CCTBX: Making Compiled Extensions

Graeme Winter, Diamond Light Source

November 13, 2013

1 Introduction

Much of the benefit of the hybrid approach to CCTBX[1] is the ability to move computationally expensive steps from Python to C++ relatively straightforwardly. It is however helpful to have an example of how to create C++ extensions and compile these with the existing compilation tools. The aim of this document is to provide this example with a toy example, toytbx. Some introductory information is however useful.

Within CCTBX C++ code is made available to Python using the Boost Python framework[2] and compiled with SCons[3]. Two steps are therefore needed to expose C++ code: writing a wrapper using Boost Python and a SConscript for compilation. Additionally a libtbx_config file is needed to specify the project dependencies. Using this libtbx does most of the work. In addition it is nice to expose functionality as "pseudo-executable" programs e.g. toytbx.task - this is also handled automatically by libtbx.

2 C++ Code

The following code defines a trivial module which contains a single function, which itself generates a Python list. The code defines a module `toytbx_ext`:

```
#include <boost/python.hpp>
#include <cctype>

namespace toytbx {
  namespace ext {

    static boost::python::list make_list(size_t n)
    {
      boost::python::list result;
      for(size_t i = 0; i < n; i++) {
```

[1] http://cctbx.sourceforge.net/current/tour.html#thinking-hybrid
[2] http://www.boost.org/libs/python/doc/
[3] http://www.scons.org/

```
        result.append(i);
      }
      return result;
    }

    void init_module()
    {
      using namespace boost::python;
      def("make_list", make_list, (arg("size")));
    }

  }
} // namespace toytbx::ext

BOOST_PYTHON_MODULE(toytbx_ext)
{
  toytbx::ext::init_module();
}
```

For more functions, extra static methods will be needed, as well as additional `def` statements in the `init_module` method.

3 Compilation

To compile the code above, SCons is used with libtbx. The first step is to define a SConscript which states the modules requirements:

```
import libtbx.load_env
Import("env_etc")

env_etc.toytbx_include = libtbx.env.dist_path("toytbx")

if (not env_etc.no_boost_python and hasattr(env_etc, "boost_adaptbx_include")):
  Import("env_no_includes_boost_python_ext")
  env = env_no_includes_boost_python_ext.Clone()
  env_etc.enable_more_warnings(env=env)
  env_etc.include_registry.append(
    env=env,
    paths=[
      env_etc.libtbx_include,
      env_etc.boost_adaptbx_include,
      env_etc.boost_include,
      env_etc.python_include,
      env_etc.toytbx_include])
  env.SharedLibrary(
    target="#lib/toytbx_ext",
    source=["ext.cpp"])
```

This can refer to other compiled modules to ensure dependencies are correctly built. With this file and ext.cpp in the toytbx directory, accessible to libtbx.configure, libtbx.configure toytbx may be run, followed by make, which will correctly compile the extension module.

If the project has dependencies, a libtbx_config file is needed for the libtbx to build projects in the correct order. The example for toytbx is shown below.

```
{
  "modules_required_for_build": ["boost", "scitbx"],
  "modules_required_for_use": ["boost_adaptbx"],
  "optional_modules": []
}
```

4 Tidying

This has generated an extension module, however it may be desirable to (i) import this nicely named as e.g. toytbx and (ii) add Python code to the module. This is best achieved by adding `__init__.py` to the toytbx directory containing:

```
from __future__ import division
try:
  import boost.python
except Exception:
  ext = None
else:
  ext = boost.python.import_ext("toytbx_ext", optional = False)

if not ext is None:
  from toytbx_ext import *
```

As with other code test cases should be written which ensure that the behaviour is correct. In this case, the following test is reasonable:

```
from toytbx import make_list

def tst_toytbx():
  assert(make_list(4) == [j for j in range(4)])
  print 'OK'

if __name__ == '__main__':
  tst_toytbx()
```

After which `cctbx.python tst_toytbx.py` should print 'OK'.

5 Command Lines

If a command-line executable is desired once again libtbx comes to the rescue. Make a directory within the toytbx called `command_line`, and within there `make_list.py`:

```
from toytbx import make_list

def main(args):
  assert(args)
  n = int(args[0])
  print make_list(n)

if __name__ == '__main__':
  import sys
  main(sys.argv[1:])
```

Rerunning libtbx.configure and make should generate executables allowing:

```
Graemes-MacBook-Pro:toytbx graeme$ toytbx.make_list 10
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

6 Being a Little More ... Flexible

Passing around Python lists is all well and good, but these are an expensive representation of the data and not much use if you want to pass the result on to other C++ code for more calculations. For this reason flex arrays are used, which have both C++ and Python representations but most critically are stored in a memory efficient way. Adding the following code snippets demonstrates the generation of flex arrays in one routine and usage of them elsewhere:

```
    // make a flex array (much more flexible)

    static scitbx::af::shared<int> make_flex(size_t n)
    {
      scitbx::af::shared<int> result;
      for(size_t i = 0; i < n; i++) {
result.push_back(i);
      }
      return result;
    }

    // using flex arrays

    static int sum(scitbx::af::shared<int> array)
```

```
    {
      int result = 0;
      for (size_t i = 0; i < array.size(); i++) {
result += array[i];
      }
      return result;
    }

    void init_module()
    {
      using namespace boost::python;
      def("make_list", make_list, (arg("size")));
      def("make_flex", make_flex, (arg("size")));
      def("sum", sum, (arg("array")));
    }
```

These behave very similarly to lists in Python, however in the Python code the following *must* be added before these arrays are used:

```
import scitbx.array_family.flex
```

otherwise nasty errors will result. Accordingly, the test should ideally be extended to:

```
import scitbx.array_family.flex
import toytbx

def tst_toytbx():
  assert(toytbx.make_list(4) == [j for j in range(4)])
  assert(sum(toytbx.make_flex(10)) == toytbx.sum(toytbx.make_flex(10)))
  print 'OK'

if __name__ == '__main__':
  tst_toytbx()
```

to make sure everything is well behaved.

7 Acknowledgements

This was prepared based on the fable module, after a pointer from Nat Echols and remembering things from Nick Sauter and Ralf Grosse-Kunstleve.