subdomains

November 1, 2022

1 1.5 Spaces and forms on subdomains

In NGSolve, finite element spaces can be defined on subdomains. This is useful for multiphysics problems like fluid-structure interaction.

In addition, bilinear or linear forms can be defined as integrals over regions. *Regions* are parts of the domain. They may be subdomains, or parts of the domain boundary, or parts of subdomain interfaces.

In this tutorial, you will learn about

- defining finite element spaces on `Regions`,
- `Compress`ing such spaces,
- integrating on `Regions`s, and
- set operations on `Region` objects via `Mask`.

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

1.0.1 Naming subdomains and boundaries

We define a geometry with multiple regions and assign names to these regions below:

[ ]:
```
geo = SplineGeometry()
geo.AddRectangle((0,0), (2,2),
                 bcs=["b","r","t","l"],
                 leftdomain=1)
geo.AddRectangle((1,0.9), (1.3,1.4),
                 bcs=["b2","r2","t2","l2"],
                 leftdomain=2, rightdomain=1)
geo.SetMaterial (1, "outer")
geo.SetMaterial (2, "inner")
mesh = Mesh(geo.GenerateMesh(maxh=0.1))
```

- These statements define two rectangular subdomain regions named “inner”, “outer”. (Note how the ordering in `SetMaterial` is 1-based, not 0-based.)

- The bottom, right, top and left parts of the outer rectangle’s boundaries define boundary regions, respectively labeled “b”, “r”, “t”, “l”.

- Similarly, the bottom, right, top and left parts of the inner rectangle's boundaries are regions named "b2", "r2", "t2", "l2".
- When you select the `Mesh` tab in the Netgen window's Graphical User Interface (GUI) and double click on a point, the two subdomains are rendered in different colors. (You should also see the subdomain outlines in the Netgen window when you select the `Geometry` tab.)

1.0.2 A finite element space on a subdomain

[ ]:
```
fes1 = H1(mesh, definedon="inner")
u1 = GridFunction(fes1, "u1")
u1.Set(x*y)
Draw(u1)
```

Note how u_1 is displayed only in the `inner` region in the Netgen GUI.

Do not be confused with `ndof` for spaces on subdomains:

[ ]:
```
fes = H1(mesh)
fes1.ndof, fes.ndof
```

Although `ndof`s are the same for spaces on one subdomain and the whole domain, `FreeDofs` show that the real number of degrees of freedom for the subdomain space is much smaller:

[ ]:
```
print(fes1.FreeDofs())
```

One can also glean this information by examining `CouplingType` of each degrees of freedom of the space, which reveals that there are many unknowns of type `COUPLING_TYPE.UNUSED_DOF`.

[ ]:
```
for i in range(fes1.ndof):
    print(fes1.CouplingType(i))
```

It is possible to remove these dofs (thus making `GridFunction` vectors smaller) as follows.

[ ]:
```
fescomp = Compress(fes1)
print(fescomp.ndof)
```

1.0.3 Integrating on regions

You have already seen how boundary regions are used in setting Dirichlet boundary conditions.

[ ]:
```
fes = H1(mesh, order=3, dirichlet="b|l|r")

u, v = fes.TnT()
gfu = GridFunction(fes)
```

Such boundary regions or subdomains can also serve as domains of integration.

[ ]:
```
f = LinearForm(fes)
f += u1 * v * dx(definedon=mesh.Materials("inner"))
```

```
f += 0.1 * v * ds(definedon=mesh.Boundaries("t"))
f.Assemble()
```

Here the functional f is defined as a sum of two integrals, one over the inner subdomain, and another over the top boundary:

$$f(v) = \int_{\Omega_{inner}} u_1 v\, dx + \int_{\Gamma_{top}} \frac{v}{10}\, ds$$

The python objects `dx` and `ds` represent volume and boundary integration, respectively. They admit a keyword argument `definedon` that is **either** a `Region` **or** a `str`, as can be seen from the documentation: look for `definedon: Optional[Union[ngsolve.comp.Region, str]]` in the help output below.

```
[ ]: help(dx)
```

