`unit_cell_refinement.py`

Overview

The `unit_cell_refinement.py` Python example starts with a list of 2-theta peak positions derived from a powder diffraction experiment, and associated Miller indices obtained with an indexing program. The six unit cell parameters a,b,c,alpha,beta,gamma are refined to minimize the least-squares residual function:

```
sum over all Miller indices of (two_theta_obs -
two_theta_calc)**2
```

The refinement starts with the unit cell parameters a=10, b=10, c=10, alpha=90, beta=90, gamma=90. A general purpose quasi-Newton LBFGS minimizer is used. The LBFGS minimizer requires:

- an array of *parameters*, in this case the unit cell parameters.
- the *functional* given the current parameters; in this case the functional is the residual function above.
- the *gradients* (first derivatives) of the functional with respect to the parameters at the point defined by the current parameters.

To keep the example simple, the gradients are computed with the finite difference method.

Before and after the minimization a table comparing `two_theta_obs` and `two_theta_calc` is shown. The script terminates after showing the refined unit cell parameters.

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

Input data

For simplicity, the input data are included near the beginning of the example script:

```
two_theta_and_index_list = """\
  8.81   0  1  1
 12.23   0  0  2
 12.71   0  2  0
...
 31.56   0  1  5
 32.12   2  2  3
""".splitlines()
```

Here two steps are combined into one statement. The first step is to define a multi-line Python string using triple quotes. In the second step the Python string is split into a Python list of smaller Python strings using the standard `splitlines()` method. It is instructive to try the following:

- Temporarily remove the `.splitlines()` call above and `print repr(two_theta_and_index_list)`. This will show a single string with embedded new-line characters:
  -
  - `'    8.81   0  1  1\n   12.23   0  0  2\n   12.71   0  2  0\n ...`
  - `31.56   0  1  5\n   32.12   2  2  3\n'`
- At the command prompt, enter libtbx.help str to see the full documentation for the Python `str` type including the documentation for `splitlines()`:
  -
  - `|  splitlines(...)`
  - `|      S.splitlines([keepends]) -> list of strings`
  - `|`
  - `|      Return a list of the lines in S, breaking at line boundaries.`
  - `|      Line breaks are not included in the resulting list unless keepends`
  - `|      is given and true.`
- With the `.splitlines()` call added back, `print repr(two_theta_and_index_list)` again. This will show a Python list of strings:
  -
  - `['    8.81   0  1  1', '   12.23   0  0  2', '   12.71   0  2  0', ...`
  - `'   31.56   0  1  5', '   32.12   2  2  3']`

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

Conversion of input data to flex arrays

The list of Python strings as obtained above has to be converted to *flex arrays* to be suitable for calculating the residual sum. This is achieved with the following Python statements:

```
from cctbx.array_family import flex

two_thetas_obs = flex.double()
miller_indices = flex.miller_index()
for line in two_theta_and_index_list:
  fields = line.split()
  assert len(fields) == 4
  two_thetas_obs.append(float(fields[0]))
```

```
  miller_indices.append([int(s) for s in fields[1:]])
```

The first statement imports the `cctbx.array_family.flex` module, which provides a number of array types, e.g. `int`, `double`, `std_string` and `miller_index`. These `flex` arrays can be one-dimensional or multi-dimensional. The numeric array types provide a comprehensive set of functions for element-wise operations such as `>`, `+`, `sin`, and reduction functions such as `sum`, `min`, `min_index`. Try libtbx.help cctbx.array_family.flex and libtbx.help cctbx.array_family.flex.double to see a listing of the available features. Almost all facilities provided by the `flex` module are implemented in C++ and are therefore very fast.

In the `unit_cell_refinement.py` example the input data are converted to the flex array types `double` and `miller_index`. The statement:

```
two_thetas_obs = flex.double()
miller_indices = flex.miller_index()
```

instantiate brand-new one-dimensional arrays. The initial size of both arrays is `0`, as can be verified by inserting `print two_thetas_obs.size()` and `print miller_indices.size()`. The next three statements loop over all lines in `two_theta_and_index_list`:

```
for line in two_theta_and_index_list:
  fields = line.split()
  assert len(fields) == 4
```

Each line is split into fields. Use libtbx.help str again to obtain the documentation for the `split()` method. The `assert` statement reflects good coding practices. It is not strictly needed, but the following two statements assume that the input line consists of exactly four fields. If this is not the case, non-obvious error messages may result. Using `assert` statements is an easy way of obtaining more reasonable error messages. They also help others understanding the source code.

The `fields` are still strings, but it is easy to convert the first field to a Python `float` instance and to append it to the `two_thetas_obs` array:

```
two_thetas_obs.append(float(fields[0]))
```

The same result could be achieved in three steps:

```
s = fields[0] # Python lists are indexed starting with 0
v = float(s) # conversion from str -> float
two_thetas_obs.append(v)
```

However, the one-line version is not only shorter but clearly better for two main reasons:

- we don't have to invent names for the intermediate results (`s` and `v` in the second version); therefore the code is easier to read.
- the intermediate results are automatically deleted, i.e. the corresponding memory is released immediately.

A full equivalent of the one-line version is actually even longer:

```
s = fields[0]
v = float(s)
del s
two_thetas_obs.append(v)
del v
```

The conversion of the Miller indices is more involved. Three integer indices have to be converted from strings to integers and finally added to the `miller_indices` array. It could be done like this:

```
h = int(fields[1])
k = int(fields[2])
l = int(fields[3])
miller_indices.append([h,k,l])
```

However, anyone who has spent frustrating hours debugging silly copy-and-paste mistakes like `k = int(fields[1])` will prefer the alternative using Python's List Comprehension syntax:

```
hkl = [int(s) for s in fields[1:]]
```

This is equivalent to the longer alternative:

```
hkl = []
for s in fields[1:]:
  hkl.append(int(s))
```

Of course, that's hardly a gain over the simple copy-and-paste solution, but the list comprehension solution clearly is, and since it is *one* expression it can be directly used as an argument for `miller_indices.append()`.

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

Calculation of 2-theta angles

Bragg's law tells us how to compute diffraction angles `theta`. In typical textbook notation:

```
lambda = 2 * d_hkl * sin(theta)
```

In Python we cannot use `lambda` as a variable name since it is a reserved keyword for functional programming; therefore we use the variable name `wavelength` instead. Rearranging Bragg's equation leads to:

```
2 * sin(theta) = wavelength / d_hkl
```

I.e. we need to obtain two pieces of information, the `wavelength` and the *d-spacing* `d_hkl` corresponding to a Miller index `hkl`.

The input data in the `unit_cell_refinement.py` script are derived from a powder diffraction experiment with copper k-alpha radiation. A lookup table for the corresponding wavelength is compiled into the `cctbx.eltbx.wavelengths` module (libtbx.help cctbx.eltbx.wavelengths.characteristic):

```
from cctbx.eltbx import wavelengths
wavelength = wavelengths.characteristic("CU").as_angstrom()
```

We don't have to calculate `d_hkl` explicitly since the `cctbx.uctbx.unit_cell` object "knows" about Bragg's law and also how to compute `d_hkl` given unit cell parameters (libtbx.help cctbx.uctbx.unit_cell). This allows us to write:

```
from cctbx import uctbx
unit_cell = uctbx.unit_cell((10,10,10,90,90,90))
two_thetas_calc = unit_cell.two_theta(miller_indices,
wavelength, deg=True)
```

Conveniently the `two_theta()` method computes all `two_thetas_calc` in one call, given an array of `miller_indices`. Now that we have both `two_thetas_obs` and `two_thetas_calc` it is a matter of two lines to show a nice table:

```
for h,o,c in zip(miller_indices, two_thetas_obs,
two_thetas_calc):
  print "(%2d, %2d, %2d)" % h, "%6.2f - %6.2f = %6.2f" % (o,
c, o-c)
```

Use libtbx.help zip to learn about Python's standard `zip()` function, or consult the Python tutorial section on Looping Techniques for more information. The `print` statement can be understood by reading the tutorial section on Fancier Output Formatting.

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

Computation of the least-squares residual

Given `two_thetas_obs` and `two_thetas_calc`, the least-squares residual defined in the first section can be computed with three nested calls of functions provided by the `flex` module introduced before:

```
flex.sum(flex.pow2(two_thetas_obs - two_thetas_calc))
```

The inner-most call of a `flex` function may not be immediately recognizable as such, since it is implemented as an *overloaded* `-` operator. In a more traditional programming language the element-wise array subtraction may have been implemented like this:

```
element_wise_difference(two_thetas_obs, two_thetas_calc)
```

Python's operator overloading gives us the facilities to write `two_thetas_obs - two_thetas_calc` instead. This expression returns a new array with the differences. The `flex.pow2()` function returns another new array with with the squares of the differences, and the `flex.sum()` function finally adds up the squared differences to return a single value, the least-squares residual. All intermediate arrays are automatically deleted during the evaluation of the nested expression as soon as they are no longer needed.

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

Gradient calculation via finite differences

The Finite Difference Method approximates the gradients of a function `f(u)` at the point `x` with the simple formula:

```
df/du = (f(x+eps) - f(x-eps)) / (2*eps)
```

`eps` is a small shift. This method is also applicable for computing partial derivatives of multivariate functions. E.g. the gradients of `f(u,v)` at the point `x,y` are approximated as:

```
df/du = (f(x+eps,y) - f(x-eps,y)) / (2*eps)
df/dv = (f(x,y+eps) - f(x,y-eps)) / (2*eps)
```

The main disadvantage of the finite difference method is that two full function evaluations are required for each parameter. In general it is best to use analytical gradients, but it is often a tedious and time consuming project to work out the analytical expressions. Fortunately, in our case we have just six parameters, and the runtime for one function evaluation is measured in micro seconds. Using finite differences is exactly the right approach in this situation, at least as an initial implementation.

To facilitate the gradient calculations, the residual calculation as introduced before is moved to a small function:

```
def residual(two_thetas_obs, miller_indices, wavelength,
unit_cell):
  two_thetas_calc = unit_cell.two_theta(miller_indices,
wavelength, deg=True)
  return flex.sum(flex.pow2(two_thetas_obs -
two_thetas_calc))
```

The finite difference code is now straightforward:

```
def gradients(two_thetas_obs, miller_indices, wavelength,
unit_cell, eps=1.e-6):
  result = flex.double()
  for i in xrange(6):
    rs = []
    for signed_eps in [eps, -eps]:
      params_eps = list(unit_cell.parameters())
      params_eps[i] += signed_eps
      rs.append(
        residual(
          two_thetas_obs, miller_indices, wavelength,
          uctbx.unit_cell(params_eps)))
    result.append((rs[0]-rs[1])/(2*eps))
  return result
```

The `list()` constructor used in this function creates a copy of the list of unit cell parameters. The chosen `eps` is added to or subtracted from the parameter `i`, and a new `uctbx.unit_cell` object is instantiated with the modified parameters. With this the residual is computed as before. We take the difference of two residuals divided by `2*eps` and append it to the resulting array of gradients.

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

LBFGS minimization

As outlined at the beginning, the `residual()` and `gradients()` are to be used in connection with the quasi-Newton LBFGS minimizer as implemented in the `scitbx.lbfgs` module. A minimal recipe for using this module is shown in the scitbx/scitbx/examples/lbfgs_recipe.py script:

```
class refinery:

  def __init__(self):
    self.x = flex.double([0])
    scitbx.lbfgs.run(target_evaluator=self)

  def compute_functional_and_gradients(self):
    f = 0
    g = flex.double([0])
    return f, g

  def callback_after_step(self, minimizer):
    pass
```

This `refinery` class acts as a `target_evaluator` object for the `scitbx.lbfgs.run()` function. Basically, the `target_evaluator` object can be any user-defined type, but it has to conform to certain requirements:

- `target_evaluator.x` must be a `flex.double` array with the parameters to be refined.
- `target_evaluator.compute_functional_and_gradients()` must be a function taking no arguments. It has to return a floating-point value for the functional, and a `flex.double` array with the gradients of the functional with respect to the parameters at the current point `target_evaluator.x`. The size of the gradient array must be identical to the size of the parameter array.
- `target_evaluator.callback_after_step()` is optional. If it is defined, it is called with one argument after each LBFGS step. The argument is an instance of the scitbx::lbfgs::minimizer C++ class and can be analyzed to monitor or report the progress of the minimization. `target_evaluator.callback_after_step` may or may not return a value. If the returned value `is True` the minimization is terminated. Otherwise the minimization continues until another termination condition is reached.

The `scitbx.lbfgs.run(target_evaluator=self)` call in the `__init__()` method above initiates the LBFGS procedure. This procedure modifies the `self.x` (i.e. `target_evaluator.x`) array in place, according to the LBFGS algorithm. Each time the algorithm requires an evaluation of the functional and the gradients, the `compute_functional_and_gradients()` method is called. I.e. the `refinery` object calls `scitbx.lbfgs.run()` which in turn calls a method of the `refinery` object. The term `callback` is often used to describe this situation. Note that both `compute_functional_and_gradients()` and `callback_after_step()` are callback functions; they are just called in different situations.

Based on the `residual()` and `gradients()` functions developed before, it is not difficult to customize the recipe for the refinement of unit cell parameters:

```
class refinery:

  def __init__(self, two_thetas_obs, miller_indices,
wavelength, unit_cell):
    self.two_thetas_obs = two_thetas_obs
    self.miller_indices = miller_indices
    self.wavelength = wavelength
    self.x = flex.double(unit_cell.parameters())
    scitbx.lbfgs.run(target_evaluator=self)

  def unit_cell(self):
    return uctbx.unit_cell(iter(self.x))

  def compute_functional_and_gradients(self):
    unit_cell = self.unit_cell()
    f = residual(
      self.two_thetas_obs, self.miller_indices,
self.wavelength, unit_cell)
    g = gradients(
      self.two_thetas_obs, self.miller_indices,
self.wavelength, unit_cell)
    print "functional: %12.6g" % f, "gradient norm: %12.6g" %
g.norm()
    return f, g

  def callback_after_step(self, minimizer):
    print "LBFGS step"
```

With this class all required building blocks are in place. The refinement is run and the results are reported with these statements:

```
refined = refinery(
  two_thetas_obs, miller_indices, wavelength,
unit_cell_start)
print refined.unit_cell()
```

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

To try goes studying over

The title of this section is Google's automatic translation of the German saying "Probieren geht ueber studieren." It is an invitation to copy and run this and other example scripts. Insert `print` statements to develop a better understanding of how all the pieces interact. Use `print list(array)` to see the elements of `flex` arrays. It may also be useful to insert `help(obj)` to see the attributes and methods of `obj`, where `obj` can be any of the objects created in the script.

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

Exercise (not very hard)

Read the 2-theta angles and Miller indices from a file.

Hint: Learn about `import sys` and `sys.argv`.

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

Exercise (not very hard)

import `iotbx.command_line.lattice_symmetry` and instantiate the `metric_subgroups` class with the refined unit cell.

Hint: Look for `"P 1"` in the `run()` function in `iotbx/iotbx/command_line/lattice_symmetry.py`.

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

Exercise (harder)

Estimate the start parameters from 6 indices and associated 2-theta angels.

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

Exercise (advanced)

Work out a way to use analytical gradients. The best solution is not clear to me (rwgk). You may have to refine metric tensor elements instead of the unit cell parameters to keep the problem manageable. See d_star_sq_alternative.py for a possible start. Compare the time it took you to work out the code with the analytical gradients to the time it takes to implement the finite difference method.

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

Exercise (requires creativity)

Study adp_symmetry_constraints.py. Use:

```
constraints = sgtbx.tensor_rank_2_constraints(
```

```
  space_group=space_group,
  reciprocal_space=False,
  initialize_gradient_handling=True)
```

to reduce the number of unit cell parameters to be refined.

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