LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-TR-237219

Leveraging Python Interoperability Tools to Improve Sapphire's Usability

Abel Gezahegne, Nicole S. Love

December 11, 2007

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Leveraging Python Interoperability Tools to Improve Sapphire's Usability

Abel Gezahegne Nicole S. Love
Lawrence Livermore National Laboratory

December 9, 2007

1 Abstract

The Sapphire project at the Center for Applied Scientific Computing (CASC) develops and applies an extensive set of data mining algorithms for the analysis of large data sets. Sapphire's algorithms are currently available as a set of C++ libraries. However many users prefer higher level scripting languages such as Python for their ease of use and flexibility. In this report, we evaluate four interoperability tools for the purpose of wrapping Sapphire's core functionality with Python. Exposing Sapphire's functionality through a Python interface would increase its usability and connect its algorithms to existing Python tools.

2 Introduction

The Sapphire project at the Center for Applied Scientific Computing (CASC) develops a scientific data mining software for the analysis of large multi-dimensional data sets from computer simulations, scientific experiments and observations in the form of images, videos, points in space, or mesh data. To support such a wide range of analysis needs, Sapphire takes an end-to-end approach to data mining and uses an extensible and modular software architecture. Sapphire provides a multitude of algorithms for each of the modules in the data mining pipeline from preprocessing of the raw data, identification and extraction of objects of interest to pattern recognition. Also the modular architecture enables the omission of some modules depending on the analysis needs. This process of data analysis is iterative and interactive as users experiment with different algorithms and/or parameters and omit parts of the data mining pipeline in order to obtain the desired results. Due to the nature of analysis process and the modular architecture of the software, providing Sapphire's functionality through a scripting language such as Python would increase its usability, flexibility and allow for rapid prototyping. Also as the Sapphire user base grows to include users in DNT, NIF, and DHS, who are accustomed to using software via a scripting language, a Python interface for Sapphire would increase its usability by providing users with a flexible way to analyze their data sets. It would also create a bridge between Sapphire's extensible set of analysis algorithms and the users' existing Python workflows. In this report, we evaluate four public-domain interoperability tools for the purpose of wrapping Sapphire's Scientific Data Mining Software with Python. Then we provide some examples on how to create Python binding using the tool we have chosen. Following the examples we demonstrate how to use the Python libraries created for Sapphire. Then we will compare the usage and timing of the C++ and Python Sapphire libraries.

3 Interoperability Tools

Language interoperability tools in a nutshell, provide the means of communication between different languages. These tools allow software developers to focus on more important issues than spend valuable time on the tedious and error prone process of manually writing glue code between languages. Since no one language is best suited for all applications, interoperability tools allow developers to exploit the strength of different languages in their application. Such as the performance of compiled languages and the flexibility of scripting languages. These tools

also allow the incorporation of legacy code into current framework, with out which the code would need to be rewritten. In this section, we provide a brief overview of four interoperability tools, namely Babel, Boost.Python, SWIG and SIP.

3.1 Overview and Evaluation

3.1.1 Babel

Interoperability tools such as Boost.Python, SWIG and SIP focus on extending C/C++ functionality to a scripting language such as Python. Babel, developed by the Components project at Lawrence Livermore National Laboratory, focuses on a much broader issue of language interoperability. Instead of a one source and one target model, Babel enables communication between all supported languages as both the source and the target. Thus, a software library written in any one language may be accessible from another supported language. Currently Babel supports C, C++, Fortran 77, Fortran 90, Java and Python [1].

The Babel tool relies on Scientific Interface Definition Language (SIDL) to describe the software library interface. SIDL is an IDL that is specially oriented for the use of describing scientific software. It supports dynamic multi-dimensional arrays and complex numbers [2]. SIDL is object-oriented and follows the Java object model that an object may extend one class and implement multiple interfaces.

Babel, when applied to the SIDL file, creates four types of files, stub, skeleton, implementation and Intermediate Object Representation(IOR) files. The stub and skeleton files perform the translation between the calling language to the IOR and from the IOR to the implementation, respectively. The implementation file will need to be edited by the user in order to provide access to the implementation of the library [2]. In order to facilitate the editing process between multiple builds, Babel provides the means to save previous edits. Any edits performed in the implementation files that are between matching *Splicer* directives will be maintained during successive builds. Babel will also generate fragments of a makefile in order to facilitate the build process.

One of Babel's advantages is that it is able to incorporate object-oriented concepts into non object-oriented languages it supports. Babel also handles memory management as all Babel objects are reference counted. Babel supports the basic data types as well as complex numbers and multi-dimensional arrays. Among many features it supports inheritance, exception handling and method overloading. For languages that don't support method overloading user provided alternate names are used instead. Among the other tools we have reviewed Babel is the only one that can be used if bi-directional communication of the different languages is necessary. As discussed above, the user will need to create the SIDL files as well as edit the implementation files generated by Babel in order to create a useful library. As such, if bi-directional communication is not necessary and time is a factor it may not be the best choice. The Babel-Runtime distribution is also required to use the libraries generated. Although Babel's user base is growing, it is still small compared to that of other tools. Babel is a research in progress and is supported by a research effort.

