Python bindings in BornAgain

Gennady Pospelov

Scientific Computing at MLZ

October, 24 2013, ILL, Grenoble, France
BornAgain is a free and open source software package to simulate and fit small angle scattering at grazing incidence.

- see talk of Walter Van Herck
Basic software architecture

Main features:
- Programming language C++
- Core libraries consist of 30k lines of code
- Python bindings
- Mac, Linux, Windows

- Depend on 4 well established libraries: gsl, boost, eigen, fftw3

Why C++?
- Performance.
- Type safety.
- External dependencies.
- Prospective use of GPU, parallel computing.
- Developers background.
Basic software architecture

Working with BornAgain.
- Running user C++ program

User C++ program
- External graphics
  - BornAgain
    - Python bindings
      - libCore
        - Samples and algorithms
        - GSL
        - fftw3
        - Eigen
      - libFit
        - minimizers
        - ROOT

Basic software architecture

**Working with BornAgain.**
- Running user Python script

![Diagram showing the software architecture of BornAgain with Python bindings and external graphics]

- **User Python script**
  - **BornAgain**
  - **libCore**
    - Samples and algorithms
  - **libFit**
    - minimizers
  - **GSL**
  - **fftw3**
  - **Eigen**
  - **(Py)ROOT**

- **External graphics**
  - **Matplotlib**
C++ is a complex programming language to master.
Several people find themselves more comfortable with higher-level interpreted environments.

Objectives
- create sample description and run the simulation from Python
- extension of the framework functionality with Python code
  new formfactors, pair-correlation functions etc.
There are two basic methods for integrating C++ with Python.

- Extension writing
  - Python access to C++

- Embedding
  - C++ access to the Python interpreter

We are primarily concerned with extension writing.

Extension example using Python-C-API

Python is written on C
Any third-party C/C++ code can be interfaced to Python using wrapper functions

function.c

```c
int fact(int n)
{
    if(n<=1) return1;
    else return n*fact(n-1);
}
```

wrapper.c

```c
#include <Python.h>

PyObject *wrap_fact(PyObject *self, PyObject *args) {
    int n, result;
    if (!PyArg_ParseTuple(args,"i:fact",&n))
        return NULL;
    result = fact(n);
    return Py_BuildValue("i",result);
}

static PyMethodDef exampleMethods[] = {
    { "fact", wrap_fact, 1 },
    { NULL, NULL }
};

void initexample() {
    PyObject *m;
    m = Py_InitModule("example", exampleMethods);
}
```

Wrapper function
- converts function arguments from Python to C
- Returns results in Python expected form

Initialization function
- Register new methods with the Python interpreter
Extension example using Python-C-API

- Function and wrappers code might be compiled into single shared library.

```bash
$ g++ -I /usr/local/python2.7 function.c wrapper.c
$ g++ -shared function.o wrapper.o -o mymodule.so
```

- Resulting shared library can be imported in the python module

```python
import mymodule
```

Wrapping a C/C++ application

- Write a Python wrapping function for every function, variable, structure and class you want to access.
- Write initialization function.
- Compile everything into the module.

Huge amount of work!
List of technologies

Python C API
Strong candidate to wrap only a couple of functions

Boost::python
Comprehensive mapping of C++ features to corresponding Python features
Stl containers, exceptions, properties

Swig
Can create wrappers for other languages besides Python (Ruby, Java)

SIP
Created for PyQt4 package

Shiboken
Created for PySide package

Ctypes
Provides C compatible data types

Cython
Python like language to write C extensions for Python
How to select the one

- Different technologies looks pretty easy while wrapping

```cpp
void hello_world()
{
    std::cout << "Hello World!" << std::endl;
}
```

- They are getting pretty tricky to master while exposing classes

```cpp
class IBase
{
    IBase(float x, float y);
    virtual ~IBase();
    virtual init() = 0;
}
```

```python
class Derived(IBase):
    def __init__(self):
        IBase.__init__(self)
    def init(self):
        self.x = 0
```

- And what about object ownership, shared_ptr, templates?

It is hard to say if given exposing technology fits your C++ project before you actually try to apply it to the whole project.
How to select the one

Discussion of Python-C-API, Ctypes, SWIG, Cython from NumPy community
http://scipy-lectures.github.io/advanced/interfacing_with_c/interfacing_with_c.html

Detailed evaluation of boost-python vs SWIG (2003)
http://seal.web.cern.ch/seal/snapshot/work-packages/scripting/evaluation-report.html

SWIG tutorial
http://www.swig.org/papers/PyTutorial98/PyTutorial98.pdf

StackOverflow discussions
http://stackoverflow.com/questions/1492755/python-c-binding-library-comparison
http://stackoverflow.com/questions/135834/python-swig-vs-ctypes
http://stackoverflow.com/questions/456884/extending-python-to-swig-not-to-swig-or-cython
How to select the one

- What is the performance?
- Build system integration?
- Is wrapping code on Python side or on C++ side?
- How much code should be written additionally?
- Should I affect or duplicate existing C++ code?
- How big is a community?
- Is it possible to fully automatize wrappers generation?
- Do I need bindings with another languages?

We have decided to go with

**boost::python**

providing the higher level of integration between C++ and Python among all competitors.
C++ template based library which enables interoperability between C++ and Python programming languages.

