MATH521
Numerical Analysis of Partial Differential Equations

Winter 2017/18, Term 2
Timm Treskatis

Homework Assignment 12

Question 1 | 3 marks Today we're solving the linear advection equation

$$\frac{\partial u(t)}{\partial t} + \operatorname{div}(u(t)a(t)) = 0 \qquad \text{in } Q =]0,T[\times\Omega \tag{1a}$$

$$u(0) = u_0 \qquad \text{in } \Omega \tag{1b}$$

$$u(t) = g(t) \qquad \text{on } \Sigma_- = \{\, (t,x) \in]0,T[\times\partial\Omega \mid a(t,x)\cdot n(x) < 0 \,\}, \tag{1c}$$

where $a :]0,T[\times\Omega \to \mathbb{R}^2$ is a given vector field, thought of as the flow velocity of a carrier fluid, and $u_0 : \Omega \to [0,1]$ the initial concentration of a solute. $g :]0,T[\times\partial\Omega \to [0,1]$ prescribes the concentration on that part of the boundary $\partial\Omega$ where the flow of solvent is directed into the domain and we use n to denote the outward pointing unit normal vectors on $\partial\Omega$. For simplicity we assume in what follows that a and g do not depend on time.

Recall that upwind discontinuous Galerkin methods employ the bilinear form

$$\sum_{T\in\mathcal{T}^h} -\int_T ua\cdot\nabla v \,\mathrm{d}x + \sum_{e\in\mathcal{E}^h}\int_e [v]u_{\mathrm{up}}a\cdot n_+ \,\mathrm{d}s$$

to discretise the transport term $\operatorname{div}(ua)$.

(a) We define the positive and negative parts of a function f by

$$f^{\mathrm{pos}} = \frac{f+|f|}{2} \qquad \text{and} \qquad f^{\mathrm{neg}} = \frac{f-|f|}{2}.$$

Re-write the edge integral in the upwind DG form in terms of g, u_+, u_-, $(a\cdot n_+)^{\mathrm{pos}}$, $(a\cdot n_+)^{\mathrm{neg}}$ and $[v]$:

- If e is an interior edge:

$$\int_e [v]u_{\mathrm{up}}a\cdot n_+ \,\mathrm{d}s = \int_e [v]u_{\mathrm{up}}\left[(a\cdot n_+)^{\mathrm{pos}} + (a\cdot n_+)^{\mathrm{neg}}\right] \mathrm{d}s$$
$$= \int_e [v]u_+(a\cdot n_+)^{\mathrm{pos}} \,\mathrm{d}s + \int_e [v]u_-(a\cdot n_+)^{\mathrm{neg}} \,\mathrm{d}s$$

  The second line follows from the fact that $(a\cdot n_+)^{\mathrm{pos}} = 0$ when $a\cdot n_+ < 0$, and therefore $u_{\mathrm{up}} = u_+$ wherever the integrand in non-zero. Similarly, $u_{\mathrm{up}} = u_-$ in the second term when $(a\cdot n_+)^{\mathrm{neg}}$ is non-zero.

- If e is an exterior edge (no $\pm$ subscripts or [ ] brackets needed, there is only one neighbouring triangle):

$$\int_e [v]u_{\mathrm{up}}a\cdot n_+ \,\mathrm{d}s = \int_e vu_{\mathrm{up}}\left[(a\cdot n)^{\mathrm{pos}} + (a\cdot n)^{\mathrm{neg}}\right] \mathrm{d}s$$
$$= \int_e vu(a\cdot n)^{\mathrm{pos}} \,\mathrm{d}s + \int_e vg(a\cdot n)^{\mathrm{neg}} \,\mathrm{d}s$$

  Where we have used the fact that $u = g$ when $a\cdot n < 0$.

(b) Complete the FEniCS script `hw12.py` to solve the linear advection equation with

$$T = 2\pi \qquad a(x) = \begin{pmatrix} -x_2 \\ x_1 \end{pmatrix} = r \begin{pmatrix} -\sin\phi \\ \cos\phi \end{pmatrix} \text{ in polar coordinates} \qquad g(x) \equiv 0$$

and the domain and initial data provided. Note that with these data, the advection equation rotates the initial field u_0 around the origin in anticlockwise direction, like a rigid body on a turntable.

Use a moderate degree (e.g. $r = 1$ or 2) for the DG-discretisation in space and use the method we are most familiar with, the θ-method, for time stepping. Any value of θ which results in a stable scheme is fine.

