Home
Download
How to install
Getting started
Documentation
Tutorials
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

cctbx - Computational Crystallography Toolbox

The Computational Crystallography Toolbox (cctbx) is an open-source library of reusable software components for macromolecular structure determination (Grosse-Kunstleve et al., 2002. The cctbx components are written both in C++ (a compiled language) and Python (a flexible scripting language. This approach is very efficient because high-level algorithms (such as refinement protocols) can be rapidly developed in Python, while computationally intensive algorithms can be implemented in C++. The Boost.Python Library is used to expose the C++ interfaces, classes and functions to Python (Abrahams & Grosse-Kunstleve, 2003).
The cctbx is released under an Open Source license to allow unrestricted use and continued development

What the cctbx can do for you

The cctbx has many modules that are specifically developped for applications in structural biology: crystallography and cryo-EM. For example, the cctbx contains algorithms for the handling of unit cells, space groups and atomic scatterers.
The cctbx can used to develop new algorithms for structural biology. It also allows easy scripting of basic tasks.

About the cctbx

The Computational Crystallography Toolbox (*cctbx*) is being developed as the open source component of the *Phenix* system. The goal of the *Phenix* project is to advance automation of macromolecular structure determination. *Phenix* depends on the cctbx, but not vice versa. This hierarchical approach enforces a clean design as a reusable library. The *cctbx* is therefore also useful for small-molecule crystallography and even general scientific applications.

To maximize reusability and, maybe even more importantly, to give individual developers a notion of privacy, the *cctbx* is organized as a set of smaller modules. This is very much like a village (the *cctbx* project) with individual houses (modules) for each family (groups of developers, of any size including one).

The *cctbx* code base is available without restrictions and free of charge to all interested developers, both academic and commercial. The entire community is invited to actively participate in the development of the code base. A sophisticated technical infrastructure that enables community based software development is provided by GitHub. This service is also free of charge and open to the entire world.

The *cctbx* is designed with an open and flexible architecture to promote extendability and easy incorporation into other software environments. The package is organized as a set of ISO C++ classes with Python bindings. This organization combines the computational efficiency of a strongly typed compiled language with the convenience and flexibility of a dynamically typed scripting language in a strikingly uniform and very maintainable way.

Use of the Python interfaces is highly recommended, but optional. The *cctbx* can also be used purely as a C++ class library.

Organization

The cctbx project contains several modules:

- **libtbx:**
  The build system common to all other modules. This includes a very thin wrapper around the SCons software construction tool. It also contains many useful frameworks and utilities to simplify application development, including tools for regression testing, parallelization across multiprocessor systems and managed clusters, and a flexible, modular configuration syntax called PHIL (Python Hierarchial Interface Language) used throughout the cctbx.
  API Documentation for libtbx

- **boost_adaptbx:**
  A very small adaptor toolbox with platform-independent instructions for building the Boost.Python library.
  API Documentation for boost_adaptbx

- **scitbx:**
  Libraries for general scientific computing (i.e. libraries that are not specific to crystallographic applications). This includes a family of high-level C++ array types, a fast Fourier transform library, and a C++ port of the popular L-BFGS quasi-Newton minimizer, and many mathematical utilities, all including Python_ bindings. These libraries are separated from the crystallographic code base to make them easily accessible for non-crystallographic application developers.
  API Documentation for scitbx

- **cctbx:**
  Libraries for general crystallographic applications, useful for both small-molecule and macro-molecular crystallography. The libraries in the cctbx module include algorithms and data structures for the handling of crystal symmetry, basic geometry restraints, reflection data, atomic displacement parameters, X-ray scattering, and high-level building blocks for refinement algorithms.
  Note the distinction between the CCTBX *project* and the cctbx *module*.
  API Documentation for cctbx

- **iotbx:**
  Libraries for reading and writing common file formats, including PDB, CIF, many reflection formats, electron density maps, and sequences. API Documentation for iotbx

- **mmtbx:**
  Functionality specific to macromolecular crystallography. This includes all of the machinery required for setup of geometry restraints, bulk solvent correction and scaling, analysis of macromolecular diffraction data, calculation of weighted map coefficients, and most of the methods implemented in phenix.refine. The majority of infrastructure for the MolProbity validation server (and Phenix equivalent) is also located here. API Documentation for mmtbx

- **xfel:**
  Software for processing serial data collected using an X-ray free electron laser. Includes spotfinding, integration, data clustering/filering and merging tools. API Documentation for xfel

- **smtbx:**
  Functionality specific to small-molecule crystallography, including a complete refinement program (smtbx.refine).

- **dxtbx:**
  The Diffraction Image Toolbox, a library for handling X-ray detector data of arbitrary complexity from a variety of standard formats. (Also used by routines in iotbx.)

- **Other:**
  Many additional libraries have more specialized functionality, including:
  - spotfinder - fast detection of Bragg peaks in diffraction images
  - ucif - the core CIF I/O library (used by iotbx)
  - rstbx - Reciprocal Space Toolbox, used for data processing
  - gltbx - OpenGL bindings, including a wxPython-based viewer framework
  - crys3d - Modules for the display of molecules, electron density, and reciprocal space data
  - fable - a program (and compatibility library) for porting Fortran77 to C++
  - wxtbx - wxPython controls used in the Phenix GUI and various utilities
  - cma_es - a library of derivative-free optimization methods

Acknowledgments

We would like to thank David Abrahams for creating the amazing Boost.Python library and for patiently supporting the entire open source community. We would like to thank Airlie McCoy for allowing us to adapt some parts of the Phaser package (FFT structure factor calculation). Kevin Cowtan has contributed algorithms for the handling of reciprocal space asymmetric units. We are also grateful for his development of the Clipper library from which we have adapted some source code fragments. 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.
The cctbx git development tree is hosted by GitHub.

Home
Download
How to install
Getting started
Documentation
Tutorials
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

cctbx - Computational Crystallography Toolbox

The Computational Crystallography Toolbox (cctbx) is an open-source library of reusable software components for macromolecular structure determination (Grosse-Kunstleve et al., 2002. The cctbx components are written both in C++ (a compiled language) and Python (a flexible scripting language. This approach is very efficient because high-level algorithms (such as refinement protocols) can be rapidly developed in Python, while computationally intensive algorithms can be implemented in C++. The Boost.Python Library is used to expose the C++ interfaces, classes and functions to Python (Abrahams & Grosse-Kunstleve, 2003).
The cctbx is released under an Open Source license to allow unrestricted use and continued development

What the cctbx can do for you

The cctbx has many modules that are specifically developped for applications in structural biology: crystallography and cryo-EM. For example, the cctbx contains algorithms for the handling of unit cells, space groups and atomic scatterers.
The cctbx can used to develop new algorithms for structural biology. It also allows easy scripting of basic tasks.

cctbx_project

Public

Edit Pins Watch 5 Fork 121 Star 252

master 285 Branches 58 Tags Go to file t Add file Code

russell-taylor Fixing regression tests. f26361f · 5 hours ago 38,556 Commits

.azure-pipelines	Clean clutter	yesterday
.github/workflows	README: switch nighly conda package tests to GitHub Ac...	last year
boost	boost_adaptbx: change imports	5 years ago
boost_adaptbx	boost: add temporary flag for Boost Timer deprecation	9 months ago
cbflib_adaptbx	boost: switch deprecated boost/timer.hpp to boost/timer/t...	last year
cctbx	Clean clutter	yesterday
cctbx_website	Use libtbx.env to find files in modules for build_cctbx_api...	2 weeks ago
chiltbx	scons: prevent installed modules from recompiling	5 years ago
clipper_adaptbx	boost_adaptbx: change imports	5 years ago
cma_es	scons: prevent installed modules from recompiling	5 years ago
cootbx	cootbx: make file finding more portable	9 months ago
crys3d	crys3d: update syntax for 3d viewer	2 weeks ago
cudatbx	cudatbx: clean up build	2 months ago
dox.sphinx	py3: first stage of xrange to range	6 years ago
dox	Add documentation for debug builds and heap analysis (#...	4 years ago
fable	add some missing functions to intrinsics.hpp	3 months ago
fast_linalg	boost_adaptbx: change imports	5 years ago
fftw3tbx	boost_adaptbx: change imports	5 years ago
gltbx	crys3d: update syntax for 3d viewer	2 weeks ago
iota	Pick up iota from new repository location (#469)	5 years ago
iotbx	clarify use of set_unit_cell_crystal_symmetry in map_ma...	yesterday
kokkostbx	Merge changes made for Frontier (#934)	last year
libtbx	libtbx: add option for adding extra lines to the Python scri...	yesterday
mmtbx	Fixing regression tests.	5 hours ago
msvc9.0_include	More Visual Studio compatibility	2 years ago
omptbx	scons: prevent installed modules from recompiling	5 years ago
prime	XFEL GUI updates (#992)	last year
qttbx	Fix qttbx/viewers/__init__.py to find ChimeraX in python 2.7	2 years ago
rsttbx	Handle more recent versions of INTEGRATE.HKL in cfttbx ...	2 weeks ago
scitbx	Add __init__.py so pdoc3 will run	2 months ago
serialtbx	Spring 2025 XFEL GUI updates (#1065)	2 months ago
simtbx	XFEL GUI updates (#992)	last year
smtbx	smtbx: remove pytest dependency	8 months ago
sphinx	Refactoring out usage of hierarchy.input.	2 years ago
spotfinder	boost: switch deprecated boost/timer.hpp to boost/timer/t...	last year
tbxx	Windows support for tbxx::time_accu	10 years ago
ucif	Fix spelling of "minimum" in comments.	3 years ago
wxtbx	wxtbx: update expected FlexGridSizer function signature	2 weeks ago
xfel	XFEL GUI: query bugfix	2 weeks ago
.cluttertignore	libtbx: add function for getting the version	4 years ago
.git-notifier	Add commit email notification configuration file	6 years ago
.gitattributes	Pick up iota from new repository location (#469)	5 years ago
.gitignore	adding mahalanobis distance	10 months ago
.gitmodules	Update submodule xfel/euxfel/definitions to point to nexu...	3 years ago
CHANGELOG.rst	Update CHANGELOG.rst for 2025.5 release	2 weeks ago
CODE_OF_CONDUCT.md	Create CODE_OF_CONDUCT.md	6 years ago
CONTRIBUTING.md	Update CONTRIBUTING.md	4 years ago
COPYRIGHT.txt	Bump year for license and copyright notices	3 months ago
LICENSE.txt	Bump year for license and copyright notices	3 months ago
README.md	README: add instructions for installing chem_data	2 months ago
azure-pipelines.yml	CI: reorganize pipelines	5 years ago

README Code of conduct License

Computational Crystallography Toolbox

Azure Pipelines failed | conda-forge v2025.5 | Nightly conda package tests passing
Platforms linux-64,osx-arm64,win-64,osx-64,linux-ppc64le,linux-aarch64 | DOI 10.5107/zenodo.1285985

Table of Contents

- Introduction
- Installation
- Building a development version
- Contributing to the cctbx
- Nightly builds and tests

Introduction

The Computational Crystallography Toolbox (cctbx) is being developed as the open source component of the Phenix project. The goal of the Phenix project is to advance automation of macromolecular structure determination. Phenix depends on the cctbx, but not vice versa. This hierarchical approach enforces a clean design as a reusable library. The cctbx is therefore also useful for small-molecule crystallography and even general scientific applications.

The cctbx also provides some of the key component of the Olex 2 software. Olex 2 is dedicated to the workflow of small molecule crystallographic studies. It features a powerful and flexible refinement engine, olex2.refine, which is developed as part of the cctbx, in the smtbx top-module.

To maximize reusability and, maybe even more importantly, to give individual developers a notion of privacy, the cctbx is organized as a set of smaller modules. This is very much like a village (the cctbx project) with individual houses (modules) for each family (groups of developers, of any size including one).

The cctbx code base is available without restrictions and free of charge to all interested developers, both academic and commercial. The entire community is invited to actively participate in the development of the code base. A sophisticated technical infrastructure that enables community based software development is provided by GitHub. This service is also free of charge and open to the entire world.

The cctbx is designed with an open and flexible architecture to promote extendability and easy incorporation into other software environments. The package is organized as a set of ISO C++ classes with Python bindings. This organization combines the computational efficiency of a strongly typed compiled language with the convenience and flexibility of a dynamically typed scripting language in a strikingly uniform and very maintainable way.

