AutoWIG Documentation

Release 0.1

P. Fernique, C. Pradal

May 31, 2017

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.

Summary

Status

License see *License* section

Authors see *Authors* section

CHAPTER 1

Installation

1.1 Test it with Docker

Note: Docker [Mer14] is an open-source project that automates the deployment of Linux applications inside software containers.

We provide **Docker** images to enable to run **AutoWIG** on various platforms (in particular Windows and MacOS). For the installation of **Docker**, please refers to its documentation. Then, you can use the `statiskit/autowig` **Docker** image to run **AutoWIG**:

```
$ docker run -i -t -p 8888:8888 statiskit/autowig
```

A list of all available images can be found here. The image tagged `latest` is unstable, it could be preferable to use the one attached with the AutoWIG paper submitted in Journal of Computational Science (tagged `v1.0.0`) as follows:

```
$ docker run -i -t -p 8888:8888 statiskit/autowig:v1.0.0
```

For convenience, examples are presented in **Jupyter** notebooks. You can therefore proceed – in the container's terminal – as follows to run examples:

1. Launch the Jupyter notebook with the following command

   ```
   $ jupyter notebook --ip='*' --port=8888 --no-browser
   ```

2. Copy the URL given in the container's terminal and paste it in your browser. This URL should looks like `http://localhost:8888/?token=/[0-9a-fA-F]+/`.

3. Click on the notebooks you want to run (denoted by `*.ipynb`) and then click on Run All item of the Cell top menu bar.

Warning: For some systems as Ubuntu, **Docker** requires root permissions (see this page for more information).

1.2 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** 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 python-clanglite python-scons python-dev␣
↪libdev -c statiskit -c conda-forge
```

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 the wrappers generation. If the problem persits, please refers to the **StatisKit** documentation concerning the configuration of the development environment.

1.3 Installation from source code

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

CHAPTER 2

Documentation

2.1 User guide

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

2.1.1 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.

2.1.2 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.

2.2 Examples

Note: In the following section, we present some examples using **AutoWIG** in order to emphasize particular aspects of the wrapping process. This examples can be executed on-line on a Binder server. They can be found in the *doc/examples* directory and are recognizable by their `*.ipynb` extension.

Warning: Binder does not provide webhooks that could allow to rebuild the **Docker** image at each changes. In order to able to run these examples, it is possible that you need to rebuild the **Docker** image. For this, go to the Binder status page and click on on the *rebuild* button.

Here are the pre-executed examples:

2.2.1 Wrapping a basic library

We here aim at presenting the interactive wrapping workflow. For the sake of simplicity, we consider a basic example of *C++* library.

First, import **AutoWIG**.

```
In [ ]: import autowig
```

Then, to install and compile the *C++* library we use available **Conda** recipes.

```
In [ ]: !conda remove libbasic -y
        !conda build -q basic/conda/libbasic -c statiskit
        !conda install -y -q libbasic --use-local -c statiskit
```

Once the headers have been installed in the system, we parse them with relevant compilation flags.

```
In [ ]: %%time
        import sys
        asg = autowig.AbstractSemanticGraph()
        asg = autowig.parser(asg, [sys.prefix + '/include/basic/overload.h',
                                   sys.prefix + '/include/basic/binomial.h'],
                                  ['-x', 'c++', '-std=c++11'],
                                  silent = True)
```

Since most of **AutoWIG** guidelines are respected, the `default controller` implementation is suitable.

```
In [ ]: %%time
        autowig.controller.plugin = 'default'
        asg = autowig.controller(asg)
```

In order to wrap the library we need to select the `boost_python_internal generator` implementation.

```
In [ ]: %%time
        autowig.generator.plugin = 'boost_python_internal'
        wrappers = autowig.generator(asg,
                                     module = 'basic/src/py/_basic.cpp',
                                     decorator = 'basic/src/py/basic/_basic.py',
                                     prefix = 'wrapper_')
```

The wrappers are only generated in-memory. It is therefore needed to write them on the disk to complete the process.

```
In [ ]: %%time
        wrappers.write()
```

Here is an example of the generated wrappers. We here present the wrappers for the `BinomialDistribution` class.

```
In [ ]: !pygmentize basic/src/py/wrapper_4046a8421fe9587c9dfbc97778162c7d.cpp
```

Once the wrappers are written on disk, we need to compile and install the *Python* bindings.

```
In [ ]: !conda build -q basic/conda/python-basic -c statiskit
        !conda install -y -q python-basic --use-local -c statiskit --force
```

Finally, we can hereafter use the *C++* library in the *Python* interpreter.

