cctbx news: Geometry restraints and other new features

Ralf W. Grosse-Kunstleve, Pavel V. Afonine and Paul D. Adams,
Computational Crystallography Initiative, Lawrence Berkeley National Laboratory, One Cyclotron Road, BLDG 4R0230, Berkeley, California, 94720-8235, USA - Email : RWGrosse-Kunstleve@lbl.gov ; WWW: http://cci.lbl.gov/

1: Introduction

The Computational Crystallography Toolbox (cctbx, http://cctbx.sourceforge.net/) is the open-source component of the Phenix project (http://www.phenix-online.org/). Currently we are finalizing an initial version of the Phenix refinement procedures. The emphasis of this article is an introduction to the underlying open-source libraries for the handling of geometry restraints, molecular mask calculations, bulk-solvent correction, likelihood-based target functions for crystallographic refinement, and the relative scaling between these target functions and the geometry restraints.

Some of the functionality covered in this newsletter is implemented in the new top-level `mmtbx` module ("macro-molecular toolbox") of the cctbx project. Due to technical reasons the `mmtbx` source code is not currently hosted at the SourceForge site even though it is covered by the same open license as the rest of the cctbx project. However, the full `mmtbx` sources are included in the bundles available at the http://cci.lbl.gov/cctbx_build/ download site. For the future we are planning to move the `mmtbx` code to the SourceForge site.

In order to interactively run the examples scripts shown below, the reader is highly encouraged to visit http://cci.lbl.gov/cctbx_build/ and to download one of the completely self-contained, self-extracting binary cctbx distributions (supported platforms include Linux, Mac OS X, Windows, IRIX, and Tru64 Unix). On recent machines the installation requires significantly less than one minute of time. Even on the slowest machine available to us (SGI O2, R5000, 300MHz) a binary installation takes less than three minutes without requiring any manual intervention. A cctbx installation is non-intrusive and does *not* require system privileges. Traceless removal is as easy as running `rm -rf` or dragging a single folder to the Recycle Bin. Nobody will know you did it!

All example scripts shown below were tested with cctbx build 2004_08_05_0113.

2: from cctbx import geometry_restraints

Commonly refinement programs support inclusion of prior chemical knowledge such as bond lengths and bond angles via *geometry restraints*. The cctbx implementation of six types of geometry restraints is located in the `cctbx.geometry_restraints` module. The restraint types available are:

- bond
- nonbonded repulsion
- angle
- dihedral (same as torsion)
- chirality
- planarity

The `cctbx.geometry_restraints` module is designed as a uniform library to support both small-molecule and macro-molecular refinement. In general the requirements for small-molecule and macro-molecular refinement are quite different. For example, some macro-molecular refinement programs have limited or no support for symmetry bonds (i.e. bonds to atoms generated by symmetry), or bonds involving sites on special positions. This is not surprising because of the 27141 `pdb*.ent` files found at

`ftp.rcsb.org` on July 27, 2004, only 256 include `LINK` records defining symmetry bonds (out of a total of 7078 files with `LINK` records), and only 534 files include heavy atoms on special positions (out of a total of 2746 files with atoms on special positions; most are water molecules). This means a little less than 98% of all macro-molecular structures can be refined with a program that does not correctly handle symmetry bonds or special positions.

In contrast, special positions and symmetry bonds are the norm in small-molecule crystallography, not the exception. The is particularly true for inorganic materials. In theory, a system that handles geometry restraints for the refinement of small molecules and inorganic materials will therefore immediately be able to handle all symmetry aspects of 100% of all macro-molecular structures. In practice however many small-molecule programs do not lend themselves to be used for macro-molecular work. This is due to fairly trivial nuisances such as unsuitable compiled-in limits on the number of atoms or bonds that can be processed, or more seriously, use of algorithms that scale with the square of the number of atoms and become prohibitively slow for large macro-molecular structures. Even more seriously, aspects that are crucial for the handling of macro-molecular structures may not be covered at all, such as nonbonded interactions, dihedral or chirality restraints.

The `cctbx.geometry_restraints` module was designed under the "completeness and correctness first, optimize later" paradigm. The handling of all symmetry aspects of bonded and nonbonded pair interactions is as complete as one expects to find in a small-molecule application, but the algorithms and data structures are optimized for handling a large number of atoms. I.e. the macro-molecular field will benefit from the rigorous treatment of symmetry, and the small-molecule field will benefit from speed increases. Owing to the facilities provided by the modern programming languages used (Python and C++), compiled-in limits are a problem of the past. The memory for all data is dynamically allocated.

2.1: cctbx.geometry_restraints.bond

Given the Cartesian coordinates of two bonded sites, the ideal bond length, and a weight, we can run the following Python code:

```
from cctbx import geometry_restraints
bond = geometry_restraints.bond(
  sites=[(1,2,3),(2,3,4)],
  distance_ideal=2,
  weight=10)
print "distance_model:", bond.distance_model
print "delta:", bond.delta
print "residual:", bond.residual()
print "gradients:", bond.gradients()
```

Output:

```
distance_model: 1.73205080757
delta: 0.267949192431
residual: 0.717967697245
gradients: ((3.0940107675850306, 3.0940107675850306, 3.0940107675850306),
(-3.0940107675850306, -3.0940107675850306, -3.0940107675850306))
```

The `bond` class performs all the basic computations required for gradient-driven refinement. The `residual()` is the contribution of this bond to the total "energy" of the geometry term of the target function and defined in the usual way (e.g. Hendrickson, 1985) as `weight * bond.delta**2`, where `bond.delta` is the difference `bond.distance_ideal - bond.distance_model`.