Hint: In FEniCS we use `dS` for integrals over interior edges and `ds` for integrals over exterior edges. You must add `('+')` or `('-')` to all discontinuous functions in integral expressions containing `dS`, e.g.

```
u('-')*jump(v)*dS
```

to specify on what side of the edge they should be evaluated. `jump(v)` is a shortcut for `v('+')-v('-')`.

(c) Inspect your numerical solution in ParaView, using the 'Warp by Scalar'-Filter. Recall that FEniCS generally exports solution data as continuous, piecewise linear functions, so don't be surprised when you cannot see any discontinuous jumps in your visualisation.

Describe any discrepancies between the numerical solution and the exact solution, using the appropriate terminology.

Hint: For general triangular meshes, the upwind DG(r) method discretises the advection term with an accuracy of order $r + \frac{1}{2}$, which increases to $r + 1$ on meshes with certain regularity.

Question 2 | 2 marks Make a copy of your script `hw12.py` and modify it to solve the advection-diffusion problem

$$\frac{\partial u(t)}{\partial t} + \operatorname{div}(u(t)a - D\nabla u(t)) = 0 \qquad \text{in } Q =]0,T[\times\Omega \tag{2a}$$

$$u(0) = u_0 \qquad \text{in } \Omega \tag{2b}$$

$$(u(t)a - D\nabla u(t)) \cdot n = 0 \qquad \text{on } \Sigma =]0,T[\times\partial\Omega \tag{2c}$$

instead, using $D = 0.01$ and all other parameters as in Question 1. The new boundary condition is a no-flux Robin condition. It admits an interpretation of a semi-permeable membrane which allows the solvent, but not the solute to pass through.

Discretise the diffusive flux with the symmetric interior penalty form

$$\sum_{T\in\mathcal{T}^h} \int_T D\nabla u \cdot \nabla v \, \mathrm{d}x \qquad \text{(bilinear form of conforming methods)}$$

$$- \sum_{e\in\mathcal{E}^h\setminus\mathcal{I}^h} \int_e vD\nabla u \cdot n \, \mathrm{d}s - \sum_{e\in\mathcal{I}^h} \int_e [v] \langle D\nabla u\rangle \cdot n_+ \, \mathrm{d}s \qquad \text{(consistency)}$$

$$- \sum_{e\in\mathcal{I}^h} \int_e [u] \langle D\nabla v\rangle \cdot n_+ \, \mathrm{d}s \qquad \text{(symmetry)}$$

$$+ \sum_{e\in\mathcal{I}^h} \frac{\sigma}{h_e} \int_e [u][v] \, \mathrm{d}s \qquad \text{(interior penalty)}$$

as derived in class, here written in a form that can be translated directly into FEniCS code (but note that the boundary condition has not been applied yet). Use the penalty parameter $\sigma = 0.1$ and $h_e = \langle h\rangle$.

Hint: The commands `h = CellSize(mesh)` and `avg(u)` may be helpful.

Your Learning Progress What is the one most important thing that you have learnt from this assignment?

Any new discoveries or achievements towards the objectives of your course project?

What is the most substantial new insight that you have gained from this course this week? Any *aha moment*?

