Assignment 1, Mek4250

Sebastian Gjertsen

16. mars 2016

1

a)

The H^p norm consist of the L2 norm plus all the derivatives up the hth degree.

$$||u||_1 = \Big(\int u^2 + \frac{\partial u}{\partial x}^2 \Big)^{\frac{1}{2}} = sin(\pi kx)^2 cos(\pi ly)^2 + (\pi k cos(\pi kx) cos(\pi ly) - \pi l sin(\pi kx) sin(\pi ly))^2$$

$$||u||_1 = \big(\pi^2(k^2+l^2)(\frac{1}{4})\big)^{\frac{1}{2}}$$

We get this recurring for every added degree of h. Giving:

$$||u||_h = \Big(\sum_i^h \pi^{2i}(k^2+l^2)^i \Big)^{\frac{1}{2}} \ (\frac{1}{2})$$

b)

To compute the H1 and L2 norms i used the norms from FENiCS. As can be seen in the code in the bottom of this section. We can see from the tables of errornorms that the error gets smaller with increasing number of elements and the error increases when we have more oscillations of the sine and cosine functions. The first part makes sense since we can more accurately represent the functions with more points in the mesh. And the latter makes sense because with more oscillations than elements in one direction we only get sort of parts of the function as in crosses through the element. This is in general for the error. For the H1 norm the errornorm gets bigger when we increase k and l because the derivative is gets higher for steeper functions.

LAGRANGE = 1

L2 NORM

N / k&l	1	10	100
8	0.0328	0.6775	159.9224
16	0.0085	0.3634	247.3597
32	0.0021	0.1779	2.5068
64	0.0005	0.0549	3.6142

H1 NORM

N / k&l	1	10	100

8	0.43611616	25.51151631	3232.51360546
16	0.21810459	17.23357873	4689.64045220
32	0.10904725	10.54385048	450.33238520
64	0.05452270	5.43087909	534.88053004

LAGRANGE = 2

L2 NORM

N / k&l	1	10	100
8	0.00056880	0.42444612	293.08481819
16	0.00006933	0.08864854	90.52742877
32	0.00000861	0.01017424	4.57194541
64	0.00000108	0.00113884	1.47234385

H1 NORM

N / k&l	1	10	100
8	0.03314086	17.66688309	5315.73621535
16	0.00838664	6.71715133	1657.67783190
32	0.00210537	1.96195378	702.19909726
64	0.00052716	0.51734696	270.00322078

c)

I used the norms

$$||u - u_h||_0 \leq Ch^{\alpha}||u||_1$$

$$||u - u_h||_1 \leq Ch^{\alpha}||u||_2$$

I did this exercise before the assignment was changed, and it does not change that value of the alpha if we have the norm of u on the LHS. The C will change a bit but we are only looking at if C is big or small not some exact value.
To compute C and α I used the least squares from numpy and made my own, as you can see in the code, to check that it was correct. With lagrange = 1, L2 and H1 norms, k and l 1 and 10 I got $\alpha \approx 2$ and with k and l 100 is almost 3. The C gets high as we increase k and l.
It seems in general that L2 and H1 norms produce better ($\alpha = 2$) and $C = high$
We can see in figure 1 and 2 that the least squares interpolates the best line over the points calculated for different N values. When k gets higher it is harder to find a suitable line and therefore our α and C values are gets worse.

LAGRANGE = 1

L2 and H1 norms:

k = l	C	alpha
1	0.304057859582	1.82285696122
10	9.51367285062	2.25838702398

```
100         23111.6349906    3.24004809894

H1 and H2 norms:
k = l       C                 alpha
1           0.371924037408    1.9111252817
10          0.221516452868    1.11933011599
100         0.397735012462    1.03231000867

LAGRANGE = 2

L2 and H1 norms:

k = l       C                 alpha
1           0.151996496825    3.73753264544
10          18.7979921884     3.31590714197
100         1864.5698256      2.81857174656

H1 and H2 norms:
k = l       C                   alpha
1            0.0975858435078    2.59587384003
10           0.632411889525     2.04983515805
100         0.191004437667     0.915842115932
```