2.2: cctbx.geometry_restraints.bond_simple_proxy

Of course, during structure refinement the coordinates are changed. Therefore we need a data structure, in new speak an object, with some type of reference to the bonded sites along with `distance_ideal` and `weight`. We call this object `bond_simple_proxy` and it works like this:

```
from cctbx import geometry_restraints
from cctbx.array_family import flex
sites_cart = flex.vec3_double([
  (1,2,3),
  (2,3,4),
  (1,3,5)])
bond_proxy_1 = geometry_restraints.bond_simple_proxy(
  i_seqs=[0,1],
  distance_ideal=2,
  weight=10)
bond_proxy_2 = geometry_restraints.bond_simple_proxy(
  i_seqs=[1,2],
  distance_ideal=1.8,
  weight=20)
for bond_proxy in [bond_proxy_1, bond_proxy_2]:
  bond = geometry_restraints.bond(
    sites_cart=sites_cart,
    proxy=bond_proxy)
  print "sites:", bond.sites
  print "residual:", bond.residual()
```

Output:

```
sites: ((1.0, 2.0, 3.0), (2.0, 3.0, 4.0))
residual: 0.717967697245
sites: ((2.0, 3.0, 4.0), (1.0, 3.0, 5.0))
residual: 2.97662350914
```

`sites_cart` is an array of Cartesian coordinates for three sites. The `i_seq` (Index into SEQuence of sites) are the references mentioned above; they are simply integer indices into the `sites_cart` array. A `bond_simple_proxy` is essentially a `bond` with one level of indirection. We can turn a `bond_simple_proxy` into a `bond` by providing the `sites_cart` array referenced to by the `i_seq`. Then we can use the methods of the `bond` object to obtain the desired information as shown before, for example the `residual()` as in this example or the `gradients()` as in the previous example.

2.3: cctbx.geometry_restraints.shared_bond_simple_proxy

In all likelihood we will have to handle a considerable number of bonds. Therefore the next data structure we need is an array of bond proxies. The previous example can be rewritten to use "shared" arrays with bond proxy objects as the elements:

```
bond_proxies = geometry_restraints.shared_bond_simple_proxy()
bond_proxies.append(geometry_restraints.bond_simple_proxy(
  i_seqs=[0,1],
  distance_ideal=2,
  weight=10))
bond_proxies.append(geometry_restraints.bond_simple_proxy(
  i_seqs=[1,2],
  distance_ideal=1.8,
  weight=20))
for bond_proxy in bond_proxies:
  bond = geometry_restraints.bond(
    sites_cart=sites_cart,
    proxy=bond_proxy)
```

If the number of bonds is very large as in the case of macro-molecular structures, the Python `for` loop will become a performance bottleneck. Python is a dynamically typed language and therefore very convenient to use, but the convenience is payed for with a runtime penalty of one to two orders of magnitude. The remedy is to reimplement the Python loop in C++ and to do the *vector operation* at the speed of a compiled language. Using Boost.Python (http://www.boost.org/libs/python/doc/, see also Grosse-Kunstleve & Adams, 2003), it is easy to make the C++ function available in Python. In this way we can, for example, obtain all `bond.delta` values with a single call from Python to C++:

```
bond_deltas = geometry_restraints.bond_deltas(
  sites_cart=sites_cart,
  proxies=bond_proxies)
print list(bond_deltas)
```

Output:

```
[0.2679491924311227, 0.38578643762690501]
```

The idea behind this approach is similar to the idea behind vector computers. Python is the slow but general scalar unit, C++ the fast but restricted vector unit. Filling the array of bond proxies is similar to loading the vector unit and the call from Python to C++ is the vector operation. More on the subject of combining Python and C++ can by found in the Newsletter No. 1 in this series (Grosse-Kunstleve & Adams, 2003).

As an aside, the array of bond proxies is called a "shared" array because it may have multiple owners. The lifetime of shared arrays is controlled by a reference count. If the reference count goes to zero (because all owning references go out of scope or are deleted explicitly) the memory for the array is automatically deallocated. This is one of the fundamental mechanisms used by Python and C++ for making memory management simple (compared to FORTRAN) and at the same time safe (compared to C).

3: Symmetry: Friend or Foe?

The astute reader will have noticed that symmetry was not mentioned in the introduction to the `bond`, `bond_simple_proxy`, and `shared_bond_simple_proxy` objects. How does the symmetry come into play? The `simple` part of the two latter symbols is already a hint that there must be something more complex, and that is of course the symmetry. While symmetry is always nice to look at and therefore appears to be a friend, when it comes to writing algorithms for the handling of symmetry it quickly becomes apparent that symmetry is a pretty bad foe. Symmetry introduces singularities and each singularity requires special attention. For example, the 230 crystallographic space groups can be understood as 230 unique singularities, each of which has a different set of singular positions known as special positions. Needless to say, each singularity requires a name or number and therefore we have space group symbols and numbers, Wyckoff tables, Wyckoff letters, site symmetry symbols, etc., etc. As a rule of thumb, the source code required for handling a problem complete with symmetry is at least ten

times the amount of source code required for the "simple" case. The handling of symmetry pair interactions and pair interactions involving sites in special positions is, unfortunately, not an exception.

We use the term *pair interaction* with reference to both bonded and nonbonded interactions. It comes as a little relief that the handling of bonded and nonbonded pair interactions is very similar. The main difference is the function used to compute the contributions to the total energy term for the geometry. In the case of bonded interactions it is the simple harmonic function `weight * bond.delta**2`, in the case of nonbonded interactions it is a more involved function of exponentials. However, up to the point of determining the `distance_model` required in both cases the algorithms are identical.