```
In [ ]: import basic
        binomial = basic.BinomialDistribution(1, .5)
        binomial

In [ ]: binomial.pmf(0)

In [ ]: binomial.pmf(1)

In [ ]: binomial.n = 0
        binomial

In [ ]: binomial.pmf(0)

In [ ]: try:
            binomial.set_pi(1.1)
        except basic.ProbabilityError as error:
            print error.message
        else:
            raise Exception('A `basic.ProbabilityError` should have been raise')
```

2.2.2 Wrapping a subset of a very large library

Sometimes, for a very large library, only a subset of available *C++* components is useful for end-users. Wrapping such libraries therefore requires **AutoWIG** to be able to consider only a subset of the *C++* components during the

`Generate` step. The **Clang** library is a complete *C/C++* compiler. **Clang** is a great tool, but its stable *Python* interface (i.e. **libclang**) is lacking some useful features that are needed by **AutoWIG**. In particular, class template specializations are not available in the abstract syntax tree. Fortunately, most of the classes that would be needed during the traversal of the *C++* abstract syntax tree are not template specializations. We therefore proposed to bootstrap the **Clang** *Python* bindings using the `libclang parser` of **AutoWIG**. This new **Clang** *Python* interface is called **PyClangLite** and is able to parse class template specializations. As for **libclang**, this interface is proposed only for a subset of the **Clang** library sufficient enough for proposing the new `pyclanglite parser`.

This repository already has wrappers, we therefore need to remove them.

```
In [ ]: !git clone https://github.com/StatisKit/ClangLite ClangLite
        !git -C ClangLite checkout a13322e37683012ca346595e88abc48ac591112c
```

```
In [ ]: from path import Path
        import shutil
        srcdir = Path('ClangLite')/'src'/'py'
        for wrapper in srcdir.walkfiles('*.cpp'):
            wrapper.unlink()
        for wrapper in srcdir.walkfiles('*.h'):
            wrapper.unlink()
        wrapper = srcdir/'clanglite'/'_clanglite.py'
        if wrapper.exists():
            wrapper.unlink()
        blddir = srcdir.parent.parent/'build'
        if blddir.exists():
            shutil.rmtree(srcdir.parent.parent/'build')
```

In addition to the **Clang** libraries, the **ClangLite** library is needed in order to have access to some functionalities. The `tool.h` header of this **ClangLite** library includes all necessary **Clang** headers. This library is installed using the **SCons** `cpp` target.

```
In [ ]: !conda remove libclanglite -y
        !conda build ClangLite/conda/libclanglite -c statiskit -c conda-forge
        !conda install -y libclanglite --use-local -c statiskit -c conda-forge
```

Once these preliminaries done, we can proceed to the actual generation of wrappers for the **Clang** library. For this, we import **AutoWIG** and create an empty Abstract Semantic Graph (ASG).

```
In [ ]: import autowig
        asg = autowig.AbstractSemanticGraph()
```

We then parse the `tool.h` header of the **ClangLite** library with relevant compilation flags.

```
In [ ]: %%time
        import sys
        prefix = Path(sys.prefix).abspath()
        autowig.parser.plugin = 'libclang'
        asg = autowig.parser(asg, [prefix/'include'/'clanglite'/'tool.h'],
                       flags = ['-x', 'c++', '-std=c++11',
                                '-D__STDC_LIMIT_MACROS',
                                '-D__STDC_CONSTANT_MACROS',
                                '-I' + str((prefix/'include').abspath())],
                       libpath = prefix/'lib'/'libclang.so',
                       bootstrap = False,
                       silent = True)
```

Since most of **AutoWIG** guidelines are respected in the **Clang** library, the `default controller` implementation could be suitable. Nevertheless, we need to force some *C++* components to be wrapped or not. We therefore implements a new `controller`.

