cctbx news: Phil and friends

Ralf W. Grosse-Kunstleve, Pavel V. Afonine, Nicholas K. Sauter and Paul D. Adams,
Computational Crystallography Initiative, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, BLDG 64R0121, Berkeley, California 94720-8118., USA - Email : RWGrosse-Kunstleve@lbl.gov ; WWW: 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 2005_01_22_0855.

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

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 much energy is devoted to implementing a streamlined command-line interface to the Phenix refinement algorithms. In this article we describe the new *Python-based hierarchical interchange language* (Phil) that was developed for this purpose. Other important developmemts highlighted below are our implementation of cartesian dynamics simulated annealing for macromolecular structure refinement, the significant enhancements of the `iotbx.reflections_statistics` command, the new C++ and Python interfaces to the CCP4 MTZ library, and the inclusion of PyCifRW in the *cctbx* bundles available for download.

The command-line interface to the Phenix refinement algorithms is called `phenix.refine`. The refinement algorithms require a structural model, xray data and optionally experimental phase information, typically in the form of Hendrickson-Lattman coefficients. For macromolecular refinement the ratio of experimental observations to refinable parameters is typically quite low. Geometry restraints have to be included in order to make the refinement stable. Finally, the refinement algorithms introduce a large number of parameters, such as the number of refinement cycles to run, parameters for bulk-solvent correction, simulated annealing, etc. In our current development version the number of parameters including file names and data labels is already greater than 100. This number is likely to increase significantly as we add more features in the future.

In previous issues of this newsletter we have described comprehensive utilities for reading reflection files (Newsletter No. 3), processing of structural data formatted as PDB files integrated with the handling of geometry restraints based on the CCP4 Monomer Library (Newsletter No. 4). However, until recently we had only ad-hoc solutions for the handling of the large number of algorithmic parameters. `phenix.refine` is written in Python (with C++ extensions for numerically intensive algorithms, see Newsletter No. 1). Therefore it was quite natural for us to also use Python to define parameters. For example, Python classes are quite convenient for organizing parameters:

```
from libtbx import introspection

class cartesian_dynamics:
  def __init__(self, temperature     = 300,
                     number_of_steps = 200,
                     time_step       = 0.0005):
    introspection.adopt_init_args()

class simulated_annealing:
  def __init__(self, do_simulated_annealing = False,
                     start_temperature      = 2500,
                     final_temperature      = 300,
                     cool_rate              = 25,
                     number_of_steps        = 25,
                     time_step              = 0.0005,
                     update_grads_shift     = 0.3):
    introspection.adopt_init_args()
```

A group of parameters can then be used like this:

```
my_cartesian_dynamics_params = cartesian_dynamics(number_of_steps=300)
my_simulated_annealing_params = simulated_annealing(final_temperature=200)
some_algorithm(
  cartesian_dynamics_params=my_cartesian_dynamics_params,
  simulated_annealing_params=my_simulated_annealing_params)
```

With:

```
def some_algorithm(
      cartesian_dynamics_params,
      simulated_annealing_params):
  print cartesian_dynamics_params.temperature
  print cartesian_dynamics_params.number_of_steps
  print simulated_annealing_params.start_temperature
  print simulated_annealing_params.final_temperature
```

the output is:

```
300
300
2500
200
```

This shows that we retain the default values for `temperature` and `start_temperature`, but override the values for `number_of_steps` and `final_temperature`.

2 Management of parameters: Phil is your friend

One obvious problem of the approach to parameter management outlined above is that it requires familiarity with the Python syntax. While Python is arguably one of the most elegant programming languages, it still has too much syntax for non-programmers. E.g. all Python string literals have to be in quotes and indentation is syntactically significant. It also appeared difficult to implement the advanced parameter management features introduced below working exclusively with Python syntax. Therefore we

have replaced the Python syntax with the new Phil syntax to make parameter management as simple as possible. The Phil equivalent of the examples above is:

```
refinement.cartesian_dynamics {
  temperature = 300
  number_of_steps = 200
  time_step = 0.0005
}

refinement.simulated_annealing {
  do_simulated_annealing = False
  start_temperature = 2500
  final_temperature = 300
  cool_rate = 25
  number_of_steps = 25
  time_step = 0.0005
  update_grads_shift = 0.3
}
```

The Phil syntax has only two main elements, `phil.definition` (e.g. `cool_rate=25` and `phil.scope` (e.g. `simulated_annealing { }`). To make this syntax as user-friendly as possible, strings do not have to be quoted and, unlike Python, indentation is not syntactically significant. E.g. this:

```
refinement.xray_data {
  file_name = "peak.mtz"
  labels = "Fobs" "SigFobs"
}
```

is equivalent to:

```
refinement.xray_data {
file_name=peak.mtz
labels=Fobs SigFobs
}
```

Scopes can be nested recursively. The number of nesting levels is limited only by Python's recursion limit (default 1000). To maximize convenience, nested scopes can be defined in two equivalent ways. For example:

```
refinement {
  xray_data {
  }
}
```

is equivalent to:

```
refinement.xray_data {
}
```

2.1 Beyond syntax

Phil is more than just a parser for a very simple, user-friendly syntax. Major Phil features are:

- The concepts of *master files* and *user files*. The syntax for the two types of Phil files is identical, but the processed Phil files are used in different ways. I.e. the concepts exist only at the semantical level. The "look and feel" of the files is uniform.
- Interpretation of command-line arguments as Phil definitions.
- Merging of (multiple) Phil files and (multiple) Phil definitions derived from command-line arguments.
- Automatic conversion of Phil files to pure Python objects equivalent to instances of ad-hoc Python parameter classes like the examples shown in the introduction. These pure Python objects are completely independent from the Phil system.
- The reverse conversion of (potentially modified) pure Python objects back to Phil files. This could also be viewed as a Phil pretty printer.
- Shell-like variable substitution using $var and ${var} syntax.
- `include` syntax to merge Phil files at the parser level.

