Python bindings for Nektar++

David Moxey
College of Engineering, Mathematics & Physical Sciences
University of Exeter

Nektar++ Workshop, London, UK
16th June 2017

Why Python?

- Python is a fantastic ‘glue’ language
- Easy to use and write code for
- Many non-Python libraries offer bindings: interfaces for huge amounts of software across lots of different fields
- Great for rapid prototyping of ideas and coupling with complex software
- Also much easier for GUI applications
- However pure Python can be quite slow: write C extensions instead

Talking to Python

Lots of approaches for bridging with external code:

- Can use the Python C API directly
  - ➡ Very technical, lots of code, quite hard work
- Can try to use an automatic wrapper (e.g. SWIG)
  - ➡ Quality of bindings tends to be poor
  - ➡ Do you need *every* function and class to be wrapped?
  - ➡ Technical problems: Nektar++ heavily uses shared pointers, inheritance everywhere

Boost.Python

- Solution: use Boost.Python
- Create Python classes in C++, point them towards our classes and functions
- Boost.Python then handles the Python API for you
- Clever C++ interface, can automatically convert between a lot of Python and C++ data types
- However it's a manual wrapper: produces high quality wrappings, but you have to decide how much to wrap and how to do it

Hello World!

```
#include <boost/python.hpp>

char const* greet()
{
   return "hello, world";
}

BOOST_PYTHON_MODULE(hello_ext)
{
    using namespace boost::python;
    def("greet", greet);
}
```

Boilerplate for a simple Boost.Python wrapper

Compile, link against boost.python, save as **`hello_ext.so`**

```
>>> import hello_ext
>>> print hello_ext.greet()
hello, world
```

Now launch python from same directory

A simple class

```
#include <boost/python.hpp>
struct World {
    void set(std::string msg) { this->msg = msg; }
    std::string greet() { return msg; }
    std::string msg;
};

using namespace boost::python;

BOOST_PYTHON_MODULE(hello) {
    class_<World>("World")
        .def("greet", &World::greet)
        .def("set", &World::set)
    ;
}
```

```
>>> import hello
>>> planet = hello.World()
>>> planet.set('howdy')
>>> planet.greet()
'howdy'
```

Inheritance

```
void export_StdExpansion()
{
    py::class_<StdExpansion,
               boost::shared_ptr<StdExpansion>,
               boost::noncopyable>(
                   "StdExpansion", py::no_init);
}

void export_StdQuadExp()
{
    py::class_<StdQuadExp, py::bases<StdExpansion>,
               boost::shared_ptr<StdQuadExp> >(
                   "StdQuadExp", py::init<
                   const LibUtilities::BasisKey&,
                   const LibUtilities::BasisKey&>());
}
```

Remember that Python can do multiple inheritance!

Likely problems

- Abstract classes don't exist in Python: use noncopyable and no_init
- Wrapping overloaded methods is a pain, although there are some macros to help you
- Wrapping functions that return things by reference
- Complex STL things (e.g. vector, iostream) as well as e.g. double **
- A lot of these can be sorted out (and made **more Pythonic**) using thin wrappers

Nektar++ bindings

- Current status: **experimental and incomplete**
  - ➡ LibUtilities (basis & points, SessionReader)
  - ➡ StdRegions (1D & 2D elements)
  - ➡ SpatialDomains (MeshGraph, Geometry objects)
  - ➡ LocalRegions (1D & 2D elements)

- Written as a separate package, i.e. not in the Nektar++ repository, you link against it:

  https://gitlab.nektar.info/nektar/nektar-python

Nektar++ bindings

- Has a layout very similar to the library level: filenames the same so it is clear where they live
- Maps `Array<OneD, >` to and from `numpy.ndarray`, which makes wrapping class functions quite easy
- Arrays are shallow copied to NumPy arrays
- NumPy arrays are always deep copied to Arrays
- For this we need Boost.NumPy (Boost 1.63+): if you don’t have it (quite likely), it’ll compile it (hopefully)

Very exciting demo

Caveats

- Probably quite inefficient right now
- Array to NumPy interface only works for doubles and won't support Arrays of Arrays (nothing stopping this in principle)
- Deep copy of Array - needs modifications inside of Nektar++
- Obviously not much has been wrapped yet!

My hopes with this

- Can get a user-friendly set of bindings out in order to make development easier
- Easier pre- and post-processing, maybe even writing solvers
- Rapid prototyping of new numerical ideas
- Custom GUIs (e.g. visualisation)
- Someone might find it useful!

Compiling the wrapper

- First compile Nektar++ (from master) and install as normal, then point bindings towards that directory
- Hopefully not too much of a pain, but not as seamless as Nektar++
- macOS probably easiest because I'm biased
- Please make sure you use your system's Boost library
- **Instructions on GitLab website**

In this workshop

1. Compile Nektar++ and the bindings
2. Try to run some of the examples
3. Modify them to do something more interesting
4. Then start wrapping: can you add new functions to StdExpansion? Can you wrap StdTetExp, TetGeom and TetExp? I'll help solve problems!
5. More adventurous: try to add a basic wrapper for MultiRegions::ExpList and ExpList2D, do VTK output

https://gitlab.nektar.info/nektar/nektar-python