2.5 Mixed formulation for second order equations

Motivation:

- exact flux conservation
- useful for a posteriori error estimates
- model problem for 4^{th} order problems, Stokes, ...

We consider the diffusion equation

$$\begin{array}{rcll} -\operatorname{div} \lambda \nabla u & = & f & \text{in } \Omega \\ u & = & u_D & \text{on } \Gamma_D \\ \lambda \frac{\partial u}{\partial n} & = & g & \text{on } \Gamma_N \end{array}$$

Primal variational formulation

Find $u \in H^1, u = u_D$ on Γ_D such that

$$\int_\Omega \lambda \nabla u \nabla v = \int_\Omega f v + \int_{\Gamma_N} g v \quad \forall v \in H^1, v = 0 \text{ on } \Gamma_D$$

First order system

Find scalar u and the flux σ such that

$$\lambda^{-1} \sigma = \nabla u, \quad \operatorname{div} \sigma = -f$$

with boundary conditions

$$\sigma \cdot n = g \text{ on } \Gamma_N, \quad \text{and} \quad u = u_D \text{ on } \Gamma_D$$

Mixed variational formulation

Find $(\sigma, u) \in H(\operatorname{div}) \times L_2$ such that $\sigma_n = g$ on Γ_N and

$$\int_\Omega \lambda^{-1} \sigma \tau + \operatorname{div} \sigma v + \operatorname{div} \tau u = -\int_\Omega f v + \int_{\Gamma_D} u_D \tau_n$$

for all test-functions $(\tau, v) \in H(\operatorname{div}) \times L_2$ with $\tau_n = 0$.

Here σ_n is the normal trace operator $\sigma \cdot n|_{\Gamma_N}$, which is meaningful in $H(\operatorname{div})$.

In [1]:

```
from netgen.geom2d import unit_square
from ngsolve import *
from ngsolve.webgui import Draw

mesh = Mesh(unit_square.GenerateMesh(maxh=0.1))
```

```
importing NGSolve-6.2.2204
```

In [2]:
```
source = sin(3.14*x)
ud = CoefficientFunction(5)
g = CoefficientFunction([y*(1-y) if bc=="left" else 0 for bc in mesh.GetBou
lam = 10
```

Setup and solve primal problem:

In [3]:
```
fesp = H1(mesh, order=4, dirichlet="bottom")
up, vp = fesp.TnT()

ap = BilinearForm(fesp)
ap += lam*grad(up)*grad(vp)*dx
ap.Assemble()

fp = LinearForm(fesp)
fp += source*vp*dx + g*vp * ds
fp.Assemble()

gfup = GridFunction(fesp, "u-primal")
gfup.Set(ud, BND)

r = fp.vec.CreateVector()
r.data = fp.vec - ap.mat * gfup.vec
gfup.vec.data += ap.mat.Inverse(freedofs=fesp.FreeDofs()) * r
Draw (gfup)
Draw (lam * grad(gfup), mesh, "flux-primal")
```

```
WebGuiWidget(layout=Layout(height='50vh', width='100%'), value={'gui_settin
gs': {}, 'ngsolve_version': '6.2.22…
WebGuiWidget(layout=Layout(height='50vh', width='100%'), value={'gui_settin
gs': {}, 'ngsolve_version': '6.2.22…
```

Out[3]:
```
BaseWebGuiScene
```

Solving the mixed problem

Define spaces for mixed problem. Note that essential boundary conditions are set to the $H(\mathrm{div})$-component on the opposite boundary. Creating a space from a list of spaces generates a product space:

In [4]:
```
order_flux=1
V = HDiv(mesh, order=order_flux, dirichlet="right|top|left")
Q = L2(mesh, order=order_flux-1)
fesm = V*Q
```

The space provides now a list of trial and test-functions:

In [5]:
```
sigma, u = fesm.TrialFunction()
tau, v = fesm.TestFunction()
```

The normal vector is provided as a *special* coefficient function (which may be used only at the boundary). The orientation depends on the definition of the geometry. In 2D, it is the tangential vector rotated to the right, and is the outer vector in our case. Since every CoefficientFunction must know its vector-dimension, we have to provide the spatial dimension:

In [6]:

```
normal = specialcf.normal(mesh.dim)
print (normal)
```

```
coef normal vector, real, dim=2
```

Define the forms on the product space. For the boundary term, we have to use the Trace operator, which provides the projection to normal direction.