2.2 Master files

The master files are written by the software developer and include "attributes" for each parameter, such as the type (integer, floating-point, string, unit cell, etc.) and support information for graphical interfaces. For example:

```
refinement.crystal_symmetry {
  unit_cell=None
    .type=unit_cell
    .help="Unit cell parameters."
    .input_size = 40
    .expert_level = 0
  space_group=None
    .type=space_group
    .help="Space group symbol or number."
    .input_size = 20
    .expert_level = 0
}
```

To see the full set of "attributes" for all `phenix.refine` parameters run this command:

```
iotbx.phil --show_all_attributes $MMTBX_DIST/mmtbx/refinement/__init__.params
```

The output is not shown because it is more than 1000 lines long (and still growing). Fortunately, the end-user does not have to be aware of these long master files.

2.3 User files

User files are typically generated by the application, e.g.

```
phenix.refine --show_defaults
```

will process the master file. (Since phenix is not open source this command is not available in a plain *cctbx* installation.) This command will list only the most relevant parameters, classified by the software developer as `.expert_level = 0`. For example:

```
refinement.crystal_symmetry {
  unit_cell = None
  space_group = None
}
```

The attributes are not shown. Therefore the output is much shorter compared to the `iotbx.phil` output above. Currently the output contains only 53 lines with 35 definitions.

2.4 Command-line arguments + Phil

In theory the user could save and edit the generated parameter files. However, in most practical situations this is not necessary for two reasons.

Firstly, `phenix.refine` (and in the future other *cctbx* and Phenix applications) inspects all input files and uses the information found to fill in the blanks automatically. For example the unit cell is copied from the input PDB file or, if this information is missing in the PDB file, from a reflection file. This is not only convenient, but also eliminates the possibility of typing errors.

Secondly, command-line arguments that are not file names or options prefixed with `--` (like `--show_defaults` above) are given to Phil for examination. E.g., this is a possible command:

```
phenix.refine peak.mtz model.pdb number_of_macro_cycles=10
```

Assume the first two arguments can be opened as files (the file names may be specified in any order; `phenix.refine` detects the file types automatically). Also assume that a file with the name `number_of_macro_cycles=10` does not exist. This argument will therefore be interpreted with Phil.

2.5 Merging of Phil objects

The Phil parser converts master files, user files and command line arguments to uniform Phil objects which can be merged to generate a combined set of "effective" parameters used in running the application. We demonstrate this by way of a simple, self-contained Python script with embedded Phil syntax:

```
import iotbx.phil

master_params = iotbx.phil.parse("""
  refinement.crystal_symmetry {
    unit_cell = None
      .type=unit_cell
```

```
    space_group = None
      .type=space_group
  }
  """)

user_params = iotbx.phil.parse("""
  refinement.crystal_symmetry {
    unit_cell = 10 12 12 90 90 120
    space_group = None
  }
  """)

command_line_params = iotbx.phil.parse(
  "refinement.crystal_symmetry.space_group=19")

effective_params = master_params.fetch(
  sources=[user_params, command_line_params])
effective_params.show()
```

The `master_params` define all available parameters including the type information. The `user_params` override the default `unit_cell` assignment but leave the space group undefined. The space group symbol is defined by the command line argument. `effective_params.show()` produces:

```
refinement.crystal_symmetry {
  unit_cell = 10 12 12 90 90 120
  space_group = 19
}
```

Having to type in fully qualified parameter names (e.g. `refinement.crystal_symmetry.space_group`) can be very inconvenient. Therefore Phil includes support for matching parameter names of command-line arguments as substrings to the parameter names in the master files:

```
import libtbx.phil.command_line

argument_interpreter = libtbx.phil.command_line.argument_interpreter(
  master_params=master_params,
  home_scope="refinement")

command_line_params = argument_interpreter.process(
  arg="space_group=19")
```

This works even if the user writes just `group=19` or even `e_gr=19`. The only requirement is that the substring leads to a unique match in the master file. Otherwise Phil produces a helpful error message. For example:

```
argument_interpreter.process("u=19")
```

leads to:

```
UserError: Ambiguous parameter definition: u = 19
Best matches:
  refinement.crystal_symmetry.unit_cell
  refinement.crystal_symmetry.space_group
```

The user can cut-and-paste the desired parameter to edit the command line for another trial to run the application.

2.6 Conversion of Phil objects to pure Python objects

The Phil parser produces objects that preserve most information generated in the parsing process, such as line numbers and parameter attributes. While this information is very useful for pretty printing (e.g. to archive effective parameters) and the automatic generation of graphical user interfaces, it is only a burden in the context of core numerical algorithms. Therefore Phil supports "extraction" of light-weight pure Python objects from the Phil objects. Based on the example above, this can be achieved with just one line:

```
params = effective_params.extract()
```

We can now use the extracted objects in the context of Python:

```
print params.refinement.crystal_symmetry.unit_cell
print params.refinement.crystal_symmetry.space_group
```

Output:

```
(10, 12, 12, 90, 90, 120)
P 21 21 21
```

At first glance one may almost miss that something significant has happened. However, we started out with "`space_group=19`" and now we see `P 21 21 21` in the output. This is because the `space_group` parameter was defined to be of `.type=space_group` in the master file. Associated with each type are converters to and from corresponding Python objects. In this case, the `space_group` converter produces a Python object of type:

```
print repr(params.refinement.crystal_symmetry.space_group)
```

Output:

```
<cctbx.sgtbx.space_group_info instance at 0xb64edf6c>
```

This object cannot only show the space group symbol, but has many other "methods". E.g. to print the list of symmetry operations in "xyz" notation:

```
for s in params.refinement.crystal_symmetry.space_group.group():
  print s
```

Output:

```
x,y,z
x+1/2,-y+1/2,-z
-x,y+1/2,-z+1/2
-x+1/2,-y,z+1/2
```

2.7 Conversion of Python objects to Phil objects

Phil also supports the reverse conversion compared to the previous section, from Python objects to Phil objects. For example, to change the unit cell parameters:

```
from cctbx import uctbx

params.refinement.crystal_symmetry.unit_cell = uctbx.unit_cell(
  (10,12,15,90,90,90))
modified_params = master_params.format(python_object=params)
modified_params.show()
```

Output:

```
refinement.crystal_symmetry {
  unit_cell = 10 12 15 90 90 90
  space_group = "P 21 21 21"
}
```

We need to bring in the `master_params` again because all the meta-information was lost in the `extract()` step that produced `params`. Again, a type-specific converter is used to produce a string for each Python object. We started out with `space_group=19` but get back `space_group = "P 21 21 21"` because we chose to make the converter work that way.