Code

```
from dolfin import *
import numpy as np
import matplotlib.pyplot as plt


def solver_1(N,k,l):
    mesh = UnitSquareMesh(N, N)

    V = FunctionSpace(mesh, "Lagrange", 1)

    class Right(SubDomain):
        def inside(self, x, on_boundary):
                return on_boundary and near(x[0],1)

    class Left(SubDomain):
        def inside(self, x, on_boundary):
                return on_boundary and near(x[0],0)


    right = Right()
    left = Left()
    bound = FacetFunction("size_t", mesh)
    bound.set_all(0)
    right.mark(bound, 1)
```

```
    left.mark(bound,2)
    #plot(bound); interactive()

    bc0 = DirichletBC(V, 0, right)
    bc1 = DirichletBC(V, 0, left)
    bcs = [bc0,bc1]

    u = TrialFunction(V)
    v = TestFunction(V)

    f = Expression("((pi*pi*k*k)+(pi*pi*l*l))*sin(pi*k*x[0])*cos(pi*l*x[1])",\
                k=k,l=l) #right side of equation
    u_exact = interpolate(Expression("sin(pi*k*x[0])*cos(pi*l*x[1])",\
                l=l,k=k),V) # exact solution of u

    a = inner(grad(u), grad(v))*dx
    F = f*v*dx
    u_1 = Function(V)
    solve(a==F,u_1,bcs)
    f_e = interpolate(f,V)
    return u_1,u_exact,V,mesh,f_e




    #print "mesh = %.d errornorm = %.8f  " \
    #%(N,errornorm(u_exact,u,norm_type = "H1",degree_rise = 3))
N = [2,4,8,16,32,64]
k = l = 100
b = np.zeros(len(N)) #empty arrays
a = np.zeros(len(N))

for i in range(len(N)):
    u ,u_exact,V,mesh,f_e = solver_1(N[i],k,l) # getting u computed and \
    # u exact for a given k,l and N
    e = errornorm(u_exact,u,norm_type = "l2",degree_rise = 3)
    #u_ex = project(u_exact,V)
    u_norm = norm(u_exact,"H1")#+ norm(f_e,"l2") \
    # this line is used for both h1 and h2 norm, since f is the H2 seminorm.
    b[i] = np.log(e/u_norm)
    a[i] = np.log(1./(N[i]))
    #print u_norm
    #u_norm = sum(u_ex*u_ex)**0.5
A = np.vstack([a,np.ones(len(a))]).T
alpha_1,c_1 = np.linalg.lstsq(A,b)[0]
plt.plot(a,b,'o',label='Original data',markersize = 10)
plt.plot(a,alpha_1*a+c_1,'r',label='Fitted line')
plt.legend();plt.show()
```

```
print exp(c_1), alpha_1
B = np.zeros(2)
A = np.zeros((2,2))
A[0,0] = len(N)

for i in range(len(N)): # inserting log values in the matrix and vector
    A[0,1] += a[i]
    A[1,0] += a[i]
    A[1,1] += a[i]**2
    B[1] += a[i]*b[i]
    B[0] += b[i]
#print A
c , alpha = np.linalg.solve(A, B)


print np.exp(c),alpha

plt.plot(a,b,'o',label='Original data',markersize = 10)
plt.plot(a,alpha*a+c,'r',label='Fitted line-my way')
plt.legend();plt.show()
#plt.plot(a,np.exp(alpha)*a+c); #plt.show()
#plt.plot(a,b); plt.show()

#print np.exp(x),y
```

2

a)

To solve this equation analytically we use the method separation of variables and say that $u(x,y) = U(x) * V(y)$
We insert this solution into the equation and get:

$$\mu \frac{V_{yy}}{V} = \mu \frac{U_{xx} + U_x}{U} = \lambda$$

The LHS we know has the solution $V = C_1 sin((\frac{\lambda}{\mu})^{\frac{1}{2}} y) + C_2 cos((\frac{\lambda}{\mu})^{\frac{1}{2}} y)$ and a solution to this equation is when $\lambda = 0$, giving $Y = 1$ This upholds the boundary conditions in y-direction.
The RHS has the known solution, with $\lambda = 0$:

$$U = \frac{e^{\frac{x}{\mu}} - 1}{e^{\frac{1}{\mu}} - 1}$$

This upholds the boundary condition in x-direction. Giving then:

$$u(x,y) = \frac{e^{\frac{x}{\mu}} - 1}{e^{\frac{1}{\mu}} - 1}$$

b)