- References and pointers.
- Automatic cross module type conversions.
- Efficient function overloading.
- C++ to Python exception translation.
- Default arguments.
- Keyword arguments.
- Exporting C++ iterators as Python iterators.
- Documentation strings.
- Manipulating Python objects in C++.
Example I
- Exposing class and its access functions to Python

Class to expose

```cpp
class TwoNumbers {
public:
    Num();
    float getA() const;
    void setA(float value);
    float getB() const;
    void setB(float value);
private:
    double a;
    double b;
};
```

Wrapper to create

```cpp
boost::python::class_<TwoNumbers>("TwoNumbers")
    .add_property("A", &TwoNumbers::getA, &TwoNumbers::setA)
    .add_property("B", &TwoNumbers::getB);
```

Use from Python

```python
>>> x = TwoNumbers()
>>> x.A = 3.14
>>> x.A, x.B
(3.14, 3.14)
>>> x.B = 2.17 # error!
```

Class property “B” is read only since setter member function not passed in
Example II
- Inheritance and virtual functions

Class to expose
```cpp
class Base {
    virtual ~Base() {}  
    virtual int f()  
    {  
        return 0;  
    }
};
```

Wrapper to create
```cpp
struct BaseWrap : Base, wrapper<Base>
{
    int f()
    {
        if (override f = this->get_override("f")) return f();
        return Base::f();
    }
    int default_f() { return this->Base::f(); }
};
```

Use from Python
```python
>>> base = Base()
>>> class Derived(Base):
    ...
        def f(self):
        ...
            return 42
...
>>> derived = Derived()
>>> base.f()
0
>>> derived.f()
42
```
Pyplusplus allows to expose to Python almost any C++ construction.
However, this requires writing of a large amount of wrapping code.
Fortunately, this can be tackled with Pyplusplus

Pyplusplus package
An object oriented framework for creating a code generator for boost::python library

- BornAgain C++ source code
  - Pyplusplus codegenerator
  - Boost::python wrappers
  - compiler

- We regenerate boost::python API only when public interface has changed
codegenerator.py

- Provides automatic generation of boost::python code for the whole project
- Contains a number of settings to adjust ownership policies, list of classes to expose, etc
- About 300 lines of python code

```python
from pyplusplus import module_builder

include_classes = [
    "BasicVector3D<complex<double>> ",
    "Lattice",
    "Layer",
    "Crystal"
]

mb = module_builder.module_builder_t( include_paths="BornAgain/include", cflags="-m64")

mb.class_( "IParame\text{ter\text{ized}}" ).member_function("registerParameter").exclude()
mb.class_("SampleBuilder" ).member_function("build\text{Sample}\text{"}).call_policies =
call_policies.return_value_policy( call_policies.manage_new_object )
```
Current status

- BornAgain core consists of 30k lines of code
- PythonAPI is generated automatically using boost::python + pyplusplus
- Same shared library can be used from C++ and from Python

Learning curve
- 10% of time was devoted to Python bindings during the first year
- <1% of time during the second

Achieved Python/C++ interoperability
- Inheritance and overload
- Transfer of ownership
- Templates exposing
- Python lists to std::vector and back
- Export C++ data structures into NumPy arrays
- shared_ptr to Python and back
Current status

C++

```cpp
const IMaterial *air_material =
    MaterialManager::getHomogeneousMaterial("Air", 0., 0.);
const IMaterial *particle_material =
    MaterialManager::getHomogeneousMaterial("Particle", 6e-4, 2e-8);

FormFactorCylinder *cylinder_ff = new FormFactorCylinder(5*nanometer, 5*nanometer);
Particle *particle = new Particle(particle_material, cylinder_ff)

ParticleDecoration particle_decoration;
particle_decoration.addParticle(particle, 0.5)

InterferenceFunction1DParaCrystal *interference =
    InterferenceFunction1DParaCrystal(20*nanometer, 7*nanometer, 1e3*nanometer)
particle_decoration.addInterferenceFunction(interference)

Layer air_layer(air_material);
air_layer.setDecoration(particle_decoration);
```

Python

```python
from BornAgainCore import *

air_material = MaterialManager.getHomogeneousMaterial("Air", 0.0, 0.0)
particle_material = MaterialManager.getHomogeneousMaterial("Particle", 6e-4, 2e-8)

cylinder_ff = FormFactorCylinder(5*nanometer, 5*nanometer)
cylinder = Particle(particle_material, cylinder_ff)

particle_decoration = ParticleDecoration()
particle_decoration.addParticle(cylinder, 0.5)

interference = InterferenceFunction1DParaCrystal(20*nanometer, 7*nanometer, 1e3*nanometer)
particle_decoration.addInterferenceFunction(interference)

air_layer = Layer(air_material)
air_layer.setDecoration(particle_decoration)
```
Summary

**boost::python + pyplusplus**

**Advantages**
- It is possible to establish complete interoperability between C++ and python.
- Actually no limitations have been found in exposing C++ constructions into Python.
- No C++ code should be adjusted.
- Single script regenerates PythonAPI for the whole project if needed.

**Disadvantages**
- boost::python wrappers double the amount of code.
- It is heavy templated code which double the compilation time.
- pyplusplus and gccxml are not developed anymore, no C++11 support will follow.

**Future plans.**
- Wait for ROOT version 6.0 which will use technology llvm + PyPy and have a look on it