FFPopSim: documentation for the SWIG interface

Fabio Zanini

June 26, 2013

1 Introduction

FFPopSim is a C++ library for population genetics with Python bindings. The library was not written in pure Python for two reasons:

1. Efficiency;
2. To keep the possibility to control the code flow at low level, by linking directly against the C++ library.

The Python bindings are built via SWIG. In order to understand how they work and possibly modify them, you need to understand abit of SWIG first.

1.1 How does SWIG work?

SWIG takes two types of files in input:

1. C++ header files, ending in `.h`;
2. interface files, ending in `.i`.

What SWIG does *not* look at is the `.cpp` implementation files. Whatever you write in those, it will never be part of the Python interface: only the *effects* of that code will be seen. For instance, if you do not declare a function in the header file but implement it directly, that function will be invisible to Python.

SWIG takes the C++ header files and makes a list of all constants, variables, functions, classes, etc. Then, it runs over the interface files and checks whether any of those objects needs some special translation rules; if so, it applies those rules. Finally, SWIG produces two output files:

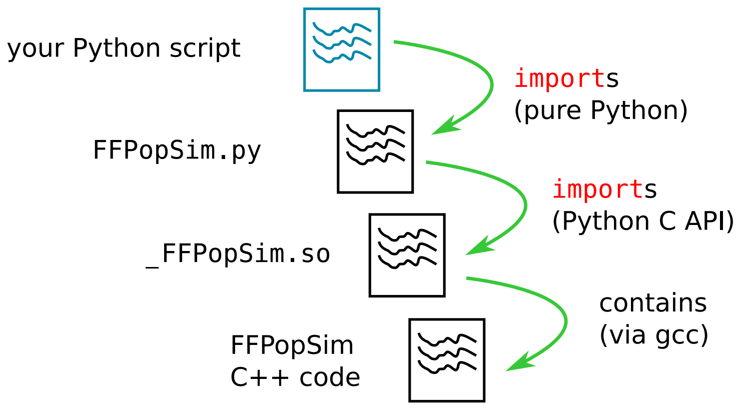

Fig. 1: Levels and relationships in FFPopSim's SWIG C++ to Python interface.

1. a Python file, `FFPopSim.py`;

2. a C++ wrap file, `FFPopSim_wrap.cpp`.

In order to get a complete Python module, you have to compile the generated wrap file and link it against the C++ library into a *shared library* file, named `_FFPopSim.so`. Note that this last step is *not* part of SWIG and can be achieved by different means: linking against a static version (`FFPopSim.a`), recompiling the whole library together, using Makefile or Python distutils and what not. Remember: in the end you are left with two files, `FFPopSim.py` and `_FFPopSim.so`, which constitute your Python module. They are to be handled as a unit (same folder, etc.). Also, never try to import `_FFPopSim`, but only `FFPopSim`!

1.2 Interface files?

C++ and Python are two very different languages. For instance, C++ has pointers, explicit types, explicit pass by value or reference, the STL library; Python has duck typing, no pointers whatsoever, a lot of builtin objects such as dictionaries, sets, tuples. Moreover, C++ and Python users expect different *styles* of programming, e.g. Python using properties in lieu of get/set methods. For these reasons, a translation between C++ and Python cannot be fully automatic and need some special rules. For example, we want to make sure that a genotype, in C++ a `boost::dynamic_bitset`, is converted into a numpy array of bool, and vice versa. All these special rules are listed in the interface files. In fact, you will spend most of your time writing such rules there.

2 SWIG in FFPopSim

2.1 General architecture

In FFPopSim, the C++ headers are in `src`, while the interface files in `src/python`. The main interface file is `FFPopSim.i`. It does three things:

1. It defines the name of the module, FFPopSim, and its docstring;
2. It includes standard pieces of SWIG code to convert STL vectors, maps, strings, numpy arrays, etc;
3. It includes a few more FFPopSim-specific interface files for the various parts of the library:
    - `ffpopsim_typemaps.i`: general recipes, called **typemaps**;
    - `ffpopsim_generic.i`: translates `ffpopsim_generic.h`
    - `ffpopsim_lowd.i`: translates `ffpopsim_lowd.h`
    - `ffpopsim_highd.i`: translates `ffpopsim_highd.h`
    - `hivpopulation.i`: translates `hivpopulation.h`

Golden rule: never touch the typemaps. Anything else you can modify pretty easiliy once you get used to the code.

In the next chapter, we'll see the typical SWIG constructs used in FFPopSim: you can copy & paste these snippets to extend FFPopSim at will. Before that, however, a few more words of explanation are needed.