Appendix A

hw12.py

```
# coding=utf-8
"""
FEniCS program: Solution of the unsteady advection equation or advection-
diffusion equation with DG-discretisation in space.

Set D = 0.0 for advection problem with boundary data g
Set D > 0.0 for diffusion problem with no-flux Robin BC's (ignores g)
"""

from __future__ import print_function
from fenics import *
from mshr import *
import numpy as np
import time

# Create a domain and mesh
domain = Circle(Point(0., 0.), 1.) - Circle(Point(0., 1.15), 0.3) - \
         Circle(Point(0., -1.15), 0.3) + Circle(Point(0.85, 0.), 0.3) + \
         Circle(Point(-0.85, 0.), 0.3)
mesh = generate_mesh(domain, 100)
n = FacetNormal(mesh)

# Function space
r = 2
V = FunctionSpace(mesh, 'DG', r)

# Problem data
a = Expression(("-x[1]", "x[0]"), degree=1) # advection velocity
g = Constant(0.0) # boundary data
D = 0.01 # diffusion constant (set D = 0.0 for advection problem)
sigma = 0.1 # interior penalty parameter

u0_expr = Expression('pow(x[0],2)+pow(x[1]-0.5,2) < 1./16. ? 1. : 0.', degree=0)
u0 = project(u0_expr, V) # initial concentration

# Parameters of the time-stepping scheme
t0 = 0. # initial time
T = 2.*pi # final time
t = t0 # current time
tsteps = 500 # number of time steps
dt = T/tsteps # time step size

# The advection operator has imaginary eigenvalues, suggesting that the
# Crank-Nicolson scheme is a good method and one should set    = 0.5.
# It should be noted, however, that upwind discretisation introduces some
# numerical diffusion which ideally should be mitigated by using an anti-
# dissipative time stepping scheme (Forward Euler, etc.). I will just make the
# safe choice of    = 0.5, which will be stable, and not try to guess the optimal
# amount of explicitness to add to the problem.
#   Note: Added implicitness seems to help the advection-diffusion problem due
#         to the diffusion term
theta = 1.0

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)

def get_M(u,v):
    # regular M(u,v) mass matrix
    M = u*v*dx
    return M

def get_B(u,v):
    # regular B(u,v) term
    B = -u*dot(a,grad(v))*dx
    f = dot(a, n('+')) # take + normal direction
    f_pos = 0.5*(f + abs(f))
    f_neg = 0.5*(f - abs(f))

    B += jump(v)*u('+')*f_pos*dS + jump(v)*u('-')*f_neg*dS # interior jumps

    f = dot(a, n) # no ambiguity of n direction on boundary
    f_pos = 0.5*(f + abs(f))
    B += v*u*f_pos*ds # exterior edge contributions

    if abs(D) > 5*DOLFIN_EPS:
        # Add Diffusion term with no-flux Robin boundary condition
        h = CellDiameter(mesh)
        he = avg(h)

        B += D*dot(grad(u),grad(v))*dx # bilinear form of conforming methods
        B -= v*dot(u*a,n)*ds # consistency, boundary edges with: D   u   n  = (ua)   n
        B -= jump(v)*dot(avg(D*grad(u)),n('+'))*dS # consistency, int. edges
        B -= jump(u)*dot(avg(D*grad(v)),n('+'))*dS # symmetry
        B += (sigma/he)*jump(u)*jump(v)*dS # interior penalty

    return B

def get_Lg(u,v):
    if abs(D) > 5*DOLFIN_EPS:
        # Adv-diff problem: no boundary data (Robin no-flux condition is used)
        Lg = Constant(0.0)*v*dx
    else:
        # Advection problem: compute RHS source term
        f = dot(a, n) # no ambiguity of n direction on boundary
        f_neg = 0.5*(f - abs(f))
        Lg = -v*g*f_neg*ds

    return Lg

# Now, we have a system (thinking of M, B as matrices, and L,u as vectors):
#   M*du/dt + B*u = L
# (Note that L may be zero, as in the case of g=0 for the advection problem or
# for no-flux Robin BC's in the advection-diffusion problem)
#
# Writing f = L-B*u, the theta method gives us:
#   M* u    = M* u    +   t (   *f(t, u   ) + (1-   )*f(t, u   ))
# And so:
#   M* u    -     t *f(t, u   ) = M* u    + (1-   )  t *f(t, u   )
#   M* u    -     t *(L-B* u    ) = M* u    + (1-   )  t *(L-B* u   )
#   (M +     t *B)* u    = M* u    + (1-   )  t *(L-B* u   ) +     t *L
#   (M +     t *B)* u    = (M - (1-   )  t *B)* u    +   t *L

M = get_M(u,v)
B = get_B(u,v)
A = assemble(M + theta*dt*B)

# I know that this source term is zero, but I thought I'd leave it in for
# generality, considering that the performance penalty for the vector addition
# is negligeable compared to the linear system solve anyways
Lg = get_Lg(u,v)
Lg_dt = assemble(dt*Lg)

# Create solver object for linear systems
solver = LUSolver(A) # Direct solver
solver.parameters.symmetric = False
solver.parameters.reuse_factorization = True

# Write initial data to file
u = Function(V, name='Concentration')
u.assign(u0)
concentration = File('hw12/advection.pvd')
concentration << (u, t)

for k in range(tsteps):
```

```
# Current time
t = t0 + (k+1)*dt
print('Step = ', k+1, '/', tsteps , 'Time =', t)

# Define right hand side
M_u0 = get_M(u0,v)
B_u0 = get_B(u0,v)
L = assemble(M_u0 - (1-theta)*dt*B_u0) + Lg_dt
solver.solve(u.vector(), L)

# Write data to file
concentration << (u, t)

# Update
u0.assign(u)
```