2.8 Extending Phil

The astute reader may have noticed that we used both `libtbx.phil` and `iotbx.phil`. Why does Phil appear to have two homes?

The best way to think about Phil is to say "Phil is libtbx.phil." The basic Phil objects storing the parsing results (`phil.definition` and `phil.scope`), the tokenizer, parser and the command line support are implemented in the `libtbx.phil` module. `iotbx.phil` *extends* Phil by adding two new types, `unit_cell` and `space_group`. The converters for these types can be found in `$IOTBX_DIST/iotbx/phil.py`. For example, this is the code for the unit cell converters:

```
class unit_cell_converters:

  def __str__(self): return "unit_cell"

  def from_words(self, words, master):
    s = libtbx.phil.str_from_words(words=words)
    if (s is None): return None
    return uctbx.unit_cell(s)
```

```
  def as_words(self, python_object, master):
    if (python_object is None):
      return [tokenizer.word(value="None")]
    return [tokenizer.word(value="%.10g" % v)
      for v in python_object.parameters()]
```

Arbitrary new types can be added to Phil by defining similar converters. If desired, the built-in converters for the basic types (`int`, `float`, `str`, etc.) defined in `libtbx.phil` can even be replaced. All converters have to have `__str__()`, `from_words()` and `as_words()` methods. More complex converters may optionally have a non-trivial `__init__()` method (an example is the `choice_converters` class in `$LIBTBX_DIST/libtbx/phil/__init__.py`).

The `iotbx.phil.parse()` function used in the examples above is a very small function which adds the `unit_cell` and `space_group` converters to Phil's default converter registry and then calls the main `libtbx.phil.parse()` function to do the actual work. Following the example of `iotbx.phil` it should be straightforward to add other domain-specific types to the Phil system.

2.9 Variable substitution

Phil supports shell-like variable substitution using $var and ${var} syntax. A few examples say more than many words:

```
import libtbx.phil

params = libtbx.phil.parse("""
  root_name = peak
  file_name = $root_name.mtz
  full_path = $HOME/$file_name
  related_file_name = ${root_name}_data.mtz
  message = "Reading $file_name"
  as_is = ' $file_name '
  """)
params.fetch(source=params).show()
```

Output:

```
root_name = peak
file_name = "peak.mtz"
full_path = "/net/cci/rwgk/peak.mtz"
related_file_name = "peak_data.mtz"
message = "Reading peak.mtz"
as_is = ' $file_name '
```

Note that the variable substitution does not happen during parsing. The output of `params.show()` is identical to the input. In the example above, variables are substituted by the `fetch()` method that we introduced earlier to merge user files given a master file.

2.10 Phil odds and ends

Phil also supports merging of files at the parsing level. The syntax is simply `include file_name`. `include` directives may appear inside scopes to enable hierarchical building of master files without the need to copy-and-paste large fragments explicitly. Duplication appears only in automatically generated user files. I.e. the programmer is well served because a system of master files can be kept free of large-scale redundancies that are difficult to maintain. At the same time the end user is well served because the indirections are resolved automatically and all parameters are presented in one uniform view.

Variable substitution and include directives smell almost like programming. However, there is a line that Phil is never meant to cross: flow control is not a part of the syntax. It is hard to imagine that a fully featured programming language could be syntactically simpler than Python. For example, there are good reasons why Python string literals have to be quoted. Otherwise Python scripts would be full of $ signs because some method is needed to distinguish strings from variable names. On the other hand, having to quote space group symbols in parameter files is a nuisance. In the future we may extend Phil as an interchange format for data other than parameters but for our programming needs we feel extremely well served by Python.

3 Refinement tools

3.1 mmtbx.refinement.f_model.manager

The goal of crystallographic structure refinement is to optimize a set of model parameters such that the model predictions best fit the experimental observations. In our terminology, *model* goes beyond atomic coordinates, displacement parameters and occupancies. A complete macromolecular model generally also includes other contributions such as scale factors, bulk-solvent correction and anisotropy correction. Furthermore, all modern refinement programs include facilities for cross-validation (e.g. for the calculation of the *R-free*).

The `phenix.refine` command mentioned earlier is based on the *mmtbx* (*Macromolecular toolbox*) module of the *cctbx*. The mathematical foundation of the *mmtbx* model parameterization is described in Afonine *et al.* (2005). It is summarized in this formula:

$$\mathbf{F}^{\text{model}} = k \exp\left(-\mathbf{h}^t \, \mathbf{U}_{\text{aniso}} \, \mathbf{h}\right)\left(\mathbf{F}^{\text{calc}} + k_{\text{sol}} \exp\left(-\frac{B_{\text{sol}} s^2}{4}\right)\mathbf{F}^{\text{mask}}\right)$$

where k is the overall scale factor (Sheriff & Hendrickson, 1987), $\mathbf{F}^{\text{calc}}$ are structure factors calculated from the atomic model, k_{sol} and B_{sol} are bulk-solvent parameters (Jiang & Brünger, 1994), $\mathbf{F}^{\text{mask}}$ are structure factors calculated from a molecular mask, $\mathbf{h}$ is a column vector with the Miller indices of a reflection, $\mathbf{h}^t$ is its transposed vector, and $\mathbf{U}_{\text{aniso}}$ is the overall anisotropic scale matrix (6 components).