2.2 Interface files speak 3+ languages

A typical chunk of an interface file looks like the following:

```
/* traits */
%feature("autodoc", "Number of traits (read-only)") number_of_traits;
const int number_of_traits;

%ignore trait;
void _get_trait(int DIM1, double* ARGOUT_ARRAY1) {
        for(size_t i=0; i < (size_t)DIM1; i++)
                ARGOUT_ARRAY1[i] = ($self->trait)[i];
}
%pythoncode {
```

```
@property
def trait(self):
    '''Traits vector of the clone'''
    return self._get_trait(self.number_of_traits)
}
```

This little part only is written in at least three languages: SWIG, C++, and Python. Here's how to recognize them:

1. lines starting with an % (percent sign) are SWIG commands (directives) and provide a self-contained, special rule. In the example, we add a Python docstring documentation to a class attribute, `number_of_traits`;

2. lines with no special frame are C++ code that interfaces to the C++ library as an external program. This code is present in the wrap file and eventually in `_FFPopSim.so`;

3. lines within `%pythoncode{}` or similar commands are pure Python: they end up in `FFPopSim.py`.

So here's the general rule: if you want to modify a rule (e.g. add a docstring, ignore or rename a variable, etc.), use a SWIG directive; if you want to add functions, classes, variables that are not speed critical, put them in a `%pythoncode{}` directive (like it is done for plot functions); finally, if you need to access the C++ library at low level or to do efficient loops, write C++ code. Unfortunately, putting three languages in a single file makes it rather messy.

In depth: didn't you say 3+? Where's the 4th?

NumPy. If you want to translate STL vectors into NumPy arrays instead of ad-hoc objects or Python lists, you have to use stuff like:

```
void _get_trait(int DIM1, double* ARGOUT_ARRAY1) {
```

The trick here are the two arguments, `DIM1` and `ARGOUT_ARRAY1`. They are used like normal input arguments in their C++ function but, in addition, they have a special meaning for SWIG by virtue of the `numpy.i` file. The latter makes sure that memory is allocated for the array before entering the function and released by Python when the array is destroyed. In a sense, lines like this are meaningful in 2 languages at the same time: C++ and the `numpy.i` mini-language. If you think this is horribly low level, think of the alternative: managing memory exchange between C++ and Python ourselves (reference counting, garbage collection, etc.)... not really exciting, uh?

2.3 Example: add a class to FFPopSim

Let us imagine you want to add a new subclass of `haploid_highd` for the HCV virus. After you have a working C++ code, proceed as follows:

1. decide where to put the *declaration* of the new class. Check the `Makefile` and the `setup.py` such that dependencies are fine there. Let us assume you created a new header file called `hcv.h`:

   ```
   #define HCV_STRAIN_ONE_A "1a"
   #define HCV_STRAIN_TWO_A "2a"

   class hcv : public haploid_highd {
   public:
           hcv(int N=0, int rng_seed=0);
           virtual ~hcv();

           // strain of HCV
           string strain;
   }
   ```

2. Create an empty file in `src/python` called e.g. `hcv.i`.

3. Modify `FFPopSim.i` and include `hcv.h` (twice) and `hcv.i` (once).

4. Open `hcv.i` and add `%ignore` directives for all objects in your header; check for compilation and runtime:

   ```
   /* Interface file for hcv.h */
   %ignore HCV_STRAIN_ONE_A;
   %ignore HCV_STRAIN_TWO_A;
   %ignore hcv;
   ```

   Of course, your class is still invisible to Python, but at least you have not messed up the other stuff!

5. Un-ignore constants, and add docstrings to them:

   ```
   %feature("autodoc", "HCV strain 1a") HCV_STRAIN_ONE_A;
   %feature("autodoc", "HCV strain 2a") HCV_STRAIN_TWO_A;
   ```

6. Un-ignore the class, add a docstring to it, extend, add documentation for the constructor, and ignore everything else: inside:

```
%feature("autodoc", "FFPopSim class for HCV") hcv;
%extend hcv {
%feature("autodoc", "Constructor for the HCV class:

Parameters:
   - N: number of viruses in the population
   - rng_seed: seed of the random number generator
") hcv;
%ignore strain;
}
```

7. Un-ignore the attribute of the class within the `%extend{}`, add docstring, check whether it works correctly:

```
%extend hcv {
...
%feature("autodoc", "HCV strain of this population") strain;
}
```

As you see, a big part of building SWIG interfaces is writing documentation. This is crucial, because the same functions will work slightly differently in Python from their C++ counterparts, and there is no Python code to look at for help! When writing docstrings, remember that the Python documentation is built using reStructured text and Sphinx, so try to respect the conventions of those programs (see the three spaces of indent under `Parameters:`?).

