MATH521
Numerical Analysis of Partial Differential Equations

Winter 2017/18, Term 2

Homework Assignment 8

Please submit the following files as indicated below:

If you haven't done so already, install ParaView on your computer. This is already included in many Linux distributions. For other operating systems, visit `https://www.paraview.org/download/`.

Question 1 | 1 mark This assignment is dedicated to a posteriori error estimates for the problem

$$\begin{aligned} -\Delta u &= f && \text{in } \Omega \\ u &= 0 && \text{on } \partial\Omega \end{aligned} \tag{P}$$

where Ω is the unit square $]0,1[^2$.

We want to solve this problem because we are interested in the average of u over the set $R = \left]\frac{1}{2},1\right[\times\left]0,\frac{1}{2}\right[$.

Following the dual weighted residual method, what is the dual problem that you have to solve for z? Write down its weak formulation, then its strong formulation. Don't forget to specify what space the solution and the test functions belong to for the weak formulation.

Hint: Indicator function.

Solution. We start by rewriting the strong form (P) of the Poisson-Dirichlet problem into the weak form.

Since we will be solving this problem with linear finite elements (as indicated in `hw8.py`), we will take the test functions to be $v \in V = H_0^1(\Omega)$. Additionally, since Ω is a convex polygonal domain (namely, the open unit square), we have that the space of linear finite elements $V^h \subset V$ is conforming.

As in the notes, we assume that the data f is an L^2-function and that the exact solution is in H^2. Now, multiplying (P) by $v \in V = H_0^1(\Omega)$, integrating by parts, and using that fact that $u, v = 0$ on $\partial\Omega$, the resulting weak form of the `Poisson-Dirichlet` problem is

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x, \quad \forall\, v \in V,$$

or

$$B(u,v) = \langle f, v\rangle, \quad \forall\, v \in V \tag{W}$$

where we have defined the bilinear form $B(u,v) \coloneqq \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x$.

Now, using the dual weighted residual method, we introduce the corresponding dual problem

$$B(v,z) = J(v), \quad \forall\, v \in V \tag{W*}$$

where $J(u)$ is the linear functional

$$J(u) = \frac{1}{|R|}\int_R u \,\mathrm{d}x \tag{1}$$

representing the quantity of interest, namely the average of the solution u over the region $R = \left]\frac{1}{2},1\right[\times\left]0,\frac{1}{2}\right[$.

Now, the dual solution z and test functions v are both in the space $V = H_0^1(\Omega)$, and if we first note that we can write equation (1) as

$$J(u) = \int_\Omega \left\{\frac{1}{|R|} I_R\right\} u \,\mathrm{d}x \tag{2}$$

where

$$I_R(x) = \begin{cases} 1, & x \in R \\ 0, & x \notin R \end{cases} \tag{3}$$

is the indicator function on the set R, then due to the symmetry $B(v,z) = B(z,v)$ of the bilinear functional B, the corresponding strong form of the dual problem (W*) is simply

$$\begin{aligned} -\Delta z &= g \qquad \text{in } \Omega \\ z &= 0 \qquad \text{on } \partial\Omega \end{aligned} \tag{P*}$$

where

$$g = \frac{1}{|R|} I_R.$$

Note additionally that $|R| = \frac{1}{4}$ for $R = \left]\frac{1}{2}, 1\right[\times \left]0, \frac{1}{2}\right[$.

$\square$

Question 2 | 4 marks If the source term in Problem (P) is given as

$$f(x) = a(a+1)x_1^{a-1}x_2(1-x_2) + 2x_1(1-x_1^a)$$

for $a \geq 1$, then the analytical solution is

$$\bar{u}(x) = x_1(1-x_1^a)x_2(1-x_2)$$

which has an average of

$$\frac{3a - 2 + 2^{1-a}}{24a + 48}$$

over the set R.

For large a, this problem is numerically challenging: observe that f becomes very large near the right boundary, while it remains comparatively small elsewhere in the domain. As a result, the solution $\bar{u}$ exhibits a sharp boundary layer near $x_1 = 1$.

(a) Download the FEniCS script `hw8.py` and complete the missing commands. This script should evaluate the a posteriori estimator $\eta_{L^2} \approx \|u^h - \bar{u}\|_{L^2} = \|e^h\|_{L^2}$ as derived in class (or on Canvas, under modules). Solve Problem (P) on the given grids to complete the following table:

h	$\|e^h\|_{L^2}$	η_{L^2}	$\eta_{L^2}/\|e^h\|_{L^2}$
$\frac{1}{64}\sqrt{2}$ (64 × 64 grid)	0.00241963657483	0.066093085134	27.3152942973
$\frac{1}{128}\sqrt{2}$ (128 × 128 grid)	0.000648344109077	0.0170555074095	26.3062580052
$\frac{1}{256}\sqrt{2}$ (256 × 256 grid)	0.000162164095122	0.00437467584739	26.9768461638

Is the error overestimated or underestimated by η_{L^2}? By what factor, approximately?

