cctbx news: Fast triplet generator for direct methods, Gallery of direct-space asymmetric units, et. al.

Ralf W. Grosse-Kunstleve, Buddy Wong 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/ and http://cctbx.sourceforge.net/

Introduction

The Computational Crystallography Toolbox (cctbx - http://cctbx.sourceforge.net/) is an open-source library of reusable crystallographic algorithms. In this article we give an overview of recent developments.

Fast triplet generator for direct methods

For almost a year the cctbx has included an experimental triplet generator in the cctbx.dmtbx (direct methods toolbox) submodule. This experimental code has now been replaced by a much faster version. In addition the interface was refactored to make it more accessible. Like most components of the cctbx, the triplet generator is scriptable from Python. Here is a simple stand-alone script that generates a list of unique Miller indices, searches for triplet phase relations and shows some statistics:

```
from cctbx import dmtbx
from cctbx import miller
from cctbx import crystal
from cctbx.array_family import flex
import time

crystal_symmetry = crystal.symmetry(
  unit_cell=(10,10,15,90,90,120),
  space_group_symbol="P63/mmc")
miller_set = miller.build_set(
  crystal_symmetry=crystal_symmetry,
  anomalous_flag=False,
  d_min=1.0)
triplets = dmtbx.triplet_generator(miller_set)
print "triplets per reflection: min,max,mean: %d, %d, %.2f" % (
  flex.min(triplets.n_relations()),
  flex.max(triplets.n_relations()),
  flex.mean(triplets.n_relations().as_double()))
print "total number of triplets:", flex.sum(triplets.n_relations())
print "cpu time used: %.2f" % time.clock()
```

(The flex module provides multi-dimensional arrays and associated functions.) The output of the script on a Redhat 8.0 Linux system (2.7 GHz):

```
triplets per reflection: min,max,mean: 1476, 4560, 2283.31
total number of triplets: 666727
cpu time used: 0.43 seconds
```

In a structure solution procedure the triplet search has to be performed only once at startup. The absolute runtime for the triplet search is so short that it is negligible in the context of the complete procedure. The runtime for the largest problem that we have worked on so far (a substructure with 160 sites) is 12.6 seconds on the same platform, generating almost 15 million triplets.

The dmtbx.triplet_generator() call above creates an object containing all the triplets. This object has associated functions, called *methods* in Python terminology, or *member functions* in C++ terminology. Above we have used the n_relations() method already. Another method is apply_tangent_formula() and it is used like this:

```
amplitudes = flex.double(miller_set.size(), 1) # fake data
phases = flex.double(miller_set.size(), 0) # fake phases
new_phases = triplets.apply_tangent_formula(amplitudes, phases)
```

This changes all phases, using all phases. Other tangent refinement protocols are supported. For example:

```
# keep the first 40% of the phases fixed
selection_fixed = flex.bool(int(miller_set.size()*0.4), True)
selection_fixed.resize(miller_set.size(), False)
new_phases = triplets.apply_tangent_formula(amplitudes, phases,
  selection_fixed, use_fixed_only=True, reuse_results=True)
```

The full interface documentation is available online (C++ interfaces -> dmtbx) at the cctbx site.

Gallery of direct-space asymmetric units

We have created a reference file defining direct-space asymmetric units for the 230 crystallographic space groups, following the conventions of the International Tables for Crystallography, Volume A (ITVA). However, ITVA only defines the volumes of the asymmetric units. In contrast our reference file includes additional conditions for the facets, edges and vertices. For example, consider space group P2 (No. 3). ITVA lists the following conditions:

```
Asymmetric unit   0 <= x <= 1; 0 <= y <= 1; 0 <= z <= 1/2
```

Clearly, points with coordinates 0,y,z and 1,y,z are redundant due to the periodicity of the crystal lattice. This can be accounted for by modifying the conditions in the following way:

```
Asymmetric unit   0 <= x < 1; 0 <= y < 1; 0 <= z <= 1/2
```