3.1.2 Boost.Python

Boost.Python, developed by David Abrahams, is a unique interoperability tool for generating Python bindings for C++. Boost.Python stands out among other tools because it uses the C++ compiler to generate wrapping code. The interface definition needed By Boost.Python to wrap a library is written in C++ and has the look and feel of an IDL. Because it uses template meta-programming to examine types and functions, users are alleviated from having to learn the syntax of another IDL to describe the interface [3]. Although the syntax used for interface description seems confusing, it is nonetheless C++. For further description of the syntax and examples, refer to the Boost.Python documentation and tutorial at [4]. Because this interface definition is written in C++ it also gives the user great power for controlling the generated code. Because Boost.Python operates in the type system domain, it is able to properly handle corner cases, where other tools based on just parsing would fail. It also requires no modification of the library being wrapped. There are also other tools that are a front-end to Boost.Python that generate the interface definition. Although we have not used it for our evaluation, Py++ automatically generates the interface definition for Boost.Python. For further information on this tool refer to [5].

3.1.3 SWIG

Simplified Wrapper and Interface Generator (SWIG), is a tool for generating a scripting language interface for C and C++ libraries. It supports eighteen different target languages including Perl, Python, Tcl, Ruby, Guile and Java [6]. SWIG is one of the oldest and widely used interoperability tool. It supports all of ANSI C and most of C++, and currently SWIG developers are working on SWIG-2.0 that will have support for nearly all of ANSI C++ standards for approximately ten target languages [6]. Some of the major features of SWIG include, all ANSI C and C++ datatypes, pointers, pointers to functions, references, function and operator overloading, single and multiple inheritance, static members, C++ templates including template specialization, C++ smart pointers and support for STL [6]. One major feature SWIG does not yet support is nested classes and structs.

SWIG is not a simple C or C++ parser. It is primarily built around the C++ type system [7]. Tools that are based on parsing alone can only work with the simple cases. Since SWIG understands the full C++ type system and C++ semantics it can handle nasty corner cases.

SWIG is able to generate glue code with minimal effort from the user. It requires an "interface file" that describes the user interface of the library that needs to be wrapped. In its simplest form this file can consist of only a few directives identifying the module being created and the header file containing the library interface. The user can also manually specify all the classes and methods that ought to have a glue code generated. SWIG also provides a wide range of directives in order to further customize almost every aspect of the code generation process. These customizations provide the user with a greater flexibility in controlling the language binding, among many others, by providing the means to change behavior of how data types are mapped, ability to change the ownership of objects from one language to another, or to support cross language polymorphism. This ability to wrap software with minimal user effort as well as the ability to customize, provides the user with a wide range of flexibility and control.

3.1.4 SIP

SIP started out as a specialized version of SWIG, it even derives its name from it. SIP was developed as a tool specific to the creation of Python bindings to C and C++ libraries. The purpose in doing so, was to create a tool that is much better suited for binding between Python and C/C++ than the more general purpose tool SWIG. The original intent of developing SIP was to create Python binding for the Qt toolkit [8]. However SIP can be used for any C or C++ library.

SIP uses a specification file describing the library interface to create the Python binding codes. The specification file is similar to the header file with additional information as well as some parts removed because SIP does not include a full C++ parser [9]. SIP does support a large number of C++ features such as automatic conversion between Python and C/C++ data types, function and method overloading, templates, namespaces, exceptions as well as the support for object ownership control. For a more complete list of supported features refer to the SIP Reference Guide [9]. As with SWIG, SIP provides numerous directives and annotations that are used in a specification file, for greater user control as well as specify certain information that can not be obtained by parsing header files, such as ownership of objects. Although one can use PyQt, the Python bindings for Qt, as a set of examples, there has been criticism of its minimal or lack of complete documentation.

3.2 Chosen Tool for Sapphire

It is important to note that the literary review of these tools is not authoritative nor our experimentation with all these tools exhaustive. Our objective here is not to identify the best overall interoperability tool, if such a tool does exist. Our goal is to identify the tool(s) that are best suited for the purpose of wrapping Sapphire.

The Sapphire Scientific Data Mining Software is a collection of eighteen C++ libraries consisting of almost five hundred classes. These classes are developed with object-oriented principles including inheritance, polymorphism, method and operator overloading, templates and template specialization and smart pointers. As a result we need a tool that can support the majority of C++, with high level of automation, and would not have a large penalty on build times. For the purpose of wrapping Sapphire libraries we chose SWIG, because it supports all of ANSI C and most of ANSI C++, is mature, well documented, has a large user base and allows for a high level of automation for creating SWIG interface files. It also provides the ability to create binding for a number of other target languages, if such a need is to arise at a later time.

4 SWIG Examples

In this section we provide a few examples that illustrate the use of SWIG. As such, only a subset of SWIG's features are described here. A more complete description can be found in the SWIG documentation [6]. The SWIG mailing lists are also a great source of information.

