coefficientfunction

November 1, 2022

1 1.2 CoefficientFunctions

In NGSolve, `CoefficientFunctions` are representations of functions defined on the computational domain. Examples are expressions of coordinate variables x, y, z and functions that are constant on subdomains. Much of the magic behind the seamless integration of NGSolve with python lies in `CoefficientFunctions`. This tutorial introduces you to them.

After this tutorial you will know how to

- **define** a `CoefficientFunction`,
- **visualize** a `CoefficientFunction`,
- **evaluate** `CoefficientFunctions` at points,
- print the **expression tree** of `CoefficientFunction`,
- **integrate** a `CoefficientFunction`,
- **differentiate** a `CoefficientFunction`,
- include **parameter** in `CoefficientFunctions`,
- **interpolate** a `CoefficientFunction` into a finite element space,
- define **vector-valued** `CoefficientFunctions`, and
- **compile** `CoefficientFunctions`.

[ ]:
```
from ngsolve import *
from ngsolve.webgui import Draw
from netgen.geom2d import unit_square
import matplotlib.pyplot as plt
mesh = Mesh (unit_square.GenerateMesh(maxh=0.2))
```

1.0.1 Define a function

[ ]:
```
myfunc = x*(1-x)
myfunc   # You have just created a CoefficientFunction
```

[ ]:
```
x         # This is a built-in CoefficientFunction
```

1.0.2 Visualize the function

Use the `mesh` to visualize the function in Netgen's GUI.

[ ]:
```
Draw(myfunc, mesh, "firstfun");
```

1.0.3 Evaluate the function

[ ]:
```
mip = mesh(0.2, 0.2)
myfunc(mip)
```

Note that `myfunc(0.2,0.3)` will not evaluate the function: one must give points in the form of `MappedIntegrationPoints` like `mip` above. The `mesh` knows how to produce them.

1.0.4 Examining functions on sets of points

[ ]:
```
pts = [(0.1*i, 0.2) for i in range(10)]
vals = [myfunc(mesh(*p)) for p in pts]
for p,v in zip(pts, vals):
    print("point=(%3.2f,%3.2f), value=%6.5f"
          %(p[0], p[1], v))
```

We may plot the restriction of the `CoefficientFunction` on a line using matplotlib.

[ ]:
```
px = [0.01*i for i in range(100)]
vals = [myfunc(mesh(p,0.5)) for p in px]
plt.plot(px,vals)
plt.xlabel('x')
plt.show()
```

1.0.5 Expression tree of a function

Internally, coefficient functions are implemented as an expression tree made from building blocks like `x`, `y`, `sin`, etc., and arithmetic operations.

E.g., the expression tree for `myfunc = x*(1-x)` looks like this:

[ ]:
```
print(myfunc)
```

1.0.6 Integrate a function

We can numerically integrate the function using the mesh:

[ ]:
```
Integrate(myfunc, mesh)
```

You can change the precision of the quadrature rule used for the integration using the key word argument `order`.

1.0.7 Differentiate a function

Automatic differentiation of a `CoefficientFunction` is now possible through the `Diff` method. Here is how you get $\partial/\partial x$ of `myfunc`:

[ ]:
```
diff_myfunc = myfunc.Diff(x)
Draw(diff_myfunc, mesh, "derivative");
```

See if you can recognize an implementation of the product rule

$$\frac{\partial}{\partial x} x(1-x) = \frac{\partial x}{\partial x}(1-x) + x\frac{\partial(1-x)}{\partial x}$$

in the tree-representation of the differentiated coefficient function, printed below.

[ ]:
```
print(diff_myfunc)
```

1.0.8 Parameters in functions

When building complex coefficient functions from simple ones like `x` and `y`, you may often want to introduce `Parameter`s, which are constants whose value may be changed later.

[ ]:
```
k = Parameter(1.0)
f = sin(k*y)
Draw(f, mesh, "f");
```

The same `f` may be given a different value of `k` later:

[ ]:
```
k.Set(10)
Draw(f, mesh, "f");
```

Look at the expression tree of `f`:

[ ]:
```
print(f)
```

Note how the `Parameter` is now a **node** in the expression tree. You can differentiate a coefficient function with respect to such quantities by passing it as argument to `Diff`:

[ ]:
```
print (f.Diff(k))
```

[ ]:
```
Integrate((f.Diff(k) - y*cos(k*y))**2, mesh)
```

1.0.9 Interpolate a function

We may `Set` a `GridFunction` using a `CoefficientFunction`:

[ ]:
```
fes = H1(mesh, order=1)
u = GridFunction(fes)
u.Set(myfunc)
Draw(u);
```

- The `Set` method interpolates `myfunc` to an element `u` in the finite element space.
- `Set` does an *Oswald-type interpolation* as follows:
  - It first zeros the grid function;
  - It then projects `myfunc` in L^2 on each mesh element;
  - It then averages dofs on element interfaces for conformity.

- We will see other ways to interpolate in 2.10.

1.0.10 Vector-valued `CoefficientFunction`

Here is an example of a vector-valued coefficient function.

[ ]:
```
vecfun = CoefficientFunction((-y, sin(x)))
Draw(vecfun, mesh, "vecfun");
```

Click on `Draw Surface Vectors` in the `Visual` menu to visualize this vector field.

Another example of a vector-valued coefficient function is the gradient of the above-set `GridFunction`.

[ ]:
```
u.Set(myfunc)
gradu = grad(u)
Draw(gradu, mesh, "grad_firstfun");
```

1.0.11 Compiled `CoefficientFunction`

Evaluation of a `CoefficientFunction` at a point is usually done by traversing its expression tree and evaluating each node of the tree. When the tree has repeated nodes, this is likely wasteful. NGSolve allows you to **"compile"** a `CoefficientFunction` to increase the efficiency of its evaluation. The compilation translates the expression tree into a sequence of linear steps.

Continuing with our simple `myfunc` example, here is how to use the `Compile` method:

[ ]:
```
myfunc_compiled = myfunc.Compile()
```

Now look at the differences between the compiled and non-compiled `CoefficientFunction`:

[ ]:
```
print(myfunc)
```

[ ]:
```
print(myfunc_compiled)
```

Evaluation of the compiled function is now a linear sequence of Steps 0, 1, 2, and 3 above. We understand the printed description of Steps 2 and 3 above to mean the following.

- Step 2: (Output of Step 1) - (Output of Step 0)
- Step 3: (Output of Step 0) * (Output of Step 2)

Here is another example, along with differences in timings for integrating the coefficient function on the mesh.

[ ]:
```
f0 = myfunc
f1 = f0*y
f2 = f1*f1 + f1*f0 + f0*f0
f3 = f2*f2*f2*f0**2 + f0*f2**2 + f0**2 + f1**2 + f2**2
final = f3 + f3 + f3
finalc = final.Compile()
```

[ ]:
```
%timeit Integrate(final, mesh, order=10)
```

[ ]:
```
%timeit Integrate(finalc, mesh, order=10)
```

If your NGSolve installation has a compiler script (you likely do if you built from source using a compiler), then you have the option of letting that compiler optimize your coefficient function further. Here is an example:

[ ]:
```
finalcc = final.Compile(realcompile=True, wait=True)
```

[ ]:
```
%timeit Integrate(finalcc, mesh, order=10)
```

[ ]:
```
print(finalc)
```

[ ]:
```
print(final)
```

[ ]:
```

```