3.1: cctbx.crystal.direct_space_asu.asu_mappings

One important term we forgot to mention in the list of names and numbers required for the singularities introduced by symmetry is that of *asymmetric unit*. Before we can introduce symmetry pair interactions we have to get acquainted with the `cctbx.crystal.direct_space_asu.asu_mappings` class. The development of this class is based on the work published in the Newsletter No. 2 in this series (Grosse-Kunstleve et al., 2003). The web pages at http://cci.lbl.gov/asu_gallery/ are available for viewing the shapes of the standard asymmetric units as defined in the International Tables for Crystallography, Volume A. These shapes play a fundamental role in all cctbx algorithms involving pair interactions.
Pair interactions are commonly considered up to a certain cutoff distance, for example a maximum bond length when searching for bonds, or a maximum nonbonded distance when searching for nonbonded interactions. A fundamental consideration is that all pair interactions can be mapped by symmetry into the shape of the standard asymmetric unit expanded by a buffer region equivalent to the chosen cutoff distance. Let's dig out the simple `quartz_structure` introduced in the Newsletter No. 1 to see how this works in practice:

```
from cctbx import xray
from cctbx import crystal
from cctbx.array_family import flex
quartz_structure = xray.structure(
  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()
asu_mappings = quartz_structure.asu_mappings(buffer_thickness=2)
print "n_sites_in_asu_and_buffer:", asu_mappings.n_sites_in_asu_and_buffer()
```

The second to last line is a high-level interface provided by the `xray.structure` class for performing the process mentioned in the previous paragraph. First, the standard asymmetric unit is determined via lookup in the "reference file" introduced in Newsletter No. 2 and, if necessary, a change-of-basis transformation from the reference setting of the space group symmetry to the given setting (in the example the given setting is already the reference setting). Next, the asymmetric unit is expanded by moving the facets 2 Å out to generate the buffer region. Finally the space group symmetry is applied to the sites in order to fill the asymmetric unit including the buffer region. The end result is an instance of the class `cctbx.crystal.direct_space_asu.asu_mappings`. The output of running the example is:

```
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, Scattering, Multiplicity, Coordinates, Occupancy, Uiso
Si   Si     3 ( 0.5000  0.5000  0.3333) 1.00 0.2000
O    O      6 ( 0.1970 -0.1970  0.8333) 1.00 0.0000
n_sites_in_asu_and_buffer: 20
```

To understand the workings of the `asu_mappings` object we start with the `asu_mappings.mappings()` array provided by the object. For each scatterer in our `quartz_structure` there is one entry in the mappings array:

```
assert asu_mappings.mappings().size() == quartz_structure.scatterers().size()
for mappings in asu_mappings.mappings():
  print type(mappings), len(mappings)
```

Output:

```
<type 'tuple'> 3
<type 'tuple'> 17
```

This tells us that each element of the `asu_mappings.mappings()` array is a standard Python tuple, i.e. a list-like sequence of Python objects. We also learn that the first tuple has 3 elements and the second tuple has 17 elements. Each element represents exactly one site in the asymmetric unit or the buffer region. The first element of each tuple is always for the site in the asymmetric unit; by definition there can only be one. All following elements of each tuple represent sites in the buffer region. I.e. in this case there are 2 Si atoms in the 2 Å buffer region and 16 O atoms. To find out where they are we can use other facilities provided by the `asu_mappings` object. To keep the output short we concentrate on the 3 mappings for the Si atom:

```
for mapping in asu_mappings.mappings()[0]:
  print "i_sym_op:", mapping.i_sym_op()
  print "unit_shifts:", mapping.unit_shifts()
  print "symmetry operation:", asu_mappings.get_rt_mx(mapping)
  print
```

Output:

```
i_sym_op: 2
unit_shifts: (0, 0, -1)
symmetry operation: y,-x+y,z-1/3

i_sym_op: 0
unit_shifts: (0, 0, 0)
symmetry operation: x,y,z

i_sym_op: 1
unit_shifts: (1, 0, -1)
symmetry operation: x-y+1,x,z-2/3
```

Each `mapping` object stores the number of the symmetry operation and the unit shifts that were used to map the original site to the site in the asymmetric unit or the buffer region. To enforce consistency, a complete copy of the symmetry operations is stored inside the `asu_mappings` object. This is enables us to use `asu_mappings.get_rt_mx(mapping)` to compute the final symmetry operations giving the `mapping` object. ("`rt_mx`" stands for "rotation-translation matrix").

3.2: cctbx.crystal.neighbors_fast_pair_generator

We know that the silicon atoms in the `quartz_structure` are covalently connected to the oxygen atoms and that the Si-O bond distance is around 1.6 Å. This is how we find the bonds:

```
pair_generator = crystal.neighbors_simple_pair_generator(
  asu_mappings,
  distance_cutoff=1.7)
for pair in pair_generator:
  print pair.i_seq, pair.j_seq, pair.j_sym, pair.dist_sq**.5
```

Output:

```
0 1 0 1.61598604691
0 1 1 1.61598604691
0 1 12 1.61598604691
0 1 15 1.61598604691
1 0 1 1.61598604691
```

The `pair_generator` is a Python iterator that performs a simple-minded search with approximately N*N/2 iterations for all pair interactions within the given `distance_cutoff` of 1.7 Å, where N is the number of atoms. At each iteration we obtain a `pair` object with integer references into the `asu_mappings.mappings()` array as introduced in the previous section. The indices `pair.i_seq` and `pair.j_seq` are indices into the `asu_mappings.mappings()` array. The index `pair.j_sym` is an index into the `asu_mappings.mappings()[pair.j_seq]` tuple (see previous section). To avoid redundancies, only bonds that emanate from within the asymmetric unit are considered. Therefore we do not need a corresponding `i_sym` index; it is always 0. I.e. the three integer indices are sufficient to uniquely define a bond based on the `asu_mappings` object.

Alternatively we could generate the same list of pairs with the "fast" pair generator:

```
pair_generator = crystal.neighbors_fast_pair_generator(
  asu_mappings,
  distance_cutoff=1.7)
```

This alternative pair generator is designed for structures with a large number of sites. The interfaces of the simple and the fast pair generators are identical, but internally the fast generator is much more complex. The asymmetric unit including the buffer region is subdivided into cubes with a vertex length equivalent to `distance_cutoff`. In the search for pair interactions involving a given pivot site, only the cube of the pivot site and the 26 surrounding cubes have to be considered. The average number of sites per cube is approximately independent of the size of the structure. For a large number of sites the search time will therefore scale approximately linearly with the number of cubes instead of quadratically with the number of sites. This leads to dramatic increases in speed. For example (Linux, Xeon 2.8GHz):

```
number of atoms    time simple search    time fast search
  3500 (gere)           1.6 seconds           0.1
 59000 (groel)        377.4                  1.5
```

In practice there is no good reason for using the simple version of the pair generator. The main reason for keeping it in the library is to support a regression test that validates the fast generator.