```
In [ ]: def clanglite_controller(asg):
```

```
for node in asg['::boost::python'].classes(nested = True):
    node.is_copyable = True

for node in asg.classes():
    node.boost_python_export = False
for node in asg.functions(free=True):
    node.boost_python_export = False
for node in asg.variables(free = True):
    node.boost_python_export = False
for node in asg.enumerations():
    node.boost_python_export = False
for node in asg.enumerators():
    if node.parent.boost_python_export:
        node.boost_python_export = False
for node in asg.typedefs():
    node.boost_python_export = False

from autowig.default_controller import refactoring
asg = refactoring(asg)

if autowig.parser.plugin == 'libclang':
    for fct in asg.functions(free=False):
        asg._nodes[fct._node]['_is_virtual'] = False
        asg._nodes[fct._node]['_is_pure'] = False
    asg['class ::clang::QualType'].is_abstract = False
    asg['class ::clang::QualType'].is_copyable = True
    asg['class ::llvm::StringRef'].is_abstract = False
    asg['class ::llvm::StringRef'].is_copyable = True
    asg['class ::clang::FileID'].is_abstract = False
    asg['class ::clang::FileID'].is_copyable = True
    asg['class ::clang::SourceLocation'].is_abstract = False
    asg['class ::clang::SourceLocation'].is_copyable = True
    asg['class ::clang::TemplateArgument'].is_abstract = False
    asg['class ::clang::TemplateArgument'].is_copyable = True
    for cls in ['::clang::FriendDecl', '::clang::CapturedDecl', '::clang::OMPThreadPrivat
                '::clang::NonTypeTemplateParmDecl', '::clang::TemplateArgumentList', '::c
                '::clang::TemplateTemplateParmDecl', '::clang::CapturedDecl', '::clang::C
                '::clang::NonTypeTemplateParmDecl', '::clang::TemplateArgumentList', '::c
                '::clang::TemplateTemplateParmDecl']:
        asg['class ' + cls].is_abstract = False

asg['class ::boost::python::api::object'].boost_python_export = True
asg['class ::boost::python::list'].boost_python_export = True
asg['class ::boost::python::str'].boost_python_export = True

subset = []
classes = [asg['class ::clang::QualType'],
           asg['class ::clang::Type'],
           asg['class ::clang::Decl']]
subset += classes
for cls in classes:
    subset += cls.subclasses(recursive=True)
for cls in subset:
    if not cls.globalname.strip('class ') in ['::clang::QualType',
                                               '::llvm::StringRef',
                                               '::clang::FileID',
                                               '::clang::SourceLocation',
                                               '::clang::TemplateArgument',
                                               '::clang::FriendDecl',
```

```
                                              '::clang::CapturedDecl',
                                              '::clang::OMPThreadPrivateDecl',
                                              '::clang::NonTypeTemplateParmDecl',
                                              '::clang::TemplateArgumentList',
                                              '::clang::ImportDecl',
                                              '::clang::TemplateTemplateParmDecl']:
        cls.is_copyable = False
    else:
        cls.is_copyable = True
subset.append(asg['class ::llvm::StringRef'])

subset.append(asg['class ::clang::ASTUnit'])
subset.append(asg['class ::clang::ASTContext'])
subset.append(asg['class ::clang::SourceManager'])
subset.append(asg['class ::clang::FileID'])

subset.append(asg['class ::clang::SourceLocation'])

subset.append(asg['class ::clang::CXXBaseSpecifier'])
subset.append(asg['class ::clang::DeclContext'])
subset.append(asg['class ::clang::TemplateArgument'])

subset.append(asg['class ::clang::TemplateArgumentList'])
subset.append(asg['enum ::clang::Type::TypeClass'])
subset.append(asg['enum ::clang::AccessSpecifier'])
subset.append(asg['enum ::clang::LinkageSpecDecl::LanguageIDs'])
subset.append(asg['enum ::clang::BuiltinType::Kind'])
subset.append(asg['enum ::clang::TemplateArgument::ArgKind'])
subset.append(asg['enum ::clang::Decl::Kind'])
# subset.extend(asg['::boost::python'].classes(nested = True))
# subset.extend(asg['::boost::python'].enumerations(nested = True))
subset.extend(asg.nodes('::clanglite::build_ast_from_code_with_args'))

for node in subset:
    node.boost_python_export = True

for fct in asg['::clanglite'].functions():
    if not fct.localname == 'build_ast_from_code_with_args':
        fct.parent = fct.parameters[0].qualified_type.desugared_type.unqualified_type
    fct.boost_python_export = True

for mtd in asg['class ::clang::ASTContext'].methods(pattern='.*getSourceManager.*'):
    if mtd.return_type.globalname == 'class ::clang::SourceManager &':
            mtd.boost_python_export = True
            break

