Financial Modelling in Python
By S. Fletcher & C. Gardner

Appendix B
Boost.Python

The Boost.Python library provides a framework for seamlessly wrapping C++ classes, functions and objects to Python, and vice-versa. No special tools are used – just the C++ compiler. The library has been so designed that you should not have to change the C++ code in order to wrap it. Through the use of advanced metaprogramming techniques in Boost.Python, the syntax of the actual wrapping code, has the look of a declarative interface definition language. In this appendix we introduce the core features of Boost.Python. The sections are loosely based on the online Getting Started Tutorial in the Boost.Python distribution. For more exhaustive documentation, the reader is encouraged to consult the online reference manual at http://www.boost.org.

B.1 HELLO WORLD

Let's start with the 'Hello world' C++ function

```
char const* greet()
{
  return ''Hello world'';
}
```

The function can be exposed to Python by writing the following Boost.Python wrapper:

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

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

We can now build this as a shared library and the resulting library is a Python module. An invocation of the function from the Python command line looks like

```
>>> import hello_ext
>>> print hello_ext.greet()
```

B.2 CLASSES, CONSTRUCTORS AND METHODS

Let's consider a C++ class/struct that we want to expose to Python:

```
struct World
{
  World(std::string msg): msg(msg) {}
  void set(std::string other) { msg = other; }
```

```
   std::string greet() { return msg; }
   std::string msg;
};
```

The Boost.Python wrapper for the above class is

```
BOOST_PYTHON_MODULE(hello_ext)
{
  using namespace boost::python;
  class_<World>(''World'', init<std: :string>())
    .def(''greet'', &World::greet)
    .def(''set'', &World::set)
  ;
}
```

The `init<std::string>()` exposes the constructor. Additional constructors can be exposed by passing more `init<...>` to the `def()` member function. For example, suppose `World` has another constructor taking two doubles, the wrapping code would look like

```
class_<World>(''World'', init<std::string>())
  .def(init<double, double>())
  .def(''greet'', &World::greet)
  .def(''set'', &World::set)
;
```

If our C++ class, world had no explicit constructors, that is if its definition were to read

```
struct World
{
  void set(std::string other) { msg = other; }
  std::string greet() { return msg; }
  std::string msg;
};
```

the compiler would synthesise an implicit default constructor. In such a case, Boost.Python can expose the default constructor by default, implying that the wrapping code could be written,

```
class_<World>(''World'')
  .def(''greet'', &World::greet)
  .def(''set'', &World::set)
;
```

then in the Python interpreter, it could be exercised like this:

```
>>> planet = hello_ext.World()
```

Abstract classes without any constructors can be exposed by using the `no_init` instead, as seen in the example below:

```
class_<Abstract>(''Abstract'', no_init)
;
```

In C++ we usually avoid public access to data members because it breaks the idea of encapsulation: with access only possible via the accessor methods `set` and `get`. Python, on the other hand, allows class attribute access by default. We replicate this behaviour for wrapped C++ classes by using the `add_property` method of the class `class_` in Boost.Python. To illustrate this, suppose we wish to wrap the following C++ class

```
struct Num
{
  Num();
  float get() const;
  void set(float value);
};
```

then the Boost.Python wrapping code looks like

```
class_<Num>(''Num'')
  .add_property(''rovalue'', &Num::get)
  .add_property(''value'', &Num::get, &Num::set)
;
```

and in Python:

```
>>> x = Num()
>>> x.value = 3.14
>>> x.value, x.rovalue
(3.14, 3.14)
>>> x.rovalue = 2.17 # error!
```

Before leaving this section we need to consider constructors with default arguments. To deal with default arguments in constructors, Boost.Python has provides the (tag) type `optional`. A simple example should suffice to explain the semantics. Consider the C++ class:

```
struct X
{
  X(int a, char b = 'D', std::string c = ''constructor'',
                    double d = 0.0);
};
```

To add this constructor to Boost.Python, we simply write:

```
.def(init<int, optional<char, std::string, double> >())
```

B.3 INHERITANCE

It is also possible to wrap class hierarchies, related by inheritance, using Boost.Python. Consider the trivial inheritance structure:

```
struct Base { virtual Base(); };
struct Derived : Base {};
```

together with a set of C++ functions operating on instances of `Base` and `Derived`:

```
void b(Base*);
void d(Derived*);
Base* factory { return new Derived; }
```

The wrapping code for both the `Base` and `Derived` is

```
class_<Base>(''Base'')
  /**/
  ;
```

and

```
class_<Derived, bases<Base> >(''Derived'')
  /**/
  ;
```

where we have used `bases<..>` to indicate that `Derived` is derived from `Base`. The corresponding wrapping code for the C++ free functions looks like

```
def(''b'', b);
def(''d'', d);
def(''factory'', factory,
  return_value_policy<manage_new_object>());
```

The `return_value_policy<manage_new_object>` construct informs Python to hold the instance of the new Python `Base` object until the Python object is destroyed.