LAGRANGE ELEMENTS = 2

L2 - NORM:

mu = 1e+00 , N = 8 Numerical Error: 1.1511e-05

mu = 1e+00 , N = 16 Numerical Error: 1.4484e-06

mu = 1e+00 , N = 32 Numerical Error: 1.8175e-07

mu = 1e+00 , N = 64 Numerical Error: 2.2766e-08

mu = 1e-01 , N = 8 Numerical Error: 2.2325e-03

mu = 1e-01 , N = 16 Numerical Error: 3.0338e-04

mu = 1e-01 , N = 32 Numerical Error: 3.8830e-05

mu = 1e-01 , N = 64 Numerical Error: 4.8872e-06

mu = 1e-02 , N = 8 Numerical Error: 7.6887e-02

mu = 1e-02 , N = 16 Numerical Error: 2.7731e-02

mu = 1e-02 , N = 32 Numerical Error: 7.3478e-03

mu = 1e-02 , N = 64 Numerical Error: 1.3145e-03

mu = 1e-03 , N = 8 Numerical Error: nan

mu = 1e-03 , N = 16 Numerical Error: nan

mu = 1e-03 , N = 32 Numerical Error: nan

mu = 1e-03 , N = 64 Numerical Error: nan

We see that the error goes down with increasing number of elements and with decreasing μ. When we decrease the mu past 0.001, python cannot represent that exponential. Thats why we get nan.

c)

To calculate the C and α i used the norms

$$||u - u_h||_0 \leq Ch^\alpha$$

$$||u - u_h||_1 \leq Ch^\alpha$$

LAGRANGE ELEMENTS = 1
L2 and H1

mu = 1.0 C = 0.077217550709 alpha = 1.99848415122
mu = 0.1 C = 0.559578704657 alpha = 1.9343298292
mu = 0.01 C = 2.78391620437 alpha = 1.83179131516

LAGRANGE ELEMENTS = 2
L2 and H1

mu = 1.0 C = 0.00494279949017 alpha = 2.98835482613
mu = 0.1 C = 0.246231900242 alpha = 2.75535616309
mu = 0.01 C = 1.07890852871 alpha = 2.04562047668

We expected from theory that $\alpha \approx$ (element degree) +1.
Which is what we find. When the elements have degree 1 the alpha becomes 2 with L2 and H1 norm. And likewise for higher elements.

LAGRANGE ELEMENTS = 1
H1 norm
mu = 1.0 C = 0.298768477069 alpha = 0.998517422902
mu = 0.1 C = 4.01201969534 alpha = 0.861879498393
mu = 0.01 C = 15.4545207055 alpha = 0.378727137865

LAGRANGE ELEMENTS = 2
H1 norm

```
mu =  1.0  C =   0.0365363825599  alpha =   1.9838208629
mu =  0.1  C =   3.83850565709  alpha =   1.77422820512
mu =  0.01  C =   18.2184701683  alpha =   0.738632714997
```

When we changed to the H1 error norm we expected to get alpha the same as the degree of elements. This is what we get and like before it gets worse as we lower μ

d)

To compute the SUPG i just changed the test function to $v = v + beta * mesh.hmin() * v.dx(0)$. This method will smooth out our solution giving us a better solution when dealing with sharp corners. We see, from the table under, that the error is a 10^{-1} bigger than for Galerkin in the lower case. Putting beta lower gets us closer to Galerkin method so its expected that the error will be better for smaller beta.
SUPG can be a useful tool when we have a very sharp edge that we have a problem computing, since it smooths out the solution. Not giving a perfect answer but at least being able to compute over the domain.
We see from figure 3 that Galerkin with small μ gives oscillations and a bad solution. And figure 4 with SUPG gives a smoother and better solution.
To compute the C and alpha in the error estimates I implemented the SUPG errornorm (6.8) from the compendium. And was looking to get alpha = degree$*\frac{3}{2}$. I implemented the errornorm the following way:

```
ex_norm = assemble((u_exact.dx(0) - u.dx(0))**2*dx)
ey_norm = assemble((u_exact.dx(1) - u.dx(1))**2*dx)
e = sqrt(mesh.hmin()*ex_norm + mu*(ex_norm+ey_norm))
```