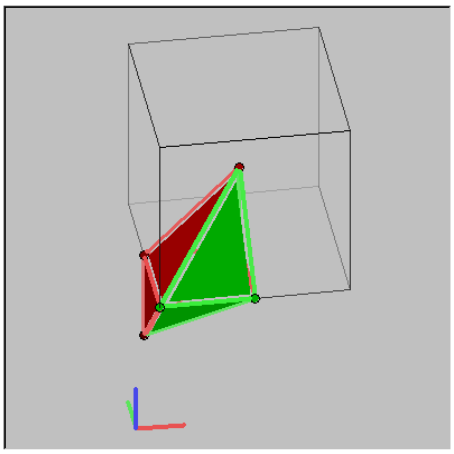
This correction might seem subtle at first, but avoids difficult problems when working with grids over the unit cell as many crystallographic algorithms do. Unfortunately there is still more to be corrected. There are two-fold axes parallel to the facets with z=0 and z=1/2. The two-fold axes generate redundancies in these facets. To account for this we need *facet-specific* conditions. In words:

> If z is exactly equal to zero or 1/2, x must be less than or equal to 1/2.

Our reference file includes additional conditions that make all 230 asymmetric units completely free of redundancies. For the cubic space groups we follow the conventions established by Koch & Fischer (1974), Acta Cryst. A 30, 490-496 (although in general only for the facets but not for the edges and vertices to keep the expressions simple).

Associated with our reference file is a set of tools, constituting about 1600 lines of Python code. These tools are used to generate our online gallery of direct-space asymmetric units, available at http://cci.lbl.gov/asu_gallery/ . For example:

Space group: P 21 3 (No. 198)



[Index] [Previous] [Next] [Grid view]

Surface area: green = inside the asymmetric unit, red = outside
Basis vectors: a = red, b = green, c = blue

Number of vertices: 6	Number of facets: 7
1/2, 1/2, 1/2 1/2, 1/2, 0 0, 1/2, 0 0, 1/2, -1/2 0, 0, 0 1/2, 0, 0	x>=0 [y<=0] x<=1/2 y<=1/2 [z>0 & x<=1/2 [z<1/2]] x-z<=1/2 [x+y<=1/2] x-z>=0 [x-y>=0] y+z>=0 y-z>=0 [Guide to notation]

The green-and-red coloring of the facets reflects the additional conditions.

All the information shown at the web pages is automatically derived from the primary information in the reference file. The primary information used to generate the web page above is encoded in this Python function:

```
def asu_198(): # P 21 3
  return (direct_space_asu('P 2ac 2ab 3')
    & x0(-y0)
    & x2
    & y2(+z0 & x2(+z2))
    & cut((-1,0,1), r1/2)(m2)
    & cut((1,0,-1), 0)(p0)
    & cut((0,1,1), 0)
    & cut((0,1,-1), 0)
  )
```

There is a one-to-one correspondence between this function and the notation shown in the box starting with Number of facets above. The notation used in the reference file was easier to work with during development and lends itself to easy algorithmic manipulation. The notation shown on the web is easier to understand for humans (follow the Guide to notation link to learn more).

That we are able to easily provide the best notation for a given purpose *without introducing difficult to manage redundancies* is due to the rich set of abstraction facilities supported by the Python language. The first thing to notice is that the Python function with the primary information is just that, a pure Python function. We did not have to invent a new syntax to parameterize the asymmetric units. When the Python

function above is called it creates a new *object*, more specifically an *instance* of the direct_space_asu *class*. *Operator overloading* is used to define the facets of the asymmetric unit. We have chosen to use the & operator for this purpose. The working of this can be better understood by considering that the code above is equivalent to:

```
def asu_198(): # P 21 3
  a = x0(-y0)
  b = x2
  c = y2(+z0 & x2(+z2))
  d = cut((-1,0,1), r1/2)(m2)
  e = cut((1,0,-1), 0)(p0)
  f = cut((0,1,1), 0)
  g = cut((0,1,-1), 0)
  return direct_space_asu('P 2ac 2ab 3') & a & b & c & d & e & f & g
```

The assignments create references to objects representing the facets. The direct_space_asu instance has a rule, in the form of the method __and__, that defines what the & operator does. Here is the *complete* code to make this work:

```
class direct_space_asu:

  def __init__(self, hall_symbol):
    self.hall_symbol = hall_symbol
    self.facets = [] # an empty list

  def __and__(self, obj):
    self.facets.append(obj) # adds the facet to the list
    return self
```

We can chain the & operators because the __and__ method returns self, which is "the direct_space_asu instance itself" (equivalent to *this in C++).

