Try_Perturb

October 17, 2020

1 Solving a Fourth Order Elliptic Singular Perturbation Problem

$$\begin{cases} \varepsilon^2\Delta^2 u - \Delta u = f & \text{in } \Omega \\ u = \partial_n u = 0 & \text{on } \partial\Omega \end{cases}$$

[1]:
```
from skfem import *
import numpy as np
from skfem.visuals.matplotlib import draw, plot
from skfem.utils import solver_iter_krylov
from skfem.helpers import dd, ddot, grad
from scipy.sparse.linalg import LinearOperator, minres
from skfem import *
from skfem.models.poisson import *
from skfem.assembly import BilinearForm, LinearForm
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
plt.rcParams['figure.dpi'] = 100

pi = np.pi
sin = np.sin
cos = np.cos
```

1.1 Problem 1

The modified Morley-Wang-Xu element method is equivalent to finding $w_h \in W_h$ and $u_{h0} \in V_{h0}$ such that

$$\begin{aligned} (\nabla w_h, \nabla \chi_h) &= (f, \chi_h) & \forall \chi_h \in W_h \\ \varepsilon^2 a_h(u_{h0}, v_h) + b_h(u_{h0}, v_h) &= (\nabla w_h, \nabla_h v_h) & \forall v_h \in V_{h0} \end{aligned}$$

where

$$a_h(u_{h0}, v_h) := \left(\nabla_h^2 u_{h0}, \nabla_h^2 v_h\right), \quad b_h(u_{h0}, v_h) := (\nabla_h u_{h0}, \nabla_h v_h)$$

Using example

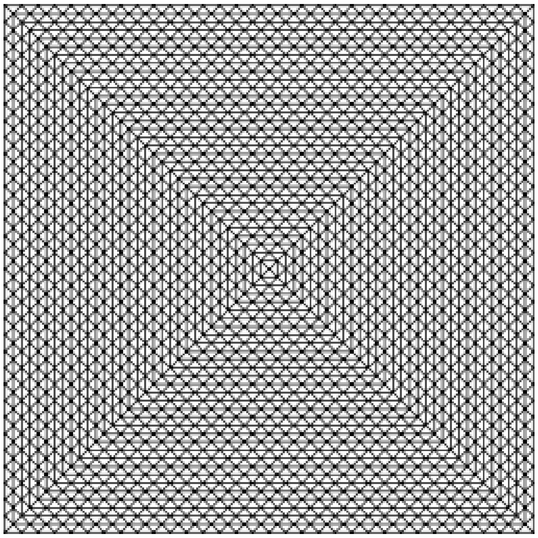
$$u(x_1, x_2) = (\sin(\pi x_1)\sin(\pi x_2))^2$$

1.1.1 Setting ϵ and generating mesh

[44]:
```
epsilon = 0

m = MeshTri.init_symmetric()
# m = MeshTri()
m.refine(5)
element = {'w': ElementTriP1(), 'u': ElementTriMorley()}
basis = {variable: InteriorBasis(m, e, intorder=4)
    for variable, e in element.items()}  # intorder: integration order for␣
 ↪quadrature

draw(m)
plt.show()
```



1.1.2 Forms for $(\nabla w_h, \nabla \chi_h) = (f, \chi_h)$

[46]:
```
@BilinearForm
def laplace(u, v, w):
    '''
    for $(\nabla w_{h}, \nabla \chi_{h})$
    '''
    return dot(grad(u), grad(v))


@LinearForm
def f_load(v, w):
```

```
    '''
    for $(f, x_{h})$
    '''
    pix = pi * w.x[0]
    piy = pi * w.x[1]
    lu = 2 * (pi)**2 * (cos(2*pix)*((sin(piy))**2) + cos(2*piy)*((sin(pix))**2))
    llu = - 8 * (pi)**4 * (cos(2*pix)*sin(piy)**2 + cos(2*piy)*sin(pix)**2 -␣
↪cos(2*pix)*cos(2*piy))
    return (epsilon**2 * llu - lu) * v
```

1.1.3 Solving w_h

[47]:
```
%%time

K1 = asm(laplace, basis['w'])
f1 = asm(f_load, basis['w'])

wh = solve(*condense(K1, f1, D=m.boundary_nodes()),␣
↪solver=solver_iter_krylov(Precondition=True))
```

```
Wall time: 19.9 ms
```