Solution. The global L^2-norm error of the solution is overestimated by approximately a factor of 27 by η_{L^2}.

□

(b) Compute a posteriori estimators $\eta_J \approx |J(u^h) - J(\bar{u})| = |J(e^h)|$ for the error in the average solution value on R, using both the expensive Strategy 1 and the cheap Strategy 2 to approximate the dual weights (cf p 55 in the notes). Complete the following table:

h	$\|J(e^h)\|$	$\|\eta_{J,1}\|$	$\|\eta_{J,2}\|$	$\|\eta_{J,1}/J(e^h)\|$	$\|\eta_{J,2}/J(e^h)\|$
$\frac{1}{64}\sqrt{2}$ (64 × 64 grid)	0.000566634238416	0.000778386852158	0.000713320888805	1.37370246869	1.25887360919
$\frac{1}{128}\sqrt{2}$ (128 × 128 grid)	0.000157153783317	0.000187641158623	0.000170875597462	1.19399708147	1.08731456447
$\frac{1}{256}\sqrt{2}$ (256 × 256 grid)	3.98056031191e-05	4.55743743355e-05	4.13531013218e-05	1.14492359779	1.0388763913

Is the error overestimated or underestimated? By what factor, approximately?

Solution. The error appears to be overestimated by a factor which is on the order of 1 and decreasing with h, possibly to a number below 1 but it is hard to tell, and my laptop runs out of memory when further refining the grid.

□

(c) For the convergence studies above we have refined the entire mesh from 64×64 to 128×128 to 256×256. The second mesh is four times larger, the third mesh even 16 times larger than the coarsest one. This makes uniform mesh refinement very expensive. We can probably compute a solution that is just as accurate as the solution on the 256×256 mesh, by refining only those triangles on the 64×64 mesh with a noteworthy contribution to the overall error.

Solve Problem (P) on the 64×64 mesh and plot the numerical solutions u^h and z^h, the cell residuals $\|r_T\|_{L^2}$, the dual weights $\|w_T\|_{L^2}$ (approximated with either the expensive or the cheap strategy) and the local error indicators η_T. What triangles of the 64×64 mesh would you refine to compute the average of u over R more accurately? A rough description like 'near the left boundary' will do. Also give a brief reason for your answer:

Solution. Inspecting the cell residuals and the dual weights (figures 3 and 4 in Appendix A), we see that error in the average of u over R comes primarily from two sources: first, error occurs due to the boundary layer near $x = 1$ as seen by $\|r_T\|_{L^2}$; second, $\|w_T\|_{L^2}$ shows that error occurs along the discontinuity of the indicator function, i.e. inside the domain at the boundary between R and the rest of Ω.

When compararing the relative effect of these two sources of error we can use the local error indicators η_T which take both r_T and w_T into account. From the plot of η_T (Figure 5), we see that the error along the boundary layer dominates, and so the cells near the edge $x = 1$ should be refined in order to most effectively lower the error of the function $J(u)$ with the least computational effort.

$\square$

(Optional) Bonus Question | 1 bonus mark Derive an *a priori* estimate for the error in the above quantity of interest. Does it agree with the numerical results from Q2(b)?

Solution. From Theorem 2.3.22 (Convergence in the L^2-Norm) in the notes, we have that, for linear finite elements, the error $e^h = \bar{u} - u^h$ of the Poisson-Dirichlet problem satisfies

$$\|e^h\|_{L^2} \leq ch^2 \|f\|_{L^2(\Omega)}$$

for some positive constant c.

Now, since the quantity of interest $J(u)$ is linear with respect to u, we have that $J(\bar{u}) - J(u^h) = J(\bar{u} - u^h) = J(e^h)$, and so

$$\begin{aligned} |J(e^h)| &= \left| \int_\Omega \left\{ \frac{1}{|R|} I_R \right\} e^h \, \mathrm{d}x \right| \\ &\leq \frac{1}{|R|} \|I_R\|_{L^2} \|e^h\|_{L^2} \\ &\leq 2ch^2 \|f\|_{L^2(\Omega)} \end{aligned}$$

where we have applied Cauchy-Schwarz in line two, and used $|R| = 1/4$, $\|I_R\|_{L^2} = 1/2$ in line 3, as well as the theorem quoted above.

Therefore, from this estimate we expect that $|J(e^h)|$ should decrease proportional to h^2. Inspecting the numerical results from Q2(b), we see that upon refining the grid size by factors of 2, $|J(e^h)|$ decreases by factors of approximately 3.6056 and 3.9480 with decreasing h, approaching the theoretical rate of $2^2 = 4$. Therefore, yes, the *a priori* error estimate does indeed agree with the numerical results.

