CCTBX for Diffraction Calculations 2

Graeme Winter, Diamond Light Source

November 27, 2012

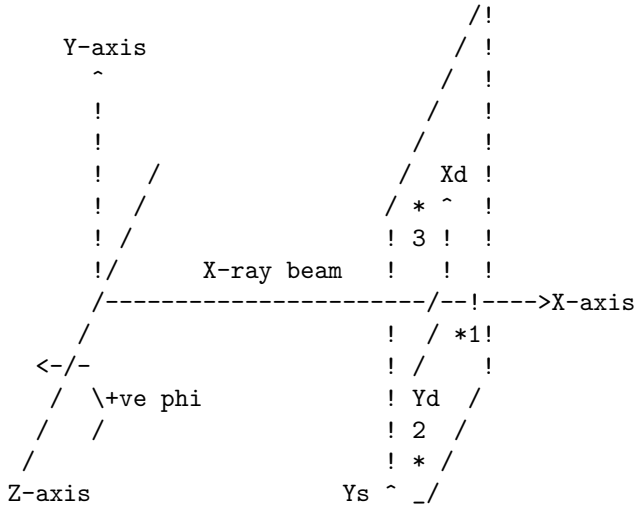
1 Introduction

Where the previous example showed "toy" diffraction calculations focussing on a diffraction image, here a more useful calculation will be performed: the assessment of the geometric completeness of a data set given the orientation matrix (from Mosflm, say) start and end spindle positions, the resolution limit and the sample symmetry. This shows some of the more useful capabilities of the CCTBX toolbox for unit cell and symmetry handling.

This example will also include much more in the way of input checking to make it more robust and to demonstrate some calculations.

2 Getting Started

The pre-requisites here are: installed and working cctbx (presumed, if you survived number 1) and a Mosflm orientation matrix. One is included for your convenience. It will be assumed in here that the experimental geometry is consistent with the operation of Mosflm... from http://www.mrc-lmb.cam.ac.uk/harry/mosflm/mosflm_user_guide.html:



```
Rotation                    ! /| Xs
 axis                       !/
                            0
```

Be warned - this is an equivalent but incompatible description to the coordinate frame used within *some* of the calculations used in CCTBX - which adopt a coordinate frame from Rossmann 1979. This shows why the use of e.g. imgCIF is valuable... the coordinate conversion matrix from Mosflm to this frame is:

$$S = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \tag{1}$$

3 Reading the Matrix File

Much of the information needed is encoded in the Mosflm matrix file, viz:

```
  0.00586721 -0.00470834  0.01151601
 -0.00697003 -0.00098276  0.01727817
 -0.00857521 -0.01155012 -0.00616458
       0.000       0.000       0.000
   0.4689419  -0.7052791   0.5316717
  -0.5570862   0.2309343   0.7976994
  -0.6853819  -0.6702617  -0.2846065
     90.3943     90.3943     45.2195     90.0000     90.0000    120.0000
       0.000       0.000       0.000
SYMM P3
```

The last record, describing the lattice symmetry, was only added more recently and will hence be ignored for this example. The file contains: $\frac{1}{\lambda}UB$, misorientation angles, U, unit cell constants and misorientation angles (again.) The orientation matrices are given in reciprocal space, such that

$$\underline{x} = R(\phi)UB\underline{h}, \tag{2}$$

where $\underline{h}$ is the Miller index of a reflection and $\underline{x}$ the reciprocal space position. The $\frac{1}{\lambda}$ term is included in there for reasons of efficiency, to scale the Ewald sphere. One side effect of Mosflm once running on a PDP 11.

In reading this matrix file, what we actually want is the $A* = UB$ matrix, so the wavelength is needed to determine the scaling:

```
def mosflm_to_rossmann(a_matrix):
    '''Convert A* matrix in Mosflm convention to Rossmann convention.'''

    from scitbx import matrix
```

```
    S = matrix.sqr((0, 0, 1, 1, 0, 0, 0, 1, 0))

    return S * a_matrix

def parse_mosflm_matrix(matrix_file):
    '''Parse the mosflm matrix file to get: U, B, wavelength, unit cell,
    returning A* = U B in the Rossmann coordinate frame.'''

    from scitbx import matrix
    from cctbx import uctbx

    tokens = open(matrix_file, 'r').read(512).replace('-', ' -').split()

    assert(len(tokens) in (30, 32))

    lUB = matrix.sqr(map(float, tokens[0:9]))
    U = matrix.sqr(map(float, tokens[12:21]))

    uc = uctbx.unit_cell(map(float, tokens[21:27]))

    # derive the wavelength

    A = lUB.inverse().elems
    a = matrix.col(A[0:3])
    b = matrix.col(A[3:6])
    c = matrix.col(A[6:9])

    wavelength = (uc.parameters()[0] / a.length() +
                  uc.parameters()[1] / b.length() +
                  uc.parameters()[2] / c.length()) / 3

    # now verify that we do not reproduce the lUB matrix - this is the
    # PDB definition...

    B = matrix.sqr(uc.fractionalization_matrix())

    # ... and this is the Mosflm version (implied)

    mB = (1.0 / wavelength) * U.inverse() * lUB

    # not important - let's just return A* (i.e. a*, b*, c*)

    return mosflm_to_rossmann((1.0 / wavelength) * lUB), uc, wavelength
```

