cctbx news

Luc J. Bourhis[a], Ralf W. Grosse-Kunstleve[b], Paul D. Adams[b]
a) University of Durham, Durham, DH1 3HP,UK and b) Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA. E-mail: rwgk@cci.lbl.gov and luc_j_bourhis@mac.com ; WWW: http://cctbx.sourceforge.net/, http://www.phenix-online.org/, http://cci.lbl.gov/

Abstract

We describe recent developments of the Computational Crystallography Toolbox.

Preamble

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). All example scripts shown below were tested with cctbx build 2007_10_29_2045.

In the following we refer to our articles in the previous editions of this newsletter as "Newsletter No. 1", "Newsletter No. 2", etc. to improve readability. The full citations are included in the reference section.

Introduction

The *Computational Crystallography Toolbox* (cctbx, http://cctbx.sourceforge.net/) is the open-source component of a structure determination suite for macro-molecular crystallography (Phenix, http://www.phenix-online.org/). However, the cctbx was started with a code base developed in the context of small-molecule crystallography, most notably SgInfo (http://cci.lbl.gov/sginfo/). Although in recent years most new developments were targeted towards macro-molecular work, the small-molecule heritage has been carefully maintained in the core modules of the cctbx project. For example, all algorithms in the `cctbx` module (for the distinction between the *cctbx module* and the *cctbx project* see Newsletter No. 1) work for all 230 crystallographic space groups and are routinely tested with symmetries not found in macro-molecular crystals.

Recently, we have started a new `smtbx` module with algorithms specifically for small-molecule work. This is very much work in progress, but below we present some related developments. We give an example of least-squares minimization with a target function and a weighting scheme commonly used in the refinement of small molecules. Another development that grew out of small-molecule context is the handling of special position constraints. We also highlight important new developments of the *Phil* system introduced in Newsletters No. 5 and No. 7. Finally, we give a brief summary of the transition of the cctbx source code repository from CVS to Subversion (SVN).

Refinement tools for small-molecule crystallographers

Refinement against F^2

Small-molecule crystallographers almost exclusively rely on the minimization of a least-square target to refine a structure. The cctbx has provided one for a long time but only for F. Since F^2 refinement is very popular among small-molecule crystallographers, a new target has recently been added to the cctbx. The class `cctbx.xray.unified_least_squares_residual` provides a single entry point to both F and F^2 refinement.

To keep the example self-contained, and also to demonstrate some tools useful to develop and debug crystallographic algorithms, instead of starting from real data, we will use randomly generated crystal structures. Atoms are randomly spread in the unit cell and the structure factors are then computed:

```
from cctbx.array_family import flex
from cctbx import xray
from cctbx import crystal
from cctbx import miller
from cctbx.development import random_structure
import random

indices = miller.build_set(
        crystal_symmetry=crystal.symmetry(unit_cell=(10,11,12, 90,105,90),
                                          space_group_symbol="P21/c"),
        anomalous_flag=False,
        d_min=0.8)
structure = random_structure.xray_structure(
  indices.space_group_info(),
  elements=['C']*6 + ['O']*2 + ['N'],
  volume_per_atom=18.6,
  random_u_iso=True)
f_ideal = structure.structure_factors(d_min=indices.d_min()).f_calc()
```

Then we extract the amplitudes or intensities and we put them on a different scale to make the example a bit more realistic:

```
f_obs = f_ideal.amplitudes()
f_obs.set_observation_type_xray_amplitude()
f_obs *= 2
f_obs_square = f_ideal.norm()
f_obs_square.set_observation_type_xray_intensity()
f_obs_square *= 3
```

In practice we would get f_obs or f_obs_square from a data file using the `iotbx` and the observation type would have been automatically set up for us.

Then we can construct the least-square targets and examine them:

```
ls_against_f = xray.unified_least_squares_residual(f_obs)
ls_against_f_square = xray.unified_least_squares_residual(f_obs_square)

residuals = ls_against_f(f_ideal, compute_derivatives=True)
print "against F: value=%.3f, scale=%.3f" % (residuals.target(),
                                             residuals.scale_factor())
residuals = ls_against_f_square(f_ideal, compute_derivatives=True)
print "against F^2: value=%.3f, scale=%.3f" % (residuals.target(),
                                               residuals.scale_factor())
```

The constructor of the class automatically recognizes whether the data passed to it are amplitudes or intensities and it adapts the computations accordingly, correctly getting a target value of 0 and a scale factor of 2 and 3, respectively, as expected in this trivial example.