4.1 Functions

As it is customary, lets start with a hello world function below.

helloWorld.hxx

```
#include <iostream>

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

In order for SWIG to create a Python binding for our `helloWorld` function we need to give it the interface file. For this case the following simple interface file will suffice.

helloWorld.i

```
%module helloWorld
%{
#include "helloWorld.hxx"
%}
%include "helloWorld.hxx"
```

In the interface file, the `%module helloWorld`, tells SWIG to encapsulate the functions wrapped in a Python module with the name `helloWorld`. Any code with in the `%{` and `%}` block is copied verbatim to the binding code SWIG generates. The `%include` statement tells SWIG to parse the given file and generate binding for the methods and classes it finds. We can also explicitly specify what methods we want wrapped, although that gives the user great control over what and how SWIG generates the wrapping code, providing the file is quicker. We will provide an example of the explicit specification later on. Once SWIG has created the Python bindings, we can use it as follows:

Python

```
>>> import helloWorld
>>> helloWorld.helloWorld()
Hello World !!!
>>>
```

4.2 Classes

To illustrate the wrapping of Classes, we have included a `Box` class header file. We will use this class as a running example to show different features of SWIG.

Box.hxx

```
#include <iostream>

class Box {
   public:
```

```
        // Constructors and Destructor
        Box() ;

        Box(int lwrXCoord, int lwrYCoord,
            int uprXCoord, int uprYCoord) ;

        Box(const Box& rhs) ;

        ~Box() ;

        // Set Box Coordinates
        void setLwrCoords(int lwrXCoord, int lwrYCoord) ;
        void setUprCoords(int uprXCoord, int uprYCoord) ;

        // Assignment Operator
        Box& operator=(const Box& rhs) ;

        // Comparison Operator
        bool operator>(const Box& rhs) const ;

        // Obtain bounding box of two boxes
        Box operator+(const Box& rhs) const ;

        // Obtain the intersection of two boxes
        Box operator*(const Box& rhs) const ;

        // Grow the box
        void grow(const int xGrow, const int yGrow) ;
        static Box grow(const Box& box, const int xGrow, const int yGrow) ;

        // Shift the box
        void shift(const int xShift, const int yShift) ;
        static Box shift(const Box& box, const int xShift, const int yShift) ;

        // Is box empty
        bool empty() const ;

        // Get Size of box
        int size() const ;

        // Output operator
        friend std::ostream& operator<<(std::ostream &os, const Box& box) ;

    private:
        int lwrXCoord ;
        int lwrYCoord ;
        int uprXCoord ;
        int uprYCoord ;
} ;
```

As in the previous example we can create a simple interface file that will wrap most of the functionality of this class. However, in this case, we have to tell SWIG more information on several aspects. In most cases SWIG is able

to handle method and operator overloading correctly. For example the `operator+` and `operator*` will be wrapped correctly as the `__add__` and `__mul__` methods respectively. Thus, they can be called from Python as one would expect with `+` and `*`, respectively. However, SWIG ignores the output operator as it ignores all friend declarations. Also the `operator<<` would be unorthodox as part of the Python environment. Instead we can use the `%extend` directive to extend the Box class and implement the `__str__` method which creates a string representation of the `Box` class. We would be able to call this method using `print` from Python. Although SWIG automatically handles method overloading, it does not support method overloading with static and non-static methods. Thus, in our case, the static version of the methods `grow` and `shift` will not be wrapped. In order to fix these issues we will have to give SWIG more information in the interface file.

Box.i

```
%module Box
%{
#define private public
#include "Box.hxx"
%}

%rename (static_grow) Box::grow(const Box&, const int, const int);
%rename (static_shift) Box::shift(const Box&, const int, const int);

%ignore *::operator= ;
%ignore operator<< ;

%include "Box.hxx"

%extend Box
{
   char * __str__()
   {
      static char buf[256] ;

      sprintf(buf, "Box: [(%g, %g), (%g, %g)]",(double) self->lwrXCoord,
                                               (double) self->lwrYCoord,
                                               (double) self->uprXCoord,
                                               (double) self->uprYCoord) ;
      return buf ;
   }
}
```

In the interface file we have used the `%rename` directive to change the name of both static methods. In this case, we have added the prefix `static_` to the static method names. We have also explicitly told SWIG to ignore the `=` and `<<` operators, this is only to silence the warnings SWIG prints, as it will already ignore them. As mentioned above the `%extend` directive is used to add the `__str__` method. Also note the `#define private public` statement within the `%{ %}` block. We have done this in order to gain access to the box coordinates. We would have used accessor methods if they had been provided in the class. We can use the `Box` class from Python as follows:

Python

```
>>> from Box import *
>>>
>>> box1 = Box(0, 0, 5, 5)
>>> box1.grow(1, 2)
>>> print box1
Box: [(-1, -2), (6, 7)]
```

```
>>>
>>> box2 = Box(3, 4, 9, 8)
>>> box3 = box1 * box2
>>> print box3
Box: [(3, 4), (6, 7)]
>>>
>>> box4 = Box.static_shift(box3, 2, 1)
>>> print box4
Box: [(5, 5), (8, 8)]
>>>
```

4.3 Templates

The `Box` class we have provided uses an `int` type for the coordinates. If instead we want use other types as well as `int`, we would need to make `Box` a template class. SWIG is able to handle template classes, however requires template instantiation. With out the instantiation, SWIG simply ignores the class and won't create any binding code for it. For the sake of brevity we will omit the modified header file and provide only the changes in the SWIG interface file.

Box.i

