Using Hermes in NCLab

Hermes is a widely used open-source C++ library for rapid development of adaptive *hp*-FEM and *hp*-DG solvers. In NCLab it is available via its Python wrappers. The wrappers follow the original C++ functionality very closely, so after understanding basic naming conventions, you should be able to create your own examples by looking at the original C++ tutorial examples which are available at `http://hpfem.org/hermes/`.

Source code of Python wrappers

If you are not sure how some Hermes functionality is used in Python, you can look it up easily. The wrappers are open source and they are available at

`https://github.com/hpfem/hermes-python.git`.

The folder and file structure of this repository is identical to the structure of the C++ library which is located at

`https://github.com/hpfem/hermes.git`.

In case that we missed to wrap something that you need, let us know and we will fix it.

Naming conventions

The following basic rules apply:

- C++ class `X` is wrapped as `PyX`.
- C++ template `T<double>` is wrapped as `PyTReal`.
- C++ template `T<complex>` is wrapped as `PyTComplex`.
- Names of methods are identical.
- Arguments of methods are identical.

Displayed tutorial examples in NCLab

Several examples from the original Hermes tutorial are available as displayed projects. To clone them, launch File Manager, and click on Project → Clone. In the table that appears, look for projects whose names start with "Hermes - Tutorial - Example ..."

Using GE and ME with Hermes

The original tutorial examples come with their own predefined meshes. However, it is easy to create custom 2D domains and meshes using the Geometry Editor (GE) and Mesh Editor (ME) and pass them to Hermes – or to your finite element program for that matter. To see how this is done, clone the displayed projects whose names start with "GE and ME".

Hermes - Tutorial - Example A03

Let us describe in more detail the Python version of the Hermes tutorial example A-linear/03-poisson.

Weak forms

We begin with defining weak forms. C++ version:

```
CustomWeakFormPoisson::CustomWeakFormPoisson(std::string mat_al, Hermes1DFunction<double>* lambda_al,
                                             std::string mat_cu, Hermes1DFunction<double>* lambda_cu,
                                             Hermes2DFunction<double>* src_term) : WeakForm<double>(1)
{
  // Jacobian forms.
  add_matrix_form(new DefaultJacobianDiffusion<double>(0, 0, lambda_al, mat_al));
  add_matrix_form(new DefaultJacobianDiffusion<double>(0, 0, lambda_cu, mat_cu));

  // Residual forms.
  add_vector_form(new DefaultResidualDiffusion<double>(0, lambda_al, mat_al));
  add_vector_form(new DefaultResidualDiffusion<double>(0, lambda_cu, mat_cu));
  add_vector_form(new DefaultVectorFormVol<double>(0, -src_term));
};
```

Python version:

```
# Define weak forms:
class PyCustomWeakFormPoisson(hermes2d.PyCustomWeakFormReal):
    def __init__(self, mat_al, lambda_al, mat_cu, lambda_cu, vol_src_term):
        # Define number of equations.
        self.super(1)

        # Jacobian forms.
        self.add_matrix_form(hermes2d.PyDefaultJacobianDiffusion(0, 0, lambda_al, mat_al))
        self.add_matrix_form(hermes2d.PyDefaultJacobianDiffusion(0, 0, lambda_cu, mat_cu))

        # Residual forms.
```

```
        self.add_vector_form(hermes2d.PyDefaultResidualDiffusion(0, lambda_al, mat_al))
        self.add_vector_form(hermes2d.PyDefaultResidualDiffusion(0, lambda_cu, mat_cu))
        self.add_vector_form(hermes2d.PyDefaultVectorFormVol(0, -src_term))
```

The line `self.super(1)` calls the constructor of the superclass with parameter `1` which is the number of equations. Notice that `HERMES_ANY` is present in the C++ version but it is omitted in the wrapper.

Loading mesh from string in XML format

This is the default way in tutorial examples. First we initialize an instance of the Mesh class. C++ version:

```
Mesh mesh;
```

Python version:

```
mesh = hermes2d.PyMesh()
```

Then we initialize an instance of the XML mesh reader. C++ version:

```
MeshReaderH2DXML mloader;
```