3.3: cctbx.crystal.pair_asu_table

The `cctbx.crystal.pair_asu_table` is the center piece of the cctbx system for the handling of pair interactions involving symmetry. The internal `process_pair` member function of this C++ extension class is the heart of the center piece. It is responsible for generating symmetrically equivalent pair interactions and for the removal of redundant interactions. A full description of the algorithm implemented by the `process_pair` function is beyond the scope of this article even though the C++ source code comprises only 41 lines (see file `cctbx/include/cctbx/crystal/pair_tables.h`). However, the following example demonstrates the most important features:

```
pair_asu_table = crystal.pair_asu_table(asu_mappings=asu_mappings)
pair_asu_table.add_all_pairs(distance_cutoff=1.7)
```

The `pair_asu_table.add_all_pairs(distance_cutoff=1.7)` statement uses the fast pair generator as described in the previous section. When the first pair is processed by the `process_pair` function, the site symmetries of the two sites involved are applied to generate all symmetrically equivalent pairs. For the simple quartz structure, this step will already generate all pairs and add them to the `pair_asu_table` object. The pairs subsequently produced by the pair generator are found by lookup in the internal table and no further processing is necessary. At this stage the `pair_asu_table.table()` object managed by the `pair_asu_table` object will hold the data:

```
pair_asu_table.show()
```

Output:

```
i_seq: 0
  j_seq: 1
    j_syms: [0, 1, 12, 15]
i_seq: 1
  j_seq: 0
    j_syms: [0, 1]
```

`pair_asu_table.table()` is the most deeply nested data structure in the entire cctbx. In Python terms it is a *list of dictionaries associating integers with lists of lists*. If this appears overly complicated consider Einstein's famous quote: "Make everything as simple as possible, but not simpler." We are certain that `pair_asu_table.table()` is as simple as possible because each level of nesting represents a clear concept necessary to fully characterize symmetry pair interactions:

- The outermost list holds one entry per atom. The `i_seq` index is implied.
- Each entry is a dictionary. The keys are the `j_seq` indices.
- The value corresponding to each `j_seq` index is a list of groups of `j_sym` indices.
- The interactions defined by the `j_sym` indices in each group are symmetrically equivalent.

Since the interactions are fully characterized it is now very simple to extract the interactions unique under symmetry. Since we are not concerned about the directionality of the pair interactions (i.e. A-B is the same as B-A) we only have to consider interactions with `i_seq <= j_seq`, and we only need the first element from each group of symmetrically equivalent interactions. This procedure is implemented as the `extract_pair_sym_table` method of `pair_asu_table`:

```
pair_sym_table = pair_asu_table.extract_pair_sym_table()
pair_sym_table.show()
```

Output:

```
i_seq: 0
  j_seq: 1
    -y,x-y,z-1/3
i_seq: 1
```

This shows that the quartz structure has only one unique Si-O bond under symmetry.
An important point to note is that `pair_sym_table` is, in contrast to `pair_asu_table`, independent of the `asu_mappings` object; hence the naming. `pair_sym_table` is therefore suitable for communicating connectivity between algorithms that may require different `asu_mapping` objects due to shifts in coordinates or modified distance cutoffs. Here is how we can re-generate a new `pair_asu_table` from a `pair_sym_table`:

```
new_asu_mappings = quartz_structure.asu_mappings(buffer_thickness=5)
new_pair_asu_table = crystal.pair_asu_table(asu_mappings=new_asu_mappings)
new_pair_asu_table.add_pair_sym_table(sym_table=pair_sym_table)
new_pair_asu_table.show()
```

Output:

```
i_seq: 0
  j_seq: 1
    j_syms: [0, 3, 55, 68]
i_seq: 1
  j_seq: 0
    j_syms: [0, 8]
```

In this case the `j_syms` have changed compared to the output of `pair_asu_table.show()` because the buffer region of `new_pair_asu_table` is larger compared to that of the initial `pair_asu_table`.
The new iotbx.show_distances command provides an easy to use interface to the core functionality described in this section. This command reads files in the simple format introduced by the `kriber` program (http://www.crystal.mat.ethz.ch/Software/Kriber). For example:

```
*quartz

P 62 2 2
 5.01 5.47
Si   0.5000  0.5000  0.3333
O    0.1970 -0.1970  0.8333
---------------------------
```

The full command is:

```
iotbx.show_distances quartz_structure --distance_cutoff=1.7
```

Output:

```
strudat tag: quartz

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, Scattering, Multiplicity, Coordinates, Occupancy, Uiso
Si   Si     3 ( 0.5000  0.5000  0.3333) 1.00 0.0000
O    O      6 ( 0.1970 -0.1970  0.8333) 1.00 0.0000
```

```
Si(1):          pair count:   4  <<  0.5000,  0.5000,  0.3333>>
  O(2):        1.6160             (  0.1970,  0.3940,  0.5000)
  O(2):        1.6160 sym. equiv. (  0.3940,  0.1970,  0.1667)
  O(2):        1.6160 sym. equiv. (  0.8030,  0.6060,  0.5000)
  O(2):        1.6160 sym. equiv. (  0.6060,  0.8030,  0.1667)
O(2):           pair count:   2  <<  0.1970, -0.1970,  0.8333>>
  Si(1):       1.6160             (  0.0000, -0.5000,  0.6667)
  Si(1):       1.6160 sym. equiv. (  0.5000,  0.0000,  1.0000)

Pair counts: [4, 2]
```

The implementation of this command can be found in the file `iotbx/iotbx/command_line/show_distances.py`.

3.4: Nonbonded exclusions

In the refinement of macro-molecular structures it is common to use nonbonded pair interactions, e.g. Lennard-Jones potentials or empirical "repulsive force fields." For sites that are not bonded but are within a certain distance (typically around 7 Å) a corresponding nonbonded energy term is added to the total energy of the geometry. Experience shows that it is highly advantageous to exclude certain nonbonded interactions. Consider this simple molecular fragment:

```
A-B-C
    |
  E-D
```

The lines indicate bonded interactions. These are often referred to as "1-2" interactions. In our fragment we find 1-2 interactions between A-B, B-C, C-D, and D-E. The nonbonded interactions A-C, B-D, and C-E are commonly referred to as 1-3 interactions, and the nonbonded interaction A-D is called a 1-4 interaction. In general, 1-2 and 1-3 interactions are excluded from the nonbonded energy term, and 1-4 interactions are attenuated.