```
...
%rename (static_grow) Box<int>::grow(const Box<int>&, const int, const int) ;
%rename (static_grow) Box<double>::grow(const Box<double>&, const double, const double);
%rename (static_shift) Box<int>::shift(const Box<int>&, const int, const int);
%rename (static_shift) Box<double>::shift(const Box<double>&,const double,const double);
...
%template (Box_Int) Box<int> ;
%template (Box_Double) Box<double> ;
```

In the interface file we have added the template instantiation on the types `int` and `double` using the `%template` directive. Given this, SWIG will create Python bindings for two classes `Box<int>` and `Box<double>`. The argument given to the `%template` directive, in this case `Box_Int` and `Box_Double` are used as the instantiation names in Python.

Python

```
>>> from Box import *
>>>
>>> box1 = Box_Double(0, 0, 5.5, 7.6)
>>> box2 = Box_Double.static_shift(box1, 6, 0)
>>> print box2
Box: [(6, 0), (11.5, 7.6)]
>>>
>>> box3 = box1 * box2
>>> box3.empty()
True
>>>
```

In a similar fashion SWIG can also generate binding code for template functions. In order for SWIG to generate the wrapper code we have to provide the template instantiations. To illustrate template functions lets look at the following `getMax` function.

getMax.hxx

```
template <class TYPE>
TYPE getMax(const TYPE& lhs, const TYPE& rhs)
{
   return (lhs > rhs) ? lhs : rhs ;
}
```

getMax.i

```
...
%template (getMax) getMax<int> ;
%template (getMax) getMax<double> ;
%template (getMax) getMax<Box<int> > ;
```

SWIG can now generate the wrapping codes for the three template instantiations. Although we could have given each instantiation a different name, we have opted to overload this function. This would avoid writing extra logic in order to figure out which function to call based on the parameter type. Note that the parameters of the methods are left out. This can be done when its not ambiguous which method we are referring to.

Python

```
>>> from Box import *
>>> from getMax import *
>>>
>>> getMax(3,4)
4
>>>
>>> box1 = Box_Int(0, 0, 5, 6)
>>> box2 = Box_Int(0, 0, 6, 6)
>>> print getMax(box1, box2)
Box: [(0, 0), (6, 6)]
>>>
```

4.4 Inheritance

SWIG supports C++ single and multiple inheritance of classes. Here we provide a shape inheritance example that is in part from the SWIG documentation, although slightly modified for the purposes of our examples.

Shape.hxx

```
class Shape {
   public :
      Shape (double x = 0.0, double y = 0.0) ;
      virtual ~Shape() ;

      void setLocation(double x, double y) ;
      double getXLocation() const ;
      double getYLocation() const ;

      virtual double area() const = 0 ;
      virtual double perimeter() const = 0 ;

      void printMe() ;

   private :
      double xLocation ;
```

```
        double yLocation ;
};
```

Circle.hxx

```
#include <iostream>
#include "Shape.hxx"

class Circle : public Shape {
   public :
      Circle (double r, double x = 0, double y = 0) ;
      virtual ~Circle() ;

      void setRadius(double r) ;
      double getRadius() const ;

      virtual double area () const ;
      virtual double perimeter() const ;

      friend std::ostream& operator<<(std::ostream& os, const Circle& c) ;

   private :
      double radius ;
};
```

Shape.i

```
%module Shape
%{
#include "Shape.hxx"
%}
%include "Shape.hxx"
```

Thus far we have used the quick form of creating SWIG interface files. For the `Circle` interface file, we will provide both methods side by side. On the left side is the quick form we have used thus far. On the right side, instead of having SWIG obtain the type declaration of `Circle` from the header file, we have provided the declaration with in the interface file. Although not evident with a small example such as this, providing the declaration in the interface file gives the developer a greater control over what SWIG generates. Obviously, more work is required and it reduces the level of automation in creating the interface files. This is not to say the developer has no control by including the header file. SWIG provides a wide range of directives that control all aspects of code generation. In fact, when creating Python bindings for Sapphire we let SWIG obtain the declarations from the header file and we use its customization features for any necessary modifications.

Circle.i

```
%module Circle
%{
#include "Circle.hxx"
%}

%ignore operator<< ;

%import "Shape.i"
```

Circle.i

```
%module Circle
%{
#include "Circle.hxx"
%}


%import "Shape.i"
```

```
%include "Circle.hxx"
```

```
class Circle : public Shape {
   public :
      Circle (double, double, double) ;
      ~Circle() ;

      void setRadius(double) ;
      double getRadius() const ;

      double area() const ;
      double perimeter() const ;
};
```

The only new information to note here is the use of the `%import` directive. It is used to obtain type declarations from another interface file with out generating any wrapping code. In this case, it is used in the `Circle` interface file to obtain the declaration of the `Shape` class. Once wrapped, we can use the `Circle` class in Python as follows:

Python

```
>>> from Circle import *
>>>
>>> c = Circle(4)
>>> c.setLocation(1,3)
>>>
>>> print c.getXLocation(), c.getYLocation()
1.0 3.0
>>>
>>> c.printMe()
Location   (1, 3)
Area       50.2655
Perimeter  25.1327
>>>
```

The `printMe` method from the `Shape` class, prints the location of the shape and calls the `area` and `perimeter` methods of the derived class in order to get the respective values.

4.5 Cross Language Polymorphism

SWIG allows for cross language polymorphism through the directors directive. Directors allow virtual method calls from C++ to reach the bottom of the inheritance chain even if that is a Python implementation. By default the director feature is disabled. To enable directors for the `Shape` class, we have to modify the interface file as follows:

Shape.i

