An Extensible Compiler for Creating Scriptable Scientific Software

David M. Beazley

University of Chicago, Chicago IL 60637, USA

Abstract. Scripting languages such as Python and Tcl have become a powerful tool for the construction of flexible scientific software because they provide scientists with an interpreted problem solving environment and they form a modular framework for controlling software components written in C,C++, and Fortran. However, a common problem faced by the developers of a scripted scientific application is that of integrating compiled code with a high-level interpreter. This paper describes SWIG, an extensible compiler that automates the task of integrating compiled code with scripting language interpreters. SWIG requires no modifications to existing code and can create bindings for eight different target languages including Python, Perl, Tcl, Ruby, Guile, and Java. By automating language integration, SWIG enables scientists to use scripting languages at all stages of software development and allows existing software to be more easily integrated into a scripting environment.

1 Introduction

One of the most difficult tasks faced by the developers of scientific software is figuring out how to make high-performance programs that are easy to use, flexible, and extensible. Clearly these goals are desirable if scientists want to focus their efforts on solving scientific problems instead of problems related to software engineering. However, the task of writing such software is usually quite difficult—in fact, much more difficult than most software engineers are willing to admit.

An increasingly popular solution to the scientific software problem is to use scripting languages such as Python or Tcl as a high-level steering language for software written in C, C++, or Fortran [8, 10, 11, 14, 16, 19]. Scripting languages provide a nice programmable user interface that is interactive and which allows more complicated tasks to be described by collections of scripts. In addition, scripting languages provide a framework for building loosely-coupled software components and gluing those components together [13]. This makes it easier to incorporate data analysis, visualization, and other data management facilities into an application without having to create a huge monolithic framework. Scripting languages are also portable and easy to utilize on high performance computing systems including supercomputers and clusters.

In addition to these benefits, the appeal of scripting languages is driven by the piecemeal software development process that characterizes a lot of scientific

software projects. Rarely do scientists set out to create a hyper-generalized program for solving everything. Instead, programs are created to solve a specific problem of interest. Later, if a program proves to be useful, it may be gradually adapted and extended with new features in order to solve closely related problems. Although there might be some notion of software design, a lot of scientific software is developed in an ad-hoc manner where features are added as they are needed as opposed to being part of a formal design specification. Scripting languages are a good match for this style of development because they can be used effectively even when the underlying software is messy, incomplete, lacking in formal design, or under continual development.

Some scientists also view scripting languages as a logical next step in the user interface of their software. For example, when programs are first developed, they are often simple batch jobs that rely upon command line options or files of input parameters. As the program grows, scientists usually want more flexibility to configure the problem so they may modify the program to ask the user a series of questions. They may even write a simple command interpreter for setting up parameters. Scripting languages build upon this by providing an interface in the form of a fully-featured programming language that provides features similar to that found in commercial scientific software such as MATLAB, Mathematica, or IDL. In fact, many scientists add an interpreter to their application just so they can obtain comparable flexibility.

2 The Problem With Scripting

All modern scripting languages allow foreign code to be accessed through a special extension API. However, to provide hooks to existing C, C++, or Fortran code, you usually have to write special wrapper functions. The role of these functions is to convert arguments and return values between the data representation in each language. For example, if a programmer wanted to access the cosine function in the math library from Python, they would write a wrapper like this:

```
PyObject *wrap_cos(PyObject *self, PyObject *args) {
  double x, result;
  if (!PyArg_ParseTuple(args,"d",&x)) return NULL;
  result = cos(x);
  return Py_BuildValue("d",result);
}
```

Even though it's not very difficult to write a few dozen wrappers, the task becomes tedious if an application contains several hundred functions. Moreover, the task becomes considerably more difficult if an application makes use of advanced programming features such as pointers, arrays, classes, inheritance, templates, and overloaded operators because there is often no obvious way to map such features to the scripting language interpreter.

