cctbx news

R.W. Grosse-Kunstleve, N.K. Sauter & P.D. Adams
Lawrence Berkeley National Laboratory, One Cyclotron Road, BLDG 4R0230, Berkeley, CA 94720-8235, USA. E-mail: RWGrosse-Kunstleve@lbl.gov ; WWW: http://cci.lbl.gov/ and http://cctbx.sourceforge.net/

1: Introduction

The Computational Crystallography Toolbox (cctbx, http://cctbx.sourceforge.net/) is an open-source library of reusable crystallographic algorithms. In this article we give an overview of recent developments. All example scripts shown below were tested with cctbx build 2004_01_16_1718.

2: Reduced cell computations

In the International Tables for Crystallography Volume A an entire chapter (No. 9) is devoted to the discussion of *Crystal Lattices*. At the center of the chapter is a treatment of *reduced bases*. By definition the basis vectors of a reduced basis are the three shortest, non-coplanar lattice vectors. Finding such a basis given a different choice of basis vectors is the subject of cell reduction algorithms. For example, the following `primitive_setting` is the result of transforming a C-centred monoclinic cell:

```
from cctbx import crystal

print "Monoclinic C-centred setting:"
monoclinic_c = crystal.symmetry(
  unit_cell=(25.0822, 5.04549, 29.4356, 90, 103.108, 90),
  space_group_symbol="C12/m1")
monoclinic_c.show_summary()
print "Number of lattice translations:", \
  monoclinic_c.space_group().n_ltr()
print

print "Primitive setting:"
primitive_setting = monoclinic_c.change_basis("-x-y,-x+y,-z")
primitive_setting.show_summary()
print "Number of lattice translations:", \
  primitive_setting.space_group().n_ltr()
```

Output:

```
Monoclinic C-centred setting:
Unit cell: (25.0822, 5.04549, 29.4356, 90, 103.108, 90)
Space group: C 1 2/m 1 (No. 12)
Number of lattice translations: 2

Primitive setting:
Unit cell: (12.7923, 12.7923, 29.4356, 102.846, 102.846, 22.7475)
Space group: Hall: -C 2y (x-y,x+y,z) (No. 12)
Number of lattice translations: 1
```

In this case the intention was to continue a structure refinement in a triclinic space group. However, the transformation `"-x-y,-x+y,-z"` leads to an unfortunate γ angle of about 23º which makes it difficult to visualize the structure. This problem can be avoided by transforming to the corresponding reduced cell:

```
print "Niggli cell:"
niggli_cell = primitive_setting.niggli_cell()
niggli_cell.show_summary()
```

Output:

```
Niggli cell:
Unit cell: (5.04549, 12.7923, 29.3711, 77.7182, 85.0727, 78.6263)
Space group: Hall: -C 2y (-x+y,z,2*x-z) (No. 12)
```

To facilitate this calculation the `uctbx` (unit cell toolbox) module of the cctbx was expanded to include several cell reduction algorithms: a Buerger reduction according to Gruber (1973), a Niggli reduction according to Krivy & Gruber (1976) and a new *minimum reduction* according to Grosse-Kunstleve et al. (2004). The example above shows that the change-of-basis operator which transforms the given basis to the reduced basis is also automatically applied to the space group symmetry (note the change from `C 1 2/m 1` to `Hall: -C 2y (-x+y,z,2*x-z)`). The automatic transformation mechanism extends to entire structures and reflection data:

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

structure_monoclinic_c = xray.structure(
  crystal_symmetry=monoclinic_c,
  scatterers=flex.xray_scatterer((
    xray.scatterer(label="Si", site=(0.1,0.2,0.3)),
    xray.scatterer(label="O", site=(0.2,0.3,0.4)))))
structure_monoclinic_c.show_summary().show_scatterers()
print

structure_niggli_cell = structure_monoclinic_c.niggli_cell()
structure_niggli_cell.show_summary().show_scatterers()
```

Output:

```
Number of scatterers: 2
At special positions: 0
Unit cell: (25.0822, 5.04549, 29.4356, 90, 103.108, 90)
Space group: C 1 2/m 1 (No. 12)
Label, Scattering, Multiplicity, Coordinates, Occupancy, Uiso
Si   Si     8 ( 0.1000  0.2000  0.3000) 1.00 0.0000
O    O      8 ( 0.2000  0.3000  0.4000) 1.00 0.0000