When setting up the nonbonded energy calculations we have to find the 1-3 and 1-4 interactions based on the pre-defined bonded (1-2) interactions. If space group symmetry is not involved this is very straightforward. However, if symmetry bonds are to be considered the situation becomes much more complex again. The algorithm required is known as "coordination sequence algorithm" and is commonly used in material science, in particular zeolite research (e.g. Brunner & Laves, 1971, Grosse-Kunstleve et al., 1996). See also the Atlas of Zeolite Framework Types available at http://www.iza-structure.org/).

3.5: cctbx.crystal.coordination_sequence

It is surprisingly easy to write a complete coordination sequence algorithm based on the `pair_asu_table` object discussed before. A `simple_and_slow` reference implementation can be found in the `cctbx.crystal.coordination_sequences` module. The complete function comprises just 36 lines of Python code. We can use this short function to easily compute the coordination sequences for the Si and O atoms in our `quartz_structure`:

```
import cctbx.crystal.coordination_sequences
term_table = crystal.coordination_sequences.simple_and_slow(
  pair_asu_table=pair_asu_table,
  max_shell=10)
for terms in term_table:
  print terms
```

Output:

```
[1, 4, 4, 12, 12, 36, 30, 84, 52, 124, 80]
[1, 2, 6, 6, 18, 18, 51, 42, 103, 62, 156]
```

The first list of terms is for the Si atom, the second for the O atom. The first term (in "shell" 0) is always 1. The 1 Si is bonded to 4 O atoms (shell 1), which are bonded to 4 new Si atoms (shell 2). Following all the bonds from these Si atoms to the next O atoms leads to 12 new O atoms (shell 3), from there to 12 new Si atoms (shell 4), etc.

In the mathematics of coordination sequences (e.g. Grosse-Kunstleve et al., 1996) it is most natural to index the coordination shells in the way shown above. Unfortunately this is not directly compatible with the nomenclature of 1-2, 1-3, and 1-4 interactions used in the macro-molecular field. The interactions accounted for in shell 1 are the 1-2 interactions, shell 2 accounts for the 1-3 interactions, and shell 3 for the 1-4 interactions.

To find the nonbonded exclusions we do have to do a little more work than just counting the number of interactions as is done by the `simple_and_slow` function. For each shell we have to keep a table of the interactions found. A much faster, optimized C++ implementation of the coordination sequence algorithm with interfaces for both simple counting and the generation of interaction tables is available in the file `cctbx/include/cctbx/crystal/coordination_sequences.h`. The Python interface to the simple and fast counting algorithm is very similar to that for the `simple_and_slow` interface:

```
term_table = crystal.coordination_sequences.simple(
  pair_asu_table=pair_asu_table,
  max_shell=10)
crystal.coordination_sequences.show_terms(
  structure=quartz_structure,
  term_table=term_table)
```

Output:

```
Si [1, 4, 4, 12, 12, 36, 30, 84, 52, 124, 80]
O [1, 2, 6, 6, 18, 18, 51, 42, 103, 62, 156]
TD10: 456.33
```

Here we make use of the `show_terms` function which shows the scatterer labels along with each list of terms and also the `TD10`, a measure of the "topological density" commonly used in the zeolite field (see http://www.iza-structure.org/).

The tabulation of the 1-3 and 1-4 interactions needed for the nonbonded exclusions is equally easy:

```
shell_asu_tables = crystal.coordination_sequences.shell_asu_tables(
  pair_asu_table=new_pair_asu_table,
  max_shell=3)
print shell_asu_tables
```

Output:

```
(<cctbx_crystal_ext.pair_asu_table object at 0x82a2bec>,
 <cctbx_crystal_ext.pair_asu_table object at 0x82a2c34>,
 <cctbx_crystal_ext.pair_asu_table object at 0x82a2c7c>)
```

The result is a Python tuple with three `pair_asu_table` objects for the 1-2, 1-3, and 1-4 interactions. The first `pair_asu_table` in the tuple is simply a reference to the original `pair_asu_table` defining the bonds. Keeping the original table together with the derived tables simplifies subsequent algorithms.

As an aside, the 36 lines of the `simple_and_slow` Python function have turned into 278 lines of C++ code in `coordination_sequences.h`. The size comparison is not quite fair because the C++ implementation works for both simple counting and tabulation of nonbonded exclusions, but a doubling or tripling of the lines of source code in the conversion from a Python reference implementation to the final C++ implementation is the norm. Unfortunately this is what we have to cope with until higher-level languages with smarter optimizers are a reality.

4: Putting everything together: cctbx.geometry_restraints.manager

The `shell_asu_tables` object of the previous section is the key data structure for the computation of the bond proxies and the nonbonded proxies. However, there is still much more to manage: we need to define the bond parameters (ideal distances and weights), nonbonded "energy types" and VdW (Van der Waals) distances, angle, dihedral, chirality and planarity restraints. Clearly we need a professional manager. It is implemented in the `cctbx.geometry_restraints.manager` module. The `manager` constructor acts as a tool for grouping all information required for the geometry restraints calculations:

```
class manager:

  def __init__(self,
        crystal_symmetry=None,
        site_symmetry_table=None,
        bond_params_table=None,
        shell_sym_tables=None,
        nonbonded_params=None,
        nonbonded_types=None,
        nonbonded_distance_cutoff=5,
        nonbonded_buffer=1,
        angle_proxies=None,
        dihedral_proxies=None,
        chirality_proxies=None,
        planarity_proxies=None):
```

