Fermilab FERMILAB-Conf-03/116 July 2003

# PyORBIT: A Python Shell for ORBIT

J.-F. Ostiguy* Fermi National Accelerator Laboratory, Batavia, IL
J. Holmes, ORNL Oak Ridge, TN †

*Abstract*

ORBIT is code developed at SNS to simulate beam dynamics in accumulation rings and synchrotrons. The code is structured as a collection of external C++ modules for SuperCode, a high level interpreter shell developed at LLNL in the early 1990s. SuperCode is no longer actively supported and there has for some time been interest in replacing it by a modern scripting language, while preserving the feel of the original ORBIT program. In this paper, we describe a new version of ORBIT where the role of SuperCode is assumed by Python, a free, well-documented and widely supported object-oriented scripting language. We also compare PyORBIT to ORBIT from the standpoint of features, performance and future expandability.

## INTRODUCTION

The philosophy of either embedding or extending a high-level scripting language has become popular for scientific applications. Typically, scientific simulation code execution time is dominated by a few computationally intensive tasks. On the other hand, bookkeeping operations such as data analysis and presentation represent a large portion of the overall development effort. This effort can potentially be significantly reduced by implementing this functionality in an interpreted language, without significantly affecting overall performance. It is in this spirit that ORBIT, a code to model the dynamics of a synchrotron in the presence of space charge, was developed at the SNS [1]. ORBIT is structured as a collection of extension modules for SuperCode, an interpreted array-oriented scripting language with a C++-like syntax originating from the early 1990's Fusion research program at LLNL. Unfortunately, SuperCode is no longer actively developed and supported. In recent years, a number of free, well-documented and stable alternative scripting languages have emerged: Perl, Python, Tcl, Ruby, Guile etc. All these languages have strengths and weaknesses. Perl has already been used sucessfully to build a framework integrating existing accelerator codes [2].

## WHY PYTHON ?

Among the many scripting languages, Python has the distinction of being fundamentally object-oriented. It supports concepts such as classes, inheritance and operator overloading. Python scripts are compiled into interpreted bytecode. Python and C++ syntax and semantics both map very well into each other, making Python a particularly good choice as middleware language to integrate functionality implemented in C++. Commonalities between Python and C++ can be summarized as follows [3]:

- both language use C-family of control structures
- support for object-orientation, functional and generic programming
- comprehensive operator overloading facilities
- collections and iterators
- support for namespaces (Python modules)
- exception handling
- Python reference semantics mirrors common C++ idioms (handle/body classes, reference-counted smart pointers)

## MODULE INTERFACE CODE GENERATION

Although Python and C++ overlap conceptually, from a low-level implementation standpoint they differ substantially. Python is implemented in C and naturally offers a C-based API for extension. Compared to C++ and Python, C has rudimentary abstraction facilities and no support for exception-handling. Writing interface code for extension modules using the C-API requires specialized knowledge; furthermore, the code tends to be complex and hard to maintain. This has stimulated the development of automated interface code generation or "wrapping" systems. The exported module interface is specified in a file processed by a specialized program which generates the necessary interface code without further user intervention. There exist at least five systems to generate python/C++ wrappers [4, 5, 6, 7, 8]: SWIG, SIP, CXX, SCXX and Boost.python. SWIG was one of the first and is probably the most widely known interface generator. It offers comprehensive support for most of the popular scripting languages including Python. Although support for the Python/C++ combination has been continuously improving, important limitations remain, one of them being the requirement that all exported templatized functions and classes be explicitly instantiated. The inter-language binding code produced by SWIG is also an unelegant mixture of Python scripts and C code. SIP is very similar in philosophy to SWIG – from which it was originally inspired – but is strictly Python/C++ specific. It was developed as a tool to produce python wrappers for the Qt library, a popular open source GUI framework. Although the results are impressive, the SIP interface specification can be complex and requires the programmer to write some low-level code. CXX wraps some part of the Python C-API in C++, managing the complexity using static metaprogramming techniques.