Number of scatterers: 2
At special positions: 0
Unit cell: (5.04549, 12.7923, 29.3711, 77.7182, 85.0727, 78.6263)
Space group: Hall: -C 2y (-x+y,z,2*x-z) (No. 12)
Label, Scattering, Multiplicity, Coordinates, Occupancy, Uiso
Si   Si     4 (-0.3000 -0.1000  0.3000) 1.00 0.0000
O    O      4 (-0.5000  0.0000  0.4000) 1.00 0.0000
```

Now for reflection data:

```
f_calc_monoclinic_c = abs(structure_monoclinic_c.structure_factors(
  d_min=1, # high-resolution limit
  anomalous_flag=False,
  algorithm="direct").f_calc())
f_calc_monoclinic_c.show_summary()
print

f_calc_niggli_cell = f_calc_monoclinic_c.niggli_cell()
f_calc_niggli_cell.show_summary()
```

Output:

```
Type of data: double, size=2168
Type of sigmas: None
Number of Miller indices: 2168
Anomalous flag: 0
Unit cell: (25.0822, 5.04549, 29.4356, 90, 103.108, 90)
Space group: C 1 2/m 1 (No. 12)

Type of data: double, size=2168
Type of sigmas: None
Number of Miller indices: 2168
Anomalous flag: 0
Unit cell: (5.04549, 12.7923, 29.3711, 77.7182, 85.0727, 78.6263)
Space group: Hall: -C 2y (-x+y,z,2*x-z) (No. 12)
```

This example also shows that the intermediate transformation (`"-x-y,-x+y,-z"` above) is not needed. The change-of-basis operator which transforms a given setting to a setting in the Niggli cell is determined automatically by the unit cell toolbox and not even presented to the user in the example above. However, if so desired the operator is of course available:

```
print "Change-of-basis original cell -> Niggli cell:"
change_of_basis_op = structure_monoclinic_c \
  .change_of_basis_op_to_niggli_cell()
print "  operator:", change_of_basis_op.c()
print "   inverse:", change_of_basis_op.c_inv()
```

Output:

```
Change-of-basis original cell -> Niggli cell:
  operator: -x-y,2*x-z,z
   inverse: 1/2*y+1/2*z,-x-1/2*y-1/2*z,z
```

Our recent paper on reduced cell algorithms (Grosse-Kunstleve et al., 2004) contains pointers to the relevant source code files in the cctbx module.

3: Determination of lattice symmetry

To support a novel auto-indexing procedure that was developed in our group (Sauter et al., 2004), we have added an algorithm for the determination of the lattice symmetry to the `sgtbx` (space group toolbox) module of the cctbx (determining the lattice symmetry is equivalent to determining the Bravais type). The algorithm can be executed via the web interface at http://cci.lbl.gov/cctbx/ or through the command line interface provided in the `iotbx` (input output toolbox) module of the cctbx. For example:

```
% iotbx.lattice_symmetry --unit_cell "12,13,14,92,88,100"

Input
=====

Unit cell: (12, 13, 14, 92, 88, 100)
Space group: P 1 (No. 1)

Angular tolerance: 3.000 degrees

Similar symmetries
==================

Symmetry in minimum-lengths cell: P 1 1 2/m (No. 10)
      Input minimum-lengths cell: (12, 13, 14, 88, 88, 80)
           Symmetry-adapted cell: (12, 13, 14, 90, 90, 80)
            Conventional setting: P 1 2/m 1 (No. 10)
                       Unit cell: (12, 14, 13, 90, 100, 90)
                 Change of basis: -x,-z,-y
                         Inverse: -x,-z,-y
      Maximal angular difference: 2.611 degrees