1.1.4 Forms for $\varepsilon^2 a_h\left(u_{h0}, v_h\right) + b_h\left(u_{h0}, v_h\right) = \left(\nabla w_h, \nabla_h v_h\right)$

$$a_h\left(u_{h0}, v_h\right) := \left(\nabla_h^2 u_{h0}, \nabla_h^2 v_h\right), \quad b_h\left(u_{h0}, v_h\right) := \left(\nabla_h u_{h0}, \nabla_h v_h\right)$$

[20]:
```
@BilinearForm
def a_load(u, v, w):
    '''
    for $a_{h}$
    '''
    return ddot(dd(u), dd(v))


@BilinearForm
def b_load(u, v, w):
    '''
    for $b_{h}$
    '''
    return dot(grad(u), grad(v))


@BilinearForm
def wv_load(u, v, w):
    '''
    for $(\nabla \chi_{h}, \nabla_{h} v_{h})$
    '''
```

```
    return dot(grad(u), grad(v))
```

1.1.5 Setting boundary conditions

[21]:
```
def easy_boundary(basis):
    '''
    Input basis
    ----------------
    Return D for boundary conditions
    '''

    dofs = basis.find_dofs({
        'left': m.facets_satisfying(lambda x: x[0] == 0),
        'right': m.facets_satisfying(lambda x: x[0] == 1),
        'top': m.facets_satisfying(lambda x: x[1] == 1),
        'buttom': m.facets_satisfying(lambda x: x[1] == 0)
    })

    D = np.concatenate((dofs['left'].nodal['u'], dofs['right'].nodal['u'],
                        dofs['top'].nodal['u'], dofs['buttom'].nodal['u'],
                        dofs['left'].facet['u_n'], dofs['right'].facet['u_n'],
                        dofs['top'].facet['u_n'], dofs['buttom'].facet['u_n']))
    return D
```

1.1.6 Solving u_{h0}

[22]:
```
%%time

D = easy_boundary(basis['u'])
K2 = epsilon**2 * asm(a_load, basis['u']) + asm(b_load, basis['u'])
f2 = asm(wv_load, basis['w'], basis['u']) * wh
uh0 = solve(*condense(K2, f2, D=D),␣
 ↪solver=solver_iter_krylov(Precondition=True)) # cg
```

```
Wall time: 65.8 ms
```

1.1.7 Computing L_2 H_1 H_2 error with u_{h0} and u

[45]:
```
def exact_u(x, y):
    return (sin(pi * x) * sin(pi * y))**2

def dexact_u(x, y):
    dux = 2 * pi * cos(pi * x) * sin(pi * x) * sin(pi * y)**2
    duy = 2 * pi * cos(pi * y) * sin(pi * x)**2 * sin(pi * y)
    return dux, duy
```

```
def ddexact(x, y):
    duxx = 2*pi**2*cos(pi*x)**2*sin(pi*y)**2 - 2*pi**2*sin(pi*x)**2*sin(pi*y)**2
    duxy = 2*pi*cos(pi*x)*sin(pi*x)*2*pi*cos(pi*y)*sin(pi*y)
    duyx = duxy
    duyy = 2*pi**2*cos(pi*y)**2*sin(pi*x)**2 - 2*pi**2*sin(pi*y)**2*sin(pi*x)**2
    return duxx, duxy, duyx, duyy

@Functional
def L2uError(w):
    x, y = w.x
    return (w.w - exact_u(x, y))**2

def get_DuError(basis, u):
    duh = basis.interpolate(u).grad
    x = basis.global_coordinates().value
    dx = basis.dx  # quadrature weights
    dux, duy = dexact_u(x[0], x[1])
    return np.sqrt(np.sum(((duh[0] - dux)**2 + (duh[1] - duy)**2) * dx))

def get_D2uError(basis, u):
    dduh = basis.interpolate(u).hess
    x = basis.global_coordinates().value  # coordinates of quadrature points [x,␣
 ↪y]
    dx = basis.dx  # quadrature weights
    duxx, duxy, duyx, duyy = ddexact(x[0], x[1])
    return np.sqrt(
        np.sum(((dduh[0][0] - duxx)**2 + (dduh[0][1] - duxy)**2 +
                (dduh[1][1] - duyy)**2 + (dduh[1][0] - duyx)**2) * dx))
```