4 Generating Reflections

Generating the full list of reflections is relatively straightforward, essentially just computing the cube of all possible reflections which certainly exceeds the resolution limit we have, then trimming out the reflections which turn out to be outside the limit:

```
def generate_reflection_indices(uc, dmin):
    '''Generate the possible reflection indices from a unit cell object: N.B.
    that these are not symmetry reduced.'''

    maxh, maxk, maxl = uc.max_miller_indices(dmin)

    indices = []

    for h in range(-maxh, maxh + 1):
        for k in range(-maxk, maxk + 1):
            for l in range(-maxl, maxl + 1):

                if h == 0 and k == 0 and l == 0:
                    continue

                if uc.d((h, k, l)) < dmin:
                    continue

                indices.append((h, k, l))

    return indices
```

Strictly speaking, to be useful this should also take in to account absent reflections due to the crystal symmetry. For e.g. $P2_12_12_1$ this is not so important, but for centred lattices e.g. I23 half of the reflections will be absent. The observant may remark that this was the case for the first introduction document - the lattice was centred, so in fact the calculation of reciprocal space distance was incorrect! Removing the absent reflections is straightforward, given the spacegroup number (N.B. it is presumed that the standard setting is used!)

```
def remove_absent_indices(indices, space_group_number):
    '''From the given list of indices, remove those reflections which should
    be systematic absences according to the given space group.'''

    from cctbx.sgtbx import space_group, space_group_symbols

    sg = space_group(space_group_symbols(space_group_number).hall())

    present = []
```

```
    for hkl in indices:
        if not sg.is_sys_absent(hkl):
            present.append(hkl)

    return present
```

Given that we now have the full list of reflections, all that is now needed is to determine the ϕ settings where they will be observed, removing the reflections which are in the blind region and hence cannot be observed:

```
def generate_intersection_angles(a_matrix, dmin, wavelength, indices):
    '''From an A matrix following the Mosflm convention and the list of
    indices, return a list of phi, (h, k, l) where (typically) there will be
    two records corresponding to each h, k, l.'''

    from rstbx.diffraction import rotation_angles
    from scitbx import matrix
    import math

    ra = rotation_angles(dmin, a_matrix, wavelength, matrix.col((0, 1, 0)))

    phi_hkl = []

    r2d = 180.0 / math.pi

    for i in indices:
        if ra(i):
            phis = ra.get_intersection_angles()
            phi_hkl.append((phis[0] * r2d % 360, i))
            phi_hkl.append((phis[1] * r2d % 360, i))

    return phi_hkl
```

5 Simulation and Symmetry

So far we have the full set of observable reflections, with their original indices. For this calculation the reduced symmetry is more interesting, as this corresponds to the unique reflections we're interested in. So - first step is to determine the reflections observed between two ϕ positions, then the second is to reduce this to the asymmetric unit to determine the completeness.

Selecting the refections observed is trivial, given the list of phi, hkl:

```
def select_reflections(phi0, phi1, phi_hkl):
    '''Select reflections in range phi0 to phi1 inclusive.'''

    return [ph[1] for ph in phi_hkl if (ph[0] >= phi0 and ph[0] <= phi1)]
```

This is also a cute example of list comprehensions. This would be several lines of code in most languages. Reducing the observations to the asymmetric unit is less straightforward, as this uses a cctbx.miller array (a flex array.)

```
def reduce_reflections_to_asu(space_group_number, indices):
    '''Reduce reflection indices to asymmetric unit.'''

    from cctbx.sgtbx import space_group, space_group_symbols
    from cctbx.array_family import flex
    from cctbx.miller import map_to_asu

    sg = space_group(space_group_symbols(space_group_number).hall())

    miller = flex.miller_index(indices)

    map_to_asu(sg.type(), False, miller)

    return [hkl for hkl in miller]
```

Finally, all that is needed is a little glue:

```
def strategy(a_matrix, dmin, symmetry):
    '''Compute which 45 degree wedge gives the best completeness of data.'''

    a_star, uc, wavelength = parse_mosflm_matrix(a_matrix)
    indices = generate_reflection_indices(uc, dmin)
    present = remove_absent_indices(indices, symmetry)
    n_unique = len(set(reduce_reflections_to_asu(symmetry, indices)))
    observable = generate_intersection_angles(a_star, dmin, wavelength,
                                              present)
    dphi = 45.0
    for phi0 in 0.0, 45.0, 90.0, 135.0, 180.0, 225.0, 270.0, 315.0:
        observed = select_reflections(phi0, phi0 + dphi, observable)
        n_unique_wedge = len(set(reduce_reflections_to_asu(symmetry,
                                                           observed)))
        print '%6.2f %6.4f' % (phi0, float(n_unique_wedge) / float(n_unique))

    return
```

Which gives, for the example included:

```
graeme$ cctbx.python cctbx_introduction_2.py cctbx_introduction_2.mat 1.5 178
  0.00 0.7971
 45.00 0.9901
 90.00 0.9734
135.00 0.9838
180.00 0.7971
225.00 0.9901
270.00 0.9734
315.00 0.9838
```