During refinement, $\mathbf{F}^{\text{model}}$ is usually calculated many times in different contexts, many parameters are updated at different schedules, and various statistics are printed repeatedly to report the refinement progress. Moreover, some refinement strategies require complete sets of intermediate parameters to be stored for later reference. To meet these needs in a general and reusable way, all model parameters for the crystallographic contribution to the refinement target are grouped by the `mmtbx.refinement.f_model.manager` class. In the following we develop a self-contained Python script to highlight major features of this class. Since we need data to work with, but also want the example to be self-contained, we start by generating a random structure:

```
from cctbx.development import random_structure
from cctbx import sgtbx

space_group_info = sgtbx.space_group_info(
  symbol="P212121")
n_sites = 500
structure = random_structure.xray_structure(
  space_group_info = space_group_info,
  elements         = ["N"]*(n_sites),
  volume_per_atom  = 50,
  anisotropic_flag = False,
  random_u_iso     = True)
```

We use this `structure` to compute ideal observations `f_obs`:

```
d_min = 2.0
f_obs = abs(structure.structure_factors(
  d_min          = d_min,
  anomalous_flag = False).f_calc())
```

Next we introduce two types of errors: missing atoms and coordinate errors with a certain `max_shift`:

```
from cctbx import xray

fraction_missing = 0.1
max_shift = 0.2
n_keep = int(round(structure.scatterers().size()
                   * (1-fraction_missing)))
partial_structure = xray.structure(
  special_position_settings=structure)
partial_structure.add_scatterers(
  structure.scatterers()[:n_keep])
partial_structure.replace_scatterers(
  partial_structure.random_shift_sites(
    max_shift_cart=max_shift).scatterers())
```

As before we compute structure factors, this time for the `partial_structure`:

```
f_calc = partial_structure.structure_factors(
  d_min          = d_min,
  anomalous_flag = False).f_calc()
```

For our demonstration we need an array of R-free flags (also known as a test set). We could generate the R-free flags in one line, but we break the code up for clarity:

```
from cctbx.array_family import flex

n_reflections = f_calc.data().size()
partitioning = flex.random_permutation(size=n_reflections) % 10
```

At this point `partitioning` is an integer array with randomly assigned but uniformly distributed values from 0 to 9. Insert `print list(partitioning)` to display the array. The next line turns this integer array into a boolean array. At the same time we build a `cctbx.miller.array` (Newsletter No. 1) with the same indexing set as `f_obs` but with the boolean array as the data:

```
r_free_flags = f_obs.array(data=(partitioning == 0))
```

Finally we have all the pieces we need to initialize the main object of this demonstration:

```
import mmtbx.refinement.f_model

f_model_manager = mmtbx.refinement.f_model.manager(
  f_calc = f_calc,
  f_obs = f_obs,
  r_free_flags = r_free_flags)
f_model_manager.show()
```

The output of the show() method is:

```
f_calc          =  <cctbx.miller.array object at 0xb5e9ff6c>
f_obs           =  <cctbx.miller.array object at 0xb60e170c>
f_mask          =  <cctbx.miller.array object at 0xb5eccb4c>
r_free_flags    =  <cctbx.miller.array object at 0xb5e9ff0c>
u_aniso         =  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
k_sol           =  0.0
b_sol           =  0.0
scale_work      =  1.0
scale_test      =  1.0
alpha           =  None
beta            =  None
sf_algorithm    =  None
target_name     =  None
target_functors =  None
```

Our f_model_manager maintains references to the input arrays (f_calc, f_obs, r_free_flags). We also see a new f_mask array used for bulk-solvent correction and all the parameters introduced above. Some parameters are not defined (None), but these are not needed in this example. As is, the f_model_manager is already able to answer certain questions, for example, what are the current values of R-work and R-free:

```
print f_model_manager.r_work()
print f_model_manager.r_free()
```

Output:

```
0.275794965768
0.28068823468
```

(The output may vary since we are working with a random structure.) More detailed information is just waiting for us:

```
f_model_manager.r_factors_in_resolution_bins(
  reflections_per_bin = 100,
  max_number_of_bins  = 10)
```

Output:

```
 Bin     Resolution       No. Refl.      R-factors
number     range          work   test     work   test
  1: 20.6581 -  3.8191    988    111   0.2778 0.2555
  2:  3.8191 -  3.0344    924    114   0.2596 0.2610
  3:  3.0344 -  2.6517    914    107   0.2649 0.2750
  4:  2.6517 -  2.4097    899    104   0.2713 0.2887
  5:  2.4097 -  2.2372    906     95   0.2856 0.2998
  6:  2.2372 -  2.1054    927     80   0.2992 0.2971
  7:  2.1054 -  2.0001    884    106   0.2846 0.3438
```

If model parameters are updated the `f_model_manager` automatically recomputes all dependent values:

```
f_model_manager.update(
  k_sol = 1.2,
  b_sol = 30.0)
```

This centralized, concise facility is extremely helpful in developing new refinement strategies.

At any stage, $\mathbf{F}^{\text{model}}$ according to the formula above, or just the bulk-solvent correction can easily be extracted:

```
f_model = f_model_manager.f_model()
f_bulk = f_model_manager.f_bulk()
```

Detailed and uniform statistics can easily be displayed in various contexts. For example:

```
f_model_manager.show_fom_phase_error_alpha_beta_in_bins(
  reflections_per_bin = 100,
  max_number_of_bins  = 10)
```

Output:

```
|-----------------------------------------------------------------------------|
|R-free likelihood based estimates for figures of merit, absolute phase error,|
|and distribution parameters alpha and beta (Acta Cryst. (1995). A51, 880-887)|
|                                                                             |
| Bin     Resolution        No. Refl.   FOM   phase err.   Alpha         Beta |
|number     range         work   test        <|p-p c|>                       |
|  1: 20.6581 -  3.8191    988    111  0.8414  20.7174     0.9663    5782.0632|
|  2:  3.8191 -  3.0344    924    114  0.8190  23.9808     0.9663    5782.0632|
|  3:  3.0344 -  2.6517    914    107  0.8208  24.0338     0.9372    4108.1081|
|  4:  2.6517 -  2.4097    899    104  0.8058  25.6936     0.9226    3272.3595|
|  5:  2.4097 -  2.2372    906     95  0.7733  28.8366     0.9251    3055.6162|
|  6:  2.2372 -  2.1054    927     80  0.7806  28.1986     0.9304    2583.5975|
|  7:  2.1054 -  2.0001    884    106  0.7484  31.1348     0.9304    2583.5975|
|-----------------------------------------------------------------------------|
```