Use of the Python interfaces is highly recommended, but optional. The cctbx can also be used purely as a C++ class library.

Installation

The easiest way to install cctbx is through the Conda package manager. We recommend the Miniforge installers since they provide a minimal environment and default to the `conda-forge` channel.

There are two packages available, `cctbx` and `cctbx-base`. The `cctbx` package is `cctbx-base` with some additional GUI packages (e.g. `wxpython`, `pyside2`, `ipython`).

With the `conda` command available, a new `cctbx-base` environment named `my_env` can be created with

```
conda create -n my_env -c conda-forge cctbx-base
```

To choose a specific version of Python, add the `python` package with the specific version

```
conda create -n my_env -c conda-forge cctbx-base python=3.8
```

Then the environment can be activated with

```
conda activate my_env
```

To install `cctbx-base` into the currently active environment, use

```
conda install -c conda-forge cctbx-base
```

The `python` package with a specific version can be added to change the version of `python` that is already installed in the active environment.

Monomer library

Some programs in cctbx require information about geometric restraints for molecules. This information is available in the `chem_data` conda package from our releases. Download the `chem_data` conda package and install in your active environment with

```
conda install <chem_data package>
```

The `chem_data` package is built from the `chem_data` and `geostd` repositories.

Building a development version

1. Download https://raw.githubusercontent.com/cctbx/cctbx_project/master/libtbx/auto_build/bootstrap.py in the directory where the cctbx and its dependencies shall be installed
2. Run `python bootstrap.py` (you may want to run it with the `--help` option first to discover the available options).

- For better compatibility with newer operating systems, `conda` packages can be used for dependencies. Add the `--use-conda` flag and the command becomes `python bootstrap.py --use-conda`. This will run the `miniconda` installer if `conda` cannot be found. The environment with the dependencies will be located in the `conda_base` directory. See the description of the `--use-conda` flag from the `--help` output for more details.

The installation will take a long while but the script will verbosely describe what it does.

Contributing to the cctbx

For a more detailed description on how to contribute to the cctbx please visit our contribution guide.

Nightly builds and tests

Azure Pipelines succeeded | conda-nightly v2025.6a13 | Platforms linux-aarch64,osx-arm64,linux-ppc64le,win-64,linux-64,osx-64

A nightly build of the `conda` packages are available on the `cctbx-nightly` channel. To use these packages, prepend `-c cctbx-nightly` as a channel to the commands above. For example, the command to create a new `my_env` environment would become

```
conda create -n my_env -c cctbx-nightly -c conda-forge cctbx-base
```

This will use the `cctbx-base` package from the `cctbx-nightly` channel, but pull the remaining dependencies from `conda-forge`.

Nightly builds are only updated if there are additional commits from the previous build.

A subset of tests is run on the current `cctbx-base` packages from the `conda-forge` and `cctbx-nightly` channels every night (10 pm Pacific) to test compatibility with the latest packages from `conda-forge`. Additional source files for `fable` and `antlr3` are needed for the tests. The nightly test details can be viewed by clicking the "Nightly conda package tests" badge near the beginning of this README.

About

Computational Crystallography Toolbox

cci.lbl.gov/docs/cctbx

crystallography cryo-em phenix cctbx

Readme
View license
Code of conduct
Activity
Custom properties
252 stars
42 watching
121 forks
Report repository

Releases 58

2025.5 Latest
2 weeks ago

+ 57 releases

Contributors 109

+ 95 contributors

Languages

Python 79.8% C++ 15.7% C 2.6% Mathematica 0.5% HTML 0.4% Cuda 0.3% Other 0.7%

© 2025 GitHub, Inc. Terms Privacy Security Status Docs Contact Manage cookies Do not share my personal information

Home
Download
How to install
Getting started
Documentation
Tutorials
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

How to install

Easy: Install using conda
Building a development version
SCons - the backbone of the cctbx build system

Easy: Install using conda

The easiest way to install cctbx is through the Conda package manager. You can get a full environment from Anaconda or just the conda package manager with the Miniconda installer.

There are two packages available, `cctbx` and `cctbx-base`. The `cctbx` package is `cctbx-base` with some additional packages (`wxpython`, `pyside2`, `ipython`).

With the `conda` command available, a new `cctbx-base` environment named `my_env` can be created with

```
conda create -n my_env -c conda-forge cctbx-base
```

To choose a specific version of Python, add the `python` package with the specific version

```
conda create -n my_env -c conda-forge cctbx-base python=3.8
```

Then the environment can be activated with

```
conda activate my_env
```

To install `cctbx-base` into the currently active environment, use

```
conda install -c conda-forge cctbx-base
```

The python package with a specific version can be added to change the version of `python` that is already installed in the active environment.

Building a development version

1. Download https://raw.githubusercontent.com/cctbx/cctbx_project/master/libtbx/auto_build/bootstrap.py in the directory where the cctbx and its dependencies shall be installed

2. Run `python bootstrap.py` (you may want to run it with the `--help` option first to discover the available options).

For better compatibility with newer operating systems, you will want to use conda packages for dependencies. Add the `--use-conda` flag and the command becomes `python bootstrap.py --use-conda`. This will run the miniconda installer if conda cannot be found. The environment with the dependencies will be located in the `conda_base` directory. See the description of the `--use-conda` flag from the `--help` output for more details.
The installation will take a long while (15 min - 1 hour, depending on your computer) but the script will verbosely describe what it does.

This creates the subdirectories `base`, `build`, and `modules`. The `base` directory contains dependencies for cctbx, `build` contains the compiled cctbx code, and `modules` contains the source code for cctbx. To keep `bootstrap.py` in your installation directory up-to-date with the version in modules, you can create a symbolic link to that version:

```
cd <installation directory>
rm bootstrap.py
ln -s ./modules/cctbx_project/libtbx/auto_build/bootstrap.py
```

Within the build directory, `cctbx` creates a file `setpaths.csh` (among others). This file must be used to initialize a new shell or process with the cctbx settings:

```
source setpaths.csh
```

There is also a `setpaths.sh` for `bash` users.

To update your cctbx installation to the latest version, you can just run:

```
python ./bootstrap.py
```

again in the `<installation directory>`. This will update the source code in the `modules` directory and recompile the changes (if necessary) in build. Occasionally, dependencies in `base` are updated. When this happens, just delete the `base` directory and rerun the `bootstrap.py` command.

To compile any local changes to the source code, enter the `build` directory and run:

```
make
```

This will actually run the `libtbx.scons` command using all available CPUs. You can also manually specify the number of CPUs to use, for example:

```
libtbx.scons -j 4
```

Note that `libtbx.scons` is just a thin wrapper around SCons. The SCons documentation applies without modification.

To run scripts with cctbx imports use the command:

```
libtbx.python your_script.py
```

(You can also use `scitbx.python`, `cctbx.python`, `iotbx.python`, etc.; all these commands are equivalent.)

For example, to run some regression tests after the compilation is finished enter:

```
source build/setpaths_all.csh
libtbx.python $SCITBX_DIST/run_tests.py
libtbx.python $CCTBX_DIST/run_tests.py --Quick
```

The output should show many OK. A Python Traceback is an indicator for problems.

SCons - the backbone of the cctbx build system

Conceptually it is a trivial task to compile and link portable source code. However, in real life this is one of the most time-consuming nuisances, in particular if multiple, diverse platforms have to be supported. In the version 1.0 release of the cctbx we made an attempt to address this with the home-made *fast track* build system. Of course home-made is often good enough, but a professional solution is almost always better, especially if it comes with no strings attached.

Fortunately such a system is now available: SCons, short for Software Construction tool. This is a perfect fit for the cctbx because the SCons developers have apparently adopted a similar "professional is better than home-made" philosophy: SCons is implemented in pure Python, and SCons configuration files (the equivalent of Makefiles) are pure Python scripts. SCons has many advantages compared to a traditional make-based build system. To quote some points from the SCons documentation:

- Global view of all dependencies – no more multiple build passes or reordering targets to build everything.
- Reliable detection of build changes using MD5 signatures – no more "clock skew detected, build may be incomplete".
- Built-in support for C, C++, Fortran, Yacc and Lex.
- Improved support for parallel builds – like make -j but keeps N jobs running simultaneously regardless of directory hierarchy.
- Building from central repositories of source code and/or pre-built targets.
- Designed from the ground up for cross-platform builds, and known to work on Linux, POSIX, Windows NT, Mac OS X, Tru64 Unix, and OS/2.

When we moved from our home-grown build system to SCons we found all these points to be perfectly true. It only took very little effort to write a small configure script for setting up a master SConstruct file based on the user's choice of which cctbx modules to use and which to ignore. New modules can easily be tied into this system simply by providing a SConstruct file in the module's top-level directory. The author of the new module has complete control over the build process. The existing settings can simply be reused, customized, or totally replaced, all within one uniform and 100% platform-independent framework, the Python language.

Home
Download
How to install
Getting started
Documentation
Tutorials
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

Getting started

Learn how to use the computational crystallography toolbox, the open source library for crystallography and cryo-EM.

Download
Installation
Running on the command line
Running as a script
Google colab with cctbx code snippets
I am completely new to cctbx. Where do I start?
I got the basics. How can I learn more?
Community and help

Download

Go to the Download page to get the most recent cctbx source code as binary installer for Linux, MacOS and Windows.

Installation

Installation is very simple and fast. After uncompressing the bundle, run the `install --prefix=` command to install on macOS and Linux. On Windows, uncompressing the zip file will provide a working copy of cctbx.

Running on the command line

Type this into a terminal window to start python from cctbx so that it knows about all the cctbx modules:

```
libtbx.python
```

The libtbx.python command runs a version of python that already knows about the cctbx so you can easily import cctbx functions. For example:

```
from iotbx.data_manager import DataManager    # Load in the DataManager
```

Running as a script

Sometimes you want to save the commands in a script. This avoids typing the same text over and over again if you want to perform some actions on many files. For example, save the following lines in a file, you can call it `hello_cctbx.py` (this page explains in more detail what the script does).

```
from iotbx.map_model_manager import map_model_manager      # load in the map_model_manager
mmm=map_model_manager()          # get an initialized instance of the map_model_manager
mmm.generate_map()               # get a model from a small library model and calculate a map for
mmm.write_map("map.mrc")         # write out a map in ccp4/mrc format
mmm.write_model("model.pdb")     # write out a model in PDB format
```

Then run the script by typing the following command in a terminal:

```
libtbx.python hello_cctbx.py
```

You just ran your first cctbx script which produced example map and model files.

Google colab with cctbx code snippets

- cctbx introduction: This colab gives an introduction to major cctbx features
- cctbx code snippets: This colab has some code snippets, mostly about models and miller arrays.

I am completely new to cctbx. Where do I start?

If you want to use the cctbx, you most likely want to do some actions on models, crystallographic data files or cryo-EM maps. The following pages give an introduction to higher level objects and methods for this purpose.

High-level cctbx objects	Overview of high level objects that represent things like data, models, restraints, etc. Learn how to create your first object, the map_model_manager..
Reading and writing files	Most scripts will act on files, so this section should be of interest to everyone. Use the DataManager to conveniently read and write data and model files.
The cctbx model manager	The model manager lets you do operations on models. Learn how to to set up the model manager, apply coordinate shifts, do atom selections and calculate model-based map coefficients.
The cctbx map manager	The map manager lets you do operations on maps. Learn how to read and save maps, change a map, convert a real-space map to a Fourier representation and back and manipulate maps that represent only a part of a full map. The tutorial also explains how to find and represent map reconstruction symmetry.
The map-model manager	The map-model manager is useful to keep track of both maps and models. This tutorial shows how to set up the manager, how to cut out a piece of a map around a part of a model and how to create masks.

I got the basics. How can I learn more?

Coming soon...

Community and help

Asking for help

- If you want to ask a question to developers and users, subscribe to the cctbx mailing list. Here are the archives of the cctbxbb.
- You can also write to the Phenix/cctbx developers directly: help@phenix-online.org

Community

Want to contribute to the cctbx? Here is the GitHub page.
Visit our contributing guide for a detailed description of guidelines and how to write tests.