```
%module(directors="1") Shape
%{
#include "Shape.hxx"
%}

%feature("director") Shape ;

%include "Shape.hxx"
```

The module statement, `%module(directors="1") Shape`, enables directors for the `Shape` class. Although directors are enabled we must still tell SWIG for which classes and methods it should create directors for. This is done using the `%feature` directive as shown in the interface file. Here we have chosen to create directors for all

virtual functions in the `Shape` class. One can also select specific methods instead of globally allowing directors for all virtual methods as follows:

Shape.i

```
...
%feature("director") Shape::area ;
%feature("director") Shape::perimeter ;
...
```

The above will create directors for the `area` and `perimeter` methods of class `Shape`. Now, one can derive from the `Shape` class in Python and provide a new definition for the virtual methods. In the following, we derive a `Square` class in Python from the wrapped `Shape` class. Note the `printMe` method at the end of our example, with out the use of directors the `Shape` class method would not have had access to the Python derived class methods.

Python

```
>>> from Shape import *
>>>
>>> class Square(Shape):
...    def __init__(self, s, x = 0, y = 0):
...       Shape.__init__(self, x, y)
...       self.side = s
...
...    def area(self):
...       return self.side * self.side
...
...    def perimeter(self):
...       return 4 * self.side
...
>>>
>>> s = Square(3, 4, 5)
>>>
>>> s.getXLocation(), s.getYLocation()
(4.0, 5.0)
>>> s.area()
9
>>> s.perimeter()
12
>>> s.printMe()
Location   (4, 5)
Area       9
Perimeter  12
>>>
```

4.6 STL

SWIG also has STL support, including vectors, maps, lists, strings and more. It will convert STL types to their most natural version in the target language. In this section we illustrate how to wrap the STL vector and modify the `Shape` class in order to show its use. First we will change the location accessor and mutator methods of the `Shape` class.

Shape.hxx

```
#include <vector>
```

```
class Shape {
   public :
      ...
      void setLocation(const std::vector<double>& loc) ;

      std::vector<double> getLocation() const ;
      ...
};
```

Shape.i

```
...
%include "std_vector.i"

%template (vector_Double) std::vector<double> ;
...
```

In the `Shape` class we have added another `setLocation` method which takes a vector of doubles as input. Although the use of vectors here is an overkill, we use it here for the purposes of the example. Another method, `getLocation` is also added that returns the `x` and `y` coordinate locations in a vector. In the interface file we have included the SWIG interface file `std_vector.i` and a template instantiation for vector of doubles.

Python