A print of this norm with varying μ and h

```
beta = h.min()
SUPG - norm
```

h/mu	1.0	0.1	0.01
2	0.231884809424	1.37374796004	1.48337689453
4	0.119830544311	1.04782853084	1.53689283747
8	0.0611456253743	0.667343103029	1.56955648944
16	0.0309330012785	0.385035688553	1.51741616324
32	0.0155652178542	0.209127121281	1.2126647182
64	0.00780854650798	0.109434586932	0.795300364213

```
mu =  1.0  C =   0.330129747596  alpha =   0.979369968414
mu =  0.1  C =   2.11805769892  alpha =   0.743376329069
mu =  0.01  C =   1.83477492663  alpha =   0.159166143481
```

```
beta = h.min()
L2 - norm
```

h/mu	1.0	0.1	0.01

```
2          0.0487459798451     0.365048142768   0.395142264119
4          0.0265760049303     0.286879143016   0.348635524567
8          0.0141685886099     0.192691162916   0.266450466069
16         0.0073698766431     0.114521156974   0.189592045188
32         0.00376694891544    0.0628907014983  0.125314777253
64         0.00190550308829    0.0330610795148  0.0769559179375

mu =  1.0   C =   0.0697885911129   alpha =   0.936689949943
mu =  0.1   C =   0.561676604788    alpha =   0.704104417791
mu =  0.01  C =   0.544386376465    alpha =   0.47773708412
```

I was expecting to see $\alpha = 1.5$ from the SUPG norm.
We can see from the L2 errornorm estimate that that our α is not where it was with Galerkin. This points to the fact that SUPG is not as accurate.

```
LAGRANGE ELEMENTS = 2
L2 NORM:
beta = h.min()*1.0

mu = 1e+00 , N = 8 Numerical Error: 1.2625e+00
mu = 1e+00 , N = 16 Numerical Error: 3.4611e-01
mu = 1e+00 , N = 32 Numerical Error: 3.3307e+00
mu = 1e+00 , N = 64 Numerical Error: 2.4616e+03

mu = 1e-01 , N = 8 Numerical Error: 2.4907e-01
mu = 1e-01 , N = 16 Numerical Error: 1.5145e-01
mu = 1e-01 , N = 32 Numerical Error: 5.2001e+00
mu = 1e-01 , N = 64 Numerical Error: 1.0097e+04

mu = 1e-02 , N = 8 Numerical Error: 2.7716e-01
mu = 1e-02 , N = 16 Numerical Error: 1.9793e-01
mu = 1e-02 , N = 32 Numerical Error: 1.3499e-01
mu = 1e-02 , N = 64 Numerical Error: 5.0506e+03
```

```
LAGRANGE ELEMENTS = 2
L2 NORM:
beta = h.min()*0.15

mu = 1e+00 , N = 8 Numerical Error: 3.0109e-02
mu = 1e+00 , N = 16 Numerical Error: 1.5232e-02
mu = 1e+00 , N = 32 Numerical Error: 7.6622e-03
mu = 1e+00 , N = 64 Numerical Error: 3.8428e-03

mu = 1e-01 , N = 8 Numerical Error: 5.7559e-02
mu = 1e-01 , N = 16 Numerical Error: 3.0749e-02
mu = 1e-01 , N = 32 Numerical Error: 1.5944e-02
mu = 1e-01 , N = 64 Numerical Error: 8.1258e-03
```