As a result, it is difficult to integrate existing software into a scripting environment without a considerable coding effort. Scientists may also be reluctant

to use scripting languages in the early stages of program development since it will be too difficult to keep the wrapper code synchronized with changes in the underlying application. Scientists may also take the drastic step of writing all new software as a hand-written extension of a particular scripting language interpreter. Although this works, it entangles the implementation of the application with the implementation of the interpreter and makes it difficult to reuse the code as a library or to integrate the system into other application frameworks. This arguably defeats the whole point of creating reusable and modular software.

3 SWIG: A Compiler for Extensions

One way to simplify the use of scripting languages is to automatically generate wrapper code using a code generation tool. SWIG (Simplified Wrapper and Interface Generator) is a special purpose compiler that has been developed for this purpose [3]. Originally developed in 1995, SWIG was first used to build scripting language bindings for the SPaSM short-range molecular dynamics code at Los Alamos National Laboratory [4]. This was one of the first large-scale parallel applications to utilize Python as a framework for computational steering and integrated data analysis [5]. Since then, SWIG has been developed as a free software project. It is found in many GNU/Linux distributions and has been used in variety of projects ranging from scientific simulations to commercial video games.

The key feature of SWIG is that it allows existing software to be integrated into a scripting environment with few if any code modifications. Furthermore, the system maintains a clean separation between the underlying application and the interpreter. Because of this, it promotes modularity and allows software to be reused in different settings. The other essential feature is that SWIG generates wrappers from standard C/C++ declarations and is fully automated. This allows the system to be incorporated into a project in a minimally intrusive manner and allows scientists to focus on the problem at hand instead of language integration issues.

3.1 A Simple Example

To illustrate the use of SWIG, suppose that a simulation program defines a few C functions like this:

```
int integrate(int nsteps, double dt);
void set_boundary_periodic();
void init_lj(double epsilon, double sigma, double cutoff);
void set_output_path(char *path);
```

To build a scripting language interface, a user simply creates a special file containing SWIG directives (prefaced by a %) and the C++ declarations they would like to wrap. For example:

```
// file: shock.i
%module shock
%{
#include "headers.h"
%}
int integrate(int nsteps, double dt);
void set_boundary_periodic();
void init_lj(double epsilon, double sigma, double cutoff);
void set_output_path(char *path);
```

This file is then processed by SWIG to create an extension module in one of several target languages. For example, creating and using a Python module often works like this:

```
$ swig -python shock.i
$ cc -c shock_wrap.c
$ cc -shared shock_wrap.o $(OBJS) -o shockmodule.so
$ python
Python 2.1 (#1, Jun 13 2001, 16:09:46)
>>> import shock
>>> shock.init_lj(1.0,1.0,2.5)
>>> shock.set_output_path("./Datafiles")
>>> shock.set_boundary_periodic()
```

In this example, a separate file was used to hold SWIG directives and the declarations to be wrapped. However, SWIG can also process raw header files or header files in which conditional compilation has been used to embed special SWIG directives. This makes it easier for scientists to integrate SWIG into their application since existing source code can be used as SWIG input files. It also makes it easier to keep the scripting language interface synchronized with the application since changes to header files (and function/class prototypes) can be tracked and handled as part of the build process.

3.2 Advanced Features

Although a simple example has been shown, SWIG provides support for a variety of advanced programming features including pointers, structures, arrays, classes, exceptions, and simple templates. For example, if a program specifies a class definition like this,

```
class Complex {
   double rpart, ipart;
public:
   Complex(double r = 0, double i = 0): rpart(r), ipart(i) { };
   double real();
   double imag();
   ...
};
```

the resulting scripting interface mirrors the underlying C++ API. For example, in Python, a user would write the following code:

```
w = Complex(3,4)           # Create a complex
a = w.real()               # Call a method
del w                      # Delete
```

When exported to the interpreter, objects are represented as typed pointers. For example, a `Complex *` in the above example might be encoded as a string containing the pointer value and type such as `_100f8ea0_p_Complex`. Type information is used to perform run-time type checking in the generated wrapper code. Type checking follows the same rules as the C++ type system including rules for inheritance, scoping, and typedef. Violations result in a run-time scripting exception.