Each of the facets of an asymmetric unit is defined by a *cut plane* dividing space into "inside" and "outside." Successive cuts lead to a bounded volume. The additional conditions mentioned before are simply cuts that only apply to a certain facet instead of globally. An intersection of two cuts forms an edge. Some asymmetric units require additional *edge-specific conditions*. I.e. a facet is defined by potentially nested expressions involving one or more cuts which are sometimes combined by boolean operators (see the function above). Giving a full account of this "cut algebra" is beyond the scope of this article, but we will show the two techniques that are at the heart of the implementation.

The first technique that we use is analogous to expression templates that C++ programmers might be familiar with. However being implemented in Python it should have a different name, say *expression objects*, and it is, of course, much more approachable. Here is the *complete* code that makes boolean expressions of cuts work:

```
class cut_expr_ops:
  def __and__(self, other): return cut_expression("&", self, other)
  def __or__(self, other): return cut_expression("|", self, other)

class cut_expression(cut_expr_ops):

  def __init__(self, op, lhs, rhs):
    self.op = op
    self.lhs = lhs
    self.rhs = rhs

class cut(cut_expr_ops):

  def __init__(self, n, c):
    self.n = n
    self.c = c
```

Note that this requires nothing but Python to work, not even the cctbx. For the meaning of `n` and `c` refer to http://cci.lbl.gov/asu_gallery/facet_notation.html .

Before we explain how the code above works let's look at some illustrative examples. Let all the letters below be references to `cut` instances. The code above enables arbitrarily complex, nested expressions:

```
a & b
a | b
a & (b | c)
a & (b & (c | d))
```

The result of each of these expressions is an *object representing the expression*. When Python evaluates the expressions it leads to the `cut_expression` class recursively keeping track of the boolean operations, i.e. the operator symbol (`&` or `|`) and the operands on both sides. The parentheses are resolved for us by Python, so we don't have to keep track of them ourselves.

For people who have worked with language translators before (e.g. yacc and lex): our expression object is equivalent to an *abstract syntax tree*. The difference to the conventional approach is that we don't have to write our own parser. Instead we use operator overloading to leverage Python itself as a parser for our domain-specific syntax. The only restriction is that our syntax must be compatible with the Python syntax. We view this as a useful restriction because it enforces a consistent and familiar syntax.

The code above uses inheritance. For readers not familiar with inheritance, the code could be rewritten as follows:

```
class cut_expression:

  def __init__(self, op, lhs, rhs):
    self.op = op
    self.lhs = lhs
    self.rhs = rhs

  def __and__(self, other): return cut_expression("&", self, other)
  def __or__(self, other): return cut_expression("|", self, other)

class cut:

  def __init__(self, n, c):
    self.n = n
    self.c = c

  def __and__(self, other): return cut_expression("&", self, other)
  def __or__(self, other): return cut_expression("|", self, other)
```

This is doing exactly the same job, but is poor style because it unnecessarily introduces explicit duplication of the same code, and potentially multiple places to fix the same bug.