A self-contained, reasonably simple example (266 lines of Python) for setting up all data structures for the bonded and nonbonded calculations can be found in the file `cctbx/cctbx/geometry_restraints/distance_least_squares.py`. This script performs a distance least squares minimization of zeolite geometries. It was developed primarily as a regression test, but covers almost all the functionality of the pioneering DLS-76 program (http://www.crystal.mat.ethz.ch/Software/DLS76). The only major DLS-76 feature missing is the refinement of unit cell parameters. The new `iotbx.distance_least_squares` command provides a simple interface to the script. In our internal test we use this command to minimize the geometries of the complete Atlas of Zeolite Framework Types (152 structures) in less than 40 seconds (Linux, Xeon 2.8GHz). This includes the automatic search for Si-Si bonds, the generation of oxygen atoms at the mid-points of the Si-Si bonds, the generation of angle restraints which are parameterized as pseudo O-O and Si-Si bonds, the generation of nonbonded interactions, and a two-stage minimization, first without a repulsive force field and in the second pass with the repulsions turned on. The successful completion of these minimizations gives us a high confidence that our system for the refinement of bonded and nonbonded pair interactions is complete and free of errors.

4.1: angle, dihedral, chirality, planarity restraints

The angle, dihedral, chirality, and planarity restraints are currently implemented in the "simple" version only, without treatment of symmetry. For our purposes this is fully sufficient and it may even be sufficient in general because angle restraints for small-molecule crystallography are often parameterized as pseudo bonds (e.g. DLS-76, see previous section). The three other restraint types are not very common in small-molecule crystallography. However, our framework is very open and symmetry-aware restraint types could probably be added without disturbing the overall organization of the

`cctbx.geometry_restraints` module. To give an example we show how to work with angle restraints:

```
from cctbx import geometry_restraints
angle = geometry_restraints.angle(
  sites=[(1,2,3),(2,3,4),(5,4,3)],
  angle_ideal=120,
  weight=1)
print "angle_model:", angle.angle_model
print "delta:", angle.delta
print "residual:", angle.residual()
print "gradients:", angle.gradients()
```

Output:

```
angle_model: 121.482154105
delta: -1.48215410529
residual: 2.19678079184
gradients: ((-69.337848889979, 1.2765767806090013e-14, 69.337848889979028),
            (63.034408081799093, -25.213763232719657, -113.4619345472384),
            (6.3034408081799089, 25.213763232719643, 44.124085657259371))
```

Comparison with the first example for defining a bond restraint shows that the interfaces are very similar. Essentially we just need three `sites` instead of two, and we have to write `angle` everywhere instead of `bond` and `distance`. The higher level support for proxies, arrays of proxies and vector operations on these arrays is also very similar. The similarities extend to dihedral and chirality restraints where we need to specify four `sites` instead of two or three. Planarity restraints are slightly different because we have to deal with a variable number of sites and each site is associated with an individual weight:

```
from cctbx import geometry_restraints
from cctbx.array_family import flex
sites_cart = flex.vec3_double([
  (-6.9, 1.3, -1.4),
  (-4.9, -1.0, 0.1),
  (-6.9, -0.6, -1.7),
  (-4.8, 0.9, 0.5)])
weights = flex.double([1, 2, 3, 4])
planarity = geometry_restraints.planarity(
  sites=sites_cart,
  weights=weights)
print "deltas:", list(planarity.deltas())
print "residual:", planarity.residual()
print "gradients:", list(planarity.gradients())
```

We don't show the rather uninteresting output. The difference to the other restraint types is that we get an array of `deltas` instead of just one value. However the important `residual()` and `gradient()` functions fit into the common framework.

5: Setting up restraints using the CCP4 Monomer Library

The CCP4 (http://www.ccp4.ac.uk/) Monomer Library is a comprehensive database of protein, nuclear acid and hetero-compound geometries. We are grateful for CCP4 to give us permission to use this library. The new `mmtbx` top-level module of the `cctbx` project (see Newsletter No. 1 for information on the overall organization of the cctbx project) includes functions for reading the monomer library files as distributed by CCP4, and to generate the geometry proxies introduced above for a given PDB file (http://www.rcsb.org/). The end result is a `cctbx.geometry_restraints.manager.manager` instance that is completely independent of the Monomer Library, the PDB file, or any other file format. The `manager` object is then used in the same minimization procedure employed by the `distance_least_squares.py` script introduced before (the minimizer is implemented in `cctbx.geometry_restraints.lbfgs`). This complete separation of file formats and core computations

makes it possible to support any other library defining geometry restraints. E.g. for the future we are planning to add support for CNS (http://cns.csb.yale.edu/) topology and parameter files.

Currently the code for working with the CCP4 Monomer Library resides in the `mmtbx/mmtbx/stereochemisty` directory. It is still being worked on in order to cleanly support PDB files with alternate conformations and it may be moved to a different place. We will describe the final result in the next newsletter.

6: Bulk solvent correction and scaling

It is well known that macromolecular crystals contain a large amount of disordered solvent reaching sometimes more than 70% of the unit cell volume. The scattering contribution of this solvent level becomes significant at low resolution starting from about 6.0 Å. There are several aspects where the appropriate modeling of low resolution data is of great importance: electron density map analysis (Urzhumtsev, 1991), crystallographic refinement (Kostrewa, 1997), precise calculation of electrostatic properties of molecules (Lecomte, 1999), and the translation search part of structure solution by the Molecular Replacement method (Fokine & Urzhumtsev, 2002a). Basically two bulk solvent models are currently in use by popular crystallographic packages: the exponential scaling model (Moews & Kretsinger, 1975; Tronrud, 1997) and the flat model (Phillips, 1980; Jiang & Brunger, 1994). The exponential scaling model is only justified for the very low-resolution data, lower than 15 Å (Podjarny & Urzhumtsev, 1997), and becomes incorrect at higher resolutions. The flat model is shown as physically more reasonable (Fokine & Urzhumtsev, 2002b) and being compared to all others models is demonstrated as more efficient in sense of both computations and quality of final result obtained (Jiang & Brunger, 1994).

