Introduction to writing IMP code

Benjamin Webb,
Sali Lab,
University of California San Francisco
(ben@salilab.org)

Overview

- Here we will cover creating a new IMP module, and writing a new restraint in C++

- For more detail, see the IMP manual at https://integrativemodeling.org/nightly/doc/manual/developing.html

- Prerequisite: build IMP from source code as per https://integrativemodeling.org/nightly/doc/manual/installation.html#installation_source

Add a new module

- From the top-level directory in the IMP source code:
  - The '`modules`' directory contains a subdirectory for each IMP module.
  - To add a new module called '`foo`', use the `tools/make_module.py` script:

```
$ tools/make_module.py foo
```

  - This will make a new subdirectory `modules/foo/`; let's take a look at its contents:

```
$ ls modules/foo
README.md   bin                 examples    pyext    test
benchmark   dependencies.py     include     src      utility
```

‘include’ directory

- This contains C++ header files that declare the public classes and other functions that are part of the module
- For classes that are not intended to be public (e.g. utility classes only used by your module itself) put them instead in the `include/internal` subdirectory
- Let’s add a new class to our module, `MyRestraint`, a simple restraint that restrains a particle to the xy plane (see the `ExampleRestraint` class in `modules/example/` for a similar class)
  - IMP convention is for class names (and the files declaring and defining them) to be CamelCase
- We do this by creating a file `MyRestraint.h` in the `modules/foo/include/` subdirectory

`MyRestraint.h` start

- The first part of the file looks like

```
#ifndef IMPFOO_MY_RESTRAINT_H
#define IMPFOO_MY_RESTRAINT_H

#include <IMP/foo/foo_config.h>
#include <IMP/Restraint.h>

IMPFOO_BEGIN_NAMESPACE
```

- The `ifndef/define` is a *header guard*, which prevents the file from being included multiple times. Convention is to use upper case `IMP<module name>_<file name>`
- All of our classes will exist in the `IMP::foo` namespace. The `IMPFOO_BEGIN_NAMESPACE` macro ensures this. It is defined in the `foo_config.h` header file
- We are going to declare a restraint, so we need to `#include` the `Restraint.h` base class definition

Class declaration

```
class IMPFOOEXPORT MyRestraint : public Restraint {
  ParticleIndex p_;
  double k_;

public:
  MyRestraint(Model *m, ParticleIndex p, double k);
  void do_add_score_and_derivatives(ScoreAccumulator sa) const
      IMP_OVERRIDE;
  ModelObjectsTemp do_get_inputs() const;
  IMP_OBJECT_METHODS(MyRestraint);
};
```

- `IMPFOOEXPORT` should be used for any class that has a `.cpp` implementation, and ensures the class can be used outside of the module (e.g. in Python)
- `IMP_OBJECT_METHODS` adds standard methods that all IMP `Objects` (like `Restraint`) are expected to provide
- Our constructor takes an IMP `Model`, a particle in that model, and a force constant
- We will also need to provide the score and inputs for the restraint (in the `.cpp` file)

`MyRestraint.h` end

- The final part of the file looks like

```
IMPFOO_END_NAMESPACE

#endif /* IMPFOO_MY_RESTRAINT_H */
```

- This just closes the namespace and header guard from the start of the file

- Next, we need to provide a definition for the class. We do this by making a corresponding file `MyRestraint.cpp` in the `modules/foo/src/` subdirectory

`MyRestraint.cpp` start

- The first part of the file looks like

```
#include <IMP/foo/MyRestraint.h>
#include <IMP/core/XYZ.h>

IMPFOO_BEGIN_NAMESPACE
```

- Similarly to the header file, we need to put everything in the `IMP::foo` namespace and include any needed header files. Here we include the previous declaration of the `MyRestraint` class. We also need the declaration of the `XYZ` decorator from `IMP::core` since we are going to be using the particle's coordinates to calculate the score.

Constructor

```
MyRestraint::MyRestraint(Model *m, ParticleIndex p,
                         double k)
    : Restraint(m, "MyRestraint%1%"), p_(p), k_(k) {}
```

- The constructor simply calls the `Restraint` base class constructor (which takes the `Model` and a human-readable name) and stores the `p` and `k` arguments in the class attributes `p_` and `k_` (IMP convention is for class attributes to end in an underscore)
- "`%1%`" is a replaced with a unique number, so multiple restraints will be named MyRestraint1, MyRestraint2, etc.

Score and derivatives

```
void MyRestraint::do_add_score_and_derivatives(ScoreAccumulator sa)
        const {
  core::XYZ d(get_model(), p_);
  double score = .5 * k_ * square(d.get_z());
  if (sa.get_derivative_accumulator()) {
    double deriv = k_ * d.get_z();
    d.add_to_derivative(2, deriv,
                        *sa.get_derivative_accumulator());
  }
  sa.add_score(score);
}
```

- We apply a simple harmonic restraint to the z coordinate to keep the particle in the xy plane
- We use the `core::XYZ` decorator to treat the particle as a coordinate
- `ScoreAccumulator` is given the score, and analytic first derivatives as well if requested

Inputs

```
ModelObjectsTemp MyRestraint::do_get_inputs() const
{
  return ModelObjectsTemp(1,
                          get_model()->get_particle(p_));
}
```