$\square$

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

- More FEniCS! It is a fun library to use, and clearly you can easily do lots of interesting things with it.

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

- I think the dual weighted residual method will be extremely useful for me - just need to learn how to apply it in the context of parabolic PDEs.

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

- No particular *aha moment*, but this was a pretty intuitive experiment, so I'm glad to see it line up with intuition.

Appendix A

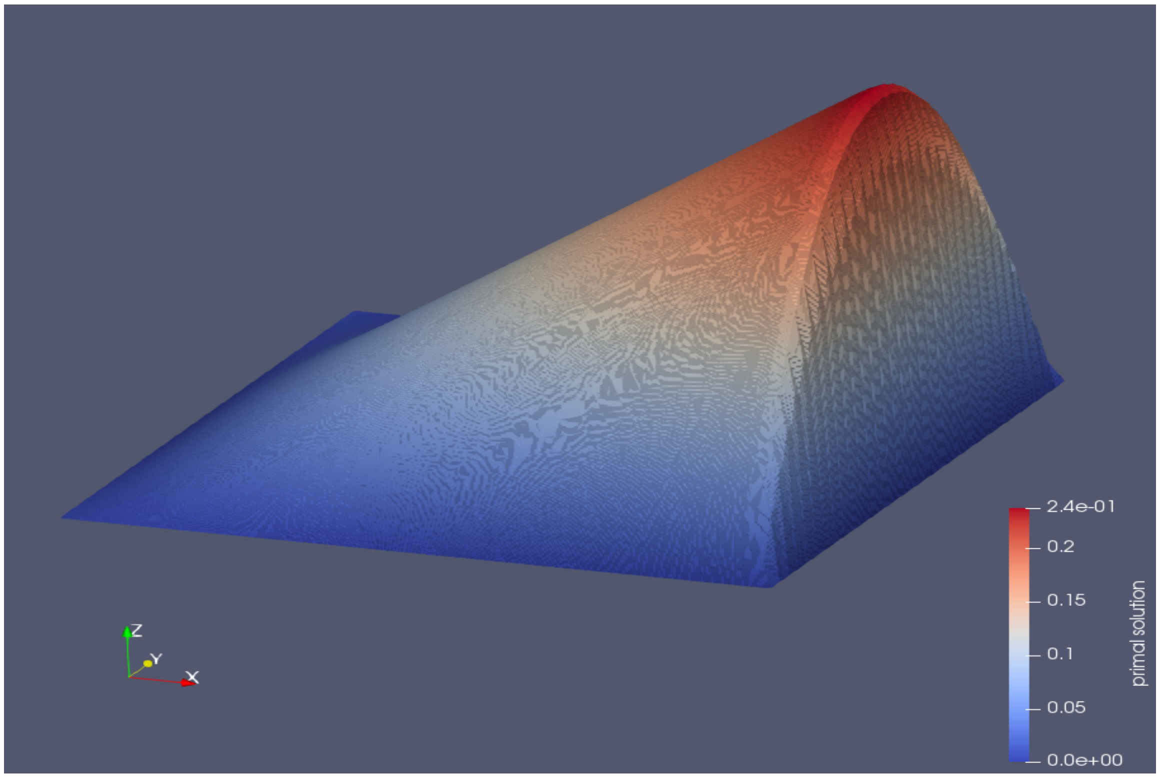

Figure 1: Solution u^h

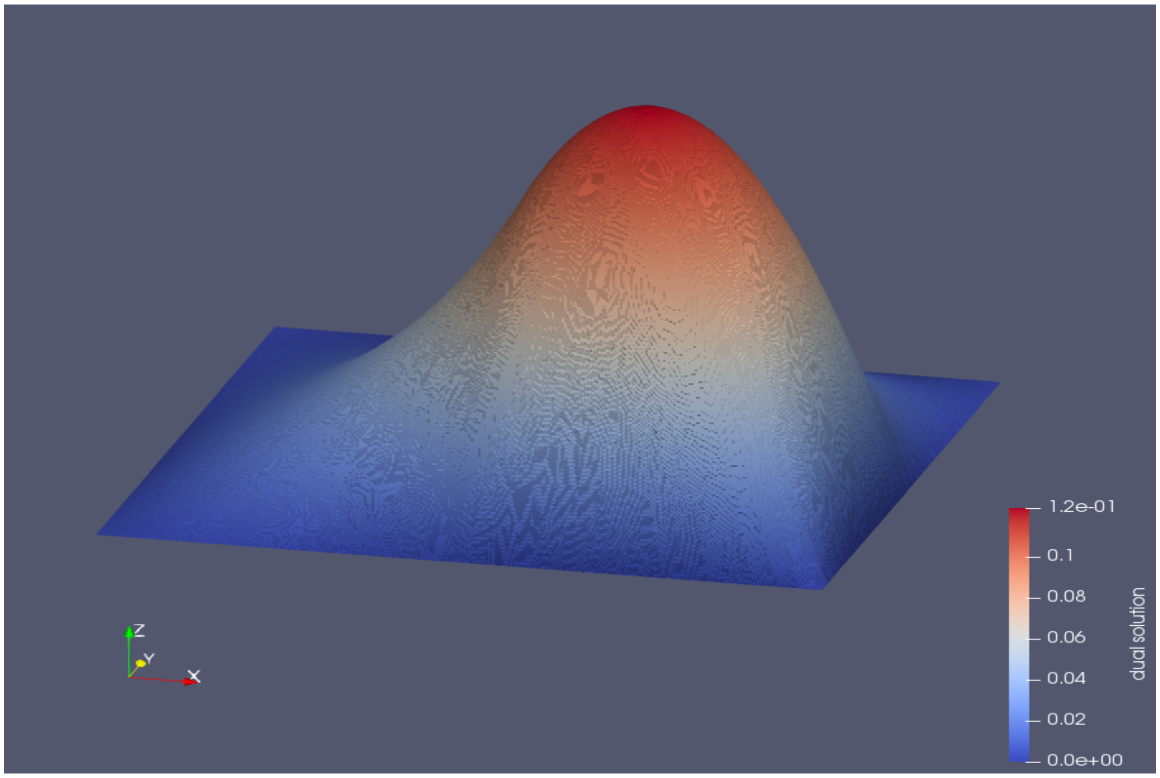

Figure 2: Dual solution z^h

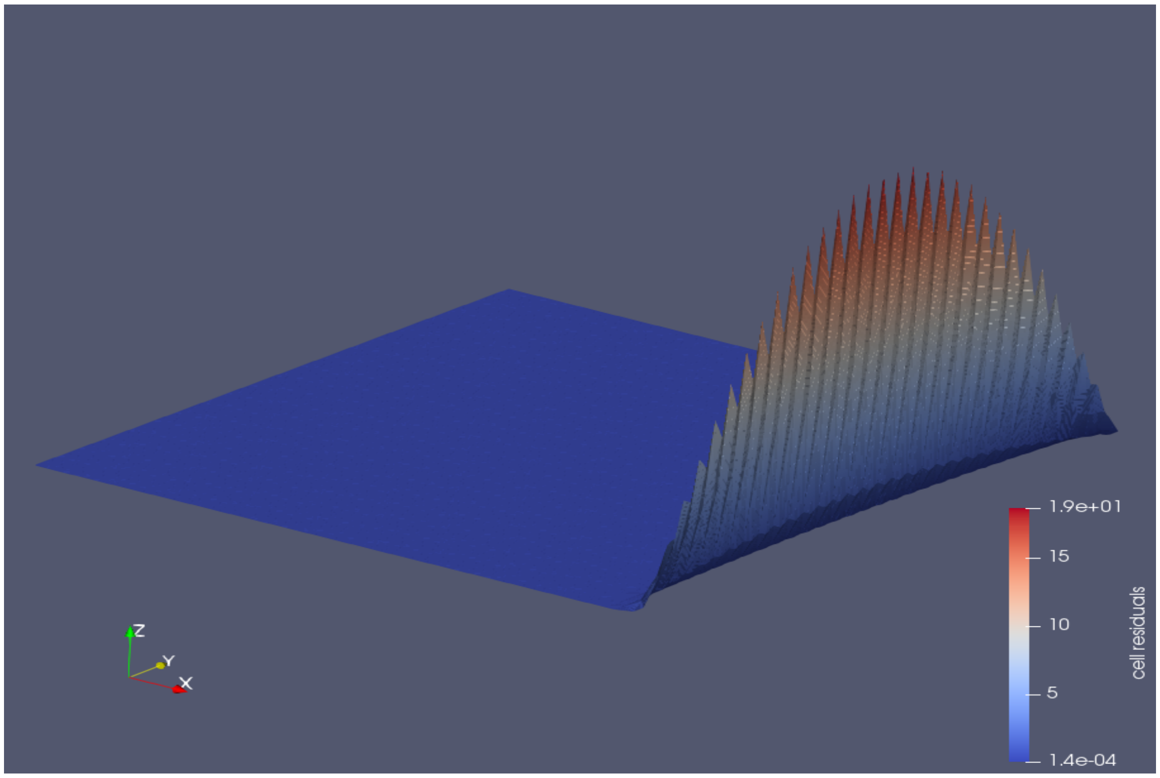

Figure 3: Norm of cell residuals $\|r_T\|_{L^2}$

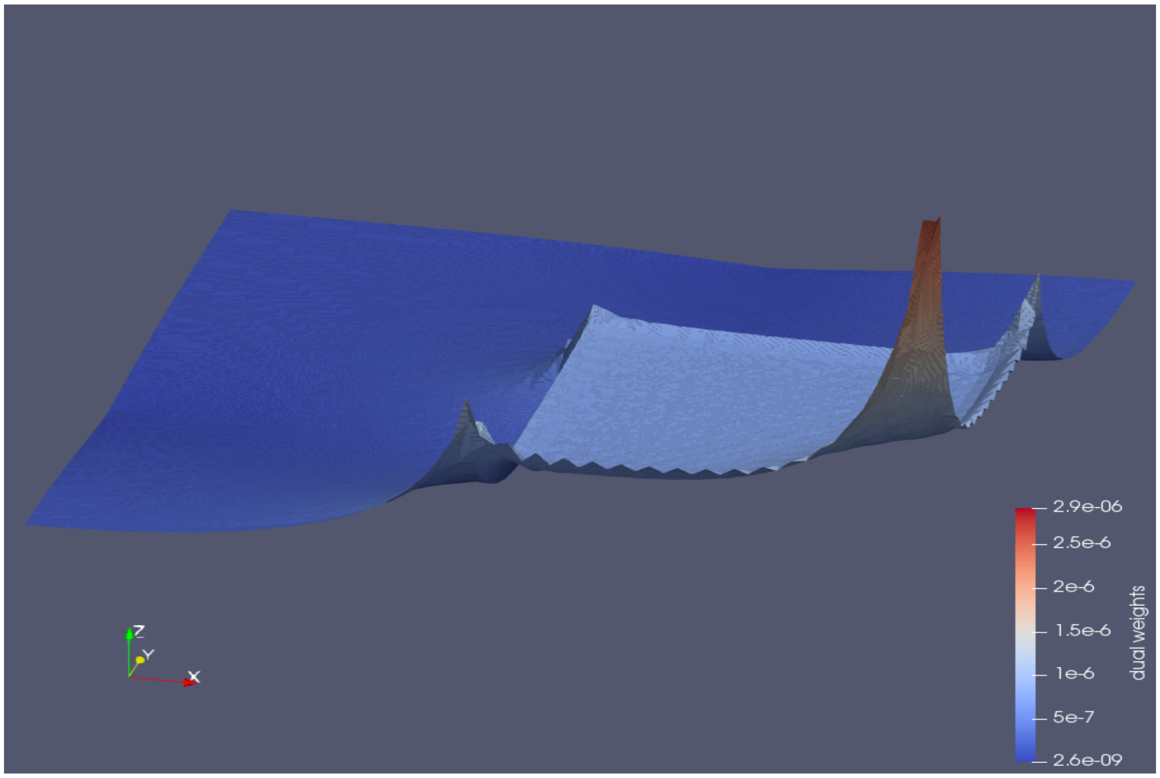

Figure 4: Norm of dual weights $\|w_T\|_{L^2}$

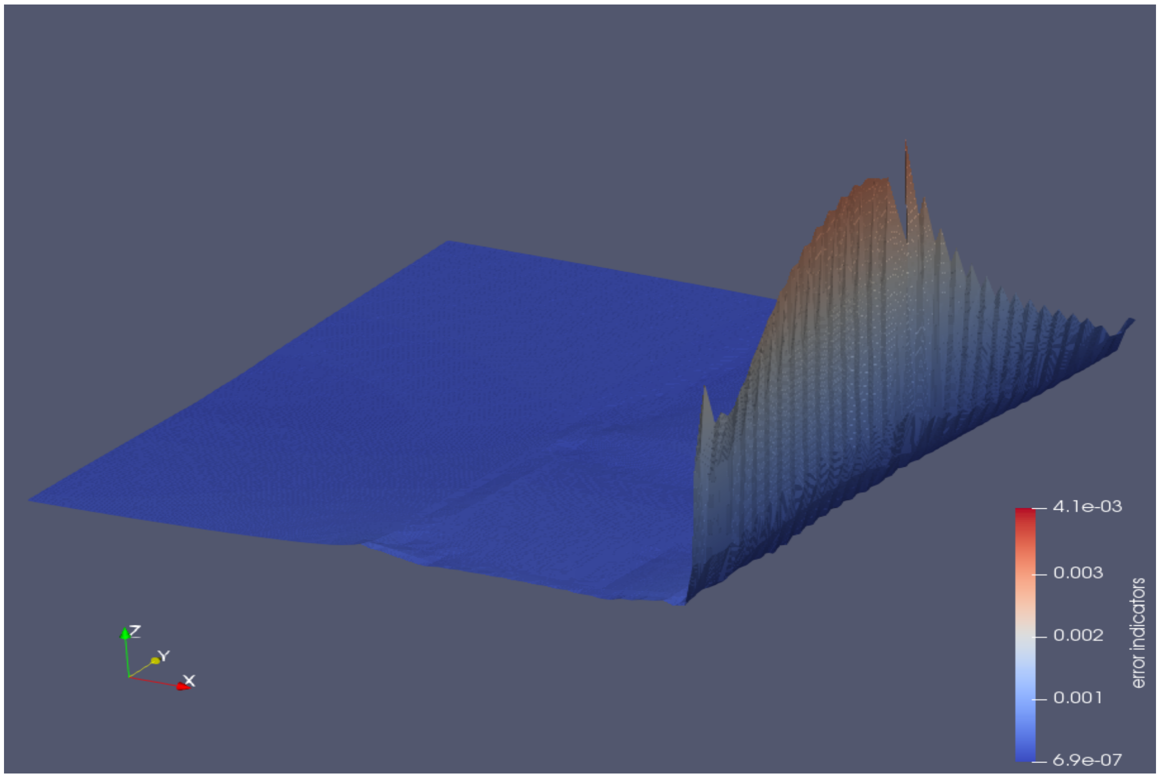

Figure 5: Local error indicators η_T

Appendix B

hw8.py

NOTE: UNICODE IS NOT DISPLAYED CORRECTLY. SEE THE ATTACHED FILE.

```
# coding=utf-8
"""
FEniCS program: A posteriori error estimates for Poisson's equation with
Dirichlet boundary conditions und the unit square.

The boundary values and the source term are chosen such that

    u(x,y) = x(1- x   )y(1-y)

is the exact solution of this problem (with a parameter a >=1).
In particular, for the region of interest R = (1/2, 1) x (0,1/2), we have

    f(x,y) = a(a+1) x       y (1-y)  + 2x(1- x   )
      J(u) = (3a-2+2         )/(24a + 48)

where J(u) is the average of the solution u over R.
"""

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

################################################################################
# DATA
################################################################################

# Parameters
N = 64 # the PDE will be solved on an NxN grid (N must be even)
a = 100.
f = Expression('a*(a+1)*pow(x[0],a-1)*x[1]*(1-x[1]) + 2*x[0]*(1-pow(x[0],a))',
               degree=3, a=a) # source term
# NB: For solving the PDE with the optimal convergence rate, a piecewise
# constant approximation (midpoint rule) would already suffice. However, we'll
# also need f later on to compute the (ideally exact) residuals, and hence
# we're using a higher-order, piecewise cubic, interpolation here.
u_D = Constant(0.0) # boundary values