We can easily evaluate the expression objects in different ways by adding pairs of methods, one to the cut class to do the actual work, and one in the cut_expression class to enable recursive expressions. For example, the is_inside() method to test if a given point is in the asymmetric unit is implemented as follows:

```
class cut_expr_ops:
  def __and__(self, other): return cut_expression("&", self, other)
  def __or__(self, other): return cut_expression("|", self, other)

class cut_expression(cut_expr_ops):

  def __init__(self, op, lhs, rhs):
    self.op = op
    self.lhs = lhs
    self.rhs = rhs

  def is_inside(self, point):
    if (self.op == "&"):
      return self.lhs.is_inside(point) and self.rhs.is_inside(point)
    else:
      return self.lhs.is_inside(point) or self.rhs.is_inside(point)

class cut(cut_expr_ops):

  def __init__(self, n, c):
    self.n = n
    self.c = c

  def is_inside(self, point):
    # do the actual work, return True or False
```

We have implemented similar pairs of methods for rewriting the cut expressions in different formats, or for change-of-basis transformations. Alternatively it is possible to evaluate the expression objects by introspection without adding additional methods. This approach is used, e.g. in the implementation of the computation of the polygon edges and vertices shown in the pictures on the web.

A second fundamental technique is used to implement the recursive nesting of cut expressions. Python's __call__ method (equivalent to operator() in C++) allows objects to act like functions. We chose this as the syntax for defining additional facet- and edge-specific conditions. For example:

```
x0 = cut((1,0,0), 0)) # equivalent to x >= 0
y2 = cut((0,-1,0), r1/2)) # equivalent to y <= 1/2
z4 = cut((0,-1,0), r1/4)) # equivalent to z <= 1/4
x0(y2)
x0(y2(z4))
```

The expression x0(y2) means that a point is inside the asymmetric unit only if x>=0. If x is exactly zero, the point is inside the asymmetric unit only if y<=1/2. The expression x0(y2(z4)) is similar, but if x is exactly zero and y is exactly 1/2 the point is inside the asymmetric unit only if z<=1/4.

This is the corresponding implementation:

```
class cut(cut_expr_ops):

  def __init__(self, n, c, cut_expr=None):
    self.n = n
    self.c = c
    self.cut_expr = cut_expr

  def __call__(self, expr):
    return cut(self.n, self.c, cut_expr=expr)
```

The reader is invited to study the implementation of the is_inside() method to see how the nested cut expressions are used. The full source code is in the directory cctbx/cctbx/sgtbx/direct_space_asu in the cctbx source code distribution (see below).

Macintosh OS 10 support

As of May 2003 the cctbx is fully functional under Mac OS 10. However, there are a few Mac-specific issues. Firstly, Python 2.3 must be available, and it must be installed as root (the Mac is the only platform with this requirement). Secondly, the only C++ compiler that currently works for us under Mac OS 10 is standard gcc 3.3 as available at http://gcc.gnu.org/ . Unfortunately compiler bugs prevent the use of optimization. Therefore the binaries are relatively slow compared to other platforms. We hope that this will improve as new gcc versions are released. Currently Apple's compiler cannot be used to install the cctbx. More detailed Mac-specific information is available at http://cci.lbl.gov/cctbx_build/mac_os_x_notes.html . We will continuously update this web page to reflect the latest developments.

Automatic builds, source code & binary bundles

We have developed a system for automatically generating self-contained cctbx source code bundles and self-extracting binary bundles for a variety of platforms. The bundles are available at http://cci.lbl.gov/cctbx_build/ , complete with build logs and results of automatic regression tests. The cctbx online documentation is also continuously updated at http://cctbx.sourceforge.net/current_cvs/ . It is highly recommended to use the current bundles instead of the old cctbx 1.0 release from November 2001. We take great care that the bundles are consistent and in working order on all platforms. The C++ interfaces have changed significantly since the 1.0 release but are very stable since the fall of 2002.

All bundles are marked with a time stamp and we are archiving all builds that worked on all platforms. Practically this is like a full traditional release, only every other day. This enables us to use a novel release policy. Instead of posting announcements *after* making significant changes we will post announcements *before* starting developments that break backwards compatibility. For example, we are planning to use Python 2.3 features shortly after Python 2.3 final becomes available (scheduled for August). We will specifically set aside a release as "the last release that worked with Python 2.2" and create a source code branch for bug fixing only.

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.