Both pure virtual and virtual functions with default implementations can be handled by Boost.Python. However, this is one of the rare instances where we have to write some extra C++ code to achieve this. Let's start with pure virtual functions. Suppose we have the following base class:

```
struct Base
{
 virtual Base() {}
 virtual int f() = 0;
};
```

What we need to do is write a little wrapper class that derives from `Base` and unintrusively hooks into the virtual functions so that a Python override can be called. The code for the wrapper class is shown below:

```
struct BaseWrap : Base, wrapper<Base>
{
  int f()
  {
    return this->get_override(''f'')();
  }
};
```

Note that we inherit from both `Base` and `wrapper<Base>`. The `wrapper` template class facilitates the job of wrapping classes that are meant to be overridden in Python. Finally, to expose `Base` we write:

```
class_<BaseWrap, boost::noncopyable>(``Base'')
  .def(``f'', pure_virtual(&Base::f))
;
```

Next we consider virtual functions with default implementations. In this instance, the `Base` class may look like

```
struct Base
{
  virtual_Base() {}
  virtual int f() { return 0; }
};
```

Again we need to introduce a C++ class to help us:

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

Just as before, the above class also implements `f`, but now we have to check if `f` has been overriden. The corresponding Boost.Python wrapper code is:

```
class_<BaseWrap, boost::noncopyable>(``Base'')
  .def(``f'', &Base::f, &BaseWrap::default_f)
;
```

Note that we expose both `&Base::f` and `&BaseWrap::default_f` because Boost.Python needs to know about both the dispatch function `f` and its default implementation `default_f`. In Python, we can now do the following:

```
>>> base = Base()
>>> class Derived(Base):
...   def f(self):
...     return 42
...
>>> derived = Derived()
>>> base.f()
0
>>> derived.f()
42
```

B.4 PYTHON OPERATORS

Boost.Python makes it extremely easy to wrap C++ operator-powered classes. A simple example should suffice. Consider the class:

```
class Vector{ /*...*/};

Vector   operator+(Vector const&, float);
Vector   operator+(float, Vector const&);
Vector   operator-(Vector const&, float);
Vector   operator-(float, Vector const&);
Vector&  operator+=(Vector&, float);
Vector&  operator-=(Vector&, float);
bool     operator<(Vector const&, Vector const&);
```

The class and operators can be mapped to Python by writing:

```
class_<Vector>(''Vector'')
  .def(self + float() )
  .def(float() + self )
  .def(self - float() )
  .def(float() - self )
  .def(self += float())
  .def(self -= float())
  .def(self < self)
;
```

B.5 FUNCTIONS

In C++ it is common to come across functions with arguments and return types that are pointers or references. The problem with such primitive types is that we don't know the owner of the pointer or referenced object. Although most C++ programmers now use smart pointers with clear ownership semantics, nevertheless there exists a lot of older C++ code with raw pointers. So Boost.Python has to be able to deal with them. The main issue to solve is the problem of dangling pointers and references. Let's consider the following simple C++ function:

```
X& f(Y& y, Z* z)
{
  y.z = z;
  return y.x;
}
```

The above function binds the lifetime of the function's return type to the lifetime of `y`, because `f` returns a reference to a member of the `y` object. If we were to naively wrap this using Boost.Python, then deleting `y` will invalidate the reference to `X`. In other words we have a dangling reference. To get round these problems, Boost.Python has the concept of call policies. In our example, we can use `return_internal_reference` and `with_custodian_and_ward` as follows:

```
def(''f'', f,
     return_internal_reference<1,
       with_custodian_and_ward<1, 2> >();
```

The `1` in `return_internal_reference<1` informs Boost.Python that the first argument of `f`, in this case `Y& y`, is the owner of the returned reference. Similarly the `1, 2` in `with_custodian_and_ward<1, 2>` informs Boost.Python that the lifetime of the second argument of `f`, in this case `Z* z`, is tied to the lifetime of the first argument `Y& y`.

It is common in C++ to overload both functions and member functions. Consider the following C++ class:

```
struct X
{
  bool f(int a);
  bool f(int a, double b);
  int f(int a, int b, int c);
};
```

To wrap the overloaded member functions into Python we need to introduce some member function pointer variables:

```
bool (X::*fx1)(int)          = &X::f;
bool (X::*fx2)(int, double)  = &X::f;
int (X::*fx3)(int, int, int) = &X::f;
```

With the member function pointer variables defined, the Boost.Python wrapping code is simply

```
.def(''f'', fx1)
.def(''f'', fx2)
.def(''f'', fx3)
```

We have seen in the above example how Boost.Python wraps function pointers. Many functions in C++ have default arguments, but C++ function pointers hold no information about default arguments. Therefore we have to write thin wrappers so that the default argument information is not lost. Consider the C++ function:

```
int f(int, double = 3.14, char const* = ''hello'');
```

then we have to write the thin wrappers:

```
int f1(int x) { f(x); }
int f2(int x, double y) { f(x, y); }
```