Home
Download
How to install
Getting started
Documentation
Table of Contents
Programming tips
Python/cctbx features
Python/cctbx hints
Multiprocessing
High level objects
Maps
Models
HKL viewer
Low level
Tutorials
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

Documentation Table of Contents

- Download: Download the cctbx for Linux, MacOS or Windows.
- Installation: Learn how to install the cctbx.
- Getting started: Overview of what to do to get started using the cctbx.
- Python/cctbx features: Non-obvious features of Python and cctbx functions.
- Python/cctbx hints: Helpful cctbx and Python hints.
- Multiprocessing: How to run any method with multiprocessing.
- High level cctbx objects:
  - Intro to high-level objects: Overview of high level objects that represent things like data, models, restraints, etc.
  - Reading and writing files: Use the DataManager to read and write data and model files.
  - The cctbx model manager: The model manager lets you do operations on models, such as apply coordinate shifts, do atom selections and calculate model-based map coefficients.
  - The cctbx map manager: The map manager lets you do operations on maps.
  - The map-model manager: The map-model manager is useful to keep track of both maps and models.
- Maps:
  - Introduction: Learn about some important map properties, such as gridding and padding. Learn how to convert between grid points and coordinates and how to get map values at specific positions.
  - Boxing: Learn how to work with a part of a map. This involves keeping track of coordinate shifts and symmetry.
- Lower level components:
  - Flex arrays: An array specifically designed for the cctbx. Very useful for handling crystallographic data.
  - Advanced features of flex arrays: An array specifically designed for the cctbx. Very useful for handling crystallographic data.
- API documentation: Browse the API documentation. Useful if you are an experienced user.

Home
Download
How to install
Getting started
Documentation
Table of Contents
Programming tips
Python/cctbx features
Python/cctbx hints
Multiprocessing
High level objects
Maps
Models
HKL viewer
Low level
Tutorials
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

Non-obvious features of Python and cctbx functions

This tutorial describes some non-obvious features of python and of the cctbx.

Is it a new object or a pointer to the old one?

Whenever you say something like:

```
b = [1,2,3]  # a list of values
a = b  # does a get the value of b or is it a pointer
```

you want to ask yourself whether a is a pointer to b (that is, they are the same object with different names), or whether a is a new (separate) object. The reason this is important is that it affects what happens to when you change something about a.

Let's look at this example. We are going to change the second element of a and see what happens to b:

```
print(b)  # prints out [1,2,3]
print(a)  # prints out [1,2,3]

a[1] = 10 # set value of second element of a
print(a)  # prints out [1,10,3]
print(b)  # prints out [1,10,3]
```

So in this case, a is just a pointer to b. We change a or b and the other changes too. We could use the list() function to instead make a new object:

```
d = list(b)  #  make a new list from b
```

Now if we change b, we don't change d:

```
print (d)  # a looks like b: [1, 10, 3]
b[0] = 20  # set value of first element of b
print(d)   # is not a pointer to b: still prints [1, 10, 3]
```

What about a slightly more complicated case where we have an object (a list in this case) that contains other objects (another list):

```
f = [6,7,8]  # f is an object (a list)
e = [1,2,f]  # e is a list with some numerics values and an object (f)
print(e)  # looks like [1, 2, [6, 7, 8]]
```

Let's make a new object with the list() function:

```
g = list(e)  #  make a new list and call it g
```

The result is not exactly what you might expect. The object g is a new list, so as before if you change an element of e you do not change g:

```
print(g)  #  looks like [1, 2, [6, 7, 8]]
e[0]=7  # replace element 0 of e
print(e)  # e is now [7,2,[6,7,8]]
print(g)  # g is still [1, 2, [6, 7, 8]]
```

However if you change something about the object f (change a value in this object), you do change f within both e and g. Let's run this from the beginning:

```
f = [6,7,8]  # f is an object (a list)
e = [1,2,f]  # a list with some numeric values and an object (f)
g = list(e)  #  make a new list and call it g

print(g)  #  looks like [1, 2, [6, 7, 8]]
e[0] = 100  # change element 0 of e
print(e)  # changed:[100, 2, [6, 7, 8]]
print(g)  # still looks like [1, 2, [6, 7, 8]]

f[0] = 32  # change the object f
print(e)  # changed in e:[100, 2, [32, 7, 8]]
print(g)  # the object f within g changes [1, 2, [32, 7, 8]]
```

Note (see below) that if we had instead made a deep copy of e, the object f would have been deep copied as well so that changing f would not have changed g at all.

Here is documentation on how all this works for Python:
https://docs.python.org/2/reference/datamodel.html

Does the member function change the object, return a part, or return something new?

In general when you are working with an object in Python or CCTBX and applying a member function of that object, you have to ask yourself whether you are changing the original object and whether the return value (if any) is a pointer to the original object or a copy that is separate. Some things to keep in mind:

- A member function of an object might change the object itself and return nothing, or
- it might change the object and return a pointer to the changed object, or
- it might return a pointer to some part of the object, or
- it might return a completely new object or value.

The general rules in CCTBX for what to expect about something returned by a member function of some object are:

- If a calculation is carried out in order to get the object that is returned, the object that is returned is a new one and not a pointer to anything in the original object
- If a part of an array is selected with a select() function, a new array is returned, not a pointer
- If the function is just identifying some existing attribute of the object and returning it, a pointer to that existing attribute is returned, and any changes made to the new object also change the original one

Note: If your goal is to change a specific part of a CCTBX object, usually the best approach is to use selections. You select part of the object (some elements), then you then set the values of those selected elements to values that you supply. See the sections on model and flex array selections for how to do this.

In general, you have to look at the documentation or the definition of the object or carry out some tests to find out for sure what is going to happen when you access a member function of a CCTBX object. Here are some examples.

Let's set up a map_data object to work with:

```
from iotbx.ma_model_manager import ma_model_manager # load ma_model_manager
mmm=ma_model_manager() # get an initialized instance of the ma_model_manager
mmm.generate_map() # get a model from a small library model and calculate a map
map_data = mmm.ma_map_manager().ma_data()  # the map as flex.double() 3D array!
```

If we just make an assignment of map_data to a new array, these are the same object and changing one in any way changes the other:

```
map_data_copy = map_data   # just a pointer to map_data
```

If we do deep_copy() map_data, the new copy is totally separate, changing one has no effect on the other. The same holds for using the Python deepcopy() function:

```
map_data_deep_copy = map_data.deep_copy()  #  totally new array
from copy import deepcopy  # import deepcopy
map_data_deepcopy = deepcopy(map_data)  #  totally new array
```

A similar situation occurs frequently in Python. If we have a list and use the sort() member function, this does not return anything. It just changes the list itself:

```
a = [5,3,8]  # a list of numbers
a.sort()  #  sort the list.  Nothing is returned
```

If we instead use the count() member function, a value is returned and the object is not changed:

```
c = a.count(3)  # count values of 3 and return the number
print(c)  # prints 1
```

For a CCTBX flex array, selecting a part of the array generally returns a new object. Let's set up a little 1D array:

```
from scitbx.array_family import flex  # import flex
array = flex.double()  # set up a flex.double() array
array.append(100)  # put in a value of 100
array.append(200)  # and a value of 200
print(list(array))  # prints [100.0, 200.0]
```

Let's select part of the array:

```
sel = (array >= 100)  # identify array elements equal to 100
selected_data = array.select(sel)  # returns new object
print(list(selected_data))  # prints [100.0]
```

Now if we change the original array, the part that was selected and returned is not changed:

```
array[0] = 23 # change original array

print(list(selected_data))  # prints [100.0]
```

Many CCTBX objects have member functions that return the result of some calculation. In these cases the object that is returned is a new object, and changes made to this new object do not affect the original one. For example, complex_double() objects have a function part() that returns the real and imaginary parts of the object:

```
complex_array = flex.complex_double()# a complex double array
complex_array.append((2.0,3.0))  # append the complex number (2.0,3.0)
complex_array.append((2.0,4.0))  # append the complex number (2.0,4.0)
a,b = complex_array.part()  # pointers a and b to the real and imaginary parts
print(list(complex_array))  # print out the array: [(2.0,3.0), (2.0,4.0)]
print(list(a), list(b))  # prints [2.0,2.0],[3.0,4.0]
```

The parts of the complex array are calculated with the part() function. They are not existing attributes of the complex array. Now if we change a or b, the original is not changed:

```
a[1] = 90 #  change pointer to a
print(list(complex_array))  # still prints out [(2.0,3.0), (2.0,4.0)]
```

Note that this result is not obvious. If the flex.complex_double() type happened to store complex numbers as two lists, one of the real parts and one of the complex parts, then the part() function would have returned pointers to each of these two lists and changing a or b would have changed the original.

Other CCTBX functions have member functions that return a pointer to an existing attribute. A map coefficients array has attributes of indices and data. You can get a pointer to the data with the function data(). Let's get a map coefficients object:

```
map_coeffs = mmm.map_manager().map_to_fourier_coefficients()  # map coeffs
print(map_coeffs.data(100))  # (23.1320326409-33.1246074018j)
```

And use the data() function to get a pointer to the data:

```
data = map_coeffs.data()# The map coefficients themselves
print(data[0])  # The first map coefficient (23.1320326409-33.1246074018j)
```

If we change the value of data[0], it will change the map coefficients too:

```
data[0] = (10+0j)  # set value of data[0]
print(map_coeffs.data(100))  # prints (10+0j)
```

Note that the map coefficients object has another function called phases() that seems somewhat related but that performs a calculation instead of returning an existing attribute. Let's get the return result from the phases() function:

```
phases = map_coeffs.phases()  # new object with indices and phases only
```

The contents of phases.data() have been calculated from map_coeffs.data() and changing the information in phases.data() has no effect on the values in map_coeffs.data().

A final example is a comparison of the member functions as_int() and as_float() for a 3D flex array. The as_int() function returns a new object in which the data is just a pointer to the original. The as_float() function that seems similar on the surface has a different behavior: it performs a calculation and returns a new object in which the data are completely separate from the original data:

```
map_data = mmm.map_manager().map_data()  # 3D flex.double array
map_data_as_in4 = map_data.as_int()  # new object, data are shared
map_data_as_float = map_data.as_float()  # new object, new data
```

Let's print out starting values:

```
print(map_data[0], map_data_as_in4[0], map_data_as_float[0]) #
```

Now set the value of map_data and see if the others change:

```
map_data[0] = 999.  # set map_data
print(map_data[0], map_data_as_in4[0], map_data_as_float[0]) #
```

The map_data and map_data_as_5d both change, but the map_data_as_float is not changed.

Using Python deepcopy or a deep_copy() member function

One way to be sure that the object you are working with is a completely new one is to use the Python deepcopy() function or a member function of a cctbx class called deep_copy(). Each of these makes a new copy. Each of these also increases memory usage by the size of that new copy.

You use the Python deepcopy() function only when you really need it. It makes a completely new copy of whatever you specify:

```
from copy  import deepcopy  # import deepcopy
c = [1,2,[6,7,8]]  # a list with some values and a list
y = deepcopy(c)  # completely new copy of c. Change c; nothing happens to y
```

To see the use of deep_copy() in a cctbx object, let's set up a map_data object:

```
from iotbx.ma_model_manager import ma_model_manager # load ma_model_manager
mmm=ma_model_manager() # get an initialized instance of the ma_model_manager
mmm.generate_map() # get a model from a small library model and calculate a map
map_data = mmm.ma_map_manager().ma_data()  # the map as flex.double() 3D array!
print(map_data[27])  # prints arbitrary value of -8.81152407138008
```

We can create a pointer to map_data:

```
map_data_pointer = map_data  # just points to map_data
```

or make a deep copy of map_data:

```
map_data_deep_copy = map_data.deep_copy()  # completely new data
```

If we change map_data it changes map_data_pointer but not map_data_deep_copy:

```
map_data[27] = 100  # set value of map_data
print(map_data_pointer[27])  # prints 100
print(map_data_deep_copy[27])  # prints original value of -8.81152407138008
```

Note that making a deep_copy uses as much new memory as the original array, while using a pointer uses almost no memory. Therefore you want to make a deep_copy only when it is necessary.

Do not use `if (not x)` or `if (x)` expressions to test whether x is defined. Use `if (x is not None)`

In many cases you can get away with saying:

```
x = None
if (not x):  # don't use this
  print("Then x can be 0, None, False, "", [], {}, (),.....""")  # happens if x is 0, None, False
```

but you should not use this construction because it is not clear whether you really meant to do something if x is zero, or if you meant to do it if x is None, or if you meant to do it if any of these occur.