* ostiguy@fnal.gov
† Work supported by the US Department of Energy under contract number DE-AC02-76CH03000.

```
#include <boost/python.h>
...
class_<Particles>("Particles")
       .def("addMacroHerd",
              &Particles::addMacroHerd,
              "Make a main herd of macro particles")
       .staticmethod("addMacroHerd")
       .def_readwrite("nHerds",  &Particles::nHerds)
;
...
```

Figure 2: Exporting the interface of an ORBIT module using boost.python

Python, making its classes and methods available to the interpreter.

### Strings

SuperCode defines its own private string class. While semantics of this class are close to that of `std::string`, they are not identical. By default, Boost.python provides automatic conversion between Python `str` type and C++ `std::string`. However, it is also possible to define and register other type conversions. This approach is used in PyORBIT to avoid modifying the substantial amount of code that refers to the private string class.

### Function Pointers

Some ORBIT functions expect a function as an argument. Typical examples are functions used to generate initial macroparticles phase space distributions or functions used to define an RF cavity voltage program. Using the latter as an example, it is convenient to define a voltage program at the interpreter level since this is a function which is typically called once per turn. In SuperCode, function pointers are basically C-style `void (*)()`. In Python, **all** variables and functions are references to dynamically allocated objects. When a call to an ORBIT function such as `AddRampedRFCavity` is executed from Python, the arguments are effectively a list of `PyObject*`. Boost.python introspection allows basic data types to be converted before the C++ version of `AddRampedRFCavity` gets called. However, references to functions cannot be converted i.e. there is no unambiguous way to go from a `PyObject*` to a `void (*)()`. To emulate SuperCode syntax and behavior, it is necessary to introduce an additional C++ wrapper function. How this is done is illustrated in Fig. 3 where for clarity, function arguments that are not relevant have been omitted. Python calls `addRampedRFCavity` with a `PyObject*` argument. This argument is passed through a private static variable to a private function which in turn uses the boost.python facility `call<>` to interpret the python function. Finally, the private function is passed as regular C-style function pointer to the original version of `addRampedRFCavity`.

```
void addRampedRFCavity( (void(*)()) sub);

using boost::python:call;

static PyObject*  RampedRFVolt_pyobjptr = 0;

static void private_RampedRFVolt()
{
  call<void>(RampedRFVolt_pyobjptr);
  return;
}

void addRampedRFCavity(PyObject* po )
{
  RampedRFVolt_pyobjptr = po;
  addRampedRFCavity(&private_addRampedBAccel);
}
```

Figure 3: Passing a reference to a function defined in Python to a C++ extension module.

## OUTLOOK AND CONCLUSION

Using Boost.python we have sucessfully transformed ORBIT into a collection of Python modules. This has been accomplished with minimal modifications to the original code. PyORBIT performance is basically the same as that of the original ORBIT since the computationally intensive work is performed by virtually identical C++ code. A vast amount of high quality third party libraries and modules developed for Python have now become available to ORBIT users. In principle, this should allow accelerated development of new data analysis and display facilities. Finally, through boost.python, adding new C++ modules is a well-defined, well-documented and straightforward matter that does not require specialized programming knowledge. At the time of this writing, PyORBIT remains a work in progress. While it is certainly usable as it stands, some work remains to be done before it can be considered a production tool, mostly in connection with exception handling and error recovery.

## REFERENCES

[1] J. Galambos et al., "ORBIT - A Ring Injection Code with Space Charge", Proceedings of the 1999 PAC Conference, pp. 3143-3145

[2] N. Malitsky and R. Talman, AIP 391, 1996

[3] D. Abrahams and R.W. Grosse-Kunstleve, "Building Hybrid Systems with Boost.Python", PyCON 2003, Washington DC, March 2003

[4] `http://www.swig.org`

[5] `http://www.riverbankcomputing.co.uk/sip`

[6] `http://sourceforge.net/projects/cxx`

[7] `http://www.macmillan-in.com/scxx.html`

[8] `http://www.boost.org/scxx.html`

SCXX started as a lightweight version of CXX. It does not use templates and as such cannot hide as many details of the low-level Python C-API. On the other hand, it automates tedious error-prone tasks such as reference counting. Boost.python has features and goals that are similar to all the other systems. However, remarkably, it does not introduce a separate wrapping language and interface code generator. Rather, it makes use of C++ compile-time introspection capabilities and advanced metatemplate programming techniques to allow interface specification to be done in pure C++. Boost.python also goes beyond the scope of other systems by providing the following features:

- support for C++ virtual functions that can be overridden in python
- lifetime management facilities for low-level C++ pointers and references
- a safe and convenient mechanism for tying into Python's powerful serialization engine
- support for organizing extensions as Python packages with a central registry for inter-language type conversions
- automatic coherent handling of C++ lvalues and rvalues

Because of its comprehensiveness and elegance, Boost.python was selected to implement PyORBIT.

## PYORBIT IMPLEMENTATION

To facilitate the transition for users already familiar with ORBIT, an important objective was to make the feel of the new PyORBIT as close as possible to the existing one. This is mostly the case, but some important differences remain. While SuperCode is strongly typed, Python is a dynamically typed language. Variables are not declared; assigning an object to a variable creates it. All Python variables are references to PyObjects. The practical consequence is that an assignment statement such as `a = b` does not result in `a` holding a copy of `b` but rather in both `a` and `b` refering to the same object on the heap. References are counted and objects are marked for automatic deletion when no variable refers to them (automatic garbage collection). Another significant difference between Python and SuperCode is the fact that in Python numbers and strings are immutable objects. If a function has arguments whose types are immutable, the values of these arguments cannot be changed by the function. Immutability of basic data types is not as restrictive as it might seem since Python also provides a mutable container type (`list`). While the container itself cannot be modified, the elements it contains can be.

### *Array Types*

SuperCode defines the built-in array types Vector, Matrix, and Array3D (separately for integers, double and complex) patterned after Fortran arrays (stored in column-major order and 1-based). These types are extensively used in the ORBIT code and therefore have been emulated. In PyORBIT, the SuperCode implementation of the build-in array types is replaced by a new simplified templatized version. Fig. 1 provides an example of how the `ComplexMatrix` type interface is exported to Python using boost.python. At first glance, this may not seem like

```
#include <boost/python.h>
using namespace boost::python;
typedef Matrix<complex<double>
        ComplexMatrix;
 ...

 class_<ComplexMatrix > >
       ("ComplexMatrix", init<int,int>())
 .def(init<const ComplexMatrix > >())
 .def("get",     &ComplexMatrix >::get)
 .def("set",     &ComplexMatrix >::set)
 .def("__repr__",&ComplexMatrix >::print)
 .def("clear",   &ComplexMatrix >::clear)
 .def("resize",  &ComplexMatrix >::resize)
 .def(self + self)
 .def(self - self)
 .def(self * self)
 .def(self ^ self)
   ;
...
```

Figure 1: Exporting the interface of an array datatype using boost.python

C++ and a few explanations are in order. The construct `class_<type>(name)` is simply an anonymous instantiation of a templatized class named `class_`. This instantiation calls the `class_` constructor and passes the C++ `type` to it. A new python class `type` is created and associated with `name` in the Python registry. The syntax `.def` simply denotes a call to a member function named `def`. Because `object.def()` returns a reference to object, multiple calls can be chained. The resulting expression is made more readable by taking advantage of the fact that the compiler ignores whitespace. Note the last four `.def` statements which define operator overloads.

### *ORBIT Modules*

Exporting existing ORBIT modules to Python with boost.python is a relatively mechanical task. The (simplified) code in Fig. 2 provides an example. The class `Particles` is exported to Python and exposes its method `addMacroHerd` and its data `nHerds`. Note that the variable `nHerds` is mirrored in Python by a special boost.python object wich does not have the type int. The expression `Particles.nHerds = 1` in Python does not result in the creation of an immutable `int` object. Rather, for objects of type `Particles`, assigning a value to the attribute `nHerd` calls a method that sets the C++ variable `Particle::NHerds`. All this mechanics is completely transparent to the progammer and generated by the boost.python libary.Compiling the code in Fig. 2 ultimately produces a shared object module that can be dynamically imported by