`direct_methods_light.py`

Overview

The `direct_methods_light.py` Python example is designed to read two CIF files from the Acta Crystallographica Section C web page as inputs:

- required: reduced X-ray diffraction data: vj1132Isup2.hkl
- optional: the corresponding refined structure: vj1132sup1.cif

The `iotbx.acta_c` module is used to convert the diffraction data to a `cctbx.miller.array` object; this is supported by James Hester's PyCifRW library. Normalized structure factors ("E-values") are computed, and the largest E-values are selected for phase recycling with the Tangent Formula.

The Miller indices of the largest E-values are used to construct index triplets `h = k + h-k` with the `cctbx.dmtbx.triplet_generator`. The Tangent Formula is repeatedly applied to recycle a phase set, starting from random phases. After a given number of cycles, the resulting phase set is combined with the E-values. The resulting Fourier coefficients are used in a Fast Fourier Transformation to obtain an "E-map". The E-map is normalized and a symmetry-aware peak search is carried out; i.e. the resulting peak list is unique under symmetry.

If the CIF file with the coordinates is given, it is first used to compute structure factors `f_calc`. The correlation with the diffraction data is shown. Next, the CIF coordinates are compared with the E-map peak list using the Euclidean Model Matching procedure (Emma) implemented in the `cctbx`. The resulting output can be used to quickly judge if the structure was solved with the simple Tangent Formula recycling procedure.

[Complete example script] [Example output] [cctbx downloads] [cctbx front page] [Python tutorial]

Recommended reading

The unit_cell_refinement.py example introduces some important basis concepts.

Processing of `vj1132Isup2.hkl`

The first step is to get hold of the file name with the reduced diffraction data. The file name has to be specified as the first command-line argument:

```
iotbx.python direct_methods_light.py vj1132Isup2.hkl
```

In the example script, the command line argument is extracted from the `sys.argv` list provided by Python's standard `sys` module (libtbx.help sys):

```
import sys
reflection_file_name = sys.argv[1]
```

The `reflection_file_name` is used in the call of the `cif_as_miller_array()` function provided by the iotbx.acta_c module:

```
from iotbx import acta_c
miller_array =
acta_c.cif_as_miller_array(file_name=reflection_file_name)
miller_array.show_comprehensive_summary()
```

The `iotbx.acta_c` module makes use of the PyCifRW library to read CIF files. `PyCifRW` returns the CIF data items as plain strings. The `cif_as_miller_array()` function extracts the appropriate strings from the object tree returned by `PyCifRW` to construct an instance of the `cctbx.miller.array` class, which is one of the central types in the cctbx source tree. The `miller.array` class has a very large number of methods (libtbx.help cctbx.miller.array), e.g. the `show_comprehensive_summary()` method used above to obtain this output:

```
Miller array info: vj1132Isup2.hkl:F_meas,F_sigma
Observation type: xray.amplitude
Type of data: double, size=422
Type of sigmas: double, size=422
Number of Miller indices: 422
Anomalous flag: False
Unit cell: (12.0263, 6.0321, 5.8293, 90, 90, 90)
Space group: P n a 21 (No. 33)
Systematic absences: 0
Centric reflections: 83
Resolution range: 6.01315 0.83382
Completeness in resolution range: 1
Completeness with d_max=infinity: 1
```

We can see that the `miller_array` contains data and sigmas, both of type double. It also contains Miller indices, an anomalous flag, a unit cell and a space_group object. These are the primary data members. The observation type is an optional annotation which is typically added by the creator of the object, in this case the `cif_as_miller_array()` function. The information in the last five lines of the output is calculated on the fly based on the primary information and discarded after the `show_comprehensive_summary()` call is completed.

Two other `cctbx.miller.array` methods are used in the following statements in the script:

```
if (miller_array.is_xray_intensity_array()):
  miller_array = miller_array.f_sq_as_f()
```

If the `miller_array` is an intensity array, it is converted to an amplitude array. The `f_sq_as_f()` method ("sq" is short for "square") returns a new `cctbx.miller.array` instance. At some point during the evaluation of the statement the old and the new instance are both present in memory. However, after the `miller_array = miller_array.f_sq_as_f()` assignment is completed, the old `miller_array` instance is deleted automatically by the Python interpreter since there is no longer a reference to it, and the corresponding memory is released immediately.

It is very important to understand that most `miller.array` methods do not modify the instance in place, but return new objects. The importance of minimizing the number of methods performing in-place manipulations cannot be overstated. In large systems, in-place manipulations quickly lead to unforeseen side-effects and eventually frustrating, time-consuming debugging sessions. It is much safer to create new objects. In most cases the dynamic memory allocation overhead associated with object creation and deletion is negligible compared to the runtime for the actual core algorithms. It is like putting on seat belts before a long trip with the car. The 10 seconds it takes to buckle up are nothing compared to the hours the seat belts protect you.

[Complete example script] [Example output] [cctbx downloads] [cctbx front page] [Python tutorial]

Computation of E-values

Having said all the things about the dangers of in-place operations, the next statement in the script happens to be just that:

```
miller_array.setup_binner(auto_binning=True)
```

However, the operation does not affect the primary data members of the `miller_array` (unit cell, space group, indices, data, sigmas). The `setup_binner()` call initializes or re-initializes a binner object to be used in subsequent calculations. The `binner` object is understood to be a secondary data member and its state only affects the results of future calculations. In situations like this in-place operations are perfectly reasonable.

The result of the `setup_binner()` call is shown with this statement:

```
miller_array.binner().show_summary()
```

The output is:

```
unused:        - 6.0133 [ 0/0 ]
bin  1: 6.0133 - 1.6574 [57/57]
bin  2: 1.6574 - 1.3201 [55/55]
bin  3: 1.3201 - 1.1546 [55/55]
```

```
bin  4: 1.1546 - 1.0496 [45/45]
bin  5: 1.0496 - 0.9747 [55/55]
bin  6: 0.9747 - 0.9175 [55/55]
bin  7: 0.9175 - 0.8717 [48/48]
bin  8: 0.8717 - 0.8338 [52/52]
unused: 0.8338 -        [ 0/0 ]
```

This means we are ready to calculate quasi-normalized structure factors by computing `f_sq / <f_sq/epsilon>` in resolution bins:

```
all_e_values =
miller_array.quasi_normalize_structure_factors().sort(by_valu
e="data")
```

This statement performs two steps at once. First, the `quasi_normalize_structure_factors()` method creates a new `cctbx.miller.array` instance with the same unit cell, space group, anomalous flag and Miller indices as the input `miller_array`, but with a new data array containing the normalized structure factors. The `sort()` method is used immediately on this intermediate instance to sort the E-values by magnitude. By default, the data are sorted in descending order (largest first, smallest last). This is exactly what we want here. To convince yourself it is correct, insert `all_e_values.show_array()`.

[Complete example script] [Example output] [cctbx downloads] [cctbx front page] [Python tutorial]

Generation of triplets

In direct methods procedures it is typical to generate the `h = k + h-k` Miller index triplets only for the largest E-values. In the example script, the largest E-values are selected with this statement:

```
large_e_values = all_e_values.select(all_e_values.data() >
1.2)
```

Again, this statement combines several steps into one expression. First, we obtain access to the array of all E-values via `all_e_values.data()`. This array is a `flex.double` instance, which in turn has its own methods (libtbx.help cctbx.array_family.flex.double). One of the `flex.double` methods is the overloaded `>` operator; in the `libtbx.help` output look for `__gt__(...)`. This operator returns a `flex.bool` instance, an array with `bool` values, `True` if the corresponding E-value is greater than `1.2` and `False` otherwise. The `flex.bool` instance becomes the argument to the `select()` method of `cctbx.miller.array`, which finally returns the result of the whole statement. `large_e_values` is a new `cctbx.miller.array` instance with the same unit cell, space group and anomalous flag as `all_e_values`, but fewer indices and corresponding data. Of the 422 E-values only 111 are selected, as is shown by this `print` statement:

```
print "number of large_e_values:", large_e_values.size()
```

At this point all the information required to generate the triplets is available:

```
from cctbx import dmtbx
triplets = dmtbx.triplet_generator(large_e_values)
```

The `triplet_generator` is based on the cctbx::dmtbx::triplet_generator C++ class which uses a very fast algorithm to find the Miller index triplets (see the references near the top of triplet_generator.h). The `triplets` object manages all internal arrays automatically. It is not necessary to know very much about this object, but is is informative to print out the results of some of its methods, e.g.:

```
from cctbx.array_family import flex
print "triplets per reflection: min,max,mean: %d, %d, %.2f" %
(
  flex.min(triplets.n_relations()),
  flex.max(triplets.n_relations()),
  flex.mean(triplets.n_relations().as_double()))
print "total number of triplets:",
flex.sum(triplets.n_relations())
```

Here the general purpose `flex.min()`, `flex.max()`, `flex.mean()` and `flex.sum()` functions are used to obtain summary statistics of the number of triplet phase relations per Miller index. `triplets.n_relations()` returns a `flex.size_t()` array with unsigned integers corresponding to the ANSI C/C++ `size_t` type. However, the `flex.mean()` function is only defined for `flex.double` arrays. Therefore `n_relations()` has to be converted via `as_double()` before computing the mean.

[Complete example script] [Example output] [cctbx downloads] [cctbx front page] [Python tutorial]

Tangent Formula phase recycling

Starting with RANTAN, the predominant method for initiating Tangent Formula phase recycling is to generate random phases. In principle this is very easy. E.g. the `flex.random_double()` function could be used:

```
random_phases_rad =
flex.random_double(size=large_e_values.size())-0.5
random_phases_rad *= 2*math.pi
```

However, centric reflections need special attention since the phase angles are restricted to two values, `phi` and `phi+180`, where `phi` depends on the space group and the Miller index. A proper treatment of the phase restrictions is implemented in the `random_phases_compatible_with_phase_restrictions()` method of `cctbx.miller.array`:

```
input_phases =
large_e_values.random_phases_compatible_with_phase_restrictio
ns()
```

The underlying random number generator is seeded with the system time, therefore the `input_phases` will be different each time the example script is run.

The Tangent Formula recycling loop has this simple design:

```
result = input
for i in xrange(10):
  result = function(result)
```

In the example script the actual corresponding code is:

```
tangent_formula_phases = input_phases.data()
for i in xrange(10):
  tangent_formula_phases = triplets.apply_tangent_formula(
    amplitudes=large_e_values.data(),
    phases_rad=tangent_formula_phases,
    selection_fixed=None,
    use_fixed_only=False,
    reuse_results=True)
```

In this case `function()` is the `apply_tangent_formula()` method of the `triplet` object returned by the `cctbx.dmtbx.triplet_generator()` call. The function call looks more complicated than the simplified version because it requires a number of additional arguments customizing the recycling protocol. It may be interesting to try different settings as an exercise. See cctbx::dmtbx::triplet_generator for details.

[Complete example script] [Example output] [cctbx downloads] [cctbx front page] [Python tutorial]

E-map calculation

Another `cctbx.miller.array` method is used to combine the `large_e_values` with the `tangent_formula_phases` obtained through the recycling procedure:

```
e_map_coeff = large_e_values.phase_transfer(
  phase_source=tangent_formula_phases)
```

The `phase_transfer()` returns a `flex.complex_double` array of Fourier coefficients. A general proper treatment of phase restrictions is automatically included, although in this case it just corrects for rounding errors.

Given the Fourier coefficients, an E-map could be obtained simply via `e_map_coeff.fft_map()`. However, we have to think ahead a little to address a

technical detail. A subsequent step will be a peak search in the E-map. For this we will use a peak search algorithm implemented in the `cctbx.maptbx` module, which imposes certain space-group specific restrictions on the gridding of the map. For all symmetry operations of the given space group, each grid point must be mapped exactly onto another grid point. E.g. in space group P222 the gridding must be a multiple of 2 in all three dimensions. To inform the `fft_map()` method about these requirements we use:

```
from cctbx import maptbx
e_map =
e_map_coeff.fft_map(symmetry_flags=maptbx.use_space_group_sym
metry)
```

The resulting e_map is normalized by first determining the mean and standard deviation ("sigma") of all values in the map, and then dividing by the standard deviation:

```
e_map.apply_sigma_scaling()
```

Since maps tend to be large and short-lived, this is implemented as an in-place operation to maximize runtime efficiency. The `statistics()` method of the `e_map` object is used to quickly print a small summary:

```
e_map.statistics().show_summary(prefix="e_map ")
```

This output is of the form:

```
e_map max 11.9224
e_map min -2.68763
e_map mean -2.06139e-17
e_map sigma 1
```

Due to differences in the seed for the random number generator, the `max` and `min` will be different each time the example script is run. However, the `mean` is always very close to `0` since the Fourier coefficient with index (0,0,0) is zero, and `sigma` is always very close to `1` due to the prior use of `apply_sigma_scaling()`; small deviations are the accumulated result of floating-point rounding errors.

[Complete example script] [Example output] [cctbx downloads] [cctbx front page] [Python tutorial]

Peak search

Given the normalized `e_map`, the peak search is initiated with this statement:

```
```

```
peak_search =
e_map.peak_search(parameters=maptbx.peak_search_parameters(
  min_distance_sym_equiv=1.2))
```

The only purpose of the `maptbx.peak_search_parameters` class is to group the fairly large number of parameters (libtbx.help cctbx.maptbx.peak_search_parameters). This approach greatly simplifies the argument list of functions and methods involving peak search parameters. It also accelerates experimentation during the algorithm development process. Parameters can be added, deleted or renamed without having to modify all the functions and methods connected to the peak search.

In the example, the minimum distance between symmetry-related sites is set to 1.2 A. This instructs the peak search algorithm to perform a cluster analysis. The underlying distance calculations are performed for symmetry-related pairs and pairs of peaks unique under symmetry ("cross peaks"). If the `min_cross_distance` peak search parameter is not specified explicitly (as in the example), it is assumed to be equal to the `min_distance_sym_equiv` parameter.

The cluster analysis begins by adding the largest peak in the map as the first entry to the peak list. All peaks in a radius of 1.2 A around this peak are eliminated. The largest of the remaining peaks is added to the peak list, and all peaks in a radius of 1.2 A around this peak are eliminated etc., until all peaks in the map are considered or a predefined limit is reached. The example uses:

```
peaks = peak_search.all(max_clusters=10)
```

to obtain up to 10 peaks in this way. The peaks are printed in this `for` loop:

```
for site,height in zip(peaks.sites(), peaks.heights()):
  print "  (%9.6f, %9.6f, %9.6f)" % site, "%10.3f" % height
```

See the unit_cell_refinement.py example for comments regarding the standard Python `zip()` function. The Python tutorial section on Fancier Output Formatting is useful to learn more about the `print` statement.

[Complete example script] [Example output] [cctbx downloads] [cctbx front page] [Python tutorial]

Processing of `vj1132sup1.cif`

Since it is not easy to quickly judge from the peak list if the structure was solved, the `vj1132sup1.cif` file is used for verification purposes. It is processed in very much the same way as the `vj1132Isup2.hkl` file before:

```
coordinate_file_name = sys.argv[2]
```

```
xray_structure = acta_c.cif_as_xray_structure(
  file_name=coordinate_file_name,
  data_block_name="I")
```

The `cif_as_xray_structure()` call requires the name of the CIF data block name in addition to the file name. This is because Acta C coordinate CIF files may contain multiple structures (and because the `iotbx.acta_c` module is not sophisticated enough to simply "do the right thing" if the CIF file contains only one structure). The result is an instance of another central type in the cctbx source tree, `cctbx.xray.structure`. The `xray_structure` object is best understood by asking it for a summary:

```
xray_structure.show_summary()
```

The output is:

```
Number of scatterers: 13
At special positions: 0
Unit cell: (12.0263, 6.0321, 5.829, 90, 90, 90)
Space group: P n a 21 (No. 33)
```

We can also ask it for a list of scatterers:

```
xray_structure.show_scatterers()
```

The result is:

```
Label, Scattering, Multiplicity, Coordinates, Occupancy, Uiso
O1   O      4 ( 0.5896  0.4862  0.6045) 1.00 0.0356
O2   O      4 ( 0.6861  0.2009  0.7409) 1.00 0.0328
C2   C      4 ( 0.6636  0.2231  0.3380) 1.00 0.0224
H2   H      4 ( 0.7419  0.1804  0.3237) 1.00 0.0270
C1   C      4 ( 0.6443  0.3133  0.5818) 1.00 0.0222
C3   C      4 ( 0.5923  0.0216  0.2916) 1.00 0.0347
H3A  H      4 ( 0.6060 -0.0308  0.1387) 1.00 0.0520
H3B  H      4 ( 0.5153  0.0605  0.3069) 1.00 0.0520
H3C  H      4 ( 0.6104 -0.0930  0.3997) 1.00 0.0520
N1   N      4 ( 0.6395  0.3976  0.1659) 1.00 0.0258
H1A  H      4 ( 0.6815  0.5160  0.1942) 1.00 0.0390
H1B  H      4 ( 0.5680  0.4351  0.1741) 1.00 0.0390
H1C  H      4 ( 0.6544  0.3463  0.0261) 1.00 0.0390
```

Instead of writing:

```
xray_structure.show_summary()
xray_structure.show_scatterers()
```

we can also write:

```
xray_structure.show_summary().show_scatterers()
```

This approach is called "chaining". The trick is in fact very simple:

```
class structure:

  def show_summary(self):
    print "something"
    return self

  def show_scatterers():
    print "more"
    return self
```

Simply returning `self` enables chaining.

[Complete example script] [Example output] [cctbx downloads] [cctbx front page] [Python tutorial]

Correlation of F-obs and F-calc

Given the amplitudes F-obs as `miller_array` and the refined coordinates as `xray_structure`, F-calc amplitudes are computed with this statement:

```
f_calc = abs(miller_array.structure_factors_from_scatterers(
  xray_structure=xray_structure,
  algorithm="direct").f_calc())
```

This expression can be broken down into three steps. The first step is:

```
miller_array.structure_factors_from_scatterers(
  xray_structure=xray_structure,
  algorithm="direct")
```

This step performs the structure factor calculation using a direct summation algorithm (as opposed to a FFT algorithm). The result is an object with information about the details of the calculation, e.g. timings, or memory requirements if the FFT algorithm is used. If the details are not needed, they can be discarded immediately by extracting only the item of interest. In this case we use the `f_calc()` method to obtain a `cctbx.miller.array` instance with the calculated structure factors, stored in a `flex.complex_double` array. The outermost `abs()` function calls the `__abs__()` method of `cctbx.miller.array` which returns another new `cctbx.miller.array` instance with the structure factor amplitudes, stored in a `flex.double` array.

The correlation of F-obs and F-calc is computed with this statement:

```
correlation = flex.linear_correlation(f_calc.data(),
miller_array.data())
```

`flex.linear_correlation` is a C++ class (libtbx.help cctbx.array_family.flex.linear_correlation) which offers details about the correlation calculation, similar in idea to the result of the `structure_factors_from_scatterers()` above. We could discard all the details again, but the correlation coefficient could be undefined, e.g. if all values are zero, or if all values in one of the two input arrays are equal. We ensure the correlation is well defined via:

```
assert correlation.is_well_defined()
```

It is good practice to insert `assert` statements anywhere a certain assumption is made. The `cctbx` sources contain a large number of `assert` statements. They prove to be invaluable in flagging errors during algorithm development. In most situations errors are flagged close to the source. Time-consuming debugging sessions to backtrack from the point of a crash to the source of the problem are mostly avoided. Once we are sure the correlation is well defined, we can print the coefficient with confidence:

```
print "correlation of f_obs and f_calc: %.4f" %
correlation.coefficient()
```

It is amazingly high (0.9943) for the `vj1132` case.

[Complete example script] [Example output] [cctbx downloads] [cctbx front page] [Python tutorial]

Euclidean Model Matching (Emma)

Our goal is to match each peak site with a site in the `vj1132sup1.cif` file. To make this section less abstract, we start with an example result:

```
Match summary:
  Operator:
       rotation: {{-1, 0, 0}, {0, -1, 0}, {0, 0, -1}}
    translation: {0.5, 0, -0.136844}
  rms coordinate differences: 0.85
  Pairs: 8
    O1 peak01 0.710
    O2 peak09 1.000
    C2 peak03 0.896
    C1 peak02 0.662
    C3 peak04 0.954
    H3B peak07 0.619
    H3C peak08 0.979
    H1C peak06 0.900
  Singles model 1: 5
    H2   H3A   N1   H1A   H1B
```

```
  Singles model 2: 2
    peak00   peak05
```

This means, `peak01` corresponds to `O1` in the CIF file with a mismatch of 0.710 A, `peak09` corresponds to `O2` with a 1.000 A mismatch, etc. The match is obtained after inverting the hand of the `peaks` (the `rotation`) and adding `{0.5, 0, -0.136844}` to the coordinates (the `translation`). Some sites in the CIF files have no matching peaks (e.g. `N1`) and some peaks have no matching site in the CIF file (e.g. `peak00`). The overall RMS (root-mean-square) of the mismatches is 0.85. I.e. this match is not very good, except as a bad example.

In general, the comparison of two coordinate sets via pair-wise association of sites is quite complex due to the underlying symmetry of the search space. In addition to the space group symmetry, allowed origin shifts and a change of hand have to be taken into consideration. This is described in detail by Grosse-Kunstleve & Adams (2003).

The `cctbx.euclidean_model_matching` module is available for computing the pairs of matching sites. The search algorithm operates on specifically designed `cctbx.euclidean_model_matching.model` objects. I.e. we have to convert the `xray_structure` instance and the `peaks` to `cctbx.euclidean_model_matching.model` objects. Converting the `xray_structure` object is easy because the conversion is pre-defined as the `as_emma_model()` method:

```
reference_model = xray_structure.as_emma_model()
```

Converting the `peaks` object is not pre-defined. We have to do it the hard way. We start with assertions, just to be sure:

```
assert
reference_model.unit_cell().is_similar_to(e_map.unit_cell())
assert reference_model.space_group() == e_map.space_group()
```

This gives us the confidence to write:

```
from cctbx import euclidean_model_matching as emma
peak_model =
emma.model(special_position_settings=reference_model)
```

`special_position_settings` is a third central type in the cctbx source tree. It groups the unit cell, space group, and the `min_distance_sym_equiv` parameter which defines the tolerance for the determination of special positions. `emma.model` inherits from this type, therefore we can use the `reference_model` (which is an `emma.model` object) anywhere a `special_position_settings` object is required. This is more convenient than constructing a new `special_position_settings` objects from scratch.

At this stage the `peak_model` object does not contain any coordinates. We add them with this loop:

```
for i,site in enumerate(peaks.sites()):
  peak_model.add_position(emma.position(label="peak%02d" % i,
site=site))
```

The loop construct is a standard idiom (libtbx.help enumerate, Looping Techniques). `label="peak%02d" % i` creates a label of the form `peak000`, `peak001`, etc. The label and the `site` are used to construct an `emma.position` object which is finally added to the `peak_model` via the `add_position()` method.

The `emma.model_matches()` function computes a sorted list of possible matches:

```
matches = emma.model_matches(
  model1=reference_model,
  model2=peak_model,
  tolerance=1.,
  models_are_diffraction_index_equivalent=True)
```

The `tolerance` determines the maximum distance for a pair of a site in `model1` and a site in `model2`. The `models_are_diffraction_index_equivalent` parameter is used in the determination of the symmetry of the search space and has to do with indexing ambiguities. It is always safe to use `models_are_diffraction_index_equivalent=False`, but the search may be slower. If it is certain that the models are derived from the same diffraction data `models_are_diffraction_index_equivalent=True` can be used to reduce the runtime. In this case we are sure because the correlation between F-obs and F-calc is almost perfect.

[Complete example script] [Example output] [cctbx downloads] [cctbx front page] [Python tutorial]

Exercise (not very hard)

Run the example script several times. Each time the results will be different due to different random seeds. You will observe that Emma is often mislead by the hydrogens in the CIF file. To solve this problem, modify the script to remove the hydrogens from the reference model.

Hint: Find the implementation of `cctbx.xray.structure.as_emma_model()` (`cctbx/cctbx/xray/__init__.py`). Note that `scatterer` has a `scattering_type` attribute.

[Complete example script] [Example output] [cctbx downloads] [cctbx front page] [Python tutorial]

Exercise (harder)

Compute the correlation between `all_e_values` and a `cctbx.xray.structure` constructed with the top 6 peaks, using `"const"` as the `scattering_type`.

Hint: Study `iotbx/iotbx/acta_c.py` to see how the `xray_structure` is constructed from the CIF file. However, use `peak_structure = xray.structure(special_position_settings=xray_structure)`. Study `cctbx.xray.structure.__init__()` to see why this works.

[Complete example script] [Example output] [cctbx downloads] [cctbx front page] [Python tutorial]

Exercise (advanced)

Refactor the example script by splitting it up into functions and possibly classes. Compute random starting phases a given number of times and repeat the Tangent formula recycling for each. Avoid duplicate work. I.e. don't read the inputs multiple time, don't create the Emma reference model multiple times.

[Complete example script] [Example output] [cctbx downloads] [cctbx front page] [Python tutorial]

View document source. Generated on: 2005-09-28 19:27 UTC. Generated by Docutils from reStructuredText source.