if autowig.parser.plugin == 'libclang':
    for node in (asg.functions(pattern='.*(llvm|clang).*_(begin|end)')
                 + asg.functions(pattern='::clang::CXXRecordDecl::getCaptureFields')
                 + asg.functions(pattern='.*(llvm|clang).*getNameAsString')
                 + asg.nodes('::clang::NamedDecl::getQualifiedNameAsString')
                 + asg.functions(pattern='.*::clang::ObjCProtocolDecl')
                 + asg.nodes('::clang::ObjCProtocolDecl::collectInheritedProtocolPropert
                 + asg.nodes('::clang::ASTUnit::LoadFromASTFile')
                 + asg.nodes('::clang::ASTUnit::getCachedCompletionTypes')
                 + asg.nodes('::clang::ASTUnit::getBufferForFile')
                 + asg.nodes('::clang::CXXRecordDecl::getCaptureFields')
                 + asg.nodes('::clang::ASTContext::SectionInfos')
                 + asg.nodes('::clang::ASTContext::getAllocator')
```

```
                                  + asg.nodes('::clang::ASTContext::getObjCEncoding.*')
                                  + asg.nodes('::clang::ASTContext::getAllocator')
                                  + asg.nodes('::clang::QualType::getAsString')
                                  + asg.nodes('::clang::SourceLocation::printToString')
                                  + asg['class ::llvm::StringRef'].methods()):
                        node.boost_python_export = False

            if autowig.parser.plugin == 'clanglite':
                for mtd in asg['class ::clang::Decl'].methods():
                    if mtd.localname == 'hasAttr':
                        mtd.boost_python_export = False

            import sys
            from path import path
            for header in (path(sys.prefix)/'include'/'clang').walkfiles('*.h'):
                asg[header.abspath()].is_external_dependency = False

            return asg
```

This `controller` is then dynamically registered and used on the ASG.

```
In [ ]: %%time
        autowig.controller['clanglite'] = clanglite_controller
        autowig.controller.plugin = 'clanglite'
        asg = autowig.controller(asg)
```

In order to wrap a subset of the **Clang** library, we need to select the `boost_python_internal` `generator` implementation.

```
In [ ]: %%time
        autowig.generator.plugin = 'boost_python_pattern'
        wrappers = autowig.generator(asg,
                          module = srcdir/'_clanglite.cpp',
                          decorator = srcdir/'clanglite'/'_clanglite.py',
                          closure = False)
```

The wrappers are only generated in-memory. It is therefore needed to write them on the disk to complete the process.

```
In [ ]: %%time
        wrappers.write()
```

Here is an example of the generated wrappers. We here present the wrappers for the `clang::Decl` class.

```
In [ ]: !pygmentize ClangLite/src/py/wrapper_a6aedb4654a55a40aeecf4b1dc5fcc98.cpp
```

Once the wrappers are written on the disk, the bingings must be compiled and installed. This can be done using the **SCons** `py` target.

```
In [ ]: !conda build ClangLite/conda/python-clanglite -c statiskit -c conda-forge
        !conda install -y python-clanglite --use-local -c statiskit -c conda-forge
```

```
In [ ]: import autowig
        from clanglite.autowig_parser import autowig_parser
        autowig.parser['clanglite'] = autowig_parser
        autowig.parser.plugin = 'clanglite'
        from path import Path
        import sys

        for wrapper in srcdir.walkfiles('*.cpp'):
            wrapper.unlink()
        for wrapper in srcdir.walkfiles('*.h'):
            wrapper.unlink()
        wrapper = srcdir/'clanglite'/'_clanglite.py'
```

```
        if wrapper.exists():
            wrapper.unlink()

        prefix = Path(sys.prefix).abspath()

        asgbis = autowig.AbstractSemanticGraph()

        asgbis = autowig.parser(asgbis, [prefix/'include'/'clanglite'/'tool.h'],
                       flags = ['-x', 'c++', '-std=c++11',
                                '-D__STDC_CONSTANT_MACROS',
                                '-D__STDC_FORMAT_MACROS',
                                '-D__STDC_LIMIT_MACROS',
                                '-I' + str((prefix/'include').abspath()),
                                '-I' + str((prefix/'include'/'python2.7').abspath())],
                       bootstrap = False,
                       silent = True)

        autowig.controller['clanglite'] = clanglite_controller
        autowig.controller.plugin = 'clanglite'
        asgbis = autowig.controller(asgbis)

        autowig.generator.plugin = 'boost_python_pattern'
        wrappers = autowig.generator(asgbis,
                          module = srcdir/'_clanglite.cpp',
                          decorator = srcdir/'clanglite'/'_clanglite.py',
                          closure = False)