# Exact solution
uBar = Expression('x[0]*(1-pow(x[0],a))*x[1]*(1-x[1])', degree=3, a=a)
# NB: by interpolating the exact solution with piecewise polynomials of degree
#    (degree of FE solution) + 2 = 3
# we hope that the interpolation error between the exact solution and the
# interpolated uBar will be negligible compared to the error between the exact
# solution and the piecewise linear FE solution.

# Exact quantity of interest
JBar = (3*a - 2 + 2.0**(1-a))/(24*a + 48)

# Create mesh and compute extra mesh data
coarsemesh = UnitSquareMesh(N/2,N/2) # post-processing only (cheap Strategy 2)
mesh = refine(coarsemesh) # for solving PDEs (each triangle -> 4 subtriangles)
h = CellDiameter(mesh) # longest edge of each triangle
n = FacetNormal(mesh) # outward pointing normal vectors on each triangle edge

################################################################################
# SOLUTION OF PROBLEM (P)
################################################################################

# Function space and boundary conditions
V = FunctionSpace(mesh, 'P', 1)

def boundary(x, on_boundary):
    return on_boundary
```

```
bc = DirichletBC(V, u_D, boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
B = dot(grad(u), grad(v))*dx
F = f*v*dx

# Compute solution
u = Function(V, name='primal solution')
solve(B == F, u, bc)

# Compute quantity of interest
#   Hint: go back to the tutorial
#   https://fenicsproject.org/pub/tutorial/html/._ftut1004.html#___sec29
#   and refer to the green box "String expressions must have valid C++ syntax"
#   to find out how you can input a function that's defined piecewise.
x0, x1 = MeshCoordinates(mesh)
Inv_R  = 1.0/0.25

# Since the domain is ]0,1[ x ]0,1[, we may assume all x0, x1 values are in
# this range, and so we can specify the lower left corner R = ]1/2,1[ x ]0,1/2[
# simply as 0.5 <= x0 and x1 <= 0.5. Below, the multiplication of the two
# conditional expressions results in an expression which is 1 in R, and zero
# outside of R.
j = Inv_R * conditional(0.5 <= x0, 1, 0) * conditional(x1 <= 0.5, 1, 0)