- We also need to tell IMP which particles our restraint acts on by overriding the `do_get_inputs` method
- Here we just have a single particle, `p_`
- This is used to order the evaluation of restraints and constraints (a constraint which moves particle A must be evaluated before any restraint with A as an input) and for parallelization

`MyRestraint.cpp` end

- The final part of the file looks like

```
IMPFOO_END_NAMESPACE
```

- As before, we need to close the namespace

- Next, we make the C++ class available in Python. We do this by modifying the `swig.i-in` file in the `modules/foo/pyext/` subdirectory

SWIG class declaration

- First, we need to tell SWIG how to wrap the `MyRestraint` class:

```
IMP_SWIG_OBJECT(IMP::foo, MyRestraint, MyRestraints);
```

- We tell SWIG that `MyRestraint` is an IMP `Object`
  - Most IMP classes are subclasses of `IMP::Object`. These are heavyweight objects which are always passed by reference-counted pointers, and are generally not copied
  - Some simple classes (e.g. `IMP::algebra::Vector3D`) are subclasses of `IMP::Value`. These are lightweight objects which are generally passed by value or reference, and can be trivially copied

SWIG header file

- We also need to tell SWIG to parse our C++ header file:

```
%include "IMP/foo/MyRestraint.h"
```

- With the SWIG interface complete, we will be able to use our class from Python as `IMP.foo.MyRestraint`.

Documentation

- Documentation is omitted here for clarity, but all C++ headers and `.cpp` files should contain comments!
- All comments are parsed by doxygen, which uses the special comment markers `//!` and `/** */`
- You should also fill in `modules/foo/README.md` with a description of the module and the license it is released under. We recommend an open source license such as the LGPL.

Tests

- Next we should write a test case in the `modules/foo/test/` directory, by creating a new file `test_restraint.py`
- Test cases periodically verify that IMP is working correctly
- Test cases can be written in C++, but are almost always written in Python for flexibility
- IMP convention is to name a test file starting with `test_`

`test_restraint.py` start

- The first part of the file looks like

```
from __future__ import print_function, division
import IMP
import IMP.test
import IMP.algebra
import IMP.core
import IMP.foo

class Tests(IMP.test.TestCase):
```

- We need to import the IMP kernel, any other IMP modules used in the test, and our own `IMP.foo` module
- The imports from `__future__` help to ensure that our test works in the same way in both Python 2 and Python 3
- All tests should be classes that use the `IMP.test` module, which adds some IMP-specific functionality to the standard Python `unittest` module

`test_restraint.py` method

```
    def test_my_restraint(self):
        m = IMP.Model()
        p = m.add_particle("p")
        d = IMP.core.XYZ.setup_particle(m, p, IMP.algebra.Vector3D(1,2,3))
        r = IMP.foo.MyRestraint(m, p, 10.)
        self.assertAlmostEqual(r.evaluate(True), 45.0, delta=1e-4)
        self.assertLess(IMP.algebra.get_distance(d.get_derivatives(),
                                                 IMP.algebra.Vector3D(0,0,30)),
                        1e-4)
        self.assertEqual(len(r.get_inputs()), 1)
```

- We create a restraint object, request its score and derivatives (`evaluate`), and ask for inputs (`get_inputs`)
- Here we simply test by comparing to known good values using the standard `unittest` methods `assertAlmostEqual`, `assertLess`, and `assertEqual`
  - Always use `assertAlmostEqual` for floating point comparisons, never `assertEqual`

test_restraint.py end

```
if __name__ == '__main__':
    IMP.test.main()
```

- This simply runs all the tests in this file if the script is run directly from the command line with "`python test_restraint.py`"

test_restraint.py complete

```
from __future__ import print_function, division
import IMP
import IMP.test
import IMP.algebra
import IMP.core
import IMP.foo

class Tests(IMP.test.TestCase):

    def test_restraint(self):
        m = IMP.Model()
        p = m.add_particle("p")
        d = IMP.core.XYZ.setup_particle(m, p, IMP.algebra.Vector3D(1,2,3))
        r = IMP.foo.MyRestraint(m, p, 10.)
        self.assertAlmostEqual(r.evaluate(True), 45.0, delta=1e-4)
        self.assertLess(IMP.algebra.get_distance(d.get_derivatives(),
                                                 IMP.algebra.Vector3D(0,0,30)),
                        1e-4)
        self.assertEqual(len(r.get_inputs()), 1)

if __name__ == '__main__':
    IMP.test.main()
```

- Python is sensitive to whitespace, so ensure the file is indented as shown here.

Dependencies

- Finally we need to tell the IMP build system which other modules and external code the module depends on. This is done by editing the file `modules/foo/dependencies.py`:

```
required_modules = ‘core:algebra'
required_dependencies = ''
optional_dependencies = ‘'
```

- Since we use the `core` and `algebra` modules, we need to declare them as requirements for this module.
- `*_dependencies` can be used to make use of 3rd party libraries. See some existing IMP modules for examples.

Source control

- Now is a good time to store the module in source control
- Most IMP modules are stored on GitHub
- See https://github.com/salilab/pmi/ and https://github.com/salilab/npctransport for examples

Build and test it

- Build IMP from source code in the usual way. cmake will pick up the new module, then make/ninja will build it
- Test the new code with something like

```
$ ./setup_environment.sh python \
    ../imp/modules/foo/test/test_restraint.py
```