In [7]:
```
am = BilinearForm(fesm)
am += (1/lam*sigma*tau + div(sigma)*v + div(tau)*u)*dx
am.Assemble()
fm = LinearForm(fesm)
fm += -source*v*dx +  ud*(tau.Trace()*normal)*ds
fm.Assemble()

gfm = GridFunction(fesm, name="gfmixed")
```

The proxy-functions used for the forms know to which component of the product space they belong to. To access components of the solution, we have to unpack its components. They don't have own coefficient vectors, but refer to ranges of the big coefficient vector.

In [8]:
```
gfsigma, gfu = gfm.components
```

Just to try something:

In [9]:
```
gfsigma.Set(CoefficientFunction( (sin(10*x), sin(10*y)) ))
gfu.Set (x)
Draw (gfsigma, mesh, "flux-mixed")
Draw (gfu, mesh, "u-mixed")
```

```
WebGuiWidget(layout=Layout(height='50vh', width='100%'), value={'gui_settin
gs': {}, 'ngsolve_version': '6.2.22…
WebGuiWidget(layout=Layout(height='50vh', width='100%'), value={'gui_settin
gs': {}, 'ngsolve_version': '6.2.22…
```

Out[9]:
```
BaseWebGuiScene
```

Now set the essential boundary conditions for the flux part:

In [10]:
```
gfsigma.Set(g*normal, BND)
```

In [12]:
```
 # fm.vec.data -= am.mat * gfm.vec
 # gfm.vec.data += am.mat.Inverse(freedofs=fesm.FreeDofs(), inverse="umfpac
solvers.BVP(bf=am, lf=fm, gf=gfm)
Draw (gfsigma, mesh, "flux-mixed")
Draw (gfu, mesh, "u-mixed")
```

```
---------------------------------------------------------------------------
NgException                               Traceback (most recent call last)
/tmp/ipykernel_124869/747807551.py in <module>
      1 fm.vec.data -= am.mat * gfm.vec
----> 2 gfm.vec.data += am.mat.Inverse(freedofs=fesm.FreeDofs(), inverse="umfpack") * fm.vec
      3 solvers.BVP(bf=am, lf=fm, gf=gfm)
      4 Draw (gfsigma, mesh, "flux-mixed")
      5 Draw (gfu, mesh, "u-mixed")

NgException: SparseMatrix::InverseMatrix:  UmfpackInverse not available
```

Calculate the difference:

In [ ]:

```
print ("err-u:   ", sqrt(Integrate( (gfup-gfu)**2, mesh)))
errflux = lam * grad(gfup) - gfsigma
print ("err-flux:", sqrt(Integrate(errflux*errflux, mesh)))
```

Post-processing for the scalar variable

The scalar variable is approximated one order lower than the vector variable, what is its gradient. Knowing the gradient of a function more accurate, and knowing its mean value, one can recover the function itself. For this post-processing trick we refer to [Arnold+Brezzi 85]

find $\widehat u \in P^{k+1,dc}$ and $\widehat \lambda \in P^{0,dc}$ such that

$$\begin{array}{ccccll}
\int \lambda \nabla \widehat u \nabla \hat v & + & \int \widehat \lambda \hat v & = & \int \sigma \nabla \hat v & \forall \hat v \\
\int \widehat u \widehat \mu & & & = & \int u \widehat \mu & \forall \widehat \mu
\end{array}$$

In [ ]:

```
fespost_u = L2(mesh, order=order_flux+1)
fespost_lam = L2(mesh, order=0)
fes_post = fespost_u*fespost_lam

u,la = fes_post.TrialFunction()
v,mu = fes_post.TestFunction()

a = BilinearForm(fes_post)
a += (lam*grad(u)*grad(v)+la*v+mu*u)*dx
a.Assemble()
f = LinearForm(fes_post)
f += (gfsigma*grad(v)+gfu*mu)*dx
f.Assemble()

gfpost = GridFunction(fes_post)
gfpost.vec.data = a.mat.Inverse() * f.vec

Draw (gfpost.components[0], mesh, "upost")
print ("err-upost:   ", sqrt(Integrate( (gfup-gfpost.components[0])**2, mes
```

In [ ]: