State of the Toolbox: an overview of the Computational Crystallography Toolbox (CCTBX)

Ralf W. Grosse-Kunstleve and Paul D. Adams
Lawrence Berkeley National Laboratory, One Cyclotron Road, BLDG 4R0230, Berkeley, CA 94720-8235, USA
E-mail: RWGrosse-Kunstleve@lbl.gov - WWW: http://cci.lbl.gov/

1. Introduction

Back in November 2001 we released version 1.0 of the Computational Crystallography Toolbox (cctbx). In this article we give an overview of the new developments that will be available in the 2.0 release. While the code base is deemed stable and ready for release, the documentation still needs updating. We hope to be able to do this in the near future.

The cctbx is being developed as the open source component of the PHENIX system. The goal of the PHENIX project is to advance automation of macromolecular structure determination. PHENIX depends on the cctbx, but not vice versa. This hierarchical approach enforces a clean design of the cctbx as a reusable library.

To maximize reusability and, maybe even more importantly, to give individual developers a notion of privacy, the new cctbx is organized as a set of smaller modules. This is very much like a village (the cctbx project) with individual houses (modules) for each family (groups of developers, of any size including one).

The cctbx code base is available without restrictions and free of charge to all interested developers, both academic and commercial. The entire community is invited to actively participate in the development of the code base. A sophisticated technical infrastructure that enables community based software development is provided by SourceForge. This service is also free of charge and open to the entire world.

2. High level organization

The SourceForge cctbx project currently contains these modules:

libtbx
: The build system common to all other modules. This is a very thin wrapper around the SCons software construction tool. The libtbx also contains platform-independent instructions for building the Boost.Python library.

scitbx
: Contains C++ libraries for general scientific computing (i.e. libraries that are not specific to crystallographic applications): a family of high-level C++ array types, a fast Fourier transform library, and a C++ port of the popular LBFGS conjugate gradient minimizer, all including Python bindings. These libraries are separated from the crystallographic code base to make them easily accessible for non-crystallographic application developers.

cctbx
: Python and C++ libraries for general crystallographic applications, useful for both small-molecule and macro-molecular crystallography. The libraries in the cctbx module cover everything from algorithms for the handling of unit cells to high-level building blocks for refinement algorithms. Note the distinction between the cctbx *project* and the cctbx *module*. In retrospect we should have chosen a different name for the project, but the current naming reflects how the modules have evolved and it would be too disruptive to start a grand renaming.

iotbx
: The youngest member in the family: evolving libraries for reading and writing established file formats. At the moment the iotbx contains C++ and Python interfaces for reading CCP4 MTZ files and ADSC X-ray detector images.

3. The beach in the box

If you go to the beach you will find massive amounts of a material known to crystallographers as quartz, which in this case is just a fancy word for sand. As an example here is the cctbx way of playing in the sandbox:

```
from cctbx import xray
from cctbx import crystal
from cctbx.array_family import flex

quartz_structure = xray.structure(
  special_position_settings=crystal.special_position_settings(
    crystal_symmetry=crystal.symmetry(
      unit_cell=(5.01,5.01,5.47,90,90,120),
      space_group_symbol="P6222")),
  scatterers=flex.xray_scatterer([
    xray.scatterer(
      label="Si",
      site=(1/2.,1/2.,1/3.),
      u=0.2),
    xray.scatterer(
      label="O",
      site=(0.197,-0.197,0.83333),
      u=0)]))

quartz_structure.show_summary().show_scatterers()
```

Running this script with Python produces the output:

```
Number of scatterers: 2
At special positions: 2
Unit cell: (5.01, 5.01, 5.47, 90, 90, 120)
Space group: P 62 2 2 (No. 180)
Label  M  Coordinates            Occ  Uiso or Ustar
Si     3  0.5000  0.5000  0.3333 1.00 0.2000
O      6  0.1970 -0.1970  0.8333 1.00 0.0000
```

Note that the script is pure Python, even though at first sight the format might appear to be specifically designed for crystallographic data. Now let's give the `quartz_structure` a rest break at the beach:

```
from scitbx.python_utils import easy_pickle
easy_pickle.dump("beach", quartz_structure)
```

This creates a file with the name `beach` containing all the information required for restoring the `quartz_structure` **object**, which is the technical term for the entire hierarchy of data referenced by the `quartz_structure` identifier in the Python script. A very important point to notice is that the `easy_pickle` module used for storing the `quartz_structure` is not specific to our object. `easy_pickle` will store and restore (almost) any user-defined Python object.

Being automatically generated, the `beach` file is not pretty to look at, but this is not important because we can easily resurrect the original object to extract any information that we might be interested in. In a potentially *different script* on a potentially *different computer* with a potentially *different operating system*:

```
from scitbx.python_utils import easy_pickle
quartz_structure = easy_pickle.load("beach")
```

Note that it is not necessary to explicitly import the relevant cctbx modules in this script. Python's pickle module does it for us automatically after inspecting the `beach` file.

In practice a "live object" in memory is much more valuable than information stored in a good-old file format because there are often many different questions one can ask about particular real-world objects. For example, we could ask: What are the site symmetries of the atoms in the `quartz_structure`? Here is how we ask that question in Python's language:

```
for scatterer in quartz_structure.scatterers():
  print "%s:" % scatterer.label, "%8.4f %8.4f %8.4f" % scatterer.site
  site_symmetry = quartz_structure.site_symmetry(scatterer.site)
  print "  point group type:", site_symmetry.point_group_type()
  print "  special position operator:",  site_symmetry.special_op()
```

Answer:

```
Si:   0.5000   0.5000   0.3333
  point group type: 222
  special position operator: 1/2,1/2,1/3
O:   0.1970  -0.1970   0.8333
  point group type: 2
  special position operator: 1/2*x-1/2*y,-1/2*x+1/2*y,5/6
```

Another question we could ask: What are the structure factors up to a resolution of d_min=2 Angstrom?

```
f_calc = quartz_structure.structure_factors(d_min=2).f_calc_array()
f_calc.show_summary().show_array()
```

Answer:

```
Miller array info: None
Type of data: complex_double, size=7
Type of sigmas: None
Number of Miller indices: 7
Anomalous flag: None
Unit cell: (5.01, 5.01, 5.47, 90, 90, 120)
Space group: P 62 2 2 (No. 180)
(1, 0, 0) (-11.3483432953-3.90019504038e-16j)
(1, 0, 1) (-14.9620947104-25.915108226j)
(1, 0, 2) (1.46915343413-2.54464839202j)
(1, 1, 0) (-12.8387095938+0j)
(1, 1, 1) (5.39203951708-9.3392864j)
(2, 0, 0) (-1.80942693741-2.84059649279e-16j)
(2, 0, 1) (4.95031293935+8.57419352432j)
```

Now we could turn our attention to the new `f_calc` object and start asking different questions. For example: What are the d-spacings?

```
f_calc.d_spacings().show_array()
```

Answer:

```
(1, 0, 0) 4.33878727296
(1, 0, 1) 3.39927502294
(1, 0, 2) 2.31368408207
(1, 1, 0) 2.505
(1, 1, 1) 2.27753582331
(2, 0, 0) 2.16939363648
(2, 0, 1) 2.01658808355
```

Sometimes questions alone are not enough. We actually want to do something. For example select only low-resolution intensities:

```
low_resolution_only                                                    =
f_calc.apply_selection(f_calc.d_spacings().data() > 2.5)
low_resolution_only.show_array()
```

Answer:

```
(1, 0, 0) (-11.3483432953-3.90019504038e-16j)
(1, 0, 1) (-14.9620947104-25.915108226j)
(1, 1, 0) (-12.8387095938+0j)
```

Of course, the cctbx does not have a canned answer to every question, even if it is a reasonable question. Fortunately, by virtue of being a Python based system, the cctbx does lend itself to being extended and embedded in order to form answers to questions that one might come across. The cctbx has now reached a degree of completeness where this can quite often be done without venturing into the deeper and darker layers, the C++ core that we haven't so far presented.

4. At the very bottom

Python is a great language for just about everything. It is just unfortunate that we do not currently have machines smart enough to turn any Python script into efficient machine language (but visit the PSYCO web site to witness mankind stretching out its feelers in that direction). Certainly, future generations will pity us for having to resort to counting bits and bytes in order to get our work done (imagine yourself with a set of Beevers-Lipson strips getting ready for a structure factor calculation).

Some core components of the cctbx started out as 'C' libraries (SgInfo, AtomInfo). Moving from 'C' to C++ including the Standard Template Library (STL) was a major step away from the bits-and-bytes counting era. For example, switching to C++ exception handling for dealing with errors reduced the source code size significantly and resulted in much improved readability. Equally important, using `std::vector` for managing dynamically allocated memory was a huge improvement over using 'C' style raw memory allocation functions (`malloc()` and `free()`). However, the idea of using `std::vector` throughout the cctbx wasn't very satisfactory: for small arrays such as 3x3 rotation matrices the dynamic memory allocation overhead can become a rate-limiting factor, and for large arrays the copy-semantics enforce a coding style that is difficult to follow. For example, consider a member function of a space group class that computes an array of multiplicities given an array of Miller indices. The scalar version of this function would certainly look like this:

```
int multiplicity(miller::index<> const& h);
```

Here is the direct translation to a vector version:

```
std::vector<int> multiplicity(std::vector<miller::index<> > const& h);
```

However, std::vector has deep-copy semantics (the same is true for std::valarray). This results in the following:

```
std::vector<int> multiplicity(std::vector<miller::index<> > const& h)
{
  std::vector<int> result; // Constructs the array.
  result.reserve(h.size()); // Optional, but improves performance.
  for(std::size_t i=0; i<h.size();i++) { // Loops over all Miller indices.
    result.push_back(multiplicity(h[i])); // Uses the scalar overload
  }                                       // to do the actual work.
  return result; // Ouch!
}
```

"Ouch" indicates that the *entire array* is copied when the function returns! While this might still be acceptable for arrays of Miller indices which are mostly of moderate size, it becomes a real burden when dealing with large maps. But returning to the example, in order to avoid the copying overhead the function above could be coded as:

```
void multiplicity(std::vector<miller::index<> > const& h,
                  std::vector<int>& result);
```

Unfortunately this is not only harder to read, but also more difficult to use because the result has to be instantiated before the function is called. This prevents convenient chaining of the type used in the `quartz_structure` examples above.

Other major problems are the absence of a multi-dimensional array type in the STL and limited support for algebraic array operations. We considered using Blitz++, and `boost::multi_array`, but these do only partially meet our specific requirements. For small arrays we actively used `boost::array` for some time, but this was also not entirely satisfactory due to the lack of convenient constructors which again prevents chaining. So eventually we started the major effort of implementing a family of small and large array types that address all our requirements and are as uniform as possible: the scitbx array family.

5scitbx.array_family.flex

The scitbx array family forms the backbone of the cctbx project. Viewed from the C++ side the family is quite big and diverse, but viewed from the Python side things are a lot simpler, as usual. All small C++ array types are transparently mapped to standard Python tuples. This gives immediate access to the rich and familiar set of standard tools for manipulating tuples. All large array types are transparently and uniformly mapped to a group of Python types in the `scitbx.array_family.flex` module. For example:

```
from scitbx.array_family import flex
flex.double(30) # a 1-dimensional array of 30 double-precision values
flex.int(flex.grid(2,3)) # a 2-dimensional array of 2x3 integer values
```

For numeric element types the `flex` type supports algebraic operations:

```
>>> a = flex.double([1,2,3])
>>> b = flex.double([3,2,1])
```

```
>>> tuple(a+b)
(4.0, 4.0, 4.0)
>>> tuple(flex.sqrt(a+b))
(2.0, 2.0, 2.0)
```

The `flex` type also supports a large number of other functions (abs, sin, pow, etc.), slicing, and as seen in the `quartz_structure` example above, pickling.

If all this looks similar to the popular Numeric module: it is at the surface. However, there are two very important differences:

- Under the hood the `flex` types are instantiations of a C++ array type that resembles familiar STL container types as much as possible. In contrast Numeric presents itself with a raw 'C' API.
- It is straightforward to implement other `flex` types with custom user-defined element types, even outside the scitbx module. This would be extremely difficult to do with Numeric, and is virtually impossible if the user-defined types are implemented in C++.

6cctbx.array_family.flex

The `cctbx.array_family.flex` inherits all `flex` types from the `scitbx.array_family.flex` module and adds a few types specific to the cctbx module, for example:

```
from cctbx.array_family import flex
flex.miller_index(((1,2,3), (2,3,4)))
flex.hendrickson_lattman(((1,2,3,4), (2,3,4,5)))
```

Another example is `flex.xray_scatterer` used in the `quartz_structure` above. The cctbx specific C++ code for establishing these Python types is fairly minimal (about 470 lines for exposing 6 types, including full pickle support and all copyright statements). This approach can therefore be easily adopted for user-defined types in other modules.

7. A balancing act

Python's **convenience of use** is directly related to the way the Python type system works: all type information is evaluated at runtime. For example consider this trivial function:

```
def plus(a, b):
  return a + b
```

It works instantly for many different argument types:

```
>>> plus(1, 2) # integer values
3
>>> plus(1+2j, 2+3j) # complex values
(3+5j)
>>> plus(['a', 'b'], ['c', 'd']) # lists
['a', 'b', 'c', 'd']
```

It works because the meaning of `a + b` is determined at runtime based on the actual types of `a` and `b`.

The **runtime efficiency** of C++ code is directly related to the way the C++ type system works: type information is usually evaluated at compile-time (virtual functions are an exception which we will not

consider here). Fortunately C++ has a very powerful mechanism that helps us avoid explicitly coding polymorphic functions over and over again:

```
template <typename T>
T plus(T const& a, T const& b)
{
  return a + b;
}
```

This template function is automatically *instantiated* for a given type `T` as used:

```
int a = 1;
int b = 2;
int c = plus(a, b); // implicitly instantiates plus with T==int
```

Given a system that is based on both Python and C++ we have the freedom of choosing the quick-and-easy runtime polymorphism offered by Python, or the more efficient compile-time polymorphism offered by C++.

An important consideration in deciding which solution is the most appropriate for a given problem is that a polymorphic Python function requires very little memory at runtime. In contrast, each new instantiation of a template function eventually results in a complete copy of the corresponding machine language instructions tailored for the specific types involved. This point may seem subtle at first, but being overly generous with the use of C++ compile-time polymorphism can lead to very large executable sizes and excessive compile times.

A comparison of the `plus` Python function and its C++ counterpart shows that the notational overhead of the C++ syntax can easily double the size of the source code. Therefore a programmer, given the choice, will naturally lean towards the Python solution until the runtime penalty due to the dynamic typing is prohibitive for a given application. However, when putting a dynamically typed system and a statically typed system together there are situations where it is important to carefully consider the best balance.

8. Hybrid systems

Considerations of the type discussed in the previous section directly lead to the following situation:

```
>>> a = flex.int((1,2,3))
>>> b = flex.double((2,3,4))
>>> a * b
TypeError: unsupported operand type(s) for *:
'scitbx_boost.array_family.flex_scitbx_ext.int' and
'scitbx_boost.array_family.flex_scitbx_ext.double'
```

In passing we note that there is a simple solution which will produce the desired result:

```
a.as_double() * b
```

However, for the purpose of this discussion let's pretend that this solution does not exist. Of course the first question is: what is the reason for the apparently stupid limitation?

As mentioned before, the Python `flex` types are implemented as instantiations of C++ class templates. This ensures that all array operations are very fast. However, from the discussion in the previous section it follows that exposing the full class with its many member functions to Python **for each element type** (`int`, `double`, `miller::index<>`, etc.) creates very sizable object files. If only *homogeneous*

operators `int * int`, `double * double`, etc.) are used the combined size of the object files scales linearly with the number of element types involved. However, if the library is expanded to support *heterogeneous* operators (`int * double`, `double * int`, etc.) the combined object files grow proportional to the square of the number of array element types involved! With current technology this is simply prohibitive.

Limitations of the kind discussed here will apply to any hybrid dynamically/statically typed system. In the broader picture the limitation shown above is just one typical example. If we want to enjoy the many benefits of using Python *and* have a system that produces results with a reasonable runtime efficiency we have to adopt the approach of sparsely sampling the space of possible C++ template instantiations. For this idea to work in practice we need a powerful and easy to use language-integration tool as discussed in the next section.

9. Building bridges

The cctbx project has evolved together with the Boost.Python library. All Python/C++ bindings in the cctbx project are implemented using this library. Here is a simplified example of how it works in practice:

This is the C++ class that we want to use from Python:

```
//! Parallelepiped that contains an asymmetric unit.
class brick
{
  public:
    //! Constructor.
    /*! Determines the parallelepiped given a space group type.
     */
    explicit
    brick(space_group_type const& sg_type);

    //! Formats the information about the brick as a string.
    /*! Example: 0<=x<=1/8; -1/8<=y<=0; 1/8<z<7/8
     */
    std::string as_string() const;

    //! Tests if a given point is inside the brick.
    bool is_inside(tr_vec const& p) const;
};
```

These are the corresponding Boost.Python bindings:

```
class_<brick>("brick", no_init)
  .def(init<space_group_type const&>())
  .def("__str__", &brick::as_string)
  .def("is_inside", &brick::is_inside)
;
```

And here is how the class is used in Python:

```
>>> from cctbx import sgtbx
>>> brick = sgtbx.brick(sgtbx.space_group_type("I a -3 d"))
>>> print brick
0<=x<=1/8; -1/8<=y<=0; 1/8<z<7/8
>>> brick.is_inside(sgtbx.tr_vec((0,0,0)))
0
```

Typically it only takes a few minutes to implement the Python bindings for a new class or function. Since it usually takes orders of magnitudes longer to implement C++ classes and functions the extra time spent on the Python bindings is in general negligible.

10. Thinking hybrid

Boost.Python's ease of use enables us to *think hybrid* when developing new algorithms. We can conveniently start with a Python implementation. The rich set of precompiled tools included in the scitbx and the cctbx gives us a head start because many operations are already carried out at C++ speed even though we are only using Python to assemble the new functionality. If necessary, the working procedure can be used to discover the rate-limiting sub-algorithms. To maximize performance these can be reimplemented in C++, together with the Python bindings needed to tie them back into the existing higher-level procedure.

To give an example, this approach was used in the development of the *Euclidean model matching algorithm* found in the cctbx (`cctbx/euclidean_model_matching.py`). This algorithm is used to compare heavy-atom substructures or anomalously scattering substructures from isomorphous replacement or anomalous diffraction experiments. The algorithm was first implemented in about 300 lines of pure Python. We wrote another 200 lines of comprehensive regression tests for thoroughly debugging the implementation. For a while the pure Python code actually worked fast enough for us, until we started to work with a very large substructure with 66 anomalous scatterers. Some simple optimizations of the Python implementation resulted only in a modest speedup, but after replacing about 30 lines of Python with a C++ implementation the algorithm runs about 50 times faster.

Of course there is still more to gain by reimplementing the entire algorithm in C++. However, one has to keep in mind that developing C++ code is typically much more time-consuming than developing in Python. For example, the 30 lines of Python mentioned in the previous paragraph turned into more than 100 lines of C++, not counting the additional 13 lines for the `Boost.Python` bindings. It is also important to keep in mind that developing maintainable C++ code requires much more hard-earned working experience than developing useful Python code. C++ has many pitfalls that one must learn to avoid. In contrast the Python language is structured in a way that steers even the novice programmer onto safe routes. In fact, Python was originally conceived as a language for teaching programming. Amazingly this heritage is still preserved even though Python has grown to be a very richly featured language.

11. A piece of cake

Conceptually it is a trivial task to compile and link portable source code. However, in real life this is one of the most time-consuming nuisances, in particular if multiple, diverse platforms have to be supported. In the version 1.0 release of the cctbx we made an attempt to address this with the home-made *fast track* build system. Of course home-made is often good enough, but a professional solution is almost always better, especially if it comes with no strings attached.
Fortunately in the meantime such has system has become available: SCons, short for Software Construction tool. This is a perfect fit for the cctbx because the SCons developers have apparently adopted a similar "professional is better than home-made" philosophy: SCons is implemented in pure Python, and SCons configuration files (the equivalent of Makefiles) are pure Python scripts. SCons has

many advantages compared to a traditional make-based build system. To quote some points from the SCons documentation:

- Global view of all dependencies -- no more multiple build passes or reordering targets to build everything.
- Reliable detection of build changes using MD5 signatures -- no more "clock skew detected, build may be incomplete".
- Built-in support for C, C++, Fortran, Yacc and Lex.
- Improved support for parallel builds - like make -j but keeps N jobs running simultaneously regardless of directory hierarchy.
- Building from central repositories of source code and/or pre-built targets.
- Designed from the ground up for cross-platform builds, and known to work on Linux, POSIX, Windows NT, Mac OS X, Tru64 Unix, and OS/2.

When we moved from our home-grown build system to SCons we found all these points to be perfectly true. It only took very little effort to write a small configure script for setting up a master SConstruct file based on the user's choice of which cctbx modules to use and which to ignore. New modules can easily be tied into this system simply by providing a SConstruct file in the module's top-level directory. The author of the new module has complete control over the build process. The existing settings can simply be reused, customized, or totally replaced, all within one uniform and 100% platform-independent framework, the Python language.

12. Bundling up

As mentioned in the introduction we are currently in the process of completing and extending the documentation of the cctbx modules. When we are finished we will create a self-contained bundle including the four cctbx modules and all its dependencies (Python, Boost, SCons). For Windows and potentially for some Unix platforms we will provide binary distributions that allow users to directly start working with the Python interfaces without worrying about the compilation. However, since SCons will be included, anybody with a working C++ compiler and a burning desire to work in that language has immediate access to all the tools that we are using in our development.