Jh = assemble(j*u*dx) # quantity of interest computed from numerical solution

# L  -error
error_L2 = errornorm(uBar, u, 'L2')
# J-error
error_J = np.abs(Jh-JBar)

################################################################################
# A POSTERIORI ERROR ESTIMATION:  L  -NORM
################################################################################

# All error indicators and cell-wise norms are constant on each triangle
W = FunctionSpace(mesh, 'DG', 0) # space of piecewise constant functions
w = TestFunction(W) # hat functions which 1 on one triangle and = 0 elsewhere

# (Squares of) local error indicators
etaT_L2 = ( h**4 * (-div(grad(u))-f)**2 * w * dx +
             0.5 * avg(h)**3 * jump(grad(u),n)**2 * avg(w) * dS )
# NB: For the test function w which is = 1 on triangle T, this expression
# returns the error indicator on this triangle T.
#    avg(w) = average of
#                  (w evaluated on one side of the edge)
#              and
#                  (w evaluated on the other side of the edge)
#
#              | 1/2 (on the edges of triangle T)
#            = |
#              | 0 (on all other edges)

# Assemble the vector which contains the (squares of) local error indicators
etaT_L2_vec = np.abs(assemble(etaT_L2))
# NB: etaT_L2 is a linear form like <f,v>
#     etaT_L2_vec is a vector like the load vector fh
# Here the absolute value only makes sure that none of the error indicators are
# negative (which could happen due to round-off errors).

# A posteriori error estimate of the L  -error
eta_L2 = np.sqrt(np.sum(etaT_L2_vec))

print('=== A POSTERIORI ERROR ESTIMATION:  L  -NORM ===========================')
print('\n')
print(' L  -error          ||eh|| =', error_L2)
print('estimator              _L   =', eta_L2)
print('ratio         _L  /||eh|| =', eta_L2/error_L2)
print('\n')


################################################################################
# A POSTERIORI ERROR ESTIMATION: QUANTITY OF INTEREST
################################################################################

## EXPENSIVE STRATEGY 1 ########################################################

# Function space and boundary conditions
Z_fine = FunctionSpace(mesh, 'P', 2)
bc = DirichletBC(Z_fine, u_D, boundary)

# Define variational problem
zh = TrialFunction(Z_fine)
v = TestFunction(Z_fine)
B = dot(grad(zh), grad(v))*dx
J = j*v*dx
# NB: We have to re-assemble the entire system from scratch since we're now
# using quadratic and not linear elements. Cannot re-use anything from the
# primal problem. It even has more degrees of freedom -> bigger linear system.

# Compute solution
zh = Function(Z_fine, name='dual solution')
solve(B == J, zh, bc) # numerical solution of the dual problem

# Piecewise quadratic approximation of the exact solution
z = zh
# Its piecewise linear interpolant
Iz = interpolate(z,V)

# NB: In FEniCS we can only add / subtract functions that belong to the same
# function space. We need z-Iz, however:
#    z belongs to Z_fine = {piecewise quadratic functions on mesh}
#    Iz belongs to V = {piecewise linear functions on mesh}
# Therefore, we re-write the piecewise linear function as a piecewise quadratic
# function (with quadratic terms = 0):
Iz = interpolate(Iz,Z_fine)

# Local error indicators
etaT_J1 = (-div(grad(u))-f)*(z-Iz)*w*dx + jump(grad(u),n)*(z-Iz)*avg(w)*dS

# Assemble the vector which contains the local error indicators
etaT_J1_vec = np.abs(assemble(etaT_J1))

# Expensive a posteriori error estimate of the J-error
eta_J1 = np.sum(etaT_J1_vec)

print('=== A POSTERIORI ERROR ESTIMATION: QUANTITY OF INTEREST (EXPENSIVE) ==')
print('\n')
print('J-error         |J(eh)| =', error_J)
print('estimator           _J1  =', eta_J1)
print('ratio       _J1/|J(eh)| =', eta_J1/error_J)
print('\n')


## CHEAP STRATEGY 2 ############################################################

# Function space and boundary conditions
Z_fine   = FunctionSpace(mesh, 'P', 2)
Z_coarse = FunctionSpace(coarsemesh, 'P', 2) # quadratic space on courser mesh
bc = DirichletBC(V, u_D, boundary)

# Define variational problem
zh = TrialFunction(V)
v = TestFunction(V)
B = dot(grad(zh), grad(v))*dx
J = j*v*dx