Instead be clear and say:

```
if (x is None):  # Use this instead
  print("x is None")  # happens if x is None (only)
```

Similarly be clear and say:

```
if (x is not None):  # Use this too
  print("x is not None")  # happens unless x is None (only)
```

Default in a function call set to an empty list (don't do this)

You might be tempted to make a function call with a default argument that is an empty list, like this:

```
def my_bad_function(value, current_list = []):  # don't do this
  current_list.append(value)  # current_list from previous call
  return current_list  # returns current_list
```

This is supposed to use an empty list as the value of current_list if current_list is not supplied. This works the first time you do it:

```
print(my_bad_function(1))  # prints [1],...current_list was []
```

However the second time you call it without specifying the value of the default current_list, the value of current_list from the first call gets used:

```
print(my_bad_function(2))  # prints [1, 2] ...current_list was [1]
```

You can get the behavior that (presumably) you want by using None in the function call and setting the value of current_list to [] if its value is None:

```
def better_function(value, current_list = None):  # ok way
  if current_list is None:  # catch uninitialized current_list
    current_list = []  # set its value to []
  current_list.append(value)  # works
  return current_list # returns current_list

print(better_function(1))  # prints [1],...current_list was []
print(better_function(2))  # prints [2],...current_list was []
```

Use map_data.tricubic_interpolation with caution if your map is not periodic

Some CCTBX map functions assume that the map repeats indefinitely with a periodicity equal to the size of the map in each direction. This can sometimes result in map functions returning values that are not what you expect. An example is the tricubic_interpolation function.

You can access the value of a map at a grid point simply by referring to it: value = map_data[i,j,k] gives the value of the map at the grid point (i,j,k).

You can get interpolated values of a map with the tricubic_interpolation function, which works on fractional coordinates. If your map has a size of (10,10,10), then interpolated_value = map_data.tricubic_interpolation(0.1,0.2,0.3) will give you the value at the same grid point (and these values will match).

Now what happens if we use grid points or fractional coordinates that are outside the bounds of the map? A CCTBX map with a size of (10,10,10) often would have gridding running from (0,0,0) to (9,9,9). The grid point (10,10,10) typically is assumed to have a value the same as the grid point at (0,0,0). This is referred to as "wrapping", as the map wraps around with a period equal to the size of the map.

If you try to access the value of the (10,10,10) grid point with value = map_data(10,10,10) you will get an error saying that an index is out of range (all three are out of range because the map only goes from 0 to 9 in each direction.)

On the other hand, if you try to access the value at the same place using interpolated_value = map_data.tricubic_interpolation(10.0,10.0,10.) you will get the value of the map at (0,0,0).

Why the difference? The tricubic_interpolation assumes that the map is wrapped. The interpolation process uses map values at grid points around the (x,y,z) coordinate to estimate the value at (x,y,z). If one of these grid points falls inside the bounds of the map, the value is taken directly from the map. Any grid points that do not fall inside the bounds of the map (0 to 9 in each direction in our example) are mapped inside the bounds by adding or subtracting multiples of 10 (or whatever the number of grid points along each direction).

Consequently the tricubic_interpolation function will give the same answers for (x,y,z) and (x+i, y+j, z+k) where i, j and m are any integers.

If your map is actually not wrapped, that is, your map does not repeat indefinitely, you need to make sure you do not use tricubic_interpolation to refer to any points outside the bounds of your map. Additionally, be aware that tricubic_interpolation can give incorrect values for points that are near the edges of your map because it may use values from the opposite face of the map when it is interpolating.

d_spacings() method of the miller array object has -1 for f000 term

The value of f_map.d_spacings().data() is -1 for f000 term and not infinity,where f_map is a set of map coefficients (particularly note for those created from a map). Be sure to have a special case for handling this if an f000 term may be present (as is usually the case for cryo_EM maps).
For example:

```
# The d_spacings get a value of -1 for the f000 term. Set it to a big number
d00 = f_map.d_spacings().data()
d00.set_selected(d00 == 0, 1.e+40)
sphere_reciprocal = f_map.q_function(f=rad_smooth) # top hat function
```

Home
Download
How to install
Getting started
Documentation
Table of Contents
Programming tips
Python/cctbx features
Python/cctbx hints
Multiprocessing
High level objects
Maps
Models
HKL viewer
Low level
Tutorials
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

Helpful CCTBX and Python hints

This tutorial describes how to take advantage of features of python and of the cctbx.

How to add the values in two tuples together
Using the tuple() method to display cctbx arrays
Carrying out vector operations with the col method
Using the parts() function to access parts of a complicated object
Always return a group_args object if your method returns more than one item
Use flex arrays to speed up operations on elements of an array
Using the min_distance_between_any_pair_with_id method to get the closest pair of atoms in two sets of coordinates
Indexing python and cctbx arrays
Slicing python and cctbx arrays
Using the approx_equal() function to compare numbers or arrays

How to add the values in two tuples together

You might want to add the values in two tuples together:

```
a = (1,2,3)  # a tuple of values
b = (6,6,6) # another tuple of the same length
```

but there is no standard operation that will do this for you. A simple solution is to use the cctbx "col" method:

```
from scitbx.matrix import col  # import the col methods
c = col(a) + col(b)  #  col objects of the same length can be added together
d = tuple(c)  # now d is a tuple again
print(d)   #  now we get (7,8,9)
```

Using the tuple() method to display cctbx arrays

If you try to print out a cctbx array, usually you will get something that is not so useful:

```
from scitbx.array_family import flex # import the flex methods
a = flex.double((1,2,3))  # make a flex array
print(a)  # print it  # you get something like
print(tuple(a))  # now you can see what is in it: (1.0, 2.0, 3.0)
```

Carrying out vector operations with the col method

The scitbx method "col" has many handy vector operations. Here are some ways to use them

Let's get the dot product of two vectors:

```
from scitbx.matrix import col  # import the col methods
a = (1,2,3)   # a tuple
b = (5,3,9)   #  another tuple
d = col(a).dot(col(b))  # dot product of a and b
print(d)          # dot product is 38
```

There are other handy operators:

```
from scitbx.matrix import col  # import the col methods
a = (1,2,3)  # a tuple
b = (5,3,9)  # another tuple
ca = col(a)  # col object based on a
cb = col(b)  # col object based on b
print( ca.dot(cb))  # dot product = 38
print( tuple(ca.cross(cb)))  # cross product = (9, 6, -7)
```

Note the use of tuple() to convert the cross product to a tuple.

Using the `parts()` function to access parts of a complicated object

A number of low-level cctbx objects have a member function called `parts()` that returns individual flex arrays corresponding to specific parts of the overall object. For example, `parts()` allows access to the real and imaginary parts of a complex array:

```
complex_array = flex.complex_double() # a complex double array
complex_array.append((1+2j))   # append the complex number (1+2i)
complex_array.append((23-6j))   # append the complex number (23-6i)
a,b = complex_array.parts()  # pointers a and b to the real and imaginary parts
print(list(complex_array))  # print out the array: [(1+2j), (23-6j)]
print(list(a), list(b))  # prints ([1.0, 23.0], [2.0, -6.0])
```

Let's set up a map_model_manager object that we can use as an example..:

```
from iotbx.map_model_manager import map_model_manager # load map_model_manager
mmm=map_model_manager() # get an initialized instance of the map_model_manager
mmm.generate_map()# get a model from a small library model and calculate a map
map_coeffs = mmm.map_manager().map_as_fourier_coefficients() # get map coeffs
complex_double_array = flex.complex_double()  # a complex double array
indices = map_coeffs.indices()1  # array of indices
sites_cart = mmm.model().get_sites_cart()  # coordinates
```

Some cctbx objects with a `parts()` member function include:

- `a,b = complex_double_array.parts()` # real and imaginary arrays
- `x,y,z = sites_cart.parts()` # individual arrays for x, y, z
- `x,y,z = flex.vec3_double().parts()` # any vec3_double_array

Some other cctbx objects have specific methods to get individual parts:

- `phases = map_coeffs.phases(deg=True)` # phases
- `amplitudes = map_coeffs.amplitudes()` # amplitides
- `indices = map_coeffs.indices()` # indices of Fourier coeffs
- `h,k,l = indices.as_vec3_double().parts()` # individual arrays for indices h,k,l
- `fourier_data = map_coeffs.data()` # Fourier coefficients (complex)

Always return a `group_args` object if your method returns more than one item

If you have a method that returns a couple items, you might be tempted to just return them by themselves:

```
def return_a_and_b(a,b): # simple function
  return a,b   # just return a couple values
```

Don't be tempted. Return instead a `group_args` object:

```
def return_a_and_b(a,b): # simple function
  from libtbx import group_args   #  import group_args
  result = group_args(                 #
    group_args_type = 'just returning a and b',  # a name for this group_args
     a = a,   #   can refer to a as result.a
     b = b)   #  and to b as result.b
  return result  #
```

Why? Because you are going to add a third, a fourth, a tenth...return item and then it is a mess if you return them as a list. Return a group_args object and they are all packaged together nicely and they even have a label:

```
result = return_a_and_b(1,2)   # call our function
print(result)   # prints out value of a and b and the label
print(result.a)   # print out value of a
```

Use flex arrays to speed up operations on elements of an array

Flex arrays operate using optimixed C++ code, while Python loops are run without much optimization and take as much as 100 times as long. Try to take advantage of flex arrays whenever you can. Here is an example of a little method to find the furthest distance from the center of a set of points:

```
def furthest(sites_cart):  # furthest from center
  center = sites_cart.mean()   # flex method to get mean of vec3_double() array
  diffs = sites_cart - center  # subtract a vector from a vec3_double() array
  norms = diffs.norms()        # get an array of lengths of the diffs array
  return norms.min_max_mean().max  # get maximum value

print(furthest(sites_cart))  # print the result
```

Note that this carries out all the little summations and squaring in C++, not Python, so that it is very fast.

Using the min_distance_between_any_pair_with_id method to get the closest pair of atoms in two sets of coordinates

You might have two sets of coordinates (maybe two sets of CA positions) and you want to identify the closet pair taking one from each set of coordinates. There is a handy function for this:

```
from scitbx.array_family import flex # import the flex methods
a = flex.vec3_double((1,1,1),(2,2,2), (3,3,3)))  # make a flex array
b = flex.vec3_double((3,2,2),(1,1.2,1.2),(2,2,2), (4,3,3)))  # make a flex array
dist, i, j = a.min_distance_between_any_pair_with_id()  # find closest pair
print(dist, i, j)  # prints  (0.2999999999999993, 0, 1)
```

...the first element of array a is close to the second (index 1) of array b

Then you might want to do something with these two coordinates that are close:

print(a[i], b[j]) # prints ((1.0, 1.0, 1.0), (1.1, 1.2, 1.2))

Indexing python and cctbx arrays

You often will want to extract one element from a python or cctbx array. This section gives some helpful hints on doing this. Python and cctbx work the same way for arrays. Suppose we have a very small array:

```
a = [5,6,7]  # a list of 3 values
```

Now we might want the first element of this array:

```
print(a[0])  # prints 5 (indexing starts at zero)
```

Or the third element of this array:

```
print(a[2])  # prints 7
```

You can also get the third element of this array by counting backwards from the end, where the "-1" index is the last one:

```
print(a[-1])  # prints 7
```

The way this works is that the index "-1" is first added to the number of elements in the array and then that is the new index:

```
index = -1
index_to_use = len(a) + index   # this is done in the background
print( index, index_to_use, a[index],a[index_to_use]) # prints (-1, 2, 7, 7)
```

As you would expect, you can count backwards to the first element in the array:

```
print(a[-3])  # prints 5
```

What happens if you count back too far (taking the fourth from the end)? You get an error:

```
print(a[-4])  # prints IndexError: list index out of range

```

Hint: if you use negative indices, always think of them as their values plus the length of the array. Any resulting value less than zero or equal to or larger than the length of the array is not allowed.