```
>>> from Circle import *
>>>
>>> c = Circle(3)
>>> c.getLocation()
(0.0, 0.0)
>>>
>>> c.setLocation([4,5])
>>> c.getLocation()
(4.0, 5.0)
>>>
>>> loc = Shape.vector_Double(2)
>>> loc[0] = 6
>>> loc[1] = 7
>>> c.setLocation(loc)
>>> c.printMe()
Location   (6, 7)
Area       28.2743
Perimeter  18.8496
```

As we can see from above, one can use a Python tuple or list as well as a SWIG wrapped STL vector as input to `setLocation` method. However, if the method changes the vector in place we will have to use the wrapped vector. More information on the use of STL is available in the SWIG documentation as well as the examples with the source code.

5 Python libraries usage

5.1 Overview

Using SWIG we have been able to provide a Python interface to the Sapphire Scientific Data Mining Software. We call the Python wrapped version of Sapphire, PySapphire. In this section we provide some information on how to use the PySapphire libraries and its difference with the C++ version of Sapphire.

To begin coding applications using PySapphire, the directory containing the PySapphire libraries should be added to the `PYTHONPATH` environment variable.

5.2 Coding Differences

There are both inherent differences in coding C++ and Python as well as differences due to wrapper implementation. Inherent differences such as importing modules verses including libraries are common to Python users. This section lists differences that were observed during testing of the PySapphire libraries.

Import vs. Include

In the Sapphire software, there are many templated classes. To gain access to the various types in Python the modules can be loaded using a `from` statement. For example:

Python	C++
`from di_RegData2D import *`	`#include ''di_RegData2D.hxx''`
`source = di_RegData2D_Float()`	`di_RegData2D<float> source;`

The `from` statement imports all the templates of the class. The module name is the same as the C++ class name. Other than specifying the type in the extension of the class name, there are few differences in coding a Sapphire application with Python.

Types

The Sapphire software has many templated classes. We have used a particular scheme for the template instantiation names used in Python. The instantiation names are created by adding a postfix to the Sapphire class name based on the type the class is templated upon. There are four categories of types which are handled by this naming scheme, 1) single word C++ built-in types, 2) double word C++ built-in types, 3) C++ STL types or Sapphire classes, 4) a pointer to any one of the previous three types. In each of these cases the postfix starts with an underscore and uses the following rules. For the single word C++ built-in types, such as, `int`, `short`, `char`, `double`, `float`, and `bool`, the postfix extension is the type with the first letter capitalized. As for the `unsigned int`, `unsigned char`, `unsigned long`, and `long int`, these extensions have the first letter of the first word capitalized, followed by the second word with the first letter capitalized. If the type is a C++ STL template class or Sapphire class, the postfix is the class name. A Sapphire class template which itself is templated, follows the same rules, as seen in the example of `tbox_Array2D_tbox_Vector2D_Int`. The last category of types, pointers, are handled by adding `Ptr` to postfix that follows the above listed rules. Example extensions for the various types follows:

Category	Python	C++
1	`tbox_Array2D_Int`	`tbox_Array2D<int>`
1	`di_RegData2D_Short`	`di_RegData2D<short>`
1	`tbox_Array2D_Char`	`tbox_Array2D<char>`
1	`tbox_Array2D_Double`	`tbox_Array2D<double>`
1	`tbox_Array2D_Float`	`tbox_Array2D<float>`
2	`tbox_Array2D_UInt`	`tbox_Array2D<unsigned int>`
2	`di_RegData2D_UChar`	`di_RegData2D<unsigned char>`
2	`tbox_Array2D_ULong`	`tbox_Array2D<unsigned long>`
2	`di_RegData2D_LInt`	`di_RegData2D<long int>`
3	`vector_string`	`vector<string>`
3	`tbox_Array2D_tbox_Histogram1D`	`tbox_Array2D<tbox_Histogram1D>`
3	`tbox_Array2D_tbox_Vector2D_Int`	`tbox_Array2D<tbox_Vector2D<int> >`

Category	Python	C++
4	tbox_Array2D_IntPtr	tbox_Array2D<int *>
4	tbox_Array2D_UIntPtr	tbox_Array2D<unsigned int *>
4	tbox_Array2D_tbox_Histogram1DPtr	tbox_Array2D<tbox_Histogram1D *>
4	tbox_Array2D_tbox_Vector2D_IntPtr	tbox_Array2D<tbox_Vector2D<int *> >
4	tbox_Array2D_tbox_Vector2DPtr_Int	tbox_Array2D<tbox_Vector2D<int> *>
4	tbox_Array2D_tbox_Vector2DPtr_IntPtr	tbox_Array2D<tbox_Vector2D<int *> *>

This scheme is used because its more natural when read from left to right. It does look complicated when it comes to the nested template pointer version in the fourth category shown above. Note the location of `Ptr` in the last three cases, this way we can disambiguate between different positions of the pointer.

Assignment vs. Declaration

In C++, variables must be declared before use, while in Python, variables must be assigned before use. For example:

Python	C++
num = 1	int addone, num = 1;
addone = num + 1	addone = num + 1;

In this example, the type of `addone` and `num` need to be declared in C++ before assignment. In Python, no type is specified, `addone` and `num` can be directly assigned. In most cases assignments in Sapphire specify a deep copy of the object, in PySapphire an assignment is a reference copy, thus points to the same object.

Vectors and Arrays

STL vectors are used in Sapphire as input and output of various methods. Vectors are implemented in the Python wrapper by extending vector with the type of the vector as in the example above. The vector module does not need to be imported it is incorporated in the wrapper. An example of usage:

Python	C++
...	...
texture = sdp_FeaExtTextureWD2D_Float(...)	texture = sdp_FeaExtTextureWD2D<float>(...);
...	...
feature_names = vector_string()	vector<string> feature_names;
texture.getFeatureNames(feature_names)	texture.getFeatureNames(feature_names);

Arrays are also available, but need to be imported to use. The array module can also be used to access data which are pointers in Sapphire, described in the next section. SWIG does provide a wrapped array, this array however will only work for arrays generated in Python given a certain size. Equally important we needed to access arrays returned via a pointer with out creating any new memory. To that end we wrapped our own array. The array is used as follows:

Python	C++
from array import *	
arr = array_Int(10)	int arr[10];
arr[0] = 1	arr[0] = 1;

Pointers

There are no pointers in Python, therefore, to access the data the array module from the previous example is used. When given a pointer, instead of an initial size, the array will not create its own copy, rather advance

through the given address location. Note that there is no bound checking for this array object and it will delete allocated memory only if it was the creator. Example:

Python	C++
`from di_RegData2D import *`	`#include ‘‘di_RegData2D.hxx’’`
`from array import *`	
`tempReg = di_RegData2D_Int()`	`di_RegData2D<int> tempReg;`
`...`	`...`
`base = tempReg.getBaseAddress()`	`int *base_ptr = (int *) tempReg.getBaseAddress();`
`base_arr = array_Int(base)`	
`base_arr[0] = 1`	`base_ptr[0] = 1;`

Printing

Objects in Sapphire which have a `print` method can be print to stdout in Python using the print statement. This is accomplished by having a string representation of the object. An example of printing to stdout:

Python	C++
`from di_RegData2D import *`	`#include ‘‘di_RegData2D.hxx’’`
`tempReg = di_RegData2D_Int()`	`di_RegData2D<int> tempReg;`
`...`	`...`
`print tempReg`	`tempReg.print();`

`None` vs. `NULL`

`None` is an empty place holder in Python, similar to `NULL` in C++.

`True/False` vs. `true/false`

`True/False`, in Python, is equivalent to `true/false`, in C++.

6 Comparison of Python to C++

In order to compare the differences between Python and C++, we implemented the same functionality in Python and C++. We chose to implement a background subtraction method (sdp_BGModelMedian2D). Background subtraction is commonly used in image analysis algorithms to remove or reduce areas that are not of interest. Our implementation applies the method to various sizes of image sequences. The sample codes of our implementation in both Python and C++ are provided along with process timing results.

6.1 Python Code:

```
#!/usr/apps/python/2.4.3/bin/python

from di_RegData2D import *
from util_Input import *

from sdp_BGModelMedian2D import *
from sdp_ThresholdingMaxInterVar2D import *

import sys, os

if len(sys.argv) < 3:
    print ("usage: test_BGModelMedian2D.py <input format> <start>:<inc>:<end>")
```

```
    print ("example: test_BGModelMedian2D.py image%04d.fits 1:1:300")
    sys.exit(-1)

input_form = sys.argv[1]
pattern = sys.argv[2]

sp_idx = pattern.find(":")

start = int(pattern[0:sp_idx])
pattern = pattern[sp_idx+1:len(pattern)]

sp_idx = pattern.find(":")
inc = int(pattern[0:sp_idx])
pattern = pattern[sp_idx+1:len(pattern)]

end = int(pattern)

threshold = sdp_ThresholdingMaxInterVar2D_Float(20)

bgmodel = sdp_BGModelMedian2D_Float(threshold)
bgmodel.resetModel()

source = di_RegData2D_Float()

for framenum in range(start,end+1,inc):
    filename = input_form % framenum
    uInput = util_Input(filename)
    uInput.get(source)

    if (not(bgmodel.isModelReady())):
        bgmodel.initModel(source)
        continue
    else:
        fgmask = di_RegData2D_Int()
        bgmodel.apply(source, fgmask)
        backgnd = bgmodel.getBGImage()

    foregnd = di_RegData2D_Float()
    foregnd = source - backgnd

    bgndfile = "./bkgnd_results/py_backgnd%04d.fits" % framenum
    os.system("rm -f " + bgndfile)
    backgnd.createFits(bgndfile)

    fgndfile = "./bkgnd_results/py_foregnd%04d.fits" % framenum
    os.system("rm -f " + fgndfile)
    foregnd.createFits(fgndfile)
```

6.2 C++ Code:

```
#include <stdio.h>
#include <math.h>
```

```
#include <stdlib.h>
#include <iostream>
#include <sstream>
#include <vector>
#include <string>

#include "Sapphire_config.hxx"
#include "di_FITSData.hxx"
#include "di_RegData2D.hxx"
#include "util_Input.hxx"
#include "sdp_BGModelMedian2D.hxx"
#include "sdp_ThresholdingMaxInterVar2D.hxx"

using namespace std;
using namespace Sapphire;

int main(int argc, char* argv[])
{
   if(argc < 3)
   {
      cerr << "usage: test_BGModelMedian2D <input format> <start>:<inc>:<end>" << endl;
      cerr << "example: test_BGModelMedian2D image%04d.fits 1:1:300" << endl;
      exit(-1);
   }

   di_RegData2D<float> in;

   string input_form = argv[1];

   string pattern = argv[2];
   istringstream str_pattern(pattern);

   char tok[5];

   int start, inc, end;


   str_pattern.getline(tok, 5, ':');
   start = atoi(tok);

   str_pattern.getline(tok, 5, ':');
   inc = atoi(tok);

   str_pattern.getline(tok, 5);
   end = atoi(tok);

   char filename[128], cmd[128];

   sdp_Thresholding2D<float> *threshold = new sdp_ThresholdingMaxInterVar2D<float>(20);

   sdp_BGModelMedian2D<float> bgmodel(threshold);
   bgmodel.resetModel();
```

```
di_RegData2D<float> source, backgnd, foregnd;


for (int framenum = start; framenum <= end; framenum++)
{
   sprintf(filename, input_form.c_str(), framenum);

   util_Input input(filename);

   input.get(source);

   if (!(bgmodel.isModelReady()))
   {
      bgmodel.initModel(source);
      continue;
   }
   else
   {
      di_RegData2D<int> fgmask;
      bgmodel.apply(source, fgmask);
      backgnd = bgmodel.getBGImage();
   }

   foregnd = source - backgnd;

   sprintf(filename, "./bkgnd_results/c_backgnd%04d.fits", framenum);
   sprintf(cmd, "rm -f %s", filename);
   system(cmd);
   backgnd.createFits(filename);


   sprintf(filename, "./bkgnd_results/c_foregnd%04d.fits", framenum);
   sprintf(cmd, "rm -f %s", filename);
   system(cmd);

   foregnd.createFits(filename);