Based on the arguments above, we implemented the flat bulk solvent model in the `mmtbx.bulk_solvent` module. The bulk solvent modeling and scaling procedure contains four main steps: molecule mask calculation, structure factors calculation from the mask, determination of solvent parameters ksol and Bsol, and determination of the overall anisotropic scale coefficient (Sheriff & Hendrickson, 1987).
The algorithm for the mask calculation is realized as described by (Jiang & Brunger, 1994). The corresponding Python code looks like this:

```
from mmtbx.bulk_solvent import bulk_solvent_models
from mmtbx.masks import masks
from iotbx import reflection_file_reader
from iotbx import pdb
pdb_file = "1F8T.pdb"
hkl_file = "1F8T.hkl"
xray_structure = pdb.as_xray_structure(pdb_file)
refl = reflection_file_reader.any_reflection_file(file_name=hkl_file)
refl_arrays = refl.as_miller_arrays(crystal_symmetry=xray_structure)
f_obs = refl_arrays[0].resolution_filter(d_min=2.5)
f_calc = f_obs.structure_factors_from_scatterers(
  xray_structure=xray_structure).f_calc()
mask_manager = masks.mask_utils(
  structure=xray_structure,
  mask_grid_step=f_obs.d_min()/4.,
  shell=5.0,
  shrink=1.0,
  rsolv=1.0)
f_mask = mask_manager.sf_from_mask(f=f_obs)
f_mask.set_info("Mask structure factors")
f_mask.show_summary()
print "Accessible surface fraction:", \
  mask_manager.accessible_surface_fraction()
print "Contact surface fraction:", \
  mask_manager.contact_surface_fraction()
```

Output:

```
Miller array info: Mask structure factors
Observation type: None
Type of data: complex_double, size=15897
Type of sigmas: None
Number of Miller indices: 15897
Anomalous flag: 0
Unit cell: (72.24, 72.01, 86.99, 90, 90, 90)
Space group: P 21 21 21 (No. 19)
Accessible surface fraction: 0.330885416667
Contact surface fraction: 0.45850308642
```

The bulk solvent structure factors and parameters ksol and Bsol can be calculated by adding the following lines to the previous code:

```
bulk_solvent_manager = bulk_solvent_models.bulk_solvent(
  verbose=-1,
  f_obs=f_obs,
  f_calc=f_calc,
  f_mask=f_mask,
  aniso_scale_flag=0001,
  bulk_solvent_correction_flag=0001)
print "Flat model bulk solvent parameters: ", \
  bulk_solvent_manager.ksol_bsol()
f_bulk = bulk_solvent_manager.f_bulk()
f_bulk.set_info("Bulk solvent structure factors")
f_bulk.show_summary()
```

Output:

```
Flat model bulk solvent parameters:  (0.31000000000000011, 38.0)
Miller array info: Bulk solvent structure factors
Observation type: None
Type of data: complex_double, size=15897
Type of sigmas: None
Number of Miller indices: 15897
Anomalous flag: 0
Unit cell: (72.24, 72.01, 86.99, 90, 90, 90)
Space group: P 21 21 21 (No. 19)
```

All major refinement programs use minimizers to determine the bulk solvent parameters and the anisotropic scaling matrix. However there are a number of difficulties to this approach:

1. The low-resolution diffraction data may not be of sufficient quality or completeness.
2. The starting values for ksol and Bsol may be far from the correct values.
3. The parameters ksol and Bsol are highly correlated. Therefore the minimizer may have difficulties finding a path to the global minimum.
4. Optimizing a function of two exponentials is generally a difficult problem.

These considerations have lead us to choose a more robust procedure. As was demonstrated by Fokine & Urzhumtsev (2002b), the values for ksol and Bsol are distributed around 0.35 $e\text{Å}^{-3}$ and 46 Å^2, respectively. Therefore we decided to implement a grid search procedure for the determination of ksol and Bsol. The search is conducted in a physically meaningful range of values, [0,1] for ksol and [0,100] for Bsol. For each trial pair (ksol, Bsol) in the specified ranges we calculate the anisotropic scaling

coefficients using a gradient-driven minimizer. Finally we select the values ksol and Bsol based on the best outcome of the minimization. It should be emphasized that we use the whole resolution range of data. In contrast, Jiang & Brunger (1994) suggest a partitioning into low-resolution and high-resolution pools. This is the approach used by the CNS program (Brunger et al., 1998) to make the minimization procedure more stable. Our grid-search procedure is sufficiently robust to work without partitioning the data.
Another new feature that distinguishes our implementation of the scaling procedure from previous implementations is the use of a maximum-likelihood function as the objective function in the minimization. Even though maximum-likelihood based refinement is now very common, all existing programs use a conventional least-squares target in the determination of the bulk solvent and scaling parameters, while maximum-likelihood functions are used to determine all other parameters. This inconsistency is eliminated in the `mmtbx.bulk_solvent` module.

7: Relative scaling of crystallographic functional and restraints

Crystallographic refinement usually considers the minimization of a sum of two functions. One function is responsible for fitting the model to the experimental data and the second function introduces restraints encoding a priori knowledge, for example the geometry restraints discussed before. The two functions are generally on a different scale and it is necessary to determine an appropriate relative scale factor in order to balance the contributions to the sum. For this purpose we have implemented the procedure proposed by (Adams et al., 1997) in the `mmtbx.refinement.weight_xray_term` module, which makes use of the new `mmtbx.dynamics` module.

8: Crystallographic target functions

The new `mmtbx.refinement` module implements two crystallographic target functions in addition to the conventional least-squares and correlation target functions provided by the `cctbx.xray` module. These are the full maximum-likelihood function of Lunin et al. (2002) and its quadratic approximation (Lunin & Urzhumtsev, 1999). The calculation of the distribution parameters for the target function, "alpha" and "beta", is implemented in two ways:

- estimation by maximization of a likelihood function given a current model and observed intensities (Lunin & Skovoroda, 1995),
- determination via an exact formula (see, for example, Afonine et al., 2003).

In the future we will also implement the sigma-a algorithm and likelihood functions including experimental phase information.