Slicing python and cctbx arrays

You often will also want to extract more than one element from a python or cctbx array. This can be non-intuitive in some situations. This section gives some helpful hints on doing this. Python and cctbx work the same way for arrays. Suppose we have a very small array:

```
a = [5,6,7]  # a list of 3 values
```

Now we might want the first two elements of this array. You get them with a slice:

```
print(a[:2])  # prints [5,6]
```

... or all the elements after the first two elements of this array:

```
print(a[2:])  # prints [7]
```

Note that you can conveniently split up the array with the first k elements in one new array and the remaining elements in another:

```
k = 2
print(a[:k],a[k:])  # prints ([5, 6], [7])
```

Note that if you slice starting at element k:

```
print( a[k:])  # prints [7]
```

the resulting array starts with the same value you would get if you asked for element k:

```
print( a[k])  # prints 7
```

You can ask for a range of indices as well:

```
print( a[k:k+1])  # prints [7]
```

This can lead to some non-obvious behaviour if you are using negative indices to count backwards from the end of the array. Let's ask for a slice starting at index -1:

```
print( a[-1:])  # prints [7]
```

You get the last value of the original array. Now let's ask for a slice from -1 to 1. You might think that this would give you the first element of the array, but it gives you an empty array:

```
print( a[-1:1])  # prints []
```

Think of negative indices as the index plus the length of the array. So the -1 value is converted to the length of the array (3) plus -1, or 2. The expression we just used is equivalent to:

```
print( a[2:1])  # prints []
```

which is an empty array because there are no elements that are after the first two elements in the array and in the first one element of the array.

Using the `approx_equal()` function to compare numbers or arrays

The `approx_equal()` function is a very handy tool for comparing results of calculations. You can use it in your tests to make sure that the results are about the same as you expect, or in regular code to compare numbers or arrays. Here are some examples:

```
from libtbx.test_utils import approx_equal # import it
print( approx_equal(1,1.001))  # prints False (not same within machine precision)
print( approx_equal(1,1.+1.e-50))  # prints True (within machine precision)
print( approx_equal(1,1.001, eps = 0.1)) # prints True (within 0.1)
a=flex.double((1,2,3))  # set up an array
b = a + 0.0001   # another array that is just a little different
print( approx_equal(a,b) )  #prints False and lists differences
print(  approx_equal(a,b, eps=0.001))  # prints True (all elements within 0.001)
```

Home
Download
How to install
Getting started
Documentation
Table of Contents
Programming tips
Python/cctbx features
Python/cctbx hints
Multiprocessing
High level objects
Maps
Models
HKL viewer
Low level
Tutorials
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

How to run any method with multiprocessing

This tutorial describes how to convert (almost) any method to run with multiprocessing

Setting up a very simple method with multiprocessing
Advanced multiprocessing made easy

Setting up a very simple method with multiprocessing

Although setting up multiprocessing can seem intimidating, it is actually amazingly easy to do in most cases. We are going to set up an example where we first run without multiprocessing and then we convert the method to run with multiprocessing.

The key to easy multiprocessing is setting up your function in a certain way. We are going to separate the function that does something from the list of inputs to that function (which are going to change for the N jobs that are run).

First, let's set up some imports:

```
from __future__ import print_function  # so we can use print function easily
import sys        # import sys
from libtbx import group_args #  group_args function we will use to return items
```

We have imported the `print_function` here so that this method will work in both python 2 and python 3.

Here is our function:

```
def run_something(value):  # simple function
  return value * 2    # just returns 2 * the input value
```

And here is our list of inputs to the function that we would like to run in parallel:

```
iteration_list = [5,7,9]  # list of anything
```

Let's first set up this calculation running one by one through the values in the values in `iteration_list`. It is a good idea to start this way even if you know from the start that you want to run with multiprocessing because once you have set it up in the way we have it below it is very easy to convert.

```
def run_one_by_one(iteration_list): #
  result_list = []                          # initialize result list
  for i in range(len(iteration_list)):      # iterate over input values
    result = run_something(iteration_list[i])  # get result
    result_list.append(result)  # save result
  return result_list   # return the list of results
```

We can run `run_one_by_one` and get a list of results:

```
result_list = run_one_by_one(iteration_list)  # get results one by one
print(result_list)  # prints list of values  [10, 14, 18]
```

Now let's convert the `run_one_by_one` method to run with multiprocessing.

```
def run_in_parallel(iteration_list, nproc = 4): #
  from libtbx.easy_mp import simple_parallel  # import the simple_parallel code
  result_list = simple_parallel(      # run in parallel
    iteration_list = iteration_list,  # our list of values to run with
    function = run_something,         # the method we are running
    nproc = nproc )                   # how many processors
  return result_list   # return the list of results
```

Now we can run `run_in_parallel` and get a list of results:

```
result_list = run_in_parallel(iteration_list, nproc = 4)  # run in parallel
print(result_list)  # prints list of values  [10, 14, 18]
```

Wow that was easy. All we did was replace the explicit iteration over our iteration_list of values and calls to run_something with a call to simple_parallel with these as arguments and specifying how many processors to use.

Bottom line on this: set up your calculation as an iteration over some list of values where the results are appended to a list and the list of result is returned. Then you can use simple_parallel without changing anything else at all.

Advanced multiprocessing made easy

You can set up multiprocessing in a more complicated situation pretty easily as well. Suppose we want to run a method that is going to do something with a really big input (like a map) that is going to be the same for every job, and some inputs that are going to be different for each job (like a different set of coordinates each time).

In this example the key features are the big object (which we do not want to copy and we do not want to write to disk and we may not be able to pickle) and some list of sets of inputs, one set of inputs for each job.

This is going to look a lot like the easy example, but it will have some new features.

Here is the function we want to run. We are going to call our fixed big object `big_object`. (It can have any name, and there can be any number of such inputs, or none of them). We are going to call this function with an `info` object (a `group_args` object) that is going to be different for each job and that contains values of `info.value` and `info.index`. (Once again, this info object can contain anything you want.) In this case, we are going to multiply `info.value` by two and add the value of `big_object` at index `info.index`:

```
def run_advanced(info, big_object = None,  #
     log = sys.stdout):   #  we can specify the log in this method if we want
  output_value = info.value * 2 + big_object[info.index]   # our result
  print("Value: %s Index: %s Output value: %s" %(info.value, info.index, output_value), file = log
  return group_args( #
    group_args_type = 'Result from one job',  #
    input_info = info,   #
    output_value = output_value,)   #
```

In this example function we have added two new features:

- A log stream. If you add the keyword `log = sys.stdout` to your function and have print statements in your function that print to `log` then the `simple_parallel` procedure will handle all the text that is printed.
- We are returning a `group_args` object and not a simple value. The use of `group_args` for a return value allows returning multiple items in a simple way and allows saving and printing output text.

Note that we are returning the input `info` object as an element of the result `group_args` object. We don't have to do that, but it shows that you can return something that will identify which output came from which inputs.

Here is how we are going to set up to run this without multiprocessing. This time we are going to make our `iteration_list` be a list of the `info` `group_args` objects that specify what inputs are to be used in each job, and set our `big_object` to be a little array:

```
iteration_list = []   # initialize
from libtbx import group_args
for i in [5,7,9]:  # our values to vary for each job
  iteration_list.append(   # a list of info objects
    group_args(   # info object (group_args)
      group_args_type = 'value of info for one job',   # title
      value = i,   # value of value
      index = 2)   # value of index
    )   #

big_object = [0,1,2,3]  # just some supposedly big object
```

... and we can run it like this:

```
def advanced_run_as_is(iteration_list,  #
     big_object = None,  #
     log = sys.stdout): # run in usual way
  result_list = []   # initialize
  for i in range(len(iteration_list)):     #  iterate through jobs
    result = run_advanced(iteration_list[i],
      big_object = big_object,
      log = log)  # run job
    result_list.append(result)   #
  return result_list    # return list of results
```

We can run this and print out each result as a `group_args` object. Notice how convenient it is that each `result` records both the inputs and the output.

The use of the `log` keyword allows us to capture the log output if we want to.

```
result_list = advanced_run_as_is( #
   iteration_list, big_object = big_object,
   log = sys.stdout) #
for result in result_list:  # run through results
  print("\nOne result:\n%s" %str(result))  # print this result (it is a group_args object)
```

Now we can run the same job in parallel with just tiny changes:

```
def advanced_run_in_parallel(iteration_list,  #
      big_object = None, nproc = 4, log = sys.stdout): # run in parallel w
  from libtbx.easy_mp import simple_parallel  #
  result_list = simple_parallel(  #
    iteration_list = iteration_list, # list of varying inputs
    big_object = big_object, # any number of keyword arguments allowed
    function = run_advanced,  # function to run
    nproc = 3,   # number of processors
    verbose = False,   # non-verbose output
    log = log,
    )
  return result_list
```

With `verbose = False` the output from the first job run (actually the first batch of jobs if more than one job is in each batch as will be the case if there are many more jobs than processors used) is sent to the log stream (here that is `sys.stdout`) and all the other output is ignored. If we had set instead set `verbose = True` then all the output would be printed (hard to read but useful for debugging).

We can now run and print out each result as a `group_args` object:

```
result_list = advanced_run_in_parallel(  #
    iteration_list, #
    big_object = big_object,  #
    log = sys.stdout) #
for result in result_list:  # run through results
  print("\nOne result:\n%s" %str(result))  # print this result (it is a group_args object)
```

Home
Download
How to install
Getting started
Documentation
Table of Contents
Programming tips
Python/cctbx features
Python/cctbx hints
Multiprocessing
High level objects
Maps
Models
HKL viewer
Low level
Tutorials
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

Phenix and CCTBX Programming Suggestions

Webpages
Run tests on your code
Make tests for your code
Write code that is compatible with mmCIF format
Be aware of how the objects in Phenix/cctbx work
Safe use of map_model_manager, map_manager, and model in Phenix

This site gives helpful programming tips to get started with cctbx.

Webpages

- Unit tests
- Detailed expectations about contributing to CCTBX
- How to make a Program template
- cctbx programming tips
- Features of high-level cctbx managers:
  - cctbx documentation overview
  - Phenix documentation overview
  - Phenix quick reference
- Features of cctbx maps
- Read about conversion to mmCIF-compliant code here.

Run tests on your code

Before committing anything to the Phenix or CCTBX repository, run all the tests and make sure they all pass.

```
phenix_regression.test_all_parallel nproc=72 sort
```

When done, test_1.log will have a summary of your tests. The "sort" runs slow tests first.

If you are in a cycle of development where a lot of tests are failing and you want to run just tests that are affected by changes you have made and have not committed, use the keyword "git" instead of "sort". It will only run a subset of tests.
Note: be sure to run all tests before you commit your changes.

Make tests for your code

- "If it isn't tested, it doesn't work"
- CCTBX methods: Make a unit test for every functional unit.
- Phenix methods (often high-level): Make unit tests if feasible, otherwise make very quick-running tests that exercise all important functionalities.
- You can read how to make a unit test here
- You can read detailed expectations about contributing to cctbx here

Write code that is compatible with mmCIF format

Overall suggestions

- Use the Program template and do all other read/write with the data_manager present in the template. You can read how to make a Program template here:
- When you write a file, capture the actual file name (which may end in .cif):
  `fn = self.data_manager.write_model_file(model, target_file_name)`
- Create tests with models that cannot fit in PDB format. You can do this with pdbtools:
  `phenix.pdbtools model.pdb output.file_name=model.cif old_id=A new_id=AXZLONG`
- When you have Phenix set up, you can see details of how to write mmCIF-compliant code by typing
  `phenix.pdb_cif_conversion`

Notes for existing code

- Convert to using the Program template and the data_manager if possible.
- Convert all the following to use the data_manager (none are mmCIF-compliant):
  - model.model_as_pdb()
  - hierarchy.as_pdb_string()
  - hierarchy.write_pdb_file()
  - For models, use:
    `fn = self.data_manager.write_model_file(model, target_file_name)`
  - For hierarchies, write out the model that it is part of instead.
  - If your hierarchy is not part of a model, you can use:
    ```
    m = hierarchy.as_model_manager(crystal_symmetry = crystal_symmetry)
    file_name = 'mypdb.pdb'
    fn = self.data_manager.write_model_file(m, file_name)
    ```
- Find and fix problems in myfile.py with:
  `libtbx.find_pdb_mmcif_problems myfile.py`
- Code that parses PDB-formatted text should be rewritten as it is not mmCIF-compatible.

Be aware of how the objects in Phenix/cctbx work

"Is it a new object or a pointer to the old one?"" This is an important but subtle feature of Python programming. Any time you make an object you need to be able to answer this question.

Member functions of cctbx objects sometimes return a new object and other times return a pointer to the old one. This makes a difference because if it is a pointer when you change the object it also changes the original.

CCTBX functions may change the object itself, or return a part of the object, or return something entirely new.

You can read all about this here.