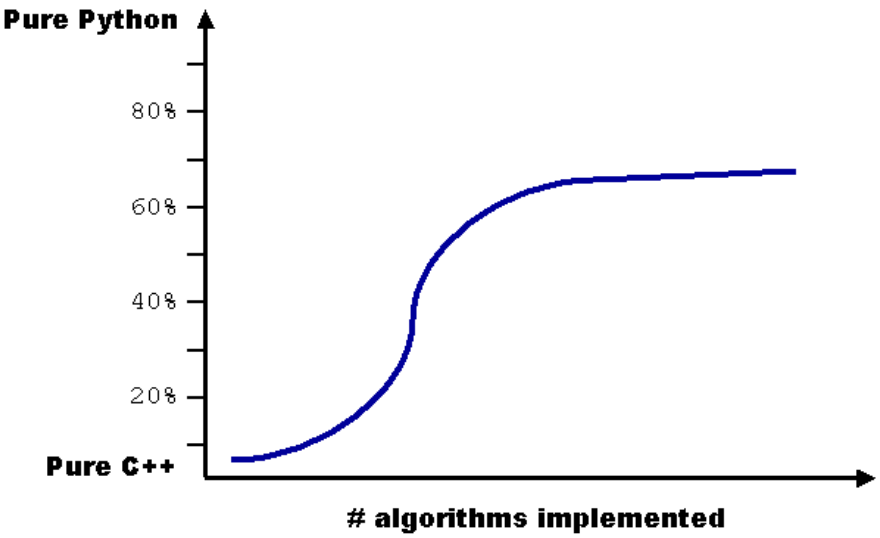
We'd like to emphasize that all C++ libraries in the cctbx project can still be used without Python, as was the case in cctbx version 1.0. The only restriction is that our build system obviously depends on Python. But, Python being more portable than C++, this should never really be a restriction in practice. And of course there is nothing which prevents our C++ libraries from being compiled with other tools.

13. Conclusion

The cctbx 2.0 covers a wide spectrum of fundamental algorithms and data structures needed for crystallographic software development. However, there is still an important blank spots on the map, namely fast Fourier transform based computations of derivatives of the structure factor equation. This is the last essential block of functionality required for building an efficient refinement program for macro-molecular structures. After the release we will focus our attention on removing this blank spot from the map.

Of course there is a plethora of other more specific algorithms that are needed to further the automation of macro-molecular structure determination, and some of these will still be general enough to warrant inclusion in a library to enable their use in different contexts. Therefore we expect the cctbx project to

grow continually for some time to come. The split of the cctbx project into several submodules ensures that this growth will be manageable, and that developers that only need a subset of the functionality do not have carry around large unused packages.
Looking back, the cctbx started out mainly as a library of C++ classes with 'C' heritage, and for a while the growth was mainly concentrated on the C++ parts. However, a very important difference between the 1.0 release and the upcoming 2.0 release is that the Python parts now constitute a much more significant fraction of the total sources. We expect this trend to continue, as illustrated qualitatively in this figure:



This figure shows the ratio of newly added C++ and Python code over time as new applications are implemented. We expect this ratio to level out near 70% Python. From an inside viewpoint the increasing ability to solve new problems mostly with the easy-to-use Python language rather than a necessarily more arcane statically typed language is the return on a major investment, namely our involvement in the development of Boost.Python. From an outside viewpoint we hope that the ability to solve some problems entirely using only Python will enable a larger group of scientist to participate in the rapid development of new algorithms. It is also important to notice that Python is an ideal language for integrating diverse existing tools, no matter which language they are written in. If portability is not an issue this can be a great solution to some problems. We are convinced that the cctbx can be very useful as an intelligent mediator between otherwise incompatible crystallographic applications.

14. Acknowledgments

We would like to thank David Abrahams for creating the amazing Boost.Python library and for patiently supporting the entire open source community. The iotbx module is developed and maintained by Nick Sauter who also made many suggestions regarding the design of the other modules. We would like to thank Airlie McCoy for allowing us to adapt some parts of the Phaser package (FFT structure factor calculation). Kevin Cowtan has contributed algorithms for the handling of reciprocal space asymmetric units. We are also grateful for his development of the Clipper library from which we have adapted some source code fragments. Our work was funded in part by the US Department of Energy under Contract No. DE-AC03-76SF00098. We gratefully acknowledge the financial support of NIH/NIGMS.

Link to references: http://cci.lbl.gov/publications/