Now, if you got here, you're over the hardest! The following chapter provides you with typical code snippets for your convenience.

3 Reference snippets

3.1 Constants or read/write global variables

Just add a docstring:

```
%feature("autodoc", "<Python documentation>") myconstant;
```

3.2 Read-only class attributes

Python has no private attributes, nor get/set methods. You can still achieve the same functionality with proterties:

```
%extend myclass {
...
%ignore myvar;
double _get_myvar() { return $self->myvar; }
%pythoncode {
@property
def myvar (self):
    '''<YOUR DOCUMENTATION>'''
    return self._get_myvar()
}
...
}
```

The trick here is that functions or variables starting with an underscore, albeit public, are considered dangerous in Python, so the user should not touch them. Moreover, the `%ignore` directive only works at the Python level, so our inline C++ function can still work (via the `$self` trick).

3.3 Classes

You must include the following: a docstring for the class, a docstring for the constructor, and `%extend` directive, and string representation methods:

```
%feature("autodoc", "Class documentation") myclass;
%extend myclass {
%feature("autodoc", "Constructor documentation") myclass;
%feature("autodoc", "x.__str__() <==> str(x)") __str__;
%feature("autodoc", "x.__repr__() <==> repr(x)") __repr__;
const char* __str__() {
        static char buffer[255];
        sprintf(buffer, "...", ...);
        return &buffer[0];
}
const char* __repr__() {
        static char buffer[255];
        sprintf(buffer, "...", ...);
        return &buffer[0];
}
...
}
```

See python **__repr__** and **__str__** for more details.

3.4 Pointers

SWIG can pass pointers to Python just right, but you cannot dereference them from there, so usually dealing with bare pointers as input/output function arguments is a bad idea. The typical solution is to write a C++ function that works the same but without the pointer, e.g.:

```
%ignore myfunction;
%rename (myfunction) myfunction2;
vector <int> myfunction2 () {
// this function returns a vector instead of fiddling with
// C-style arrays
...
}
```

Note: when working with objects, you sometimes want a function to return a copy of the object. For instance, you do not want the naïve Python user to modify the genealogy within `haploid_highd`, for fear of damage. Sometimes, however, you want to use pointers, to *modify extant objects*. For instance, if your population has an attribute called `location` which is a struct and you want to modify it from Python, you will need a pointer to the extant object, not a copy!

3.5 Converting `std::<whatever>` into Python builtins

Standard translation tables for STL objects can be less than optimal. If efficiency is not an issue, you can use this trick:

```
%rename (_myfunction) myfunction;
%pythoncode {
def myfunction(myargs):
    return list(_myfunction(myargs)) # or dict(...) for maps, etc.
}
```

If you get an error because the object you have is not convertible into a list/dict/..., check `FFPopSim.i` for the appropriate include statements for the STL. If this is not sufficient, you might need a typemap. Good luck.

3.6 Errors and exceptions

The C++ part of FFPopSim uses two styles of error handling. Some functions simply return nonzero values if an error has occurred (C style errors), other functions `throw` actual C++ exceptions. Typically, the latter style is employed in all cases where returning an `int` is not an option, e.g. for class constructors (they return an instance of the class). How are errors propagated to Python and handled there?

> **Remember:** Error messages should be helpful for the user, not generic or too low-level.

C style error handling

C style errors need no special translation, but look weird to Python users – this generates still another WTF moment each time. To make a better interface, you can use the following recipe:

```
%exception myfunction {
  $action
  if (result) {
     PyErr_SetString(PyExc_RuntimeError,"Error in the C++ function.");
     SWIG_fail;
  }
}
```

Note: `PyExc_RuntimeError` is only one of the many possible standard Python exceptions. Look at the Python C API website for more examples.
Note: the `if (result)` construct is very vaguely documented in SWIG, hence this kind of code is probably fragile.

C++ style exceptions

If the C++ library `throw`s an error that is not caught by Python, the user won't be happy. Here's how you do that:

```
%exception myfunction {
        try {
                $action
        } catch (int err) {
                PyErr_SetString(PyExc_RuntimeError,"Error in the C++ function.")
                SWIG_fail;
        }
}
```

Here we assumed the thrown exception is an `int` (this is the typical case in FFPopSim), but it can be anything really.