        wrappers.write()
```

```
In [ ]: !conda remove python-clanglite -y
        !conda build ClangLite/conda/python-clanglite -c statiskit -c conda-forge
        !conda install -y python-clanglite --use-local -c statiskit -c conda-forge
```

2.2.3 Wrapping a template library

A template library is a library where there are only template classes that can be instantiated. Wrapping such libraries therefore requires **AutoWIG** to be able to consider various *C++* template classes instantiations during the `Parse` step. It is therefore required to install the `pyclanglite parser`.

The **Standard Template Library (STL)** library is a *C++* library that provides a set of common *C++* template classes such as containers and associative arrays. These classes can be used with any built-in or user-defined type that supports some elementary operations (e.g. copying, assignment). It is divided in four components called algorithms, containers, functional and iterators. **STL** containers (e.g. `std::vector`, `std::set`) are used in many *C++* libraries. In such a case, it does not seem relevant that every wrapped *C++* library contains wrappers for usual **STL** containers (e.g. `std::vector< double >`, `std::set< int >`). We therefore proposed *Python* bindings for sequence containers (i.e. `pair`, `array`, `vector`, `deque`, `forward_list` and `list` of the `std` namespace) and associative containers (`set`, `multiset`, `map`, `multimap`, `unordered_set`, `unordered_multiset`, `unordered_map` and `unordered_multimap` of the `std` namespace). These template instantiations are done for *C++* fundamental types (`bool`, `signed char`, `unsigned char`, `char`, `wchar_t`, `int` (with sign modifiers `signed` and `signed` combined or not with size modifiers `short`, `long` and `long long`), `float`, `double`, `long double`) and strings (`string`, `wstring` of the `std` namespace). For ordered associative containers both `std::less` and `std::greater` comparators are used. We here only illustrate the procedure on the `std::vector` template class. For the complete procedure refers to the `AutoWIG.py` file situed at the root of the **PySTL** repository.

```
In [ ]: !git clone https://github.com/StatisKit/STL STL
        !git -C STL checkout b9569c67ebc59482dc99a8fa11aa685faebc981d
```

Then, to install and compile the *C++* library we use available **Conda** recipes.

```
In [ ]: !conda build -q STL/conda/libstatiskit_stl -c statiskit
        !conda install -y -q libstatiskit_stl --use-local -c statiskit
```

As presented below, in order to wrap a template library, the user needs to write headers containing aliases for desired template class instantiations.

```
In [ ]: !pygmentize STL/src/cpp/STL.h
```

Once these preliminaries done, we can proceed to the actual generation of wrappers for the **PySTL** library. For this, we import **AutoWIG** and create an empty Abstract Semantic Graph (ASG).

We need then to install the *C++* headers. This is done using the `cpp` target in **SCons**.

```
In [ ]: !scons cpp -C STL
```

Once the headers habe been installed in the system, we parse headers with relevant compilation flags.

```
In [ ]: !scons autowig -c -C STL
        !scons autowig -C STL
```

Here is an example of the generated wrappers. We here present the wrappers for the `std::vector< int >` class.

```
In [ ]: !pygmentize STL/src/py/wrapper/wrapper_6b9ae5eac40858c9a0f5e6e21c15d1d3.cpp
```

Once the wrappers are written on disk, we need to compile and install the *Python* bindings.

```
In [ ]: !conda build STL/conda/python-statiskit_stl -c statiskit
        !conda install -y python-statiskit_stl --use-local -c statiskit --force
```

Finally, we can hereafter use the *C++* library in the *Python* interpreter.

```
In [ ]: from statiskit.stl import VectorInt
        v = VectorInt()
        v.push_back(-1)
        v.push_back(0)
        v.push_back(1)
        v
```

```
In [ ]: list(v)
```

```
In [ ]: v[0]
```

```
In [ ]: v[0] = -2
        v[0]
```

```
In [ ]: VectorInt([0, 1])
```

2.3 Frequently Asked Questions

Note: Frequently asked questions about the project and contributing.

2.3.1 How to use AutoWIG on Windows or MacOS ?

Currently, **AutoWIG** binaries for Windows or MacOs X are proposed and can be installed using **Conda** but are not guaranteed to be working perfectly. However, we provide a **Docker** image that can be used on these operating systems. Please follow the *Test it with Docker* procedure.

CHAPTER 3

License

AutoWIG is distributed under the CeCILL license.

Note: CeCILL license is LGPL compatible.

CHAPTER 4

Authors

- Pierre Fernique
- Christophe Pradal