[38]:
```
epsilon = 1

L2_list = []
Du_list = []
D2u_list = []
h_list = []
m = MeshTri()

for i in range(1, 7):
    m.refine()

    element = {'w': ElementTriP1(), 'u': ElementTriMorley()}
    basis = {variable: InteriorBasis(m, e, intorder=4)
        for variable, e in element.items()}  # intorder: integration order for␣
 ↪quadrature

    K1 = asm(laplace, basis['w'])
    f1 = asm(f_load, basis['w'])
```

```
    wh = solve(*condense(K1, f1, D=m.boundary_nodes()),␣
↪solver=solver_iter_krylov(Precondition=True))

    D = easy_boundary(basis['u'])
    K2 = epsilon**2 * asm(a_load, basis['u']) + asm(b_load, basis['u'])
    f2 = asm(wv_load, basis['w'], basis['u']) * wh
    uh0 = solve(*condense(K2, f2, D=D),␣
↪solver=solver_iter_krylov(Precondition=True))

    U = basis['u'].interpolate(uh0).value

    L2u = np.sqrt(L2uError.assemble(basis['u'], w=U))
    Du = get_DuError(basis['u'], uh0)
    H1u = Du + L2u
    D2u = get_D2uError(basis['u'], uh0)
    H2u = Du + L2u + D2u
    print('Case 2^-' + str(i))
    print('L2 error of uh0:', L2u)
    print('H1 error of uh0:', H1u)
    print('H2 error of uh0:', H2u)
    h_list.append(m.param())
    Du_list.append(Du)
    L2_list.append(L2u)
    D2u_list.append(D2u)
```

```
Case 2^-1
L2 error of uh0: 0.14312157458419236
H1 error of uh0: 1.1285182985411255
H2 error of uh0: 13.136803545627988
Case 2^-2
L2 error of uh0: 0.041409209806485096
H1 error of uh0: 0.6019549703041572
H2 error of uh0: 8.166025291658702
Case 2^-3
L2 error of uh0: 0.00905868775220735
H1 error of uh0: 0.17818886544221169
H2 error of uh0: 4.013397269740482
Case 2^-4
L2 error of uh0: 0.0020300577442618198
H1 error of uh0: 0.046720223553143266
H2 error of uh0: 1.9423395021729348
Case 2^-5
L2 error of uh0: 0.0004859434175848888
H1 error of uh0: 0.011822005583285031
H2 error of uh0: 0.9550957673989472
Case 2^-6
```

```
L2 error of uh0: 0.00011996444727651908
H1 error of uh0: 0.0029644451905044395
H2 error of uh0: 0.47396480908547806
```

[39]:

```python
hs = np.array(h_list)
L2s = np.array(L2_list)
Dus = np.array(Du_list)
D2us = np.array(D2u_list)

H1s = L2s + Dus
H2s = H1s + D2us
print('epsilon =', epsilon)
print('  h     L2u    H1u    H2u')
for i in range(H2s.shape[0] - 1):
    print(
        '2^-' + str(i + 2),
        ' {:.2f}  {:.2f}  {:.2f}'.format(-np.log2(L2s[i + 1] / L2s[i]),
                                         -np.log2(H1s[i + 1] / H1s[i]),
                                         -np.log2(H2s[i + 1] / H2s[i])))
```

```
epsilon = 1
  h     L2u    H1u    H2u
2^-2  1.79  0.91  0.69
2^-3  2.19  1.76  1.02
2^-4  2.16  1.93  1.05
2^-5  2.06  1.98  1.02
2^-6  2.02  2.00  1.01
```