By passing `compute_derivatives=True`, we require the computation of the derivatives of the L.S. target function with respect to F_{calc} (h) for each Miller index h. They are available as `residuals.derivatives()`. Derivatives now brings us to discuss refinement.

To demonstrate refinement, we will first add a perturbation to the crystal structure as well as allowing the refinement of atom sites:

```
perturbed_structure = structure.random_shift_sites(max_shift_cart=0.2)
for s in perturbed_structure.scatterers():
        s.flags.set_grad_site(True)
```

The simplest refinement engine in the cctbx is `cctbx.xray.minimization.lbfgs`. For a refinement against F^2:

```
refining_structure = perturbed_structure.deep_copy_scatterers()
optimiser = xray.lbfgs(
                target_functor=ls_against_f_square,
                xray_structure=refining_structure,
                structure_factor_algorithm="direct")
print "Initial L.S. residual:%.3f" % optimiser.first_target_value
structure.show_scatterers()
print "Final L.S. residual:%.3f" % optimiser.final_target_value
refining_structure.show_scatterers()
```

The LBFGS algorithm (Liu & Nocedal, 1989) used to minimize the target function needs to compute its partial derivatives with respects to the refined parameters (atomic position here). This is where the derivatives which we showed to be obtainable by `residuals.derivatives()` above come into play: they are combined, by using the chain rule, with the derivatives of F_{calc} (h) with respect to the refined parameters.

Least-squares weights

By default, `xray.unified_least_squares_residual` tries to make sensible choices for the least-squares weights, i.e. unit weights for refinement against F and the so-called quasi-unit weights for refinement against F^2, which are $1/(4\ F^2_{obs})$. The latter choice is known to result in a more stable refinement. However, almost universally, experimental error estimates are available, as `f_obs_square.sigmas()`, and the least-square weights should take advantage of them. The pure statistical weights $1/\sigma^2$ are rarely used in small molecule crystallography. The most popular choice is that of the ShelXL program (http://shelx.uni-ac.gwdg.de/), $1/(\sigma(F^2_{obs})^2 + (a\ P)^2 + b\ P)$ where $P = 1/3\ \max(0, F^2_{obs}) + 2/3\ F^2_{calc}$, which down-weights the stronger reflection while reducing statistical bias by the use of P (Wilson 1976).

This weighting scheme is available in the module `cctbx.xray.weighting_schemes`:

```
weighting = xray.weighting_schemes.shelx_weighting()
shelx_weighted_ls_against_f_square = xray.unified_least_squares_residual(
        f_obs_square, weighting=weighting)
```

This least-square target can then be used with the `lbfgs` minimizer. The default is that of ShelXL, i.e. a=0.1 and b=0, which is best suited for early refinements when the structure is still incomplete. All the other weighting schemes we mentioned are also provided by that module (the reader is invited to read the comments for each Python class).

Special position constraints

The traditional way to deal with the refinement of crystal structures with atoms on special position is to constrain the latter never to move away from the special position, in effect minimizing the target function under a set of constraints. However, the minimizer introduced above does not work in such a manner and neither does the more sophisticated `mmtbx.refinement.minimization.lbfgs`, mainly because this is less important for macro-molecular models, which usually have very few atoms on special positions. One can safely, at each refinement cycle, let those atoms move away from the special positions and move them back before the next cycle. In contrast, in the small molecule world it is not uncommon that all atoms are on special positions and the ad-hoc treatment is more problematic.

The cctbx does actually provide all the tools to find, store and apply special position constraints. We will therefore start by introducing those fundamental tools:

```
crystal_symmetry = crystal.symmetry(
    unit_cell=(10,10,10,90,90,90),
    space_group_symbol="Pm3m")
crystal_symmetry.show_summary()

special_position_settings = crystal_symmetry.special_position_settings(
  min_distance_sym_equiv=0.5)
```

The site symmetry of a position given in fractional coordinates is then obtained by:

```
site_symmetry = special_position_settings.site_symmetry(
   site=(0.3, 0.31, 0.111))
```

The given position is at a Cartesian distance smaller than 0.5 A from a special position with x=y. The `site_symmetry` algorithm computes the exact location of the nearest special position (Grosse-Kunstleve and Adams, 2002). The resulting `site_symmetry.exact_site()` is (0.305, 0.305, 0.111).