Thus - `dx(definedon=mesh.Materials("inner"))` may be replaced by `dx('inner')`,

and similarly,

- `ds(definedon=mesh.Boundaries("t")` may be replaced by `ds('t')`.

```
[ ]: f = LinearForm(fes)
     f += u1*v*dx('inner') + 0.1*v*ds('t')
     f.Assemble()
```

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

     # Solve the problem:
     gfu.vec.data = a.mat.Inverse(fes.FreeDofs()) * f.vec
     Draw (gfu)
```

1.0.4 More about Region objects

```
[ ]: mesh.GetMaterials()     # list all subdomains
```

```
[ ]: mesh.GetBoundaries()    # list boundary/interface regions
```

```
[ ]: mesh.Materials("inner") # look at object's type
```

```
[ ]: mesh.Boundaries("t")     # same type!
```

1.0.5 Operations with Regions

Print region information:

[ ]:
```
print(mesh.Materials("inner").Mask())
print(mesh.Materials("[a-z]*").Mask())  # can use regexp
print(mesh.Boundaries('t|l').Mask())
```

Add regions:

[ ]:
```
io = mesh.Materials("inner") + mesh.Materials("outer")
print(io.Mask())
```

Take complement of a region:

[ ]:
```
c = ~mesh.Materials("inner")
print(c.Mask())
```

Subtract regions:

[ ]:
```
diff = mesh.Materials("inner|outer") - mesh.Materials("outer")
print(diff.Mask())
```

Set a piecewise constant `CoefficientFunction` using the subdomains:

[ ]:
```
domain_values = {'inner': 3.7,  'outer': 1}
values_list = [domain_values[mat]
               for mat in mesh.GetMaterials()]
cf = CoefficientFunction(values_list)
Draw(cf, mesh, 'piecewise')
```

Coefficients on boundary regions are given similarly using `mesh.GetBoundaries()`.

1.0.6 Extra exercises to try out

Let's make a linear function that equals 2 at the bottom right vertex of current domain (0,2) x (0,2) and equals zero at the remaining vertices.

[ ]:
```
bdry_values = {'b': x, 'r': 2-y}
values_list = [bdry_values[bc]
               if bc in bdry_values.keys() else 0
               for bc in mesh.GetBoundaries()]
cf = CoefficientFunction(values_list)
Draw(cf, mesh, 'piecewise')
```

Pitfall! Look at the GUI output. This is not the function we wanted!

What happened here? The object `cf` has no information on boundary regions, so it cannot associate the list of values to boundary regions. By default, the list of values are assumed to be subdomain values.

To associate these values to boundary regions, we use a `GridFunction` and `Set`, which also lets us view an extension of these boundary values into the domain.

[ ]:
```
g = GridFunction(H1(mesh), name='bdry')
g.Set(cf, definedon=~mesh.Boundaries(''))
Draw(g, max=2, min=0)
```

If you think that specifying the whole boundary using

`~mesh.Boundaries('')`

is convoluted, then you may use

`g.Set(cf, definedon=mesh.Boundaries('b|r')).`

[ ]:
```
g.Set(cf, definedon=mesh.Boundaries('b|r'))
Draw(g, max=2, min=0)
```

What happens if you `Set` using `b` and then on `r`?

[ ]:
```
g.Set(cf, definedon=mesh.Boundaries('b'))
g.Set(cf, definedon=mesh.Boundaries('r'))
Draw(g, max=2, min=0)
```

Pitfall! Do you see an unexpected behavior in the GUI?

If so, realize that each `Set` begins by zeroing the grid function - so any previous values are lost!

What happens if you use

`g.Set(cf, BND)`

instead?

Pitfall!

[ ]:
```
g = GridFunction(H1(mesh))
g.Set(cf, BND)
Draw(g, max=2, min=0)
```

We do not obtain the previous result because the keyword `BND` directs `Set` to only set the data in boundary regions marked as `dirichlet`.

[ ]:
```
g = GridFunction(H1(mesh, dirichlet='b|r'))
g.Set(cf, BND)
Draw(g, max=2, min=0)
```