After refinement is is often very helpful to inspect electron density maps. Since the `f_model_manager` controls all essential data for the calculation of maps, it is most natural to add a map generation method. For example:

```
fft_map = f_model_manager.electron_density_map(
  map_type = "2m*Fobs - alpha*Fmodel")
```

Or:

```
fft_map = f_model_manager.electron_density_map(
  map_type = "k*Fobs - n*Fmodel",
  k        = 2,
  n        = 1)
```

To end this demonstration, we bring in the *iotbx* utilities for writing maps in XPLOR format:

```
import iotbx.xplor.map

fft_map.as_xplor_map(
  file_name="2fo-fm.xplor",
  title_lines=["2*Fobs - Fmodel"],
  gridding_first=(0,0,0),
  gridding_last=fft_map.n_real())
```

Happy viewing! -- Well, admittedly it is not very interesting to view maps of random structures, but it works just the same given real data and real models.

The complete script can be found in the *cctbx* installation:

```
$MMTBX_DIST/mmtbx/examples/f_model_manager.py
```

3.2 Bulk-solvent correction and anisotropic scaling

In the previous issue of the Newsletter (No. 3) we briefly described a protocol for the determination of flat bulk-solvent model parameters and anisotropic scaling parameters. In the current version of the *cctbx* we have generalized this protocol significantly. The main features currently available are:

1. In addition to the least-squares target function presented before, a maximum-likelihood crystallographic target function can be used for the determination of the bulk-solvent and scale parameters. This enables a uniform overall strategy for maximum-likelihood model refinement since all parameters (bulk solvent, scale and atomic) can be refined against the same target function.
2. Three options for defining the bulk-solvent parameters (k_{sol}, B_{sol}) and the anisotropic scale matrix $\mathbf{U}_{aniso}$:
    a. Manual assignment. This is potentially useful at the beginning of structure refinement when the model has many errors.
    b. Minimization of a crystallographic target function using the *LBFGS* minimizer. This is a quick and precise way of determining k_{sol}, B_{sol} and $\mathbf{U}_{aniso}$ if a model of reasonable quality is available and the experimental data extend to sufficiently low resolution. However, this algorithm fails to produce physically reasonable parameters in some situations. This experience was the motivation for implementing the more sophisticated protocol outlined below.
    c. Combined *LBFGS* minimization and grid search algorithm (Afonine *et al*, 2005). This is the most robust procedure for the determination of k_{sol}, B_{sol} and $\mathbf{U}_{aniso}$. However, it is also the most time-consuming option.

The bulk-solvent and scaling algorithms are implemented in the `mmtbx.bulk_solvent` module.

3.3 Simulated annealing refinement

Simulated annealing is a time-tested tool for escaping local minima in crystallographic refinement (Brünger *et al*, 1987). Recently we have implemented a simulated annealing algorithm for restrained molecular dynamics in the *cctbx.* This enables us to take full advantages of combined simulated annealing and maximum-likelihood model refinement (Adams *et al*, 1997; Brunger & Adams, 2002).
The simulated annealing algorithms are implemented in the `mmtbx.cartesian_dynamics` module.

3.4 Building of hydrogen atoms

Fourier syntheses at subatomic resolution (d_{min} < 1.0 Å) usually reveal the presence of hydrogen atoms. At lower resolutions this information is lost. Therefore a general refinement program has to provide different strategies depending on the resolution of the data. If ultra-high resolution data are available, hydrogens can be explicitly included in the refinement, for example using the riding hydrogen model (Sheldrick, 1995). At lower resolutions the inclusion of hydrogens in the refinement target for the X-ray data is likely to lead to overfitting. However in this case the hydrogens should still be considered in the definition of the geometry restraints, and this has been shown to improve atomic models even in the absence of atomic resolution data (Richardson et al. 2003). In addition, refinement against neutron diffraction data requires appropriate modeling of hydrogen atoms.

As a first step towards covering these cases we have implemented a hydrogen building procedure for the standard amino acid residues. In most cases the hydrogen positions are geometrically well defined. However, there are some cases where the positions are not unambiguously determined, such as -CH3, -OH in a tyrosine residue. To account for this, our procedure consists of two steps. In the first step we place all expected hydrogen atoms in appropriate positions. If ambiguities exist, we place the affected hydrogens arbitrarily in a one of the allowed positions. In the second step we perform model regularization by refinement against geometry restraints (see Newsletter No. 4). Optionally, this can be combined with Cartesian dynamics to escape from local minima.

The hydrogen building algorithms are implemented in the `mmtbx.hydrogens` module.

3.5 Maximum-likelihood tools

Previously we had implemented an amplitude-based maximum-likelihood target function (Lunin *et al*, 2002), its quadratic approximation (Lunin & Urzhumtsev, 1999), and a procedure for estimating the distribution parameters (alpha, beta) according to Lunin & Skovoroda (1995). Recently we have extended the set of maximum-likelihood tools by these methods:

R-free likelihood-based estimation of model phase errors and figures of merit

This procedure is based on the algorithm described by Lunin & Skovoroda (1995). The mean phase errors and figures of merit are determined in narrow resolution bins using test reflections only. The procedure provides relatively precise and unbiased values for these parameters. The algorithms are available via methods of the `mmtbx.refinement.f_model.manager` class introduced in section 3.1, e.g.:

```
figures_of_merit = f_model_manager.figures_of_merit()
phase_errors = f_model_manager.phase_errors()
```

Coefficients for Fourier Syntheses

It was straightforward to implement the calculation of "best" coefficients for Fourier syntheses, $\left[2m_s F_s^{\text{obs}} - \alpha_s F_s^{\text{model}}\right]\exp\left(i\varphi_s^{\text{calc}}\right)$, where m_s are figures of merit and $\alpha_s = \left\langle\cos\left(\mathbf{s}, \Delta\mathbf{r}\right)\right\rangle$ (Urzhumtsev et al., 1996; Read, 1986 uses the notation D_s). The `f_model_manager.electron_density_map()` method demonstrated in section 3.1 provides an interface to these algorithms.