Then:

```
site_constraints = site_symmetry.site_constraints()
```

gives access to the constraints on the exact site:

```
>>> site_constraints.n_independent_params()
2
>>>
site_constraints.independent_params(all_params=site_symmetry.exact_site())
(0.305, 0.111)
```

The other way around, one can determine all the site coordinates from the independent ones:

```
>>> site_constraints.all_params(independent_params=(0.2, 0.1))
(0.2, 0.2, 0.1)
```

The most interesting facility for refinement is the handling of gradients. To illustrate it, let us introduce a simple function f of the site coordinates x,y,z (during a refinement, f would instead be the L.S. target for example, considering its dependence on the coordinates of one atom only):

```
def f(x,y,z): return -x + 2*y + 3*z
```

which we then restrict onto the special position locus:

```
def g(u,v):
  x, y, z = site_constraints.all_params((u,v))
  return f(x,y,z)
```

We wish to compute the derivatives of g with respect to u and v knowing the derivatives of f with respect to x,y,z which are easily read directly from its definition: (df/dx, df/dy, df/dz) = (-1,2,3). Those derivatives of g are then easily obtained with:

```
>>> independent_gradients = site_constraints.independent_gradients(
      all_gradients=flex.double((-1,2,3)))
(1.0, 3.0)
```

We have so far demonstrated only site constraints and only in fractional coordinates, but the cctbx provides more facilities:

```
frac_adp_constraints = site_symmetry.adp_constraints()
```

gives access to the constraints on ADPs in fractional coordinates whereas:

```
cart_adp_constraints = site_symmetry.cartesian_adp_constraints(
  crystal_symmetry.unit_cell())
```

deals with the same constraints on ADPs in Cartesian coordinates.

It should be noted that the `site_symmetry` object caches all the constraints it gives access to: the necessary symmetry computations are performed only once when `site_constraints`, `adp_constraints` or `cartesian_adp_constraints` are called for the first time and the results are reused later on.

The cctbx special position constraints we have just expounded are the basis of the refinement engine `lbfgs` in the module `smtbx.refinement.minimization`. It is to be used exactly as `cctbx.xray.minimization.lbfgs`, the only difference being that atoms on special positions have their site coordinates and ADPs constrained properly. Our plan is to develop `smtbx.refinement.minimization.lbfgs` over time into the equivalent of the more sophisticated and versatile `mmtbx.refinement.minimization.lbfgs`.

Phil developments

Phil (Python-based hierarchical interchange language) is a module for the management of application parameters and, to some degree, inputs. Phil was first introduced in Newsletter No. 5. In Newsletter No. 7 we presented a complete example application which uses Phil for the handling of program parameters.

Overall, Phil has been very stable after an intense development push almost three years ago. Some features were added in response to needs that arose as part of the development of other applications, and a few bugs were fixed. These changes are very conservative. However, recently there has also been an important semantic change affecting "multiple definitions" and "multiple scopes". The new features and changes are reflected in the updated documentation (URL at end of this section) which is based on parts of our original article in Newsletter No. 5. Important new features and changes include:

- The handling of multiple scopes went through a few iterations eliminating bugs, inconsistencies and inefficiencies. The latest implementation (as of October 2007) is considered mature, stable and intuitive.
- The new `.fetch_diff()` method returns only objects with non-default values. This is most useful for applications with a large number of parameters. Usually most parameters are not changed by the user. `.fetch_diff()` pin-points the (few) changes.
- The previous plain `include` syntax was changed to `include file`; the new `include scope` syntax was added. The `include scope` feature builds on Python's standard import mechanism. In practice it is found to be much more useful than the `include file` feature.
- The syntax-aware comment feature now uses `!` as the comment character. For example:

  ```
  !crystal_symmetry {
    unit_cell = None
    space_group = None
  }
  ```

  The exclamation mark comments out the entire `crystal_symmetry` scope including all embedded definitions. `#crystal_symmetry` is now a normal one-line comment as expected by most users. Based on feedback, the distinction between

`#crystal_symmetry` (syntax-aware comment) and `# crystal_symmetry` (one-line comment) was found to be too subtle.

