staticcond

November 1, 2022

1 1.4 Static Condensation

Static condensation refers to the process of eliminating unknowns that are internal to elements from the global linear system. They are useful in standard methods and critical in methods like the HDG method. NGSolve automates static condensation across a variety of methods via a classification of degrees of freedom.

In this tutorial, you will learn

- how to turn on static condensation (`eliminate_internal` or `condense`),
- how to get the full solution from the condensed system using data members
  - `harmonic_extension_trans`
  - `harmonic_extension`
  - `inner_solve`,
- their relationship to a Schur complement factorization,
- types of degrees of freedom such as `COUPLING_TYPE.LOCAL_DOF`, and
- an automatic utility for solving via static condensation.

[ ]:
```
from ngsolve import *
from ngsolve.webgui import Draw
from netgen.geom2d import unit_square

mesh = Mesh(unit_square.GenerateMesh(maxh=0.4))

fes = H1(mesh, order=4, dirichlet='bottom|right')
u, v = fes.TnT()
```

1.0.1 Asking `BilinearForm` to `condense`

Keyword arguments `condense` and `eliminate_internal` are synonymous.

[ ]:
```
a = BilinearForm(fes, condense=True)
a += grad(u) * grad(v) * dx
a.Assemble()

f = LinearForm(fes)
```

```
f += 1 * v * dx
f.Assemble()

u = GridFunction(fes)
```

- The assembled matrix $A =$ `a.mat` can be block partitioned into

$$A = \begin{pmatrix} A_{LL} & A_{LE} \\ A_{EL} & A_{EE} \end{pmatrix}$$

where L denotes the set of **local** (or internal) degrees of freedom and E denotes the set of **interface** degrees of freedom.

- In our current example E consists of edge and vertex dofs while L consists of triangle dofs. (Note that in practice, L and E may not be ordered contiguously and L need not appear before E, but such details are immaterial for our discussion here.)
- The condensed system is known as the **Schur complement**:

$$S = A_{EE} - A_{EL}A_{LL}^{-1}A_{LE}.$$

- When `eliminate_internal` (or `condense`) is set to `True` in `a`, the statement `a.Assemble` actually assembles S.

1.0.2 A factorization of the inverse

- NGSolve provides
  - `a.harmonic_extension_trans` $= \begin{pmatrix} 0 & 0 \\ -A_{EL}A_{LL}^{-1} & 0 \end{pmatrix}$
  - `a.harmonic_extension` $= \begin{pmatrix} 0 & -A_{LL}^{-1}A_{LE} \\ 0 & 0 \end{pmatrix}$
  - `a.inner_solve` $= \begin{pmatrix} A_{LL}^{-1} & 0 \\ 0 & 0 \end{pmatrix}$.
- To solve

$$\begin{pmatrix} A_{LL} & A_{LE} \\ A_{EL} & A_{EE} \end{pmatrix} \begin{pmatrix} u_L \\ u_E \end{pmatrix} = \begin{pmatrix} f_L \\ f_E \end{pmatrix}$$

we use a factorization of A^{-1} that uses S^{-1}. Namely, we use the following identity:

$$\begin{pmatrix} u_L \\ u_E \end{pmatrix} = \begin{pmatrix} I & -A_{LL}^{-1}A_{LE} \\ 0 & I \end{pmatrix} \begin{pmatrix} A_{LL}^{-1} & 0 \\ 0 & S^{-1} \end{pmatrix} \underbrace{\begin{pmatrix} I & 0 \\ -A_{EL}A_{LL}^{-1} & I \end{pmatrix} \begin{pmatrix} f_L \\ f_E \end{pmatrix}}_{\begin{pmatrix} f'_L \\ f'_E \end{pmatrix}}$$

We implement this formula step by step, starting with the computation of f'_L and f'_E.