# NB: If we hadn't overwritten the data from the primal problem for the
# expensive error estimator, we could have re-used it here. FEniCS even
# provides the command 'adjoint' which computes the left hand side of the dual
# problem automatically from the bilinear form B of the primal problem.

# Compute solution
zh = Function(V, name='dual solution')
solve(B == J, zh, bc) # numerical solution of the dual problem

# Piecewise quadratic interpolant on patches of four triangles
z = interpolate(zh, Z_coarse) # Its piecewise quadratic interpolant on Z_coarse

# Its piecewise linear interpolant = ???
Iz = zh

# How do you compute their difference???
# NB: In FEniCS we can only add / subtract functions that belong to the same
# function space. We need z-Iz, however:
#    z belongs to Z_coarse = {piecewise quadratic functions on coarsemesh}
#    Iz belongs to V = {piecewise linear functions on mesh}
# Therefore, we interpolate the piecewise quadratic function z on the coarsemesh
# AND the piecewise linear function Iz on the fine mesh to the piecewise
# quadratic space Z_fine on the fine mesh; this doesn't change either of the
# function's actual shape, only puts them into the same space

# Testing that the functions don't change during interpolation:
# print('L2-error: ||z  - interpolate(z, Z_fine)|| = ', assemble( (interpolate(z, Z_fine) - z )**2 * dx(mesh) ) )
# print('L2-error: ||Iz - interpolate(Iz,Z_fine)|| = ', assemble( (interpolate(Iz,Z_fine) - Iz)**2 * dx(mesh) ) )

Iz = interpolate(Iz,Z_fine) # interp fine piecewise linear to piecewise quad
z  = interpolate(z,Z_fine)  # interp coarse piecewise quad to fine piecewise quad

# Local error indicators
etaT_J2 = (-div(grad(u))-f)*(z-Iz)*w*dx + jump(grad(u),n)*(z-Iz)*avg(w)*dS

# Assemble the vector which contains the local error indicators
etaT_J2_vec = np.abs(assemble(etaT_J2))

# Cheap a posteriori error estimate of the J-error
eta_J2 = np.sum(etaT_J2_vec)

print('=== A POSTERIORI ERROR ESTIMATION: QUANTITY OF INTEREST (CHEAP) ======')
print('\n')
print('J-error         |J(eh)| =', error_J)
print('estimator           _J2  =', eta_J2)
print('ratio       _J2/|J(eh)| =', eta_J2/error_J)
print('\n')


################################################################################
# EXPORT DATA FOR PLOTTING
################################################################################