Use of Experimental Phase Information

We are actively working on fast C++ code for a maximum-likelihood target which includes experimental phase information (MLHL target; Pannu *et al*, 1998). This code is in the *cctbx* bundles already but not yet fully tested.

4 iotbx.reflection_statistics

Recently we have enhanced the `iotbx.reflection_statistics` command significantly. The initial version (written in April 2004) can be used to compute data completeness, anomalous signals, correlations between intensities and correlations between anomalous signals of pairs of reflection arrays. All these statistics are computed both in resolution shells and as overall quantities. The latest version (written in December 2004) adds these new features:

- Automatic determination of the space group of the metric (i.e. the lattice symmetry; see also Newsletter No. 3).
- Automatic derivation of a non-redundant set of possible twin laws from first principles (Flack, 1987).
- Automatic derivation of a non-redundant set of possible reindexing matrices for comparing two datasets. The matrices are derived from first principles (see below).
- Computation of a sorted list of peaks in the native Patterson synthesis to facilitate the detection of translational non-crystallographic symmetry (NCS).
- Tests for perfect merohedral twinning using both the second moments of amplitudes (also known as Wilson ratios) and intensities (Yeates, 1997).

With the old version of the `iotbx.reflection_statistics` command correlations between pairs of reflection arrays are computed only if the unit cell parameters and the space group symmetries are identical. The new version is designed to overcome this limitation in the most general way. Internally, all arrays are transformed to a primitive setting. The change-of-basis matrices are determined with a cell reduction algorithm (see Newsletter No. 3). Each array in the primitive setting is expanded to P1. I.e. the symmetry matrices are applied to generate all equivalent Miller indices. Given a pair of reflection arrays preprocessed in this way, a newly developed algorithm performs an exhaustive search for the change-of-basis matrix that leads to the best superposition of the reduced unit cells. This algorithm employs the new `similarity_transformations()` and `bases_mean_square_difference()` methods of the `cctbx.uctbx.unit_cell` class. Associated with each unit cell is the space group of the metric as determined with the lattice symmetry algorithm outlined in the Newsletter No. 3. If the tolerances used in the computation of the cell superposition are reasonable, the metric symmetries are identical, or one is a subgroup of the other. We continue with the highest metric space group. Each symmetry operation of this space group is a possible reindexing matrix. Conceptually, we compute the correlations between two arrays for each reindexing matrix and produce a sorted list of the results. However, if any of the space groups of the two input arrays are different from P1, this leads to a redundant list. The remove these redundancies, we employ double coset decomposition (see below). To minimize the runtime, redundant correlations are never computed.

The algorithmic complexities are in stark contrast to the simple end-user interface. The universal reflection file reader described in Newsletter No. 3 is used to automatically detect and process all common file formats. A possible command for comparing reflection data is:

```
iotbx.reflection_statistics *.sca *.mtz
```

For a large number of arrays this may take a couple of minutes, but the comprehensive analyses do not require any user intervention. The potentially large output contains tags for quick searching. A guide is printed at the beginning of the output. For example:

```
Array indices (for quick searching):
   1: hg1.0_scale_anomalous.sca:i_obs,sigma
   2: hginfl_3.5_ano.sca:i_obs,sigma
   3: hgpeak_3.5_ano.sca:i_obs,sigma
   4: pb1.0_4.0_ano.sca:i_obs,sigma
   5: pbpeak_3.5_scale_anomalous.sca:i_obs,sigma
   6: pt_4.0_ano.sca:i_obs,sigma
   7: scale.sca:i_obs,sigma
   8: sm_scale_anomalous.sca:i_obs,sigma
   9: tmpb.sca:i_obs,sigma

Useful search patterns are:
    Summary i
    CC Obs i j
    CC Ano i j
  i and j are the indices shown above.
```

If we search for `CC Obs 7 1` we find:

```
CC Obs 7 1  0.956 h,-k,-l
Correlation of:
  scale.sca:i_obs,sigma
  hg1.0_scale_anomalous.sca:i_obs,sigma
Overall correlation with reindexing:  0.956 h,-k,-l
unused:         - 43.6948 [   4/20  ]  1.000
bin  1: 43.6948 -  8.6072 [2856/2950]  0.954
bin  2:  8.6072 -  6.8402 [2916/2924]  0.962
bin  3:  6.8402 -  5.9780 [3028/3036]  0.957
bin  4:  5.9780 -  5.4326 [2936/2948]  0.960
bin  5:  5.4326 -  5.0438 [2940/2960]  0.964
bin  6:  5.0438 -  4.7468 [2792/2818]  0.959
bin  7:  4.7468 -  4.5093 [3104/3124]  0.954
bin  8:  4.5093 -  4.3132 [2824/2842]  0.949
bin  9:  4.3132 -  4.1473 [2904/2930]  0.946
bin 10:  4.1473 -  4.0043 [2964/2990]  0.937
unused:  4.0043 -         [  24/72  ]  0.904

CC Obs 7 1  0.364 h,k,l
Correlation of:
  scale.sca:i_obs,sigma
  hg1.0_scale_anomalous.sca:i_obs,sigma
Overall correlation:  0.364
```

In this example the highest correlation (0.956) between the two arrays is found with the reindexing matrix `h,-k,-l`. In contrast, the correlation between the arrays as indexed originally is only 0.364.

The `iotbx.reflection_statistics` command is implemented in the file `$IOTBX_DIST/iotbx/command_line/reflection_statistics.py`.

4.1 Double coset decomposition

A useful summary of the theory of double cosets can be found in *An introduction to group theory* by Tony Gaglione, which is available online:

http://web.usna.navy.mil/~wdj/tonybook/gpthry/node44.html

Double coset decomposition is concerned with a group `g` and two subgroups `h1` and `h2`. The group `g` is partitioned into non-overlapping sets of symmetry operations equivalent under `h1` and `h2`. In the context of the algorithm outlined above, `g` is the highest space group of the metric. `h1` and `h2` are the space groups of the arrays to be compared. Each double coset represents a reindexing choice unique under `h1` and `h2`. I.e. any matrix selected from a given double coset will lead to identical correlation coefficients.