The Boost.Python wrapping code then looks like:

```
def(''f'', f);  // all arguments
def(''f'', f2); // two arguments
def(''f'', f3); // one argument
```

Fortunately Boost.Python has a macro for automatically creating the wrappers for us. For example

```
BOOST_PYTHON_FUNCTION_OVERLOADS(f_overloads, f, 1, 3)
```

The macro creates a class `f_overloads` that can be passed on to `def(...)`. The third and fourth arguments denote the minimum and maximum arguments respectively. The `def(...)` function will automatically add all the variants for us:

```
def(''f'', f, f_overloads());
```

Similarly for member function overloads, we can use the BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS macro. Suppose we had the C++ class:

```
struct X
{
  bool f(int a, int b = 0, double = 3.14);
};
```

then we would write:

```
BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS(X_overloads, X, 1, 3)
```

and the generated class X_overloads can be used as an argument to .def(...):

```
.def(``f'', &X::f, X_overloads());
```

B.6 ENUMS

Boost.Python has a clever way of wrapping C++ enums. Python has no enum type, so Boost.Python exposes them as an int. Consider the following example:

```
enum choice { red, blue };
```

the Boost.Python enum_<T> construct can be used to expose to Python:

```
enum_<choice>(``choice'')
      .value(``red'', red)
      .value(``blue'', blue)
      ;
```

The new enum type is created in the current scope, which will usually be the current module. The created Python class is derived from the Python int type and the values can be accessed in Python as follows:

```
>>> my_module.choice.red
m_module.choice.red
```

where my_module is the name of the module in which the enum is declared.

B.7 EMBEDDING

We have seen how to use Boost.Python to call C++ code from Python. In this section we are going to discuss how to call Python code from C++. The first step is to embed the Python interpreter into the C++ code. To do this, we simply #include<boost/python.hpp> and call Py_Initialize() to start the interpreter and create the __main__ module. Note that at the time of writing you must not call Py_Finalize() to stop the interpreter. This may change in future versions of Boost.Python. Although objects in Python are automatically reference-counted, the Python C API requires reference counting to be handled manually. So Boost.Python provides the handle and object class templates to automate the process. The handle template class is beyond the scope of this short primer.

The `object` class template wraps `PyObject*` and Boost.Python comes with a set of derived `object` types corresponding to Python's: `list`, `dict`, `tuple`, `str` and `long_`. Wherever appropriate, the methods of a particular Python type have been duplicated in the corresponding derived `object` type. For example, `dict` has a `keys()` method, `str` has a `upper` method, etc., and `make_tuple` is provided for declaring tuples:

```
tuple t = make_tuple(123, ``Hello, World'', 0.0);
```

Just as for Python's types, the constructors for the corresponding derived `object` types make copies. Consider the following example from the Python command line:

```
>>> l = [1, 2, 3]
>>> m = list(l) # new list
>>> m[0] = 4
>>> print l
[1, 2, 3]
>>> print m
[4, 2, 3]
```

Calling the `list` constructor makes a new `list`. Correspondingly the constructors of the derived `object` types make copies:

```
dict d(x.attr(``__dict__'')); // copies x.__dict__
```

Sometimes we need to get C++ values out of object instances. We can do this by using the `extract<T>` template functions. For example:

```
double l = extract<double>(o.attr(``length''));
dict d = extract<dict>(x.attr(``__dict__''));
```

Note that the dictionary `d` is in fact a reference to `X.__dict__`, hence writing

```
d[``whatever''] = 3;
```

modifies `x.__dict__`.

To run Python code from C++, Boost.Python provides three related functions:

```
object eval(str expression
          , object globals = object()
          , object locals = object());

object exec(str     code
          , object globals = object()
          , object locals = object());

object exec_file(str filename
          , object globals = object()
          , object locals = object());
```

`eval` evalulates a given expression, `exec` executes the given code, and `exec_file` executes the code contained in a file. All functions return the results as an `object`. The `globals` and `locals` parameters are Python dictionaries containing the globals and locals of the context in which the code is to be executed. It is almost always sufficient to use the namespace dictionary

of the __main__ module for both parameters. To do this we first use the `import` function of Boost.Python to import the __main__ module:

```
object import(str name);
```

and then get the namespace of __main__ as follows:

```
object main_module = import(``__main__'');
object main_namespace = main_module.attr(''__dict__'');
```

Now that we have the namespace we can execute a Python script, for example:

```
object ignored = exec(``result = 5**2'', main_namespace);
int five_squared = extract<int>(main_namespace[``result'']);
```

B.8 CONCLUSION

The purpose of this primer has been to introduce the reader to some of the tools provided by Boost.Python. The primer is by no means exhaustive. Indeed Boost.Python offers many more features to help the C++ programmer to seamlessly expose C++ classes to Python and embed Python into C++.