dirichlet

November 1, 2022

1 1.3 Dirichlet boundary conditions

This tutorial goes in depth into the mechanisms required to solve the Dirichlet problem

$$-\Delta u = f \quad \text{in } \Omega,$$

with a **nonzero** Dirichlet boundary condition

$$u|_{\Gamma_D} = g \quad \text{on a boundary part } \Gamma_D \subset \partial\Omega.$$

The same mechanisms are used in solving boundary value problems involving operators other than the Laplacian.

You will see how to perform these tasks in NGSolve:

- extend Dirichlet data from boundary parts,
- convert boundary data into a volume source,
- reduce inhomogeneous Dirichlet case to the homogeneous case, and
- perform all these tasks automatically within a utility.

If you are just interested in the *automatic utility,* then please skip to the last part of the tutorial.

1.0.1 Spaces with Dirichlet conditions on part of the boundary

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

mesh = Mesh(unit_square.GenerateMesh(maxh=0.2))
mesh.GetBoundaries()
```

The `unit_square` has its boundaries marked as `left`, `right`, `top` and `bottom`. Suppose we want non-homogeneous Dirichlet boundary conditions on

$$\Gamma_D = \Gamma_{left} \cup \Gamma_{right}.$$

Then, we set the space as follows:

[ ]:
```
fes = H1(mesh, order=2, dirichlet="left|right")
```

Compare this space with the one without the `dirichlet` flag:

[ ]:
```
fs2 = H1(mesh, order=2)
fes.ndof, fs2.ndof    # total number of unknowns
```

Thus, the `dirichlet` flag did not change `ndof`. In NGSolve the unknowns are split into two groups: * dirichlet dofs (or constrained dofs), * free dofs.

The facility `FreeDofs` gives a `BitArray` such that `FreeDofs[dof]` is True if and only if `dof` is a free degree of freedom.

[ ]:
```
print("free dofs of fs2 without \"dirichlet\" flag:\n",
      fs2.FreeDofs())
print("free dofs of fes:\n", fes.FreeDofs())
```

- The space `fs2` without `dirichlet` flag has only free dofs (no dirichlet dofs).
- The other space `fes` has a few dofs that are marked as *not* free. These are the dofs that are located on the boundary regions we marked as `dirichlet`.

1.0.2 Extension of boundary data

We use the standard technique of reducing a problem with essential non-homogeneous boundary conditions to one with homogeneous boundary condition using an extension. The solution u in H^1 satisfies

$$u|_{\Gamma_D} = g$$

and

$$\int_\Omega \nabla u \cdot \nabla v_0 = \int_\Omega f v_0$$

for all v_0 in $\in H^1_{0,D} = \{v \in H^1 : v|_{\Gamma_D} = 0\}$. Split the solution

$$u = u_0 + u_D$$

where u_D is an extension of g into Ω. Then we only need to find u_0 in $H^1_{0,D}$ satisfying the homogeneous Dirichlet problem

$$\int_\Omega \nabla u_0 \cdot \nabla v_0 = \int_\Omega f v_0 - \int_\Omega \nabla u_D \cdot \nabla v_0$$

for all v_0 in $H^1_{0,D}$. These are the **issues to consider** in this approach:

- How to define an extension u_D in the finite element space?
- How to form and solve the system for u_0?

Let us address the first in the following example.

Suppose we are given that

$$g = \sin(y) \qquad \text{on } \Gamma_D.$$

[ ]:
```
g = sin(y)
```

We interpolate g on the boundary of the domain and extend it to zero on the elements not having an intersection with Γ_D.

[ ]:
```
gfu = GridFunction(fes)
gfu.Set(g, BND)
Draw(gfu)
```

The keyword `BND` tells `Set` that `g` need only be interpolated on those parts of the boundary that are marked `dirichlet`.

Thus, `gfu` now contains the extension u_D. Next, we turn to set up the system for u_0.

1.0.3 Forms and assembly

In NGSolve, bilinear and linear forms are defined independently of the dirichlet flags. Matrices and vectors are set up with respect to all unknowns (free or constrained) so they may be restricted to any group of unknowns later.

[ ]:
```
u, v = fes.TnT()

a = BilinearForm(fes, symmetric=True)
a += grad(u)*grad(v)*dx
a.Assemble()
```

If $A =$ `a.mat` is the matrix just assembled, then we want to solve for

$$A(u_0 + u_D) = f \quad \Rightarrow \quad Au_0 = f - Au_D$$

or

$$\begin{pmatrix} A_{FF} & A_{FD} \\ A_{DF} & A_{DD} \end{pmatrix} \begin{pmatrix} u_{0,F} \\ 0 \end{pmatrix} = \begin{pmatrix} f_F \\ f_D \end{pmatrix} - \begin{pmatrix} A_{FF} & A_{FD} \\ A_{DF} & A_{DD} \end{pmatrix} \begin{pmatrix} u_{D,F} \\ u_{D,D} \end{pmatrix}$$

where we have block partitioned using free dofs (F) and dirichlet dofs (D) as if they were numbered consecutively (which may not be the case in practice) for ease of presentation. The first row gives

$$A_{FF}u_{0,F} = f_F - [Au_D]_F.$$

Since we have already constructed u_D, we need to perform these next steps:

- Set up the right hand side from f and u_D.
- Solve a linear system which involves only A_{FF}.

- Add solution: $u = u_0 + u_D$.

1.0.4 Solve for the free dofs

We need to assemble the right hand side of $A_{FF}u_{0,F} = f_F - [Au_D]_F$, namely

$$r = f - Au_D.$$

[ ]:
```
f = LinearForm(fes)
f += 1*v*dx
f.Assemble()

r = f.vec.CreateVector()
r.data = f.vec - a.mat * gfu.vec
```

The implementation of

$$u = u_D + \begin{pmatrix} A_{FF}^{-1} & 0 \\ 0 & 0 \end{pmatrix} r$$

by sparse solvers is achieved by the following:

[ ]:
```
gfu.vec.data += a.mat.Inverse(freedofs=fes.FreeDofs()) * r
Draw(gfu)
```

1.0.5 The automatic utility `BVP`

NGSolve also provides a `BVP` facility in the `solvers` submodule, within which the above steps are performed automatically. You provide A, f, a grid function `gfu` with your boundary condition g, and a preconditioner. Then `BVP` solves the problem with non-homogeneous Dirichlet boundary condition and overwrites `gfu` with the solution.

[ ]:
```
gfu.Set(g, BND)
c = Preconditioner(a,"local")   #<- Jacobi preconditioner
#c = Preconditioner(a,"direct") #<- sparse direct solver
c.Update()
solvers.BVP(bf=a, lf=f, gf=gfu, pre=c)
Draw(gfu)
```

[ ]:
```

```