Safe use of map_model_manager, map_manager, and model in Phenix

These managers are available when you use the Program template for your programming and your parameters include maps and models.

Get your manager with the Program template and data_manager

- The data_manager automatically provides access to a map, model, and two half maps if the user supplies them
- Normally just get a map_model_manager. This will be ready to use:
  `mmm = self.data_manager.get_map_model_manager(from_phil=True)`
- if you have just a model, get a model manager
  `m = self.data_manager.get_model(self.params.input_files.model_file)`

Use the methods of a manager to do what you want to do

- Mask all the maps around atoms in the model:
  `map_model_manager.mask_all_maps_around_atoms()`
- Get a model-building object that knows about the map and model:
  `mb = map_model_manager.model_building()`
- Get the model and map
  `m = map_model_manager.model() mm = map_model_manager.map_manager()`
- You can see all the features of these managers here.
- You can also see many specific ways to use these managers here and here.

The model or map you get from a map_model_manager is the same model or map that is still in the manager

- If you change a model or map you are changing the ones in the map_model_manager too
- If you want an unattached copy, you need to make a deep_copy().
- Some attributes of a model you might reasonably change: sites_cart, occ, b, u. These do not affect compatibility with other objects in a map_model_manager
- Some attributes of a model you should not change if it is part of a ap_model_manager: crystal_symmetry, unit_cell_crystal_symmetry, shift_cart. These are handled in the map_model_manager and affect all objects in the manager.
- You can check to make sure your map_model_manager is in sync using the method: `map_model_manager.check_consistency()` which will stop if there is anything not consistent.

Boxed maps and origin shifts

Some maps have their origin at xyz = (0,0,0), but others may have an origin at some other location, usually because they have been cut out of a larger map (they have been boxed).

You can read all about this here.

- If you have a model all by itself, or a map which has its origin on xyz = (0,0,0), origin shift is (0,0,0). All coordinates are the same as in an mmCIF or PDB file.
- The origin shift comes up if you have a map that does not have its origin at xyz = (0,0,0). If the map has an origin at xyz=(100.,0,0) then the origin shift is (-100.0,0,0). The origin shift is applied to all coordinates in all models and to the map.
  In effect, the models and maps are all shifted to place their new origin at xyz=(0,0,0). Then you can work with them and the maps and models will have the right relationships. Origin shifts in Phenix always correspond to an integral number of grid units in the map.
- The map_model_manager, model manager, and map_manager all keep track of the origin shift. You can get it with mmm.shift_cart(), model.shift_cart(), etc.
- If you write out a model or map it will be shifted back to its original place when written to your output file unless you specify otherwise.
- The hierarchy object does not know about origin shifts. If you extract a hierarchy from a model and write it out like this (not recommended):
  `text = model.get_hierarchy().as_pdb_or_mmcif_string()` then it will write the internal coordinates without shifting them back to their original location. Better to make this clear with the model object which does know about origin shifts and specifying if you want to shift back (to original location; this example does not shift back):
  `text = model.as_pdb_or_mmcif_string(do_not_shift_back=True)`

Contributing to the Computational Crystallography Toolbox (cctbx)

cctbx Developer Guidance

Table of Contents

- Short version
- Long version
- Guidance for Developing Tests
  - Location
  - Name
  - Run Time
  - 3 steps for adding steps
  - Guidelines for writing tests
  - Some useful cctbx tools for tests
  - How to use models and data
  - Using non-cctbx modules
  - How to handle errors
  - Miscellaneous
  - Example for a test
  - Good etiquette

Short version

As discussed at the Gordon Research Conference for Diffraction Methods in Structural Biology, GTB2018, the following guidelines are agreed upon for developers wishing to contribute code to cctbx:

- cctbx is open source.
- We welcome third party contributions, but note that there is a cost for managing third-party contributions.
- cctbx is not a free-for-all.
- Contributors should be encouraged to review their plans for contributions before doing a large amount of work and submitting a pull request, but that shouldn't preclude users from submitting requests.
- Interface changes and other breaking changes should be communicated to the bulletin boards (cctbxbb@phenix-online.org for example).

Expectations for contributions

- Include documentation: docstrings for functions, in-line comments, and help strings for command line programs.
- Include tests. Testing includes unit tests (individual functions) and integration tests (full program), and possibly multiple of each with different inputs.
- Run the cctbx tests. If appropriate, run tests from other projects affected by the new code.
- Do not add external dependencies without discussion.

Expectations for developers

- We will respond to contributions, at least with "I don't have time to review this right now".
- We can deny pull requests, but if we do so, we should do so with constructive feedback, following consistent rules.
- The build meeting team will review tickets and pull requests to ensure regular feedback.

Long version

The Computational Crystallography Toolbox (cctbx) is a large code base under active development by several groups worldwide. There are more than 1 million lines of code, 500 modules, and 20 active developers. It is therefore very important that all contributors follow some basic guidelines. Keep in mind that the intention is for the cctbx to provide a fully featured code base for crystallographic calculations while also remaining lightweight and straightforward to compile and install.

1. No new dependencies unless absolutely necessary

- It is important to keep the cctbx easy to compile and install. The inclusion of third-party packages that have their own dependencies is therefore strongly discouraged. This applies especially if a dependency on a new compiler language is introduced.
- Check if the desired functionality already exists in the cctbx.
- Ask other developers if the functionality exists already in the code base.
- If a new dependency is needed, **discuss with the other developers** before adding it.

2. Avoid code duplication

- It is often the case that the desired functionality already exists somewhere in the cctbx. Therefore:
  - Review the code base prior to implementing new functionality.
  - Ask other cctbx developers if they know that this functionality exists already
- Add new context independent functions into appropriate modules, not into specialized code. Example: a place for the function that calculates distance between two points is scitbx/matrix/, because it may be very tempting to inline this function into a specialized code as needed thus creating code duplication.
- Use constants from a central place. Example: use `math.pi` instead of defining `pi = 3.14` every time it is needed. Note: there are plenty of other constants available, such as `scitbx/constants/tiny_pt.hpp`, add more as needed. (Make sure not to use an OS-dependent constant)

3. Use the cctbx coding style

- The Python Style guide is generally useful Note however that where cctbx deviates from PEP8, follow cctbx (for example using 2 spaces to indent instead of 4).
- Constructs should be used to make subsequent use and testing as easy to debug as possible (i.e. standard out and standard error output)
- Printing output
  - Use `print(file, f"{args}")` and not `print(file, ...)`.
  - Use `show` function to print a summary or result of code execution instead of in-lining print statements directly into the code.
  - Avoid unconditional printing.
- Consistency. If editing an existing file, follow the code style of that file.
- Code clarity. Ideally, clearly written code does not need documentation (however, even clearly written code may benefit from documentation). Having this in mind:
  - Use self-explicable functions/classes/arguments/file names. A clear long name is better than a short cryptic name.
  - While valid exceptions exist, generally if a function does not fit a page then it is likely to benefit from being broken into smaller functions.
- Common modern computation courses emphasize the importance of in-code documentation
  - Since what is clear to one person might not so clear to another (or to the same person after several months or years).
  - Auto documentation creation (like sphinx) use the in-code documentation strings.
  - Time saving: a developer can read the documentation to understand what a function does, what are the arguments' types and formats and what it returns. This is much better than having to read and figure out what a function needs and what it does.
  - Longevity: Since developers and collaborators change over time and since the cctbx and related systems such as Phenix and DIALS are quite complex software, uncommented code might be a barrier for continuous development.
  - Update documentation to reflect code changes
- Developers are encouraged to correct severe deviations from coding styles found anywhere in codebase.
- Developers are encouraged to modify code comments when unclear, outdated or in a format that does not render well in the SPHINX automated documentation.

4. Run find clutter before commits

- `libtbx.find_clutter` primarily checks for a few common errors:
  - No from `__future__ import division`: Use `libtbx.add_from_future_import_division` to fix. Important: always use the `//` operator to indicate integer division the `/` operator is exclusively for floating point division.
  - Tabs or trailing whitespace: use `libtbx.clean_clutter` to fix.
  - Unused imports: use `libtbx.find_unused_imports_crude` to find them and remove them.
  - The use of bare except statements is prohibited, since it causes the `try...except` to catch `KeyboardInterrupt`. At a minimum use `except Exception`.
- Before submitting your code, be sure to test it again after fixing problems by running `libtbx.find_clutter`.

5. Help and guidance

- Use the cctbx mailing list to ask for guidance from other developers, and locate specific features in the current code base.
- The other cctbx developers are an invaluable resource that can be used to help with getting started in cctbx development.
- Link to the mailing list: http://www.phenix-online.org/mailman/listinfo/cctbxbb

6. Send an email to the mailing list stating the intent to submit a new code into the cctbx

- There are several sub projects within the cctbx, often embedded in the form of code trees within the main cctbx project. New code trees should not be introduced without good reason, and if abandoned the code tree should be removed. This reduces the accumulation of distracting code in the cctbx over time.
- Periodically, unused code trees may be removed from the cctbx to minimize clutter.

7. Subscribe to git check-in alerts sent via email (code changes alerts) and review the diffs.

- This will minimize surprises later when someone changes someone's code.
- Any inefficiencies or bugs spotted in diffs should be pointed out to a respective contributor or fixed by anyone who found them first.

8. Python do's and don'ts:

- Use inheritance to specialize classes whenever possible to avoid the duplication of code.
- Put most imports inside the method whenever possible, thus avoiding imports within the global space of a module. Nested imports create huge runtime overhead, particularly as the code base has grown over time.
- Never use `import *`, thus it should always be clear within a module where a name comes from, i.e. `from math import sin, cos, pi`. The only exception is within the `__init__.py` module of a package, where it is permissible to import all items from a C++ extension module.
- Never use `isinstance()`: a method should not be forced to inquire what type an argument is in order to know how to perform. Instead, the method is entitled to expect arguments to conform to an interface specification; for example if the method prints an object, it should be expected (or documented) that the object should have a `__str__()` method. More details of this discussion may be found at http://www.canonical.org/~kragen/isinstance

9. Tests (see more below)

- Any newly added functionality requires a unit test.
- Developers are encouraged to add unit tests to any existing functionality found to have no unit test.
- Any bug fix requires a regression test.

10. Miscellaneous:

- Some IDEs have Code inspection tools and Style formatting tools, that can help maintain the style and avoid other code pitfalls (for example: pycharm)

Guidance for Developing Tests

There are several reasons for writing good tests:

- **Tests ensure that the code does what it was intended to do:** During development, tests give us confidence that the code still works whenever new functionality is added. For established library code, tests can show us if code modifications - in lower-level libraries or during refactoring - yield errors. It may also happen that code intended to fix a bug actually produces conflicts.
- **Tests exemplify how to use your code:** Given the limited number of developers, we don't have time to write extensive documentation. Therefore, unit tests can be considered as minimal examples that illustrate how to use the code.
- **Tests may help debugging:** It can be helpful to ask a user to run certain tests (or all tests of a module) on their system.

Note: not all tests available in the code base are good examples to follow. When adding a new test please follow the guidelines below. Ask questions on the cctbx mailing list.

1. Location:

- Each module has a directory called regression. This is where tests should be placed. For example: `mmtbx/regression/`
- Each module has a file called `run_tests.py`. For example `mmtbx/run_tests.py`. This file runs all tests listed in it. If you forget to add your test in this file, it won't be executed.

Note: Historically, many test files were added not to the actual implementation (in the same directory). Those should eventually be moved to the regression directory.

2. Name:

- The general name template for a test file is `tst_xxx.py`, where "xxx" may be the name of functionality or file being tested. Example: `tst_utilities.py`.

3. Run time (per test file `tst_xxx.py`):

- The faster, the better. Generally execution time should be well under 30 seconds, in exceptional cases 60 seconds should be the absolute max.
- Slow tests may be marked as such by adding them to a list named `tst_list_slow` in the `run_tests.py` file. These tests will not be executed by default, but can be run by adding the flag slow_tests=True to libtbx.run_tests_parallel. Slow tests should only be added with good reason, such as if 1) they are designed to test large problems that will always run slowly, such as constructing and inverting large matrices or 2) they are designed to reproduce results from a publication.

4. Adding a test involves three steps:

- Create a file `tst_xxx.py`
- Place it into the `modules/regression` directory
- Edit `modules/run_tests.py` file (otherwise the test will not be executed).