- Again based on feedback, the `{` and `}` scope-delineation characters (compare with the `crystal_symmetry` example above) are now interpreted as one-character keywords. In contrast to the initial implementation, any `{` and `}` in strings have to be quoted. This leads to a more obvious syntax. It also enables a scope name and embedded definitions to appear on the same line. This can be convenient in certain situations, in particular for definitions specified on the command-line.
- Similarly, `;` characters are now interpreted as one-character keywords (they had no syntactical meaning in the original implementation). This enables `scope { a=1; b=2 }` all on one line.
- Related to the `{`, `}` scope-delineation change, the `${varname}` variable substitution syntax was changed to `$(varname)`.

For more details see the updated documentation: http://cctbx.sourceforge.net/libtbx_phil.html

Subversion

For six years, from April 2001, when the SourceForge cctbx project was started, to March 2007, cctbx development was based on the Concurrent Versions System (CVS, http://www.cvshome.org/). This era came to an end with the transition to the more modern Subversion system (SVN, http://subversion.tigris.org/). In the last two years, the vast majority of the open source community has gone through this transition. SVN introduces many new features, most importantly "atomic commits" which handle a set of changes as one entity, no matter how many files and directories are involved. This makes it much easier to keep track of large-scale changes, and to backtrack if necessary. Almost as important is the option to rename files and directories without introducing breaks in the development history. The need to reorganize arises frequently in the early stages of a (sub-)project. Often it is initially not very clear how all the pieces fit together. The best ideas for organizing the whole tend to crystallize only after a critical subset of the project is already implemented. SVN provides a clear path for reorganizing the sources.

The cctbx sources were converted using the `cvs2svn` script (http://cvs2svn.tigris.org/, it is a Python script!) which preserves the complete CVS development history, including all original time stamps, user names, and log messages. The last state of the cctbx CVS repository before the conversion is still available for viewing at SourceForge, but for all practical purposes it is completely obsolete.

The main cctbx page at Sourceforge (http://cctbx.sourceforge.net/) includes a link to the web view of the cctbx SVN repository. Instructions for checking out the sources using the svn command are posted under the cctbx installation instructions (http://cctbx.sourceforge.net/current/installation.html). These instructions show how to get a selection of individual cctbx modules. Note that it is also possible to get the entire cctbx project with a single command (this wasn't possible with CVS), for example:

```
svn co https://cctbx.svn.sourceforge.net/svnroot/cctbx/trunk svnroot_cctbx
```

The name of the target directory (`svnroot_cctbx`) is arbitrary. The advantage of checking out everything under one tree is that a single svn update or svn commit command works on the entire repository. The disadvantage compared to the more granular approach is that the working copy includes modules that may not be needed and therefore consumes more disk space (currently about 62 MB).

With the move to SVN, the old distinction between *anonymous* and *developer* checkouts disappeared. This is a great advantage. Everybody can checkout working copies using the exact same commands. Credentials, i.e. a user name and password, are required only for write access (e.g. svn commit). It is possible to checkout a working copy before having a SourceForge user name, make local modifications, get a user name and request cctbx svn write access, and then to commit the changes from the initial working copy.

Acknowledgments

L.J. Bourhis gratefully acknowledge the financial support of EPSRC under grant number EP/C536274/1. R.W. Grosse-Kunstleve and P.D. Adams gratefully acknowledge the financial support of NIH/NIGMS under grant number P01GM063210. Their work was supported in part by the US Department of Energy under Contract No. DE-AC02-05CH11231.

References

Grosse-Kunstleve, R.W., Adams, P.D. (2002). Acta Cryst. A58, 60-65.

Grosse-Kunstleve, R.W., Adams, P.D. (2003). Newsletter of the IUCr Commission on Crystallographic Computing, 1, 28-38.

Grosse-Kunstleve, R.W., Afonine, P.V., Sauter, N.K., Adams, P.D. (2005). Newsletter of the IUCr Commission on Crystallographic Computing, 5, 69-91.

Grosse-Kunstleve, R.W., Zwart P.H., Afonine, P.V., Ioerger, T.R., Adams, P.D. (2006). Newsletter of the IUCr Commission on Crystallographic Computing, 7, 92-105.

Liu C., Nocedal J. (1989). Mathematical Programming 45, 503-528.

Wilson, A.J.C. (1976). Acta Cryst. A32, 994-996.