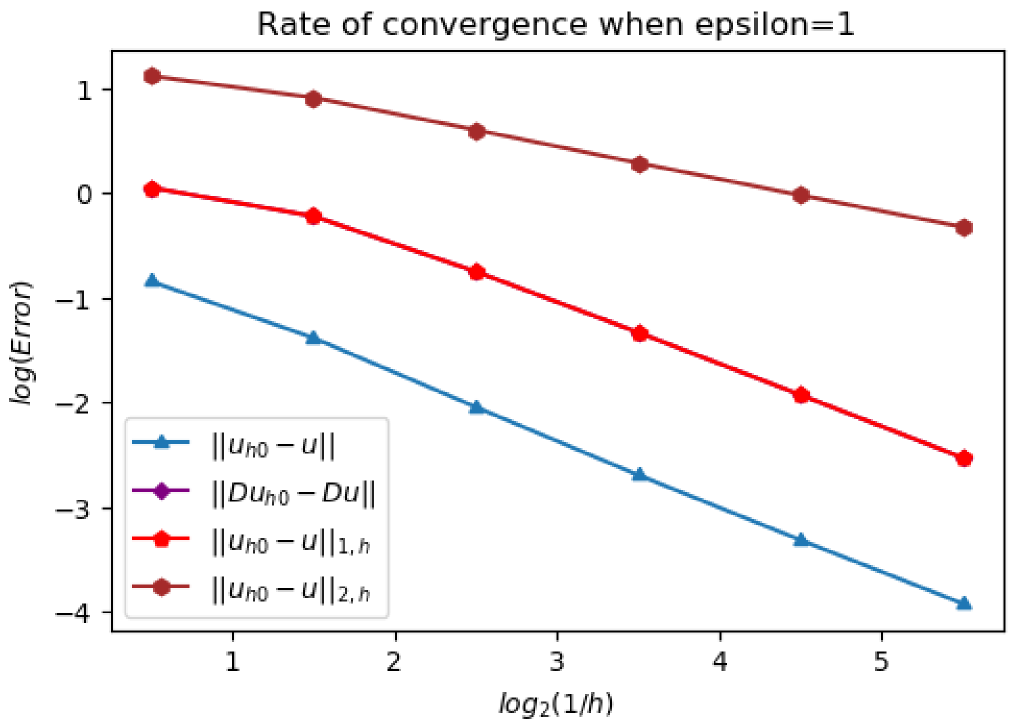
[43]:

```python
hs_Log = np.log2(hs)

L2plot, = plt.plot(-hs_Log,
                   np.log10(L2s),
                   marker=(3, 0),
                   label='$|\|u_{h0}-u\||}$')
Duplot, = plt.plot(-hs_Log,
                   np.log10(H1s),
                   marker=(4, 0),
                   label=r'$|\left\|{Du}_{h0}-Du\right\||$',
                   color='purple')
H1plot, = plt.plot(-hs_Log,
                   np.log10(H1s),
                   marker=(5, 0),
                   label=r'$|\left\|{u}_{h0}-u\right\||_{1, h}$',
                   color='red')
H2plot, = plt.plot(-hs_Log,
                   np.log10(H2s),
                   marker=(6, 0),
```

```
                    label=r'$|\left\|{u}_{h0}-u\right\||_{2, h}$',
                    color='brown')

plt.legend(handles=[L2plot, Duplot, H1plot, H2plot])
plt.title('Rate of convergence when epsilon='+str(epsilon))
plt.xlabel('$log_{2}(1/h)$')
plt.ylabel('$log(Error)$')
plt.show()
```



1.2 Problem 2

The modified Morley-Wang-Xu element method is also equivalent to

$$
\begin{aligned}
(\nabla w_h, \nabla \chi_h) &= (f, \chi_h) && \forall \chi_h \in W_h \\
\varepsilon^2 \tilde{a}_h(u_h, v_h) + b_h(u_h, v_h) &= (\nabla w_h, \nabla_h v_h) && \forall v_h \in V_h
\end{aligned}
$$

where

$$
\tilde{a}_h(u_h, v_h) := (\nabla_h^2 u_h, \nabla_h^2 v_h) - \sum_{F \in \mathcal{F}_h^\partial} (\partial_{nn}^2 u_h, \partial_n v_h)_F - \sum_{F \in \mathcal{F}_h^\partial} (\partial_n u_h, \partial_{nn}^2 v_h)_F + \sum_{F \in \mathcal{F}_h^0} \frac{\sigma}{h_F} (\partial_n u_h, \partial_n v_h)_F
$$