Symmetry in minimum-lengths cell: P -1 (No. 2)
      Input minimum-lengths cell: (12, 13, 14, 88, 88, 80)
           Symmetry-adapted cell: (12, 13, 14, 88, 88, 80)
            Conventional setting: P -1 (No. 2)
                       Unit cell: (12, 13, 14, 88, 88, 80)
                 Change of basis: -x,y,-z
                         Inverse: -x,y,-z
      Maximal angular difference: 0.000 degrees
```

The first step of the algorithm is to determine a reduced basis as presented in the previous section. The second step is to determine all two-fold axes of the crystal lattice according to Le Page (1982). This is equivalent to searching for 90º angles between lattice vectors. To account for experimental uncertainties

and rounding errors it is necessary to consider an `Angular tolerance` (δ_{max} in Le Page, 1982). In the example above deviations up to 3º are permitted. The list of two-fold axes found in the search is sorted by the angular deviation, with the smallest deviation first.

The highest-symmetry space group compatible with the given unit cell is determined by successive group multiplication starting with the first two-fold in the list. The other two-folds are added to the group one-by-one until the list is exhausted or the group multiplication leads to an infinite group (e.g. performing group multiplication starting with a four-fold axis and a six-fold axis leads to an infinite group). This procedure leads to the highest-symmetry space group because any crystallographic space group can be generated using just the two-fold axes and, for centric space groups, a centre of inversion. For example:

```
from cctbx import sgtbx
group = sgtbx.space_group() # start with P1
for two_fold in ["-x,-y,z", "-y,-x,-z", "-z,-y,-x"]:
  group.expand_smx(two_fold)
print sgtbx.space_group_info(group=group)

group.expand_smx("-x,-y,-z") # centre of inversion
print sgtbx.space_group_info(group=group)
```

Output:

```
P 4 3 2
P m -3 m
```

The source code that implements the search for two-fold axes and the group multiplication can be found in the file `cctbx/sgtbx/lattice_symmetry.cpp`.

It may happen that the highest-symmetry space group as obtained by the group multiplication has a fairly large maximum angular deviation (e.g. `2.611 degrees` in above). It is therefore of interest to compute the maximum angular deviation for each subgroup of the highest symmetry. The search for subgroups is quite simple because it is well known that any space group can be generated by performing a group multiplication starting with just two symmetry operations and, for centric space group, the center of inversion. Since any crystal lattice has a center of inversion it can be factored out of the determination of the subgroups. We can work with the acentric subgroup of the highest symmetry and add the center of inversion at the end of the procedure. A two-deep loop, each over all symmetry operations of the highest-symmetry space group, produces all possible combinations of generators for the subgroups. For example:

```
highest_symmetry = sgtbx.space_group_info("P 4 3 2").group()
for i_smx in xrange(highest_symmetry.order_p()):
  for j_smx in xrange(i_smx, highest_symmetry.order_p()):
    subgroup = sgtbx.space_group() # start with P1
    subgroup.expand_smx(highest_symmetry(i_smx))
    subgroup.expand_smx(highest_symmetry(j_smx))
```

In the worst case (cubic symmetry) this involves (24+1)*24/2 = 300 iterations since we are always working with primitive settings and acentric groups; i.e. it is always very fast.

In the complete implementation (file `cctbx/cctbx/sgtbx/subgroups.py`) duplicate subgroups are removed. The unique subgroups are sorted and the maximum angular deviation computed for each. These values can be used as a guide for deciding which symmetry to work with in higher-level applications.

The next step in the determination of all possible lattice symmetries is to make the input unit cell parameters exactly fit the given subgroup symmetry. Symmetry-adapted parameters are obtained by converting the unit cell parameters to a metrical matrix `g` (also known as metric tensor) which must be invariant under the following tensor transformation for all rotation matrices `r` in the subgroup:

```
g = r.transpose() * g * r
```

This follows directly from the transformation law for tensors (e.g. Giacovazzo, 1992, p. 130). For example:

```
from cctbx import sgtbx
from cctbx import uctbx
from cctbx import matrix

space_group = sgtbx.space_group_info("P 3").group()
unit_cell = uctbx.unit_cell("10,10,12,90,90,120")
g = matrix.sym(unit_cell.metrical_matrix())
for s in space_group:
  r = matrix.sqr(s.r().num()) # rotation part of symmetry operation
  print (r.transpose() * g * r).mathematica_form(
    one_row_per_line=True,
    format="%.2f")
```

Output:

```
{{100.00, -50.00, 0.00},
  {-50.00, 100.00, 0.00},
  {0.00, 0.00, 144.00},
}
{{100.00, -50.00, 0.00},
  {-50.00, 100.00, -0.00},
  {0.00, -0.00, 144.00},
}
{{100.00, -50.00, -0.00},
  {-50.00, 100.00, 0.00},
  {-0.00, 0.00, 144.00},
}
```

To see what happens if the unit cell parameters are not compatible with the symmetry we change the b-axis from `10` to `11`:

```
unit_cell = uctbx.unit_cell("10,11,12,90,90,120")
```

New output:

```
{{100.00, -55.00, 0.00},
  {-55.00, 121.00, 0.00},
  {0.00, 0.00, 144.00},
}
{{121.00, -66.00, 0.00},
  {-66.00, 111.00, -0.00},
  {0.00, -0.00, 144.00},
}
{{111.00, -45.00, -0.00},
  {-45.00, 100.00, 0.00},
  {-0.00, 0.00, 144.00},
}
```

An obvious way to make the parameters compatible is to average the transformed metrical matrices and to compute new unit cell parameters from the average (see also Grosse-Kunstleve et al., 2002, section 3.3). The cctbx provides an easy to use interface to this functionality:

```
print space_group.average_unit_cell(unit_cell)
```

Output:

```
(10.5198, 10.5198, 12, 90, 90, 120)
```

As the final step each subgroup along with its symmetry-adapted unit cell is transformed from the primitive setting to a standard setting. This is achieved using the algorithm of Grosse-Kunstleve (1999). The script that puts all steps of the determination of the lattice symmetry together is in the file `iotbx/iotbx/command_line/lattice_symmetry.py`.

4: N-Gaussian approximations to scattering factors

To compute X-ray structure factors a program must have access to the scattering factors of all the elements and ions involved. These data are tabulated in two forms in the International Tables for Crystallography, Volume C, section 6.1.1:

- Primary data: tables of sin(theta)/lambda and the corresponding scattering factor.
- Gaussian approximations to the primary data with 4 Gaussian terms `a * exp(-b * (sin(theta)/lambda)**2)` plus a constant term.

The Gaussian approximations are used by most crystallographic applications (e.g. SHELX, CCP4 and CNS). In general the approximations are valid up to sin(theta)/lambda = 2Å^{-1} (d_{min} = 1/4 Å). Waasmeier & Kirfel (1995) introduced approximations with 5 Gaussian terms plus a constant term that are valid up to sin(theta)/lambda = 6 Å^{-1} (d_{min} = 1/12 Å). Both libraries of approximations have been part of the cctbx for a long time (`cctbx.eltbx.xray_scattering.it1992` and `cctbx.eltbx.xray_scattering.wk1995`).

When computing structure factors for macromolecular structures the high-resolution limit is usually significantly lower than the limit of the Gaussian approximations. At the same time the number of terms plus one for the constant term in the Gaussian approximations enters, to a first approximation, as a linear factor into the CPU time required for a structure factor calculation using the FFT-based algorithm of Ten Eyck (1977). This has prompted Agarwal (1978) to suggest 2-term Gaussian approximations for the most prevalent elements in proteins: H, C, N, O, S. These approximations are valid to d_{min} = 1.5 Å with a relative error of about 1% at the highest resolution. Using these coefficients instead of the 4-plus-1-term approximations of the International Tables reduces the CPU time for sampling the electron density according to Ten Eyck (1977) by about 60%. Since the sampling is one of the rate-limiting steps in macromolecular structure refinement the gain is in practice very substantial.

To facilitate a dynamic adjustment of the number of Gaussian terms we have computed a comprehensive library of N-Gaussian approximations to the primary data for all elements and ions listed in section 6.1.1 of the International Tables. For example:

```
scattering_type: N
stol: 6.00 # d_min: 0.08, max_error: 0.0017
a: 2.7754532 1.3759575 1.0628956 1.038057 0.62582183 0.12084177
b: 15.064476 7.1774688 0.52744677 37.962277 0.18761875 0.047184388
c: 0
stol: 5.00 # d_min: 0.10, max_error: 0.0038
a: 3.2903774 1.8375163 1.0084335 0.62711549 0.23302095
b: 10.300994 30.499187 0.28689128 0.76591255 0.068219922
c: 0
stol: 3.00 # d_min: 0.17, max_error: 0.0081
a: 3.1621224 1.9855589 1.0798456 0.76723966
b: 9.9408274 29.234168 0.57566882 0.15176128
c: 0
stol: 1.70 # d_min: 0.29, max_error: 0.0097
a: 2.9995494 2.2558389 1.7278842
b: 23.27268 7.4543309 0.31622488
c: 0
stol: 0.50 # d_min: 1.00, max_error: 0.0071
a: 4.0103186 2.9643631
b: 19.971888 1.7558905
c: 0
stol: 0.17 # d_min: 2.94, max_error: 0.0088
a: 6.9671502
b: 11.43723
c: 0
```

In this example the 1-term approximation is valid up to d_{min} = 2.94 Å, the 2-term approximation up to d_{min} = 1.00 Å, etc. `max_error` is the maximum relative error over the entire resolution range from sin(theta)/lambda = 0 $Å^{-1}$ up to the `stol` shown above. The limits for the 1-term approximations for the most prevalent elements in protein structures are:

```
H: stol: 0.17 # d_min: 2.94, max_error: 0.0096
C: stol: 0.15 # d_min: 3.33, max_error: 0.0098
N: stol: 0.17 # d_min: 2.94, max_error: 0.0088
O: stol: 0.19 # d_min: 2.63, max_error: 0.0082
S: stol: 0.15 # d_min: 3.33, max_error: 0.0093
```

This table shows that 1-term approximations are fully sufficient at a resolution of 3.5 Å. The limits for the 2-term approximations are:

```
H: stol: 0.42 # d_min: 1.19, max_error: 0.0064
C: stol: 0.50 # d_min: 1.00, max_error: 0.0100
N: stol: 0.50 # d_min: 1.00, max_error: 0.0071
O: stol: 0.55 # d_min: 0.91, max_error: 0.0072
S: stol: 0.55 # d_min: 0.91, max_error: 0.0088
```

This table shows that the vast majority of protein structures can be refined using just 2-Gaussian approximations.

The library of N-Gaussian approximations is automatically used if structure factors are computed via the high-level interface (e.g. `f_calc_monoclinic_c` in the first section above). The given d_{min} is used to dynamically select the approximation with the least number of terms but a maximum relative error of less than 1%. To give an example, the savings in CPU time for sampling the electron density of the structure with the PDB access code 1HGE are:

```
Number of scatterers: 15549
Number of reflections:
  d_min=4: 39137
  d_min=3: 92401
  d_min=2: 310603
  d_min=1: 2474361

dynamic/4-plus-1 using the exp function:
  d_min=4: 0.42 s / 1.28 s = 0.33
  d_min=3: 1.09 s / 2.70 s = 0.40
  d_min=2: 2.61 s / 5.67 s = 0.46
  d_min=1: 20.75 s / 40.22 s = 0.52

dynamic/4-plus-1 using an exp table:
  d_min=4: 0.38 s / 0.91 s = 0.41
  d_min=3: 0.84 s / 1.86 s = 0.45
  d_min=2: 1.95 s / 3.92 s = 0.50
  d_min=1: 14.08 s / 27.22 s = 0.52
```

These times were collected on a 2.8 GHz Xeon computer running Windows 2000. Strictly speaking the absolute times are meaningless without the exact definition of all parameters used in the sampling of the electron density (which is beyond the scope of this article), but the parameters used are typical and the ratios shown above are roughly invariant under a change of parameters. As a rule of thumb, the sampling procedure is about twice as fast if the dynamically selected N-Gaussian approximations are used instead of the 4-plus-1 approximations from the International Tables. The results shown above can be reproduced by running the `electron_density_sampling.py` script in the directory `cctbx/cctbx/development`.

The library of N-Gaussian approximations can also be accessed at a lower level. For example:

```
from cctbx.eltbx import xray_scattering
for n_terms in [1,2]:
  table_entry = xray_scattering.n_gaussian_table_entry("C", n_terms)
  print "d_min:", table_entry.d_min()
  print "max_relative_error:", table_entry.max_relative_error()
```

```
  n_gaussian = table_entry.gaussian()
  n_gaussian.show()
  print
```

Output:

```
d_min: 3.33333333333
max_relative_error: 0.00975980236455
a: 5.9679281
b: 14.895768
c: 0

d_min: 1.0
max_relative_error: 0.00995574533403
a: 3.5435555 2.4257967
b: 25.623984 1.5036446
c: 0
```

The raw table can be found in the file `cctbx/eltbx/xray_scattering/n_gaussian_raw.cpp`.

5: Fast structure-factor gradients

For the refinement of macromolecular structures it is essential that the structure-factor and structure-factor-gradient calculations are carried out using a Fast Fourier Transform (FFT) based method. Bricogne (2001) uses the wording "spectacular increases in speed" (p. 91) in connection with such methods. The first to introduce FFT gradient calculations into crystallography was Agarwal (1978). Agarwal's original method requires the computation of a FFT for each type of refinable parameter, but "Lifchitz's reformulation" (Bricogne, 2001) removes this requirement and the Agarwal-Lifchitz procedure is routinely used in programs like TNT (Tronrud et al., 1987), CNS (Brunger, 1989) and REFMAC (Murshudov et al., 1997). The same procedure is now also available in the `cctbx.xray.structure_factors` package. Gradients w.r.t the following parameters are supported:

- coordinates
- isotropic displacement parameters ("B-factors")
- anisotropic displacement parameters
- occupancy factors
- dispersive coefficients ("f-prime")
- anomalous coefficients ("f-double-prime")

The procedure is optimized for structures both with and without anomalous scatterers. Gradients w.r.t any combination of parameters may be computed (e.g. only coordinates, or coordinates and displacement parameters simultaneously, etc.). Isotropic and anisotropic displacement parameters may be arbitrarily mixed. The memory required to store the gradients is allocated dynamically if needed. Of course, the procedure is fully scriptable from Python to maximize reusability and flexibility.

The `cctbx.xray.minimization` module is useful as a starting point to explore how the gradient calculations are used. The core calculations are implemented in C++ and can be found in the file `cctbx/include/cctbx/xray/fast_gradients.h`. Exploiting the abstraction facilities provided by C++, the code for the computation of the gradients makes heavy use of the code for the FFT structure factor calculations which is located in the same directory. As a consequence the C++ source code specific to the gradient calculations is only 620 lines long.

The cctbx also includes the much simpler direct-summation procedure for computing structure-factor gradients. Both the FFT-based procedure and the direct-summation procedure are accessible through a uniform Python interface. The desired method is selected via `algorithm="direct"` or `algorithm="fft"` as shown for the calculation of `f_calc_monoclinic_c` in the example in the first section. If `algorithm` is not specified, a heuristic procedure determines automatically which method to use.

6: Universal reflection file reader

To support the substructure determination procedure in Phenix (Grosse-Kunstleve & Adams, 2003) we have implemented a reflection file reader that automatically detects and processes these formats:

```
- merged scalepack files
- unmerged scalepack files
- CCP4 MTZ files with merged data
- CCP4 MTZ files with unmerged data (but merged files are preferred)
- d*trek .ref files
- XDS_ASCII files with merged data
- CNS reflection files
- SHELX reflection files
```

Using this reader is extremely simple:

```
from iotbx import reflection_file_reader
reflection_file = reflection_file_reader.any_reflection_file(
  file_name="gere_MAD.mtz")
miller_arrays = reflection_file.as_miller_arrays()
for miller_array in miller_arrays:
  miller_array.show_summary()
```

A fragment from the output:

```
Miller array info: gere_MAD.mtz:F(+)SEpeak,SIGF(+)SEpeak,F(-)SEpeak,SIGF(-
)SEpeak
Observation type: xray.amplitude
Type of data: double, size=23010
Type of sigmas: double, size=23010
Number of Miller indices: 23010
Anomalous flag: 1
Unit cell: (108.742, 61.679, 71.652, 90, 97.151, 90)
Space group: C 1 2 1 (No. 5)
```

Note that in this case the reader automatically combines four data columns of the input MTZ file into one object. The low-level processing of the reflection data is handled automatically as much as possible using all available information. However, sometimes the reader needs a little more help. For example when reading CNS reflection files the unit cell and space group are not available. Here is how the information can be supplied externally:

```
from iotbx import reflection_file_reader
from cctbx import crystal
reflection_file = reflection_file_reader.any_reflection_file(
  file_name="scale.hkl")
crystal_symmetry = crystal.symmetry(
  unit_cell=(108.742, 61.679, 71.652, 90, 97.151, 90),
  space_group_symbol="C2")
miller_arrays = reflection_file.as_miller_arrays(
  crystal_symmetry=crystal_symmetry)
for miller_array in miller_arrays:
  miller_array.show_summary()
```

Sometimes certain space groups are used as placeholders during data processing until the true space group is known (e.g. P222 instead of P212121). The method above can also be used to replace the information found in the reflection file with the correct symmetry information.

To minimize the need for manual entering of symmetry information, the iotbx provides a facility for extracting just the unit cell parameters and the space group from all reflection file formats shown above (all that actually contain symmetry information) and some other file formats such is CNS input files, SHELX .ins files and SOLVE input files. As before, the format is detected and processed automatically:

```
from iotbx import crystal_symmetry_from_any
crystal_symmetry = crystal_symmetry_from_any.extract_from(
  file_name="shelx.ins")
crystal_symmetry.show_summary()
```

The `crystal_symmetry` object obtained in the example can be used as an argument to the `as_miller_arrays()` method in the previous example.

7: Notes on supported platforms

In regular intervals the cctbx is automatically built on a large number of platforms and easy-to-install binary distributions are posted at http://cci.lbl.gov/cctbx_build/ . Currently, 15 different binary bundles are available (various versions of Windows, Linux, Mac OS X, IRIX, Tru64 Unix combined with different versions of Python). Not all of them are strictly needed. For example the binary bundles for Windows 2000 will also work under Windows XP. However, our approach ensures that the build procedure itself works in all these different environments.

In the last newsletter we announced limited support for Mac OS X. Fortunately the situation has improved significantly since then. The new compilers provided by Apple can now be used under both OS 10.2 and OS 10.3 and the C++ optimizers are fully functional. Python 2.3, which is required for running the cctbx under OS 10, was officially released in July 2003 and is included by default in all OS 10.3 installations.

In addition to the platforms posted at our web site, we have successfully tested the cctbx in the following environments:

- RedHat 8, Intel C++ 7.1.006, native Python (2.2.1)
- RedHat 8, Intel C++ 8.0.058, native Python (2.2.1)
- SunOS 5.9, GCC 3.3.1, Python 2.3 installed automatically from source code bundle
- SuSE SLES-8.1 (AMD64 Opteron), native gcc (3.2.2), native Python (2.2.1)

Currently we are not aware of any major platform where the cctbx could not be used.

8: Acknowledgments

We would like to thank Michael O'Keefe for giving us access to an electronic version of the primary scattering factor data. Wolfgang Kabsch kindly answered our questions regarding the XDS_ASCII reflection file format. We are grateful for the permission to use the CMTZ library provided by CCP4. Our work was funded in part by the US Department of Energy under Contract No. DE-AC03-76SF00098. We gratefully acknowledge the financial support of NIH/NIGMS.

9: References

Agarwal, R.C. (1978). Acta Cryst. A34, 791-809.
Bricogne, G. (2001). In: International Tables for Crystallography, Volume B.
Brunger, A.T. (1989). Acta Cryst. A45, 42-50.
Giacovazzo, C. (1992). Editor. Fundamentals of Crystallography. IUCr/Oxford University Press.
Grosse-Kunstleve, R.W. (1999). Acta Cryst. A55, 383-395.
Grosse-Kunstleve, R.W., Adams, P.D. (2002). J. Appl. Cryst. 35, 477-480.
Grosse-Kunstleve, R.W., Sauter, N.K., Adams, P.D. (2004). Acta Cryst. A60, 1-6.
Gruber, B. (1973). Acta Cryst. A29, 433-440.
Krivy, I. & Gruber, B. (1976). Acta Cryst. A32, 297-298.
Le Page, Y. (1982). J. Appl. Cryst. 15, 255-259.
Murshudov, G.N., Vagin, A.A. & Dodson, E.J. (1997). Acta Cryst. D53, 240-253.
Sauter, N.K., Grosse-Kunstleve, R.W., Adams, P.D. (2004). Submitted to J. Appl. Cryst.
Ten Eyck, L.F. (1977). Acta Cryst. A33, 486-492.
Tronrud, D.E., Ten Eyck, L.F., Matthews, B.W. (1987). Acta Cryst. A43, 489-501.
Waasmeier & Kirfel (1995). Acta Cryst. A51,416-431.