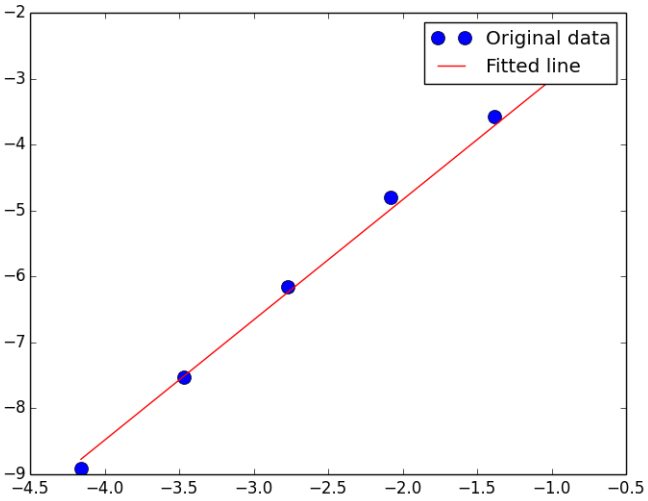
```
mu = 1e-02  , N = 8  Numerical Error: 7.8816e-02
mu = 1e-02  , N = 16  Numerical Error: 5.9183e-02
mu = 1e-02  , N = 32  Numerical Error: 3.8149e-02
mu = 1e-02  , N = 64  Numerical Error: 2.2061e-02
```

Code

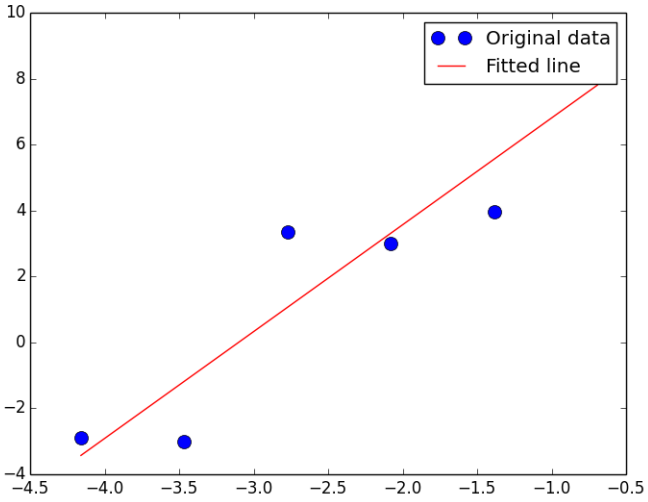
```
from dolfin import *
import numpy as np
import matplotlib.pyplot as plt
set_log_active(False)

def solver_1(N ,mu, beta):
    #print exp(1000)
    mu = Constant(mu)
    mesh = UnitSquareMesh(N, N)

    V = FunctionSpace(mesh, "CG", 1)
    V2 = FunctionSpace(mesh, "CG",2)

    class Right(SubDomain):
        def inside(self, x, on_boundary):
                return on_boundary and near(x[0],1)

    class Left(SubDomain):
        def inside(self, x, on_boundary):
                return on_boundary and near(x[0],0)


    right = Right()
    left = Left()
    bound = FacetFunction("size_t", mesh)
    bound.set_all(0)
    right.mark(bound, 1)
    left.mark(bound,2)
    #plot(bound); interactive()

    bc0 = DirichletBC(V, 1, right)
    bc1 = DirichletBC(V, 0, left)
    bcs = [bc0,bc1]

    u = TrialFunction(V)
    v = TestFunction(V)
    v = v + beta*mesh.hmin()*v.dx(0)
    f = Constant(0)

    u_exact = interpolate(Expression("(exp(x[0]/mu)-1)/(exp(1/mu)-1)",mu=mu),V2
    # sette 0 overalt utenom paa 1
    #print "norm of exact: ", norm(u_exact,"l2")
```

```
    #plot(u_exact); interactive()

    a = mu*inner(grad(u), grad(v))*dx + u.dx(0)*v*dx
    F = f*v*dx

    u_1 = Function(V)
    solve(a==F,u_1,bcs)
    #print project(u_exact - u_1,V)
    #print "mu = %.e , N = %.f Numerical Error: %.4e " %(mu,N,errornorm(u_exact

    #plot(interpolate(u_exact,V))
    #plot(u_1); interactive()
    #plt.show()
    #plt.savefig("Galerkin.png")
    return u_1,u_exact,V,mesh
#solver_1(64,0.0001,beta = 1.0)



N = [2,4,8,16,32,64]
mu = 0.01
b = np.zeros(len(N))
a = np.zeros(len(N))


for i in range(len(N)):
    u ,u_exact,V,mesh = solver_1(N[i],mu,beta=1.0)
    #ex_norm = assemble((u_exact.dx(0) - u.dx(0))**2*dx)
    #ey_norm = assemble((u_exact.dx(1) - u.dx(1))**2*dx)

    #e = sqrt(mesh.hmin()*ex_norm + mu*(ex_norm+ey_norm))
    e = errornorm(u_exact, u, norm_type = "l2",degree_rise = 3)
    print e
    #u_norm = norm(u_exact,"H1")
    #e = errornorm(u_exact,u,norm_type = "l2",degree_rise = 3)
    b[i] = np.log(e)#/u_norm)
    a[i] = np.log(mesh.hmin())#1./(N[i]))
    #print u_norm
    #u_norm = sum(u_ex*u_ex)**0.5
A = np.vstack([a,np.ones(len(a))]).T
alpha_1,c_1 = np.linalg.lstsq(A,b)[0]

print "mu = ",mu, "C = ", exp(c_1), "alpha = ", alpha_1

plt.plot(a,b,'o',label='Original data',markersize = 10)
plt.plot(a,alpha_1*a+c_1,'r',label='Fitted line')
plt.legend();plt.show()

"""
```