1.0.3 Full solution from the condensed system

- The following step implements

$$\begin{pmatrix} f'_L \\ f'_E \end{pmatrix} = \begin{pmatrix} I & 0 \\ -A_{EL}A_{LL}^{-1} & I \end{pmatrix} \begin{pmatrix} f_L \\ f_E \end{pmatrix}.$$

[ ]:
```
f.vec.data += a.harmonic_extension_trans * f.vec
```

- The next step implements part of the next matrix application in the formula.

$$\begin{pmatrix} 0 \\ u_E \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & S^{-1} \end{pmatrix} \begin{pmatrix} f'_L \\ f'_E \end{pmatrix}.$$

[ ]:
```
u.vec.data = a.mat.Inverse(freedofs=fes.FreeDofs(coupling=True)) * f.vec
```

- Note:
  - Because `a` has `condense` set, the inverse `a.mat.Inverse` actually computes S^{-1}.
  - Note that instead of the usual `fes.FreeDofs()`, we have used `fes.FreeDofs(coupling=True)` or simply `fes.FreeDofs(True)` to specify that only the degrees of freedom that are *not local* and *not Dirichlet* should participate in the inverse computations. (The underlying assumption is that Dirichlet dofs cannot be local dofs.)
- Next, we compute

$$\begin{pmatrix} u'_L \\ u_E \end{pmatrix} = \begin{pmatrix} 0 \\ u_E \end{pmatrix} + \begin{pmatrix} A_{LL}^{-1} & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} f'_L \\ f'_E \end{pmatrix}.$$

[ ]:
```
u.vec.data += a.inner_solve * f.vec
```

- Finally:

$$\begin{pmatrix} u_L \\ u_E \end{pmatrix} = \begin{pmatrix} I & -A_{LL}^{-1}A_{LE} \\ 0 & I \end{pmatrix} \begin{pmatrix} u'_L \\ u_E \end{pmatrix}$$

[ ]:
```
u.vec.data += a.harmonic_extension * u.vec
Draw(u)
```

1.0.4 Behind the scenes: `CouplingType`

How does NGSolve know what is in the index sets L and E?

- Look at `fes.CouplingType` to see a classification of degrees of freedom (dofs).

[ ]:
```
for i in range(fes.ndof):
    print(fes.CouplingType(i))
```

[ ]:
```
dof_types = {}
for i in range(fes.ndof):
    ctype = fes.CouplingType(i)
    if ctype in dof_types.keys():
```

```
        dof_types[ctype] += 1
    else:
        dof_types[ctype] = 1
dof_types
```

The `LOCAL_DOF` forms the set L and the remainder forms the set E. All finite element spaces in NGSolve have such dof classification.

Through this classification a bilinear form is able to automatically compute the Schur complement and the accompanying extension operators. Users need only specify the `eliminate_internal` option. (Of course users should also make sure their method has an invertible A_{LL}!)

1.0.5 Inhomogeneous Dirichlet boundary conditions

In case of inhomogeneous Dirichlet boundary conditions, we combine the technique of Dirichlet data extension in 1.3 with the above static condensation principle in the following code.

[ ]:
```
U = x*x*(1-y)*(1-y)           # U = manufactured solution
DeltaU = 2*((1-y)*(1-y)+x*x) # Source: DeltaU =  U

f = LinearForm(fes)
f += -DeltaU * v * dx
f.Assemble()

u = GridFunction(fes)
u.Set(U, BND)                # Dirichlet b.c: u = U on boundary

r = f.vec.CreateVector()
r.data = f.vec - a.mat * u.vec
r.data += a.harmonic_extension_trans * r

u.vec.data += a.mat.Inverse(fes.FreeDofs(True)) * r
u.vec.data += a.harmonic_extension * u.vec
u.vec.data += a.inner_solve * r

sqrt(Integrate((U-u)*(U-u),mesh))  # Compute error
```

If you find the above steps onerous, you can again use the automatic utility `solvers.BVP`, which will perform static condensation if `condense` is turned on in its input bilinear form.

Automatic utility `BVP`

[ ]:
```
u = GridFunction(fes)
u.Set(U, BND)

c = Preconditioner(a,"direct")
c.Update()
solvers.BVP(bf=a, lf=f, gf=u, pre=c)
sqrt(Integrate((U-u)*(U-u),mesh))
```