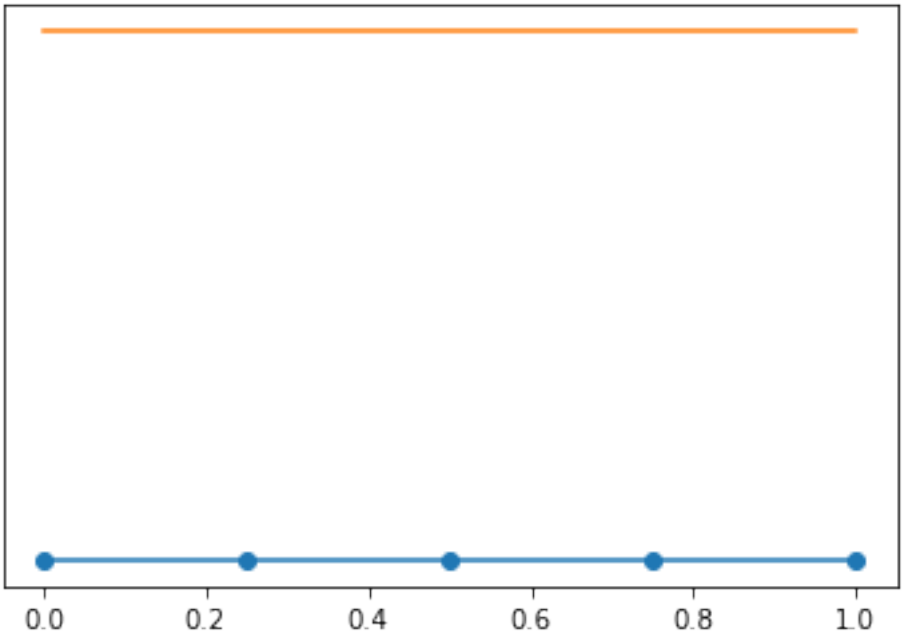
or

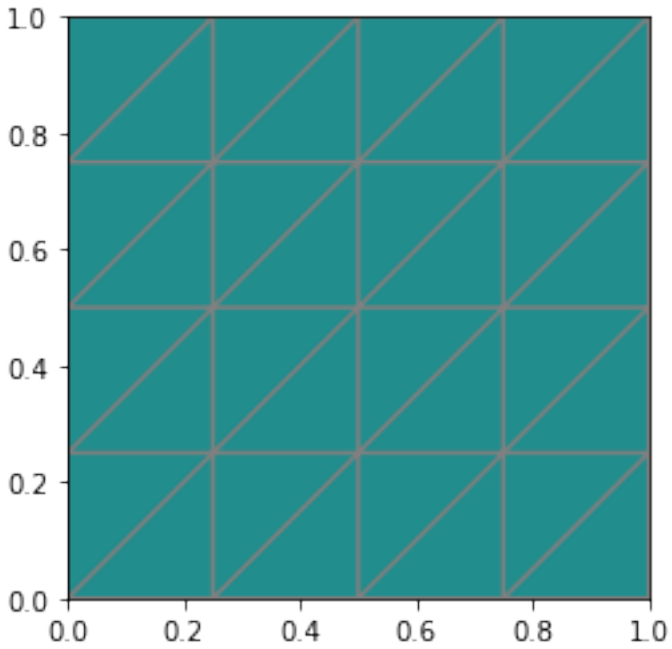
$$\varrho(u^n, v) + \Delta t(\alpha(u^{(1)})\nabla u^n, \nabla v) = \varrho(u^{(1)}, v) + \Delta t(f(\vec{x}_n, t_n), v)$$

Implementation in FEniCS software, which can handle both 1D, 2D and 3D input, follows below.

[211]:
```
from fenics import *
import numpy as np
import matplotlib.pyplot as plt

def picard(grid_size=[8,8], rho=1.0, alpha=Constant('1'),
           I=Constant('1'), f=Constant('0.0'),P=1, nt=1,
           showplot=True, T=1.0):


    mesh_options = [UnitIntervalMesh, UnitSquareMesh, UnitCubeMesh]
    dim = len(grid_size)
    mesh = mesh_options[dim-1](*grid_size)
    n = grid_size[0]

    t = 0
    dt = T/nt
    t_array = np.linspace(t, T, nt+1)

    V = FunctionSpace(mesh, 'P', P)

    u = TrialFunction(V)
    u_ = Function(V)
    v = TestFunction(V)
    u0 = I
    u1 = project(u0, V)
    uk = u1

    # Bilinear/linear form
    a = (rho*inner(u, v)+ dt*inner(alpha(uk)*nabla_grad(u), nabla_grad(v)))*dx
    L = (rho*inner(uk, v) + dt*inner(f, v))*dx


    for t in t_array:
        f.t = t
```

```
        solve(a == L, u_)
        u1.assign(u_)
        uk.assign(u_)


    if dim < 3 and showplot:
        plot(mesh)
        if dim == 2:
            plot(u_, scalarbar=True)
        else:
            plot(u_)
        plt.show()


    return u1, V, mesh, dt


n = 4
u, V, mesh, dt = picard(grid_size=[n], nt=1)
u, V, mesh, dt = picard(grid_size=[n,n], nt=1)
u, V, mesh, dt = picard(grid_size=[n,n,n], nt=1)
```





1.5 d) Constant solution

We can easily see from our original PDE that if u is a constant, the time derivative will become zero, the gradient will become zero, and the solution depends only on f and the initial conditions I. I choose the values

- $\varrho = 1$,
- $\alpha = 1$,
- $f = 0$,
- $I = 3$,

which will give $u(x, t) = 3$. We verify this with the following code.

[212]:
```
u, V, mesh, dt = picard(grid_size=[8], showplot=False, I=Constant('3'), nt=10)
u_vertex = u.compute_vertex_values(mesh)
print(u_vertex)
```

```
[3. 3. 3. 3. 3. 3. 3. 3. 3.]
```

As we can see, the numerically computed u consists of constant values of 3, which is what we expected.

1.6 e) Analytical solution

In this task we assume

- $\alpha(u) = 1$,
- $f = 0$,
- $\Omega = [0,1] \times [0,1]$,
- P1 elements,
- $I(x,y) = \cos(\pi x)$.

Then we will have the exact solution

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

- Error in space: $\mathcal{O}(\Delta x^2) + \mathcal{O}(\Delta y^2$,
- Error in time: $\mathcal{O}(\Delta t^p)$, with $p = 1$ for the Backward Euler scheme.

We have a model for an error measure:

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

with $h = \Delta t^p = \Delta x^2 = \Delta y^2$ and $K = K_t + K_x + K_y$. The measure E will be taken as the discrete L_2 norm of the solution as the nodes, which is computed by

```
vertex_values_u_exact = u_exact.compute_vertex_values(mesh)
vertex_values_u = u.compute_vertex_values(mesh)
error = vertex_values_u_exact - vertex_values_u
E = numpy.sqrt(numpy.sum(error**2)/error.size)
```

for some fixed point of time. We have that `u_exact` is a projection of the exact solution onto the function space used for `u`. The following code shows how E/h behaves as the mesh in space and time is simultanouesly refined by reducing h. We are using a fixed point in time, $T = 1.0$.

[219]:
```
n_values = [10,20,30,40,50]
T = 1.0

for n in n_values:
    u, V, mesh, dt = picard(grid_size=[n,n], showplot=False,
                            I=Expression('cos(pi*x[0])', degree=3),␣
 ↪alpha=Constant('1'),
                            f=Constant('0'), P=1, nt=n**2, T=T)

    h = dt

    u_e = Expression('exp(-pi*pi*t)*cos(pi*x[0])', t=T, degree=3)
    u_exact = project(u_e, V)

    error = u_exact.compute_vertex_values(mesh) - u.compute_vertex_values(mesh)
    E = np.sqrt(np.sum(error**2)/error.size)

    print('h={:.8f}, E/h={:.10f}'.format(h, E/h))
```

```
h=0.01000000, E/h=0.0012916229
h=0.00250000, E/h=0.0011788640
h=0.00111111, E/h=0.0011542891
h=0.00062500, E/h=0.0011443310
h=0.00040000, E/h=0.0011390887
```

The value E/h is slowly decreasing when we refine the mesh in time and space, but is approximately constant.

1.7 f) Manufactured solution

We now restrict the problem to one space dimension with $\Omega = [0, 1]$, and set

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

and $\alpha(u) = 1 + u^2$. We use `sympy` in the code below (taken from the project description) to find an $f(x,t)$ based on the u we have chosen.

[206]:
```
import sympy as sp


x, t, rho, dt = sp.symbols('x[0] t rho dt')

def a(u):
    return 1 + u**2

def u_simple(x, t):
    return x**2*(sp.Rational(1,2) - x/3)*t

for x_point in 0, 1:
    print(f'u_x({x_point},t): {sp.diff(u_simple(x, t), x).subs(x, x_point).
 ↪simplify()}')


print(f'Initial condition: {u_simple(x, 0)}')

## MMS: Full nonlinear problem
u = u_simple(x, t)
f = rho*sp.diff(u, t) - sp.diff(a(u)*sp.diff(u, x), x)
f = f.simplify()

print(f)
f_ccode = printing.ccode(f)
print(f_ccode)
```

```
u_x(0,t): 0
u_x(1,t): 0
```

```
Initial condition: 0
-rho*x[0]**2*(2*x[0] - 3)/6 + t**3*x[0]**4*(x[0] - 1)**2*(2*x[0] - 3)/3 +
t*(2*x[0] - 1)*(t**2*x[0]**4*(2*x[0] - 3)**2 + 36)/36
-1.0/6.0*rho*pow(x[0], 2)*(2*x[0] - 3) + (1.0/3.0)*pow(t, 3)*pow(x[0],
4)*pow(x[0] - 1, 2)*(2*x[0] - 3) + (1.0/36.0)*t*(2*x[0] - 1)*(pow(t,
2)*pow(x[0], 4)*pow(2*x[0] - 3, 2) + 36)
```

We will now compare the FEniCS solution ant the u given above as a function of x for a couple of t, using the code below.

[210]:
```
t_values = [0.05, 0.1, 0.5, 1., 2.]
rho = 1.0
n = 20

def alpha1(u):
    return (1 + u*u)


for t in t_values:
    u, V, mesh, dt = picard(grid_size=[n], showplot=False,
                        I=Constant('0'), alpha=alpha1, rho=rho,
                        f=Expression(f_ccode, rho=rho, t=t, degree=3),
                        P=1, nt=n**2, T=t)

    h = dt

    u_e = Expression('t*pow(x[0],2)*(0.5 - x[0]/3.)', t=t, degree=2)
    u_exact = project(u_e, V)

    error = u_exact.compute_vertex_values(mesh) - u.compute_vertex_values(mesh)
    E = np.sqrt(np.sum(error**2)/error.size)

    print('t={:.4f}, E={:.6f}'.format(t, E))
```

```
t=0.0500, E=0.000011
t=0.1000, E=0.000021
t=0.5000, E=0.000116
t=1.0000, E=0.000237
t=2.0000, E=0.000479
```

As we can see from the results, the error E is gradually increasing, which is what we would expect, since the the various numerical error sources will keep adding up for each step.

1.8 g) Sources of numerical errors

Different sources of numerical errors in the FEniCS program:

- Error in time from the finite difference approximation, which for the Backward Euler scheme is $\mathcal{O}(\Delta t)$.
- Error in space from the finite element method, which in the problems we have dealt with is $\mathcal{O}(\Delta x^2) + \mathcal{O}(\Delta y^2)$.
- Error from Picard iterations. This error will decrease when we increase the number of Picard iterations, but since we operate with only one Picard iteration in this project, this error will be significant.