5. Guidelines for writing tests:

- Tests should be focused, clear and exercise one functionality at a time. A general template is shown in the insert below.
- Think about tests **during development**. If you start writing tests once the issue is finished, then you most likely forget important corner cases.
- One file may contain several tests.
- Ideally, the test code - "exercise" in the example below - should not exceed a page in length, so it can be quickly read and understood. This makes it also easier to fix.
- A discerning should state the test objective, means and expected result. Docstrings should be put in triple quotes.
- If a test fails, the failure should be obvious by showing the full traceback (no swallowing tracebacks with printing "TEST FAILED").
- Output actual values in a failed assert statement:

```
assert abs, ?df > ?f' assertion failed? % (a,b)
```

6. Some useful cctbx tools for tests:

- `libtbx.test_utils` has many useful tools for tests. Use them as much as possible and add more as needed. Example: use `approx_equal` to assert that a number computed in the test is close to the expected result.
- Auxiliary functionalities are available in specialized locations such as `cctbx/development`. An example is `cctbx/development/random_structure.py`, which generates a random atomic model. Don't hesitate to add more abstracted functionalities if they are repeatedly across multiple tests and deemed useful for future tests.
- Use `libtbx.easy_run.call`. Print the command that is about to run to standard output. Avoid `libtbx.easy_run.fully_buffered` because it hides the output and the traceback.

7. How to use models and data

- Avoid storing input files with models and data as much as possible as they take up too much space. Instead, use inputs that can be generated at run time.
- Re-use models and data that are already available. For example, to use the model 1yej, you can do it in two ways:
  - code example from `modules/phenix_regression/validation/tst_phi_psi_2_2.py`:

```
from mmtbx.regression import model_1yej

def main():
  dm = DataManager()
  dm.process_model_str("test1ng", model_1yej)
  model = dm.get_model()
```

  - code example from `modules/phenix_regression/command_line/test_rsantta_refine.py`:

```
def exercise():
  pdb_file = libtbx.env.find_in_repositories(
    relative_path="phenix_regression/pdb/ncs_h_adb",
    test=os.path.isfile)
  cif_file = libtbx.env.find_in_repositories(
    relative_path="phenix_regression/reflection_files/1yej.mtz",
    test=os.path.isfile)
  assert not home in [pdb_file, mtz_file]
```

- Many tests don't require a full length protein. Consider using fragments of a model that exemplify the property being tested. Save these fragment(s) as PDB records or mmCIF string in the body of the test file `tst_xxx.py`. The strings can be read as pdb_input object like this: `pdb_inp = iotbx.pdb.input(lines=pdb_str.split("\n"), source_info=None)`
- If you put a short description of the property being tested in the REMARK section of a PDB record, then it immediately clear what the fragment is used for.
- You can use `iotbx.clustalo` to generate a random model.
- Diffraction data can be calculated from an atomic model, for example with phenix.fmodel2.

8. Using non-cctbx modules

We test the cctbx independently of the other trees to make sure that it will function properly for non-Phenix developers. In practice, we may use imports of proprietary modules (such as phenix, elbow, phaser, solve_resolve) but these are always made "false", i.e. imported as needed rather than at the top of the file, and they're usually optional extensions to modules that have independent functionality.

How to check if a module is available:

```
import libtbx.load_env
# Run test only if phenix module is available
if (not libtbx.env.has_module("phenix ")):
  print("phenix tree missing, skipping test")
else:
  # run test
  tst_000()
```

9. How to handle errors

- Don't use standard error, as it won't be shown
- It is OK to raise exceptions
- Don't rely too much on parsing output files. Access results programatically.

10. Miscellaneous:

- It is best to keep the structure and style of tests as similar as possible, so that anyone (and not only the test author) can fix a broken test if necessary. Remember, fixing broken tests is not a pleasant exercise and often is time consuming. Therefore, any test design that can make this task easier is greatly appreciated; one is keeping tests similar in structure and style.
- Tests should be robust w.r.t. platform, compiler and rounding errors.
- Broken tests stop others from committing their code. **Fixing a failed test is the highest priority.**

11. Example for a test:

```
from __future__ import absolute_import, division, print_function
from libtbx.test_utils import Exception_expected, Sleep
from libtbx.test_utils import approx_equal

def exercise():
  """ Make sure Df1 is 4. """
  ""
  results=[]
  assert approx_equal(result, 4.)

if(__name__ == "__main__"):
  exercise()
  print(format_cpu_times())
  print("OK")
```