9: Integration of Clipper

Thanks to generous support by Kevin Cowtan, the Clipper library (http://www.ysbl.york.ac.uk/~cowtan/clipper/clipper.html, see also Cowtan (2003) in Newsletter No. 2) is now integrated into the cctbx project and redistributed with the cctbx bundles posted at http://cci.lbl.gov/cctbx_build/ . The bundles with the build tag 2004_07_06_0816 are the first to include Clipper. Currently the Clipper libraries requiring fast Fourier transforms are not compiled in the cctbx build, but this is likely to change in the future. The supporting clipper_adaptbx adaptor toolbox in the cctbx tree provides a fully functional Python interface to the sigma-a calculations in Clipper. We will add other Python interfaces as the need arises.

10: Efficient sampling of search spaces

Indirectly Kevin Cowtan has left his mark in the cctbx project in another way. The newly added `cctbx.crystal.close_packing` module implements a hexagonal close packing sampling generator as suggested by Kevin for some time. Sampling space at the points of a hexagonal close packing instead of the points of a regular grid leads to significant speed increases in search procedures such as the molecular replacement translation search or the placement of molecular fragments in electron density maps. The `cctbx.crystal.close_packing.hexagonal_sampling` generator produces points to efficiently sample search spaces with various symmetries. Space group symmetry and Euclidean normalizer symmetry (also known as Cheshire symmetry) can be arbitrarily combined to define the symmetry of the search space. Depending on the settings, the resulting sampling points may cover three, two, one or zero dimensions. The symmetry is controlled at a high level via flags. The search-space symmetry operations including continuous allowed origin shifts are determined automatically. For example:

```
from cctbx import crystal
import cctbx.crystal.close_packing
from cctbx import sgtbx
crystal_symmetry = crystal.symmetry(
  unit_cell="255.260  265.250  184.400  90.00  90.00",
  space_group_symbol="P 21 21 2")
for use_space_group_symmetry in [True, False]:
  sampling_generator = crystal.close_packing.hexagonal_sampling(
    crystal_symmetry=crystal_symmetry,
    symmetry_flags=sgtbx.search_symmetry_flags(
      use_space_group_symmetry=use_space_group_symmetry,
      use_seminvariants=True,
      use_normalizer_k2l=False,
      use_normalizer_l2n=False),
    point_distance=2)
  print "number of sampling points:", sampling_generator.count_sites()
```

Output:

```
number of sampling points: 46332
number of sampling points: 162240
```

The whole procedure takes 0.14 seconds (Linux, Xeon 2.8GHz). See also the reference documentation for the classes `cctbx::crystal::close_packing::hexagonal_sampling_generator` and `cctbx::sgtbx::search_symmetry_flags` available at http://cctbx.sourceforge.net/ under "C++ interfaces."

11: Acknowledgments

We like thank the CCP4 project for giving us permission to use the Monomer Library, Kevin Cowtan for giving us permission to use Clipper, and the mmLib (http://pymmlib.sourceforge.net/) development team (E. Merritt and J. Painter) for giving us permission to use the `mmCIF.py` file. We gratefully acknowledge the financial support of NIH/NIGMS. Our work was supported in part by the US Department of Energy under Contract No. DE-AC03-76SF00098.

12: References

Adams, P. D., Pannu, N. S., Read, R. J. & Brunger, A. T. (1997). Proc. Natl. Acad. Sci. USA, 94, 5018-5023.
Afonine, P., Lunin, V. & Urzhumtsev, A. (2003). J. Appl.Cryst., 36,158-159.
Brunger, A. T., Adams, P. D., Clore, G. M., DeLano, W. L., Gros, P., Grosse-Kunstleve, R. W., Jiang, J.-S., Kuszewski, J., Nilges, M., Pannu, N. S., Read, R. J., Rice, L. M., Simonson, T. & Warren, G. L. (1998). Acta Cryst. D54, 905-921.
Brunner, G.O. & Laves, F. (1971). Wiss. Z. Techn. Univers. Dresden 20, 387-390.
Cowtwn, K. (2003). Newsletter of the IUCr Commission on Crystallographic Computing, 2, 4-9.
Fokine, A. & Urzhumtsev, A. (2002a). Acta Cryst., A58, 72-74.
Fokine, A. & Urzhumtsev, A. (2002b). Acta Cryst., D58, 1387-1392.
Grosse-Kunstleve, R.W., Brunner G.O., Sloane N.J.A. (1996). Acta Cryst. A52, 879-889.
Grosse-Kunstleve, R.W., Adams, P.D. (2003). Newsletter of the IUCr Commission on Crystallographic Computing, 1, 28-38.
Grosse-Kunstleve, R.W., Wong, B., Adams, P.D. (2003). Newsletter of the IUCr Commission on Crystallographic Computing, 2, 10-16.
Hendrickson, W.A. (1985). Meth. Enzym. 115, 252-270.
Jiang, J.-S. & Brunger, A.T. (1994). J. Mol. Biol. 243, 100-115.
Kostrewa, D. (1997). CCP4 Newsl. 34, 9-22.
Lecomte, C. (1999). Implications of Molecular and Materials Structure for New Technologies, NATO ASI and Euroconference, Serie E: Applied Science, pp. 23-44. Dordrecht: Kluwer.
Lunin, V.Y., Afonine, P.V. & Urzhumtsev, A. (2002). Acta Cryst. A58, 270-282.
Lunin, V.Y. & Skovoroda, T.P. (1995). Acta Cryst. A51, 880-887.
Lunin, V.Y., Urzhumtsev, A.G. (1999). CCP4 Newsletter on Protein Crystallography, 37, 14-28.
Moews, P.C. & Kretsinger, R.H. (1975). J. Mol. Biol. 91, 201-228.
Phillips, S.E.V. (1980). J. Mol. Biol. 142, 531-554.
Podjarny, A.D., Urzhumtsev, A.G. (1997). In Methods in Enzymology, Academic Press, San Diego., C.W.Carter, Jr., R.M.Sweet, eds. 276A, 641-658.
Sheriff, S. & Hendrickson, W.A. (1987). Acta Cryst. A43, 118-121.
Tronrud, D.E. (1997). Methods Enzymol. 277, 306-319.
Urzhumtsev, A. (1991). Acta Cryst. A47, 794-801.