# Exporting data from FEniCS to post-process it in ParaView works as follows:
#    1. Start with a function u.
#    2. Use the left shift command << to export it to a PVD file.
#          Piecewise constant functions are saved as values on triangles.
#          Piecewise linear functions are saved as values on the mesh points.
#          Piecewise quadratic and higher-order functions are also saved as
#          values on the mesh points only, so they will be plotted as piecewise
#          linear NOT piecewise quadratic etc.
#    3. Open the PVD file in ParaView. You normally have to click on the green
#       button 'Apply' before the function is shown.
#    4. For a 3D surface plot, go to Filters -> Alphabetical -> Warp by Scalar
#       and click 'Apply'. In the figure window, switch from '2D' to '3D' so
#       that you can rotate the surface.

# Forcing function
File('poisson/hw8_f.pvd') << interpolate(f, Z_fine)

# Solution of the primal problem
File('poisson/hw8_u.pvd') << u

# Solution of the dual problem
```

```
File('poisson/hw8_z.pvd') << zh

# Cell residuals
# We have to compute a piecewise constant function with the cell residuals as
# its function values.

# Squares of cell residuals ||rT||  _L  (T)
rT = (-div(grad(u))-f)**2 * w * dx
# Assemble the corresponding vector
rT_vec = np.sqrt(np.abs(assemble(rT)))
# Define the function with these function values
rT_fun = Function(W, name='cell residuals')
rT_fun.vector().set_local(rT_vec)
File('poisson/hw8_rT.pvd') << rT_fun

# Squares of dual weights ||wT||  _L  (T)
wT = (z-Iz)**2 * w * dx
# Assemble the corresponding vector
wT_vec = np.sqrt(np.abs(assemble(wT)))
# Define the function with these function values
wT_fun = Function(W, name='dual weights')
wT_fun.vector().set_local(wT_vec)
File('poisson/hw8_wT.pvd') << wT_fun

# Error indicators
etaT = (-div(grad(u))-f)*(z-Iz)*w*dx + jump(grad(u),n)*(z-Iz)*avg(w)*dS
# Assemble the corresponding vector
etaT_vec = np.sqrt(np.abs(assemble(etaT)))
# Define the function with these function values
etaT_fun = Function(W, name='error indicators')
etaT_fun.vector().set_local(etaT_vec)
File('poisson/hw8_etaT.pvd') << etaT_fun
```