If we do not care which matrix is selected from a given double coset, we arrive at a surprisingly simple algorithm. The following is the relevant fragment from the file `$CCTBX_DIST/cctbx/sgtbx/cosets.py`:

```
def double_unique(g, h1, h2):
  result = []
  done = {}
  for a in g:
    if (str(a) in done): continue
    result.append(a)
    for hi in h1:
      for hj in h2:
        b = hi.multiply(a).multiply(hj)
        done[str(b)] = None
  return result
```

`g`, `h1` and `h2` are instances of `cctbx.sgtbx.space_group`. The algorithm follows directly from the definition of cosets as found at the web page referenced above:

> For `a`, `b` element of `g`, we define `a` ~ `b` if and only if `h1` `a` `h2` = `b`.

`h1` `a` `h2` = `b` corresponds to `b = hi.multiply(a).multiply(hj)` in the Python code.

`result` is a Python list of representative matrices, one from each coset. Which matrices are returned depends on the order of the matrices in `g`, `h1` and `h2`. This may not always yield the "nicest" choice. However, any investment in a more sophisticated selection has little or no practical value. Typically the transformed indices are mapped into a canonical asymmetric unit (e.g. using the `map_to_asu()` method of `cctbx.miller.array`). After this manipulation the indexing set will be the same no matter which matrix from a given double coset is selected.

5 iotbx.mtz

CCP4 MTZ files are binary files containing merged or unmerged reflection data and optionally information about raw data ("batches"). For a couple of years already the *cctbx* has included C++ and Python interfaces to the CCP4 C MTZ library in the `iotbx.mtz` module. However, while the support for reading MTZ files was quite complete, creating and writing MTZ files was only partially supported. To

resolve this problem and to unify the interfaces for reading and writing, the `iotbx.mtz` module was heavily restructured. We have also added complete C++ and Python interfaces for the manipulation of MTZ batches. The `iotbx.mtz` module extends the functionality of the CCP4 C MTZ library by automatically grouping related MTZ columns into one object, `cctbx.miller.array` instances as introduced in Newsletter No. 1.

Combined with the universal reflection file reader, it is quite easy to quickly write a script for converting any of the formats processed by the reflection file reader to the MTZ format. First let's get some data to work with:

```
from iotbx import reflection_file_reader
import os

reflection_file = reflection_file_reader.any_reflection_file(
  file_name=os.path.expandvars(
    "$CNS_SOLVE/doc/html/tutorial/data/pen/scale.hkl"))
```

We are reading a CNS reflection file in the CNS tutorial. (To run this example CNS has to be installed including the tutorial.) Since the crystal symmetry is not defined in CNS reflection files, we supply this information manually:

```
from cctbx import crystal

crystal_symmetry = crystal.symmetry(
  unit_cell=(97.37, 46.64, 65.47, 90, 115.4, 90),
  space_group_symbol="C2")
```

We convert the reflection file to a list of `cctbx.miller.array` instances:

```
miller_arrays = reflection_file.as_miller_arrays(
  crystal_symmetry=crystal_symmetry)
```

Now we loop over the Miller arrays to convert them to MTZ data columns:

```
mtz_dataset = None
for miller_array in miller_arrays:
  if (mtz_dataset is None):
    mtz_dataset = miller_array.as_mtz_dataset(
      column_root_label=miller_array.info().labels[0])
  else:
    mtz_dataset.add_miller_array(
      miller_array=miller_array,
      column_root_label=miller_array.info().labels[0])
```

Let's see what we got:

```
mtz_object = mtz_dataset.mtz_object()
mtz_object.show_summary()
```

The output ends with:

```
Column number, label, number of valid values, type:
  1 H         6735/6735=100.00% H: index h,k,l
  2 K         6735/6735=100.00% H: index h,k,l
  3 L         6735/6735=100.00% H: index h,k,l
  4 F_PHGA    6735/6735=100.00% F: amplitude
  5 SIGF_PHGA 6735/6735=100.00% Q: standard deviation
  6 F_KUOF    6735/6735=100.00% F: amplitude
  7 SIGF_KUOF 6735/6735=100.00% Q: standard deviation
  8 F_NAT     6735/6735=100.00% F: amplitude
  9 SIGF_NAT  6735/6735=100.00% Q: standard deviation
```

Finally we write the MTZ file to disk:

```
mtz_object.write("pen_data.mtz")
```

Note that the `iotbx.mtz.dump pen_data.mtz` command is available to produce the same output as the `mtz_object.show()` statement in the example.

6 Integration of PyCifRW

PyCifRW is a library for reading and writing CIF (Crystallographic Information Format) files using Python. PyCifRW was developed by James Hester at the Australian National Beamline Facility (ANBF). Documentation can be found online:

```
http://www.ansto.gov.au/natfac/ANBF/CIF/
```

Recently, the PyCifRW license was changed to allow redistribution. We are very excited about this development because it allows us to include PyCifRW in the *cctbx* bundles. However, like the CCP4 I/O library and Clipper (see Newsletter No. 4), PyCifRW is not in the *cctbx* CVS tree on SourceForge. James Hester continues to develop PyCifRW in his own environment and we will update the *cctbx* bundles with the latest releases. Currently we redistribute PyCifRW version 1.19 released in November 2004.