Python version:

```
reader = hermes2d.PyMeshReaderH2DXML()
```

For the time being, mesh strings are stored directly in the worksheets, later we will move them to separate files as in the C++ version. The XML format is described in detail in the C++ tutorial:

```
mesh_stream = '<?xml version="1.0" encoding="utf-8"?>\
<mesh:mesh xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"\
 xmlns:mesh="XMLMesh"\
 xmlns:element="XMLMesh"\
 xsi:schemaLocation="XMLMesh /usr/include/hermes2d/xml_schemas/mesh_h2d_xml.xsd">\
 <variables>\
   <variable name="a" value="1.0" />\
   <variable name="m_a" value="-1.0" />\
   <variable name="b" value="0.70710678118654757" />\
 </variables>\
\
 <vertices>\
   <vertex x="0.0" y="m_a" i="0"/>\
   <vertex x="a" y="m_a" i="1"/>\
   <vertex x="m_a" y="0.0" i="2"/>\
   <vertex x="." y="0.0" i="3"/>\
   <vertex x="a" y="0.0" i="4"/>\
   <vertex x="m_a" y="a" i="5"/>\
   <vertex x="0.0" y="a" i="6"/>\
   <vertex x="b" y="b" i="7"/>\
 </vertices>\
\
 <elements>\
```

```
   <element:quad v1="0" v2="1" v3="4" v4="3" marker="Copper" />\
   <element:triangle v1="3" v2="4" v3="7" marker="Copper" />\
   <element:triangle v1="3" v2="7" v3="6" marker="Aluminum" />\
   <element:quad v1="2" v2="3" v3="6" v4="5" marker="Aluminum" />\
 </elements>\
\
 <edges>\
   <edge v1="0" v2="1" marker="Bottom" />\
   <edge v1="1" v2="4" marker="Outer" />\
   <edge v1="3" v2="0" marker="Inner" />\
   <edge v1="4" v2="7" marker="Outer" />\
   <edge v1="7" v2="6" marker="Outer" />\
   <edge v1="2" v2="3" marker="Inner" />\
   <edge v1="6" v2="5" marker="Outer" />\
   <edge v1="5" v2="2" marker="Left" />\
 </edges>\
\
 <curves>\
   <arc v1="4" v2="7" angle="45" />\
   <arc v1="7" v2="6" angle="45" />\
 </curves>\
</mesh:mesh>'
```

This string is read by the XML mesh reader and passed into the instance of the Mesh class. C++ version (reading from file `domain.xml`):

```
MeshReaderH2DXML mloader;
mloader.load("domain.xml", &mesh);
```

Python version (reading from string):

```
reader.load_stream(mesh_stream, mesh)
```

Using NCLab's GE and ME with your own FEM code

NCLab's Mesh Editor has XML output, so one way to connect your own FEM code to ME is to implement a method that reads meshes in this format. ME also provides mesh output in raw format consisting of arrays of vertices, elements with material markers, boundary edges with boundary markers, and curves. See displayed projects whose names start with "GE and ME".

Mesh refinement operations

All manual mesh refinement operations available in Hermes (see the C++ tutorial) can be also used in Python. For example, the following will perform a uniform mesh refinement. C++ version:

```
mesh.refine_all_elements();
```

Python version:

```
mesh.refine_all_elements()
```

Creating constant Dirichlet condition on certain boundary markers

C++ version:

```
DefaultEssentialBCConst<double> bc_essential(
    Hermes::vector<std::string>("Bottom", "Inner", "Outer", "Left"), FIXED_BDY_TEMP);
```

Python version:

```
bc = hermes2d.PyDefaultEssentialBCConstReal(["Bottom", "Inner", "Left", "Outer"], FIXED_BDY_TEMP)
```

Adding all BCs into a container to be passed into Space

Since there can be many different boundary conditions, all BCs are first put into a container which is then passed into Space. C++ version:

```
EssentialBCs<double> bcs(&bc_essential);
```

Python version:

```
bcs = hermes2d.PyEssentialBCsReal(bc)
```

Creating finite element space

This model problem is a linear second-order equation whose solution is continuous, and therefore we use the H^1 space. All other spaces available in C++ Hermes – $H(\mathrm{curl})$, $H(\mathrm{div})$ and L^2 – are available in Python as well. C++ version:

```
H1Space<double> space(&mesh, &bcs, P_INIT);
```

Python version:

```
space = hermes2d.PyH1SpaceReal(mesh, bcs, P_INIT)
```

Initializing weak formulation

C++ version:

```
CustomWeakFormPoisson wf("Aluminum", new Hermes1DFunction<double>(LAMBDA_AL),
                         "Copper", new Hermes1DFunction<double>(LAMBDA_CU),
                         new Hermes2DFunction<double>(VOLUME_HEAT_SRC));
```

Python version:

```
wf = PyCustomWeakFormPoisson("Aluminum", LAMBDA_AL, \
         "Copper", LAMBDA_CU, VOLUME_HEAT_SRC)
```

Initializing discrete problem

C++ version:

```
DiscreteProblem<double> dp(&wf, &space);
```

Python version:

```
dp = hermes2d.PyDiscreteProblemReal(wf, space)
```

Initializing a Solution

C++ version:

```
Solution<double> sln;
```

Python version:

```
solution = hermes2d.PySolutionReal()
```

Initializing Newton solver

C++ version:

```
NewtonSolver<double> newton(&dp, matrix_solver);
```

Python version:

```
newton = hermes2d.PyNewtonSolverReal(dp)
```

Solving via the Newton's method

C++ version:

```
newton.solve();
```

Python version:

```
newton.solve(coef)
```

The fact that the initial coefficient vector is omitted on the C++ level means that it is a zero vector by default.

Translating resulting coefficient vector into a Solution

C++ version:

```
Solution<double>::vector_to_solution(newton.get_sln_vector(), &space, &sln);
```

Python version:

```
hermes2d.PySolutionReal().vector_to_solution(newton.get_sln_vector(), space, solution)
```