As applications become more complex, SWIG may need additional input in order to generate wrappers. For example, if a programmer wanted to bind C++ overloaded operators to Python operators, they might specify the following:

```
%rename(__add__) Complex::operator+(const Complex &);
%rename(__sub__) Complex::operator-(const Complex &);
%rename(__neg__) Complex::operator-();
...
class Complex {
...
 Complex operator+(const Complex &c) const;
 Complex operator-(const Complex &c) const;
 Complex operator-() const;
};
```

Similarly, if a program uses templates, information about specific template instantiations along with identifier names to use in the target language must be provided. For example:

```
template<typename T> T max(T a, T b) { return  a>b ? a : b; }
...
%template(maxint)    max<int>;
%template(maxdouble) max<double>;
```

3.3 Support for Legacy Software

Unlike wrapper generation tools designed strictly for object-oriented programming, SWIG provides full support for functions, global variables, constants and other features commonly associated with legacy software. In addition, SWIG is able to repackage procedural libraries into an object-based scripting API. For example, if an application used a procedural API like this,

```
typedef struct {
   double re, im;
} Complex;

Complex add_complex(Complex a, Complex b);
double  real_part(Complex a);
```

the following SWIG interface will create a class-like scripting interface:

```
%addmethods Complex {
   Complex(double r, double i) {
       Complex *c = (Complex *) malloc(sizeof(Complex));
       c->re = r;
       c->im = i;
       return c;
   }
   ~Complex() { free(self); }
   double real() { return real_part(*self); }
   Complex add(Complex b) { return add_complex(*self,b); }
};
```

In this case, the resulting scripting interface works like a class even though no changes were made to the underlying C code.

3.4 Customization Features

For advanced users, it is sometimes desirable to modify SWIG's code generator in order to provide customized integration between the scripting environment and the underlying application. For example, a user might want to interface their code with an array package such as Numeric Python [7]. By default, SWIG does not know how to perform this integration. However, a user can customize the code generator using a typemap. A typemap changes the way that SWIG converts data in wrapper functions. For instance, suppose an application had several functions like this:

```
void settemp(double *grid, int nx, int ny, double temp);
double avgtemp(double *grid, int nx, int ny);
void plottemp(double *grid, int nx, int ny, double mn, double mx);
```

Now suppose that a programmer wanted to pass a Numeric Python array as the `grid` parameter and associated `nx` and `ny` parameters. To do this, a typemap rule such as the following can be inserted into the SWIG interface file:

```
%typemap(in) (double *grid, int nx, int ny) {
   PyArrayObject *array;
   if (!PyArray_Check($input)) {
      PyErr_SetString(PyExc_TypeError,"Expected an array");
```

```
      return NULL;
   }
   array = (PyArrayObject *)
        PyArray_ContiguousFromObject(input, PyArray_DOUBLE, 2, 2);
   if (!array) {
       PyErr_SetString(PyExc_ValueError,
            "array must be two-dimensional and of type float");
       return NULL;
   }
   $1 = (double *) array->data;  /* Assign grid */
   $2 = array->dimensions[0];    /* Assign nx   */
   $3 = array->dimensions[1];    /* Assign ny   */
}
```

When defined, all subsequent occurrences of the argument sequence `double *grid, int nx, int ny` are processed as a single numeric array object. Even though the specification of a typemap requires detailed knowledge of the underlying scripting language API, these rules only need to be defined once in order to be applied to hundreds of different declarations.