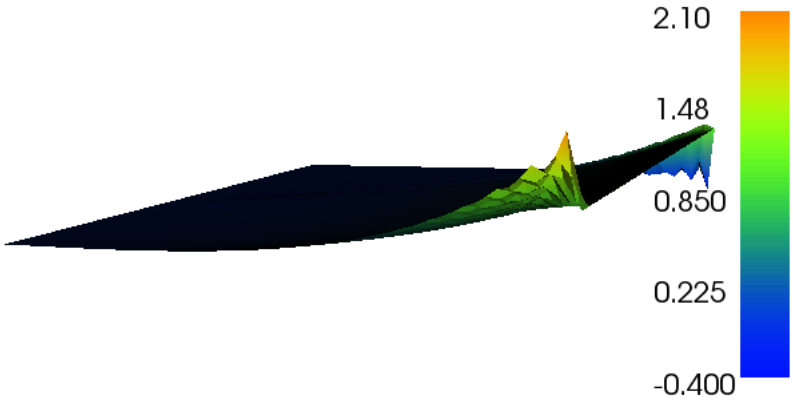
```
mu␣=[1,␣0.1␣,0.01,␣0.001];␣N␣=␣[8,16,32,64];
for␣m␣in␣mu:
␣␣␣␣for␣i␣in␣N:
␣␣␣␣␣␣␣␣solver_1(i,m,beta=0.15)"""
```



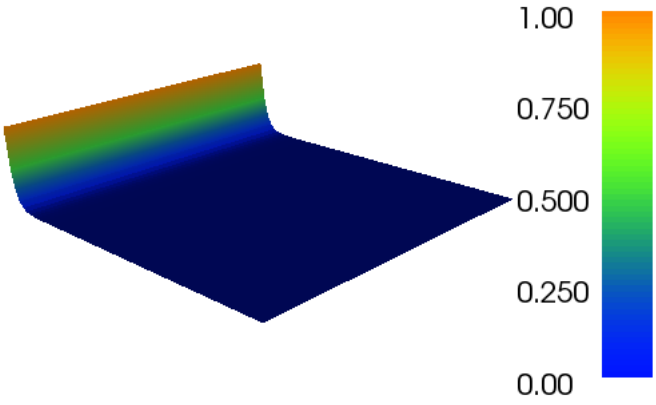
Figur 1: Least Squares, k = l = 1 , polynomial = 1



Figur 2: Least Squares, k = l = 100, polynomial = 1



Figur 3: Galerkin with $\mu = 0.0001$, N = 64



Figur 4: SUPG with $\mu = 0.0001$, N = 64