}
return(1);
```

6.3 Timing Efficiency

C++ is a compiled language meaning the C++ code is processed to produce an executable for the specific machine. This results in fast program execution. But, Python is an interpreted language, where the executable is designed to run on any machine. As a result, Python code does not take advantage of any specific machine configuration and therefore, program execution is relatively slow compared to compiled code. We examined the process timing of the background subtraction method implemented in both C++ and Python. Figure 1 shows the process timing of both implementations for various sizes of image sequences, where the y-axis is a log scale. As expected, the C++ implementation is significantly (order of magnitude) faster than the Python implementation.

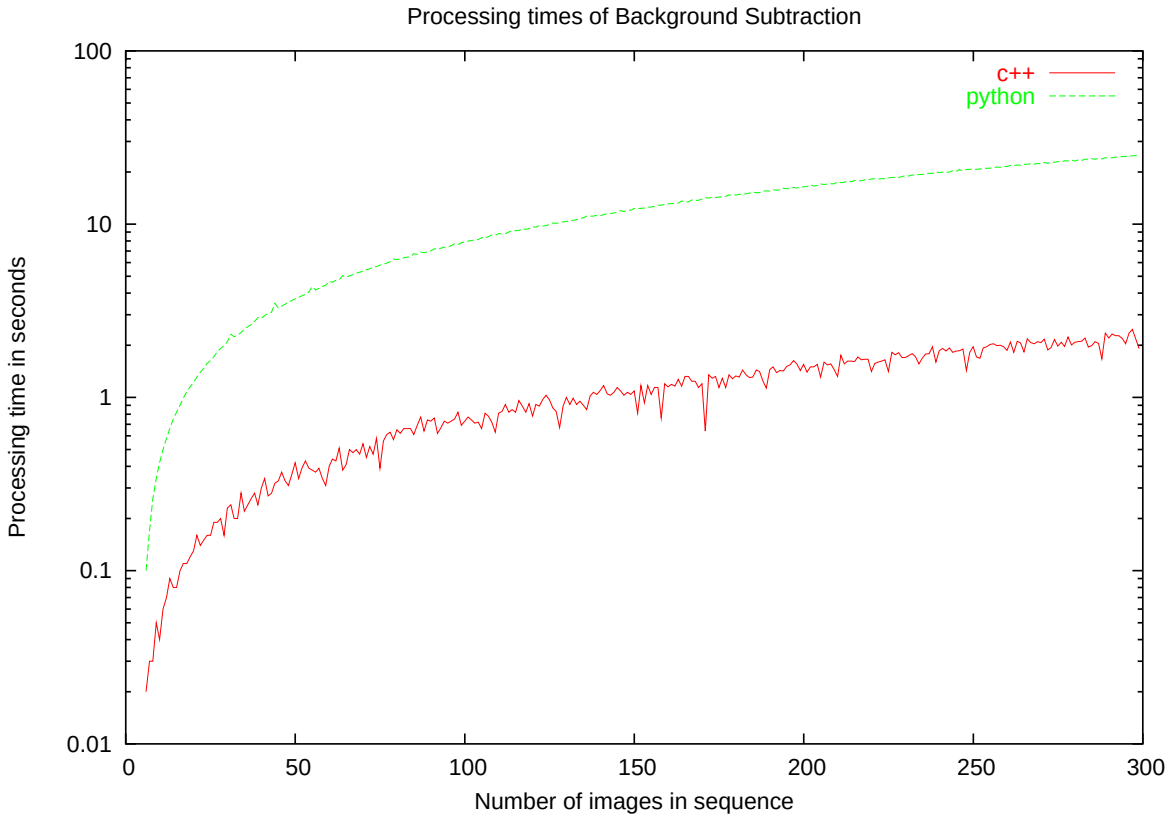

Figure 1: Process timing for a background subtraction method (sdp_BGModelMedian2D), various size image sequences were tested.

7 Conclusions

The purpose of this project, as stated above was to expose a selective set of Sapphire's functionality through a Python interface, in order to increase its usability. To that end, we have done a literary review and some experimentation of the four interoperability tools Babel, Boost.Python, SWIG and SIP. All these tools have been successfully used in many projects. The purpose of our review and evaluation was not to find the best of the four, but to identify the best suited for wrapping the Sapphire libraries. We chose SWIG for this purpose, as it supports a breadth of C and C++ functionality, is well documented, has a large user base, is easy to use, supports multiple target languages, and has a greater level of automation, compared to the other tools, for generating the SWIG interface files and the makefiles. As a result, we were able to generate Python binding not just for a select few of Sapphire's functionality, rather we were able to do so for all of Sapphire.

8 Acknowledgements

We thank the Computational Directorate Technology Base project for funding this project. This work was performed under the auspices of the U.S. Department of Energy by the University of California Lawrence Livermore National Laboratory under Contract W-7405-Eng-48. UCRL-TR-??????

References

[1] Babel, `https://computation.llnl.gov/casc/components/babel.html`.

[2] Tamara Dahlgren, Thomas Epperly, Gary Kumfert and James Leek. "Babel Users' Guide," November 2, 2007, UCRL-SM-230026.

[3] David Abrahams and Ralf W. Grosse-Kunstleve, "Building Hybrid Systems with Boost.Python," March 19, 2003.

[4] Python.Boost, `http://www.boost.org/libs/python/doc/`.

[5] Py++, `http://www.language-binding.net/pyplusplus/pyplusplus.html`.

[6] Simplified Wrapper and Interface Generator, `www.swig.org`.

[7] David M. Beazley, "Why Extension Programmers Should Stop Worrying About Parsing and Start Thinking About Type Systems," November 9, 2002.

[8] SIP, `http://www.riverbankcomputing.co.uk/sip/`.

[9] Jonathan Riddell, "Phil Thompson Talks About PyQt," August 8, 2006. `http://dot.kde.org/1155075248`.