To modify the handling of declarations, the `%feature` directive is used. For example, if a programmer wanted to catch a C++ exception in a specific class method and turn it into a Python exception, they might write the following:

```
%feature("except") Object::getitem {
    try { $action } catch (IndexError) {
       PyErr_SetString(PyExc_IndexError, "bad index");
       return NULL;
    }
}
class Object {
   virtual Item *getitem(int index);
};
```

Although `%feature` is somewhat similar to a traditional compiler pragma, it is a lot more powerful. For instance, when features are defined for a class method as shown, that feature is propagated across an entire inheritance hierarchy. Therefore, if `Object` was a base class, the exception handler defined would be applied to any occurrence of `getitem` found in derived classes. Feature declarations can also be parameterized with types. This allows them to be precisely attached to specific methods even when those methods are overloaded. This behavior is illustrated by the `%rename` directive example in section 3.2 (which is really just a special form of `%feature` in disguise).

4 SWIG Internals

One of SWIG's most powerful features is its highly extensible design. C/C++ parsing is implemented using an extended version of the C preprocessor and

a customized C++ parser. These components differ from a traditional implementation due to issues related to mixing special SWIG directives and C++ declarations in the same file. For instance, certain information from the preprocessor is used during code generation and certain syntactic features are parsed differently in order to make SWIG easier to use. SWIG also does not concern itself with parsing function bodies.

Internally, SWIG builds a complete parse tree and provides a traversal API similar to that in the XML-DOM specification. Nodes are built from hash tables that allow nodes to be annotated with arbitrary attributes at any stage of parsing and code generation [2]. This annotation of nodes is the primary mechanism used to implement most of SWIG's customization features.

To generate code, parse tree nodes are handled by a hierarchical sequence of handler functions that may elect to generate wrapper code directly or forward the node to another handler after applying a transformation. The behavior of each target language is defined by providing implementations of selected handler functions in C++ class. Minimally, a language module only needs to implement handlers for generating low-level function and variable wrappers. For example:

```
class MinimalLanguage: public Language {
public:
  void main(int argc, char *argv[]);
  int  top(Node *n);
  int  functionWrapper(Node *n);
  int  constantWrapper(Node *n);
  int  nativeWrapper(Node *n);
};
```

However, in order to provide more advanced wrapping of classes and structures, a language module will generally implement more handlers.

5 Limitations

SWIG is primarily designed to support software development in C and C++. The system can be used to wrap Fortran as long as the Fortran functions are described by C prototypes. SWIG is also not a full C++ compiler. Certain C++ features such as nested classes and namespaces aren't currently supported. Furthermore, features such as overloaded methods, operators, and templates may require the user to supply extra directives (as illustrated in earlier sections). More complicated wrapping problems arise due to C++ libraries that rely almost entirely on generic programming and templates such as the STL or Blitz++ [17, 18]. Although SWIG can be used to wrap programs that use these libraries, providing wrappers to the libraries themselves would be problematic.

6 Related Work

The problem of creating scripting language extension modules has been explored extensively in the scripting language community. Most scripting languages have

tools that can assist in the creation of extension modules. However, few of these tools are designed to target multiple target languages. Scripting language extension building tools can sometimes be found in application frameworks such as Vtk [12]. However, these tend to be tailored to the specific features of the framework and tend to ignore programming features required for them to be more general purpose. A number of tools have been developed specifically for scientific applications. For example, pyfort and f2py provide Python wrappers for Fortran codes and the Boost Python Library provides an interesting alternative to SWIG for creating C++ class wrappers [1, 9, 15].

Work similar to SWIG can also be found in the meta-programming community. For example, the OpenC++ project aims to expose the internals of C++ programs so that tools can use that information for other tasks [6]. Using such information, it might be possible to generate scripting language wrappers in a manner similar to SWIG.

7 Conclusions and Future Work

SWIG is a compiler that simplifies the integration of scripting languages with scientific software. It is particularly well-suited for use with existing software and supports a wide variety of C++ language features. SWIG also promotes modular design by maintaining a clean separation between the scripting language interface and the underlying application code.

