AutoWIG Documentation

Release 0.1

P. Fernique, C. Pradal

Jul 03, 2018

Contents

High-level programming languages, such as *Python* and *R*, are popular among scientists. They are concise, readable, lead to rapid development cycles, but suffer from performance drawback compared to compiled language. However, these languages allow to interface *C*, *C++* and *Fortran* code. In this way, most of the scientific packages incorporate compiled scientific libraries to both speed up the code and reuse legacy libraries. While several semi-automatic solutions and tools exist to wrap these compiled libraries, the process of wrapping a large library is cumbersome and time consuming. **AutoWIG** is a *Python* library that wraps automatically compiled libraries into high-level languages. Our approach consists in parsing *C++* code using the **LLVM/Clang** technologies and generating the wrappers using the **Mako** templating engine. Our approach is automatic, extensible, and applies to very complex *C++* libraries, composed of thousands of classes or incorporating modern meta-programming constructs.

CHAPTER 1

Citation

If you use AutoWIG in a scientific publication, we would appreciate citations:

> Fernique P, Pradal C. (2018) AutoWIG: automatic generation of python bindings for C++ libraries. PeerJ Computer Science 4:e149 https://doi.org/10.7717/peerj-cs.149

External Ressources

CHAPTER 2

Installation

2.1 Installation from binaries

In order to ease the installation of the **AutoWIG** software on multiple operating systems, the **Conda** package and environment management system is used. To install **Conda**, please refers to its documentation or follow the installation instructions given on the **StatisKit** documentation. Once **Conda** is installed, you can install **AutoWIG** binaries into a special environment that will be used for wrapper generation by typing the following command line in your terminal:

```
conda create -n autowig python-autowig -c statiskit
```

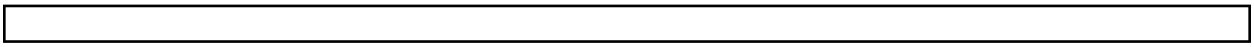
Warning: When compiling wrappers generated by **AutoWIG** in its environment some issues can be encountered at compile time or run time (from within the *Python* interpreter) due to compiler or dependency incompatibilies. This is why it is recommended to install **AutoWIG** in a separate environment that will only be used for wrappers' generation. If the problem persits, please refers to the **StatisKit** documentation concerning the configuration of the development environment.

2.2 Installation from source code

For installing **AutoWIG** from source code, please refers to the **StatisKit** documentation concerning the configuration of the development environment.

Warning: **AutoWIG** and **ClangLite** repositories are considered as submodule of the **StatisKit** repository. To update these repositories and benefit from the last development, you must first go to these submodules and pull the code from the actual repositories. This step, described below, has to be as soon as the **StatisKit** repository is cloned.

```
cd StatisKit
cd share
cd git
cd ClangLite
git pull origin master
```



CHAPTER 3

Documentation

3.1 User guide

Note: In this section, we introduce wrapping problems and how **AutoWIG** aims at minimize developers effort. Basic concepts and conventions are introduced.

3.1.1 Problem setting

Consider a scientist who has designed multiple *C++* libraries for statistical analysis. He would like to distribute his libraries and decide to make them available in *Python* in order to reach a public of statisticians but also less expert scientists such as biologists. Yet, he is not interested in becoming an expert in *C++*/*Python* wrapping, even if it exists classical approaches consisting in writing wrappers with **SWIG** [Bea03] or **Boost.Python** [AG03]. Moreover, he would have serious difficulties to maintain the wrappers, since this semi-automatic process is time consuming and error prone. Instead, he would like to automate the process of generating wrappers in sync with his evolving *C++* libraries. That's what the **AutoWIG** software aspires to achieve.

3.1.2 Automating the process

Building such a system entails achieving some minimal features:

***C++* parsing** In order to automatically expose *C++* components in *Python*, the system requires parsing full legacy code implementing the last *C++* standard. It has also to represent C++ constructs in Python, like namespaces, enumerators, enumerations, variables, functions, classes or aliases.

Documentation The documentation of *C++* components has to be associated automatically to their corresponding *Python* components in order to reduce the redundancy and to keep it up-to-date in only one place.

Pythonic interface To respect the *Python* philosophy, *C++* language patterns need to be consistently translated into *Python*. Some syntax or design patterns in *C++* code are specific and need to be adapted in order to obtain a functional *Python* package. Note that this is particularly sensible for *C++* operators (e.g. `()`, `<`, `[]`) and

corresponding *Python* special functions (e.g. `__call__`, `__lt__`, `__getitem__`, `__setitem__`) or for object serialization.

Memory management *C++* libraries expose in their interfaces either raw pointers, shared pointers or references, while *Python* handles memory allocation and garbage collection automatically. The concepts of pointer or references are thus not meaningful in *Python*. These language differences entail several problems in the memory management of *C++* components into *Python*. A special attention is therefore required for dealing with references (`&`) and pointers (`*`) that are highly used in *C++*.

Error management *C++* exceptions need to be consistently managed in *Python*. *Python* doesn't have the necessary equipment to properly unwind the *C++* stack when exception are thrown. It is therefore important to make sure that exceptions thrown by *C++* code do not pass into the *Python* interpreter core. All *C++* exceptions thrown by wrappers must therefore be translated into *Python* errors. This translation must preserve exception names and contents in order to raise informative *Python* errors.

Dependency management between components The management of multiple dependencies between *C++* libraries with *Python* bindings is required at run-time from *Python*. *C++* libraries tends to have dependencies. For instance the *C++* **Standard Template Library** containers [PLMS00] are used in many *C++* libraries (e.g `std::vector`, `std::set`). For such cases, it doesn't seem relevant that every wrapped *C++* library contains wrappers for usual **STL** containers (e.g. `std::vector< double >`, `std::set< int >`). Moreover, loading in the *Python* interpreter multiple compiled libraries sharing different wrappers from same *C++* components could lead to serious side effects. It is therefore required that dependencies across different library bindings can be handled automatically.

3.2 Frequently Asked Questions

Note: Frequently asked questions about the project and contributing.

CHAPTER 4

Tutorials

CHAPTER 5

Authors

- Pierre Fernique
- Christophe Pradal