PyCifRW in a *cctbx* installation is used in the same way as described in the PyCifRW documentation. Let's try it out. We develop a self-contained Python script by starting with embedded CIF syntax:

```
file("quartz.cif", "w").write("""
  data_global
  _chemical_name Quartz
  _cell_length_a 4.9965
  _cell_length_b 4.9965
  _cell_length_c 5.4570
  _cell_angle_alpha 90
  _cell_angle_beta 90
  _cell_angle_gamma 120
  _symmetry_space_group_name_H-M 'P 62 2 2'
  loop_
  _atom_site_label
  _atom_site_fract_x
  _atom_site_fract_y
  _atom_site_fract_z
  Si   0.50000   0.00000   0.00000
  O   0.41520   0.20760   0.16667
  """)
```

At this point we have created a file `quartz.cif`. Now we parse it with PyCifRW:

```
from PyCifRW.CifFile import CifFile

cif_file = CifFile("quartz.cif")
cif_global = cif_file["global"]
print cif_global["_chemical_name"]
```

Output:

```
Quartz
```

Looks like a good start! But we want more. For example, structure factors. For this we have to process the rest of the data in the CIF file. First we determine the crystal symmetry:

```
from cctbx import uctbx, sgtbx, crystal

unit_cell = uctbx.unit_cell([float(cif_global[param])
  for param in [
  "_cell_length_a","_cell_length_b","_cell_length_c",
  "_cell_angle_alpha","_cell_angle_beta","_cell_angle_gamma"]])
space_group_info = sgtbx.space_group_info(
  symbol=cif_global["_symmetry_space_group_name_H-M"])
crystal_symmetry = crystal.symmetry(
  unit_cell=unit_cell,
  space_group_info=space_group_info)
crystal_symmetry.show_summary()
```

Output:

```
Unit cell: (4.9965, 4.9965, 5.457, 90, 90, 120)
Space group: P 62 2 2 (No. 180)
```

Now we turn our attention to the list of coordinates and create a new `cctbx.xray.structure` instance:

```
from cctbx import xray

structure = xray.structure(crystal_symmetry=crystal_symmetry)
for label,x,y,z in zip(cif_global["_atom_site_label"],
                       cif_global["_atom_site_fract_x"],
                       cif_global["_atom_site_fract_y"],
                       cif_global["_atom_site_fract_z"]):
  scatterer = xray.scatterer(
    label=label,
    site=[float(s) for s in [x,y,z]])
  structure.add_scatterer(scatterer)
structure.show_summary().show_scatterers()
```

Output:

```
Number of scatterers: 2
At special positions: 2
Unit cell: (4.9965, 4.9965, 5.457, 90, 90, 120)
Space group: P 62 2 2 (No. 180)
Label, Scattering, Multiplicity, Coordinates, Occupancy, Uiso
Si   Si     3 ( 0.5000  0.0000  0.0000) 1.00 0.0000
O    O      6 ( 0.4152  0.2076  0.1667) 1.00 0.0000
```

Just one more hoop and we have the structure factors:

```
f_calc = structure.structure_factors(d_min=2).f_calc()
abs(f_calc).show_summary().show_array()
```

Output:

```
Miller array info: None
Observation type: None
Type of data: double, size=7
Type of sigmas: None
Number of Miller indices: 7
Anomalous flag: False
Unit cell: (4.9965, 4.9965, 5.457, 90, 90, 120)
Space group: P 62 2 2 (No. 180)
(1, 0, 0) 15.708493924
(1, 0, 1) 36.2626337008
(1, 0, 2) 7.77312576362
(1, 1, 0) 14.9039425672
(1, 1, 1) 0.975009858138
(2, 0, 0) 15.8407980479
(2, 0, 1) 13.6738859288
```

Note that this is almost what we had in Newsletter No. 1. The main difference is that we start from a CIF file rather than the plain *cctbx* interfaces.

The complete script can be found in the *cctbx* installation:

```
$PYCIFRW_DIST/example_quartz.py
```

7 Acknowledgments

We like to thank James Hester for writing PyCifRW and for his hard work concerning the PyCifRW license. 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.

8 References

Adams, P. D., Pannu, N. S., Read, R. J. & Brünger, A. T. (1997). *Proc. Natl. Acad. Sci*. **94**, 5018-5023.

Afonine, P.V., Grosse-Kunstleve, R.W. & Adams, P. D. (2005). Submitted.

Brunger, A. T & Adams, P. D. (2002). *Acc. Chem. Res*. **35**, 404-412.

Brünger, A. T., Kuriyan, J., Karplus, M. (1987). *Science*. **235**, 458- 460.

Flack, H.D. (1987). *Acta Cryst.* **A43**, 564-568.

Grosse-Kunstleve, R.W., Adams, P.D. (2003). *Newsletter of the IUCr Commission on Crystallographic Computing*, 1, 28-38. http://www.iucr.org/iucr-top/comm/ccom/newsletters/2003jan/

Grosse-Kunstleve, R.W., Sauter, N.K., Adams, P.D. (2004). *Newsletter of the IUCr Commission on Crystallographic Computing*, 3, 22-31. http://www.iucr.org/iucr-top/comm/ccom/newsletters/2004jan/

Grosse-Kunstleve, R.W., Afonine, P.V., B., Adams, P.D. (2004). *Newsletter of the IUCr Commission on Crystallographic Computing*, 4, 19-36. http://www.iucr.org/iucr-top/comm/ccom/newsletters/2004aug/

Jiang, J.-S. & Brünger, A. T. (1994). *J. Mol. Biol*. **243**, 100-115.

Lunin, V.Y. & Skovoroda, T.P. (1995). *Acta Cryst*. **A51**, 880-887.

Lunin, V.Y. & Urzhumtsev, A. (1999). *CCP4 Newsletter on Protein Crystallography*, **37**, 14-28.

Lunin, V.Y., Afonine, P.V. & Urzhumtsev, A. (2002). *Acta Cryst.,* **A58,** 270-282.

Pannu, N. S., Murshudov, G. N., Dodson, E. J. & Read, R. J. (1998). *Acta Cryst*. **D54**, 1285-1294.

Read, R.J. (1986). *Acta Cryst*. **A42**, 140-149.

Richardson, J.S., Arendall, W.B. III, and Richardson, D.C. (2003). *Methods Enzymol.* **374**, 385-412.

Sheldrick, G.M. (1985). SHELXS86. Program for the Solution of Crystal Structures. Univ. of Göttingen, Germany.

Sheriff, S. & Hendrickson, W. A. (1987). *Acta Cryst*. **A43**, 118-121.

Urzhumtsev, A.G., Skovoroda, T.P., Lunin, V.Y. (1996). *J. Appl. Cryst.* **29**, 741-744.

Yeates, T.O. (1997). *Methods Enzymol.* **276**, 344-358.