By using an automatic code generator such as SWIG, computational scientists will find that it is much easier to utilize scripting languages at all stages of program development. Furthermore, the use of a scripting language environment encourages modular design and allows scientists to more easily construct software that incorporates features such as integrated data analysis, visualization, database management, and networking.

Since its release in 1996, SWIG has been used in hundreds of software projects. Currently, the system supports eight different target languages including Guile, Java, Mzscheme, Perl, PHP, Python, Ruby, and Tcl. Future work is focused on improving the quality of code generation, providing support for more languages, and adding reliability features such as contracts and assertions. More information about SWIG is available at `www.swig.org`.

8 Acknowledgments

Many people have helped with SWIG development. Major contributors to the current implementation include William Fulton, Matthias Köppe, Lyle Johnson, Richard Palmer, Luigi Ballabio, Jason Stewart, Loic Dachary, Harco de Hilster, Thien-Thi Nguyen, Masaki Fukushima, Oleg Tolmatcev, Kevin Butler, John Buckman, Dominique Dumont, David Fletcher, and Gary Holt. SWIG was originally developed in the Theoretical Physics Division at Los Alamos National

Laboratory in collaboration with Peter Lomdahl, Tim Germann, and Brad Holian. Development is currently supported by the Department of Computer Science at the University of Chicago.

References

1. Abrahams, D.: The Boost Python Library. `www.boost.org/libs/python/doc/`.
2. Aho, A., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading, Massachusetts. (1986)
3. Beazley, D.: SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++. In Proceedings of USENIX 4th Tcl/Tk Workshop. (1996) 129-139
4. Beazley, D., Lomdahl, P.: Message-Passing Multi-Cell Molecular Dynamics on the Connection Machine 5. Parallel Computing. **20** (1994) 173-195.
5. Beazley, D., Lomdahl, P.: Lightweight Computational Steering of Very Large Scale Molecular Dynamics Simulations. In Proceeding of Supercomputing'96, IEEE Computer Society. (1996).
6. Chiba, S.: A Metaobject Protocol for C++. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). (1995) 285-299.
7. Dubois, P., Hinsen, K., Hugunin, J.: Numerical Python. Computers in Physics. **10(3)** (1996) 262-267.
8. Dubois, P.: The Future of Scientific Programming. Computers in Physics. **11(2)** (1997) 168-173.
9. Dubois, P.: Climate Data Analysis Software. In Proceedings of 8th International Python Conference. (2000).
10. Gathmann,F.: Python as a Discrete Event Simulation Environment. In Proceedings of the 7th International Python Conference. (1998).
11. Hinsen, K.: The Molecular Modeling Toolkit: A Case Study of a Large Scientific Application in Python. In Proceedings of the 6th International Python Conference. (1997) 29-35.
12. Martin, K.: Automated Wrapping of a C++ Class Library into Tcl. In Proceedings of USENIX 4th Tcl/Tk Workshop. (1996) 141-148.
13. Ousterhout, J.: Scripting: Higher-Level Programming for the 21st Century. IEEE Computer. **31(3)** (1998) 23-30.
14. Owen, M.: An Open-Source Project for Modeling Hydrodynamics in Astrophysical Systems. IEEE Computing in Science and Engineering. **3(6)** (2001) 54-59.
15. Peterson, P., Martins, J., Alonso, J.: Fortran to Python Interface Generator with an application to Aerospace Engineering. In Proceedings of 9th International Python Conference. (2000).
16. Scherer, D., Dubois, P., Sherwood, B.: VPython: 3D Interactive Scientific Graphics for Students. IEEE Computing in Science and Engineering. **2(5)** (2000) 56-62.
17. Stroustrup, B.: The C++ Programming Language, 3rd Ed. Addison-Wesley, Reading, Massachusetts. (1997).
18. Veldhuizen, T.: Arrays in Blitz++. In Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98), Springer-Verlag. (1998).
19. White, R., Greenfield, P.: Using Python to Modernize Astronomical Software. In Proceedings of the 8th International Python Conference. (1999).