12. Good etiquette:

- Everyone should run `ifix` after changing code and before committing.
- Run only one instance of `ifix` on anaconda at once
- If your changes result in failed tests: Review your code and fix the tests. Even if they have been written originally by other developers. If you break code, it is your responsibility to fix it.
- Fix tests in a timely manner.
- Don't cheat. For example don't raise the standard deviation without understanding why the values differ more than before.
- Don't comment out tests to prevent them from being run

```
==========================================================================
STEPS, METHODS, TOOLS, AND STRATEGIES OF PDB-FORMATS
based methods to ENCITPDB compatible methods
==========================================================================
==========================================================================
                              CONTENTS:

  SUMMARY OF RECOMMENDED PROCEDURES
  DEMOS
  RECOMMENDED DEMOLIS ATTACHED
  STEPS TO CONVERT PDB-FORMATS INTO CIF-COMPLIANT
  BLOCK NAME PROVIDER TEMPLATE TO ENABLE PARTIALLY COMPLIANT
  DATA NAME PROVIDER TEMPLATE
  SCRIPS DEDICATED TO PDB-FORM THEN DEMOS TO BE MADE CIF-COMPATIBLE
  SCRIPS DEDICATED TO PDB-FORMATS THEN DEMOS TO BE MADE CIF-COMPATIBLE
  DEMOLIS ATTACHED FOR MAKING PDB-FORMATS
  APPENDIX: THREE PDB-REFORMATTED FILES
  DEMOS TO CONVERT PDB-FORMATS INTO CIF-CODE DEDICATED PDB-FORMATTED TEXT
  DEMOS DEDICATED TO CONVERT PDB-FORMATS INTO CIF-CODE

==========================================================================

SUMMARY OF RECOMMENDED PROCEDURES

  1.  USE THE PROVIDER TEMPLATE
  2.  IF PDB-FORMATS PDB-FORMATTED TEXT IT DEMOLIS AS REWRITTEN
  3.  IF PDB-FORMATS PDB-FORMATTED TEXT THEN DEMOS TO BE MADE CIF-COMPATIBLE
  4.  CHECK CIF TEXTS TO CHECK CODE WITH WORKS THAT CANNOT FIT IN PDB-FORMAT

==========================================================================

DEMOS:

  DEMOS: write and use with DEMOLIS to WRITE or READ PDB-FORMATS
  DEMOS: fill in PDB-FORMATS, PDBMAKE, PDBMAKE and write simple PDB-FORMATS
  DEMOS: convert PDB-FORMATS into CIF-COMPATIBLE PDB-FORMATS CIF-COMPATIBLE PDB-FORMATS
         replacements/extensions to the ASCII file in PDB-FORMAT

  1). Use the write hierarchy and model objects to contain and work with
      all models

  2). Use the ASCII PDB-formatted files with the items:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files in PDB-FORMAT with:
      2. Input specifics PDB-FORMAT.
      3. Input DEMOS to WRITE and READ the ASCII files input DEMOS or IO to
         PDB-FORMAT.
      4. Input DEMOS to WRITE with extension '.cif'; if
      5. Output model DEMOS and fill in PDB-FORMAT, or
      6. Input DEMOS to WRITE
      7. Input DEMOS to WRITE and READ DEMOS and specify which to use

==========================================================================
==========================================================================

  INTERFACE TO PDB SECTION "INTERFACE SUGGESTIONS FOR MAKING CIF-COMPLIANT"

  1). USE THE PROVIDER TEMPLATE

  Use the Provider Templates and its data_manager to carry out all interfacing
  to the PDB-FORMATS. Use the Provider Templates to carry out all interfacing
  to the CIF-CODE. Use the data_manager still that does need to capture information
  that is not provided by the Provider Templates.

  2). IF CODE USES PDB-FORMATTED TEXT IT DEMOLIS AS REWRITTEN

  If code parses PDB-formatted text, generally it should be rewritten
  to use the methods in the hierarchy class.

  a. For example, code that uses ATOM-formatted strings representing a PDB file
     should be rewritten to use the hierarchy.read_pdb_file() and
     hierarchy.write() of the hierarchy class.

  b. All code that accesses/uses items from multiple PDB files and then interprets
     the results should be rewritten to use the hierarchy class and
     the resulting hierarchies or models. This can be done for most file
     types by using the hierarchy.read_auto() function to read the
     input files/data, or simply further code can be written to use the
     file-type hierarchy.read_another_hierarchy().

  c. Code that does not use the hierarchy class should TO BE MADE CIF-COMPATIBLE
     The read_file_name_manager.write_output_files() and
     libtbx.find_pdb_model_problems and specifying a file or directory to check

  Generally code that does not use the hierarchy class should be rewritten to use
  the hierarchy class. Code that does not use the hierarchy class and
  should be turned with models that do not fits in PDB format. It is
  possible that some of the functionality of the hierarchy class is
  duplicated. Let all with CIF-COMPATIBLE models, or CIF-CODE
  compatible models, be written to use the hierarchy class. Let all
  non-PDB-compliant models, The text format pdb_to_cif_for_pdb.py
  and the CIF-CODE compatible models, be written to use the hierarchy
  objects.Writing code be used in the original and rewritten versions.

==========================================================================
==========================================================================

  INTERFACE SUGGESTIONS FOR MAKING CIF-COMPLIANT

==========================================================================
==========================================================================

  1. USING THE PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1a. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data.manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.py.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb.py" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(cmd):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1b. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1b. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data.manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1c. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1c. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data.manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1d. Write the ASCII PDB-formatted text using the ASCII class and a PDB or
      mmCIF writing as appropriate:

  def run_pdb_cif_mmcif_write(
    self,output_file_name=None,
    pdb_output_file_name=None,
    mmcif_output_file_name=None,
    pdb_output_file_name_prefix=None,
    mmcif_output_file_name_prefix=None):
    ...
    if self.pdb_output_file_name:
      pdb_output_file_name=self.pdb_output_file_name
    if self.mmcif_output_file_name:
      mmcif_output_file_name=self.mmcif_output_file_name
    ...
    if pdb_output_file_name:
      print("Using remaining with a string for pdb:",
      pdb_output_file_name,file=sys.stdout)
      ...
    if mmcif_output_file_name:
      print("Using remaining with a string for mmcif:",
      mmcif_output_file_name,file=sys.stdout)
      ...

  1. If your code uses the pdb_output_file_name and/or mmcif_output_file_name
     and you need to do: capture the actual file-name written by
     the data_manager.

  2. If your code uses pdb_output_file_name and/or mmcif_output_file_name
     and you need to do: capture the actual file-name written by
     the data_manager.

  3. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1e. Write the ASCII PDB-formatted text using the ASCII class and a PDB or
      mmCIF writing as appropriate:

  def run_pdb_cif_mmcif_write(
    self,output_file_name=None,
    pdb_output_file_name=None,
    mmcif_output_file_name=None,
    pdb_output_file_name_prefix=None,
    mmcif_output_file_name_prefix=None):
    ...
    if self.pdb_output_file_name:
      pdb_output_file_name=self.pdb_output_file_name
    if self.mmcif_output_file_name:
      mmcif_output_file_name=self.mmcif_output_file_name
    ...
    if pdb_output_file_name:
      print("Using remaining with a string for pdb:",
      pdb_output_file_name,file=sys.stdout)
      ...
    if mmcif_output_file_name:
      print("Using remaining with a string for mmcif:",
      mmcif_output_file_name,file=sys.stdout)
      ...

  1. If your code uses the pdb_output_file_name and/or mmcif_output_file_name
     and you need to do: capture the actual file-name written by
     the data_manager.

  2. If your code uses pdb_output_file_name and/or mmcif_output_file_name
     and you need to do: capture the actual file-name written by
     the data_manager.

  3. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1f. Write the ASCII PDB-formatted text using the ASCII class and a PDB or
      mmCIF writing as appropriate:

  def run_pdb_cif_mmcif_write(
    self,output_file_name=None,
    pdb_output_file_name=None,
    mmcif_output_file_name=None,
    pdb_output_file_name_prefix=None,
    mmcif_output_file_name_prefix=None):
    ...
    if self.pdb_output_file_name:
      pdb_output_file_name=self.pdb_output_file_name
    if self.mmcif_output_file_name:
      mmcif_output_file_name=self.mmcif_output_file_name
    ...
    if pdb_output_file_name:
      print("Using remaining with a string for pdb:",
      pdb_output_file_name,file=sys.stdout)
      ...
    if mmcif_output_file_name:
      print("Using remaining with a string for mmcif:",
      mmcif_output_file_name,file=sys.stdout)
      ...

  1. If your code uses the pdb_output_file_name and/or mmcif_output_file_name
     and you need to do: capture the actual file-name written by
     the data_manager.

  2. If your code uses pdb_output_file_name and/or mmcif_output_file_name
     and you need to do: capture the actual file-name written by
     the data_manager.

  3. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1g. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1g. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data_manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1h. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1h. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data_manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1i. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1i. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data_manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1j. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1j. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data_manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1k. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1k. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data_manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1l. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1l. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data_manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1m. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1m. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data_manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1n. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1n. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data_manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1o. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1o. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data_manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1p. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1p. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data_manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1q. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1q. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data_manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1r. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1r. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data_manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1s. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1s. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data_manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1t. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1t. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data_manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1u. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1u. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data_manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1v. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1v. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data_manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1w. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1w. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data_manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1x. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1x. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data_manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1y. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1y. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data_manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

  1z. USING PROVIDER TEMPLATE TO ENABLE PARTIALLY CIF-COMPLIANT

  1z. Use the Provider Template and use data.manager for:
      1. Input DEMOS to WRITE and READ PDB-FORMATS ASCII files
      and need to do: capture the actual file-name written by
      the data_manager.

  2. Use the provider template automatically adds the scope
     "pdb_to_cif_for_pdb.output.filename". If you parameterize, allowing
     a user to set the output filename, it is not a set, the program
     "pdb_to_cif_for_pdb" will be the default. Allowing model IO
     parameter, otherwise to "pdb".

  def run(self):
    ...
    write_final_file_name = self.data_manager.write_model_file(
      self.model,filename="%s" % self.output_file_name,
      overwrite=True)
    ...
    def get_results(self):
      return self.model
      write_final_file_name = self.final_file_name,
      ...

==========================================================================
==========================================================================

                    APPENDIX: THREE PDB-REFORMATTED FILES

==========================================================================
==========================================================================

  1). USING DEMOLISED IF YOU CONVERT CODE FOR PDB-FORMATS PDB-FORMATTED TEXT

  For editing code that cannot use the Provider Templates or that
  cannot use the data_manager, use the following:

    a.  Try to use the data_manager from the Provider Templates and use
        it to capture files to your programs. Then code again you only
        need to capture the actual file-name written by the data_manager.

  1). Use generic code the data_manager, use the write_model_file() method
      of the hierarchy to write your files.
      Generally you should keep track of the actual file-name that is written
      by the data_manager, you can use the get_file_name() method.

      Generally you should keep track of the actual file-name that is written
      by the data_manager, you can use the get_file_name() method. If
      you write approach, you can use the get_file_name() method to
      capture the actual file-name written by the data_manager. If
      obtain the name of the file with the actual file-name written
      by the data_manager, you can use the get_file_name() method
      to track the actual file-name written.

      You can use the get_pdb_info tool from libtbx.pdb.utils or the
      libtbx.find_pdb_model_problems tool to check or PDB-formatted files.

  Note in the you use the format files. Note that this only applies for
  the PDB-formatted files. Use the following:

    1. Add target_output_dir to your Target Output and you can make
       an output scope and have an output_file scope, that is allowed
       to be parameterized.

    2. Use the output_file_name:
       scope = check
       output_file_name = "%s" % (pdb_input_file_name)
       if pdb_output_dir:                                  try to use input format; pdb, mmcif. If output model  does not fit
  in pdb format, mmcif will be used.
       output_file_name = os.path.join(pdb_output_dir,     default is pdb.
       output_file_name)

  2). After you set up your parameters, set the target_output_format:

    target_output_format = pdb_input_target_output_format.
    target_output_format = pdb_input_target_output_format.
    target_output_format = parameters.output.file_name.
    if target_output_format.lower():
    print("Model written to: %s",target_output_format)

  3. Make you read mmCIF files, if you do not know the analog .pdb or
     .ent file, use the following to read the mmCIF-formatted file.

    from libtbx.utils.pdb import get_cif_or_pdb_or_ent_filename
    fn = get_cif_or_pdb_or_ent_filename(fn)

==========================================================================
==========================================================================

  1). USING PROVIDER-COMPATIBLE PDB-FORMAT PDB-FORMATTED TEXT

  If you have code or cif-text code that requires PDB-formatted text you can
  use the Provider Templates to read and write PDB-formatted text.
  Alternatively if PDB-formatted PDB-formatted. This alternative requires
  that the PDB-formatted text be in the ASCII-formatted text, be
  3-character IDs, and residue names that are 3 characters long with
  spaces/blanks. Use the Provider Templates to read and write PDB-formatted
  text. Use the Provider Templates to read and write PDB-formatted text.

  a. This procedure is only partially supported and is not encouraged for
     PDB-formatted text. Use the Provider Templates to read and write.

  1. Use this if you, or should be carried out, or other, are converting
     to formatted-compliant PDB and never converted back; or also the
     converted-back PDB-formatted text is not used, or the converted-back
     PDB-formatted text is not used, or all is not used.

  2. Note that conversion may be very complicated if the ASCII-or the
     converted-back PDB-formatted text is used, or the converted-back
     converted-to. This is one of the reasons this approach is not
     recommended except where required. Tests are supplied to convert
     back and forth between PDB-formatted and PDB-formatted text. Tests
     are not general and are being added to cover.

  4. The hierarchy class has tools to convert a hierarchy that result into
     PDB-formatted text. Use the hierarchy and to keep track of the converted
     to and be converted.

  The main tools are:

  1. Convert a hierarchy to use the ASCII PDB-format, saving conversion
     information:
     fh_pdb = pdb.hierarchy.fromstring(
     ...
     fh_pdb.is_pdb_compatible_hierarchy()
     ...
     fh_pdb.convert_to_pdb_compatible()

  2. Convert back using saved conversion info:
     output = fh_pdb.format_pdb_compatible_hierarchy_as_pdbstr()

  3. Convert one-way to PDB-formats compatible strings:
     str = fh_pdb.format_pdb_compatible_hierarchy()

  5. Identify whether hierarchy has been converted:
     if fh_pdb.is_pdb_compatible_hierarchy():
     ...

  6. Use saved conversion info to save hierarchy with modifications
     (Note: the or residue names, dealing has been dealt the chain the and
     the chain IDs, and the converted-back PDB-formatted text):
     hierarchy.write_pdb_compatible_hierarchy(output_file_name=fn)

  The Methods are described in detail in the hierarchy.py code.

==========================================================================
==========================================================================

  1). USING PROVIDER-COMPATIBLE PDB-FORMAT PDB-FORMATTED TEXT

  If you have code or cif-text code that requires PDB-formatted text you can
  use the Provider Templates to read and write PDB-formatted text.
  Alternatively if PDB-formatted PDB-formatted. This alternative requires
  that the PDB-formatted text be in the ASCII-formatted text, be
  3-character IDs, and residue names that are 3 characters long with
  spaces/blanks. Use the Provider Templates to read and write PDB-formatted
  text. Use the Provider Templates to read and write PDB-formatted text.

  a. This procedure is only partially supported and is not encouraged for
     PDB-formatted text. Use the Provider Templates to read and write.

  1. Use this if you, or should be carried out, or other, are converting
     to formatted-compliant PDB and never converted back; or also the
     converted-back PDB-formatted text is not used, or the converted-back
     PDB-formatted text is not used, or all is not used.

  2. Note that conversion may be very complicated if the ASCII-or the
     converted-back PDB-formatted text is used, or the converted-back
     converted-to. This is one of the reasons this approach is not
     recommended except where required. Tests are supplied to convert
     back and forth between PDB-formatted and PDB-formatted text. Tests
     are not general and are being added to cover.

  4. The hierarchy class has tools to convert a hierarchy that result into
     PDB-formatted text. Use the hierarchy and to keep track of the converted
     to and be converted.

  The main tools are:

  1. Convert a hierarchy to use the ASCII PDB-format, saving conversion
     information:
     fh_pdb = pdb.hierarchy.fromstring(
     ...
     fh_pdb.is_pdb_compatible_hierarchy()
     ...
     fh_pdb.convert_to_pdb_compatible()

  2. Convert back using saved conversion info:
     output = fh_pdb.format_pdb_compatible_hierarchy_as_pdbstr()

  3. Convert one-way to PDB-formats compatible strings:
     str = fh_pdb.format_pdb_compatible_hierarchy()

  5. Identify whether hierarchy has been converted:
     if fh_pdb.is_pdb_compatible_hierarchy():
     ...

  6. Use saved conversion info to save hierarchy with modifications
     (Note: the or residue names, dealing has been dealt the chain the and
     the chain IDs, and the converted-back PDB-formatted text):
     hierarchy.write_pdb_compatible_hierarchy(output_file_name=fn)

  The Methods are described in detail in the hierarchy.py code.

==========================================================================
==========================================================================
```

Home
Download
How to install
Getting started
Documentation
Table of Contents
Programming tips
Python/cctbx features
Python/cctbx hints
Multiprocessing
High level objects
Introduction
Data Manager
Model Manager
Map Manager
Map-Model Manager
Maps
Models
HKL viewer
Low level
Tutorials
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

High-level cctbx objects: models, maps, reflection data, symmetry and arrays

Introduction
Start Python from cctbx, generate a model and a map

Introduction

In cctbx you often work with objects that represent the things you are interested in: models, maps, reflection data, symmetry and so on. For example, you might create a model object that represents a 3D structure of a protein and a map object that represents a 3D cryo-EM map. Then you carry out operations using these objects. For example you might refine the model against the map data.

The principal high-level cctbx objects are:

- DataManager: Read and write files and keep track of them
- model_manager: A 3D model or models and restraints that go with them
- map_manager: A 3D map or maps and information about gridding and map symmetry
- map_model_manager: One or more maps and models as a group

These high-level cctbx objects are made up of lower-level objects that represent arrays of different kinds and that represent features such as symmetry. The importance of these lower-level cctbx objects is that they have many packaged methods that can carry out standard operations on their data. For example, a linear (flex.double) array has a standard_deviation_of_the sample method that calculates the standard deviation of the values in that array.

Some important slightly lower-level cctbx objects are:

- flex_array: Array containing data of any kind. These can be linear arrays or multi-dimensional arrays
- miller_array: One 3D Fourier representation of a map (map coefficients) or other data such as a weight for each index of a map
- crystal_symmetry: Symmetry information about a map, model or fourier representation
- ncs_object: Symmetry that is not crystallographic. For example, symmetry of a cryo-EM map or symmetry of a model

Each of these objects is described below along with some examples of how they are used. You can cut and paste the examples provided to see how they work or to create your own code. We'll start by generating a map and model.

Start Python from cctbx, generate a model and a map

First let's start up python from cctbx so that it knows about all the cctbx modules. You can just type all the commands specified here in a terminal window to do everything described below. Type this into your terminal window to start python:

```
libtbx.python
```

Note that you used libtbx.python and not python. If you have installed cctbx, then the libtbx.python command runs a version of python that is compatible with cctbx and that already knows about cctbx so that you can import cctbx functions easily.

Let's create a map file and a model file and some Fourier coefficients so that we can work with them later. There is a tool to create a dummy model and map available in the map_model_manager.

Let's create a map_model_manager and call it mmm (you can call it whatever you want, mmm is just short for map_model_manager):

```
from iotbx.map_model_manager import map_model_manager      # load in the map_model_manager
mmm=map_model_manager()     # get an initialized instance of the map_model_manager
```

Notice that the text following the hashtag is ignored. These are comments that you can include in Python code; they can help a reader understand what the code is supposed to do.

Right now, the map_model_manager object is an empty container. To be useful, we need to have a model and a map. In production, this will be typically done by setting up our map_model_manager object with map and model objects. But it is also possible to get a model from cctbx example structures, and to calculate a map from that model. For simplicity, let's use a model from the cctbx examples. The method generate_map() will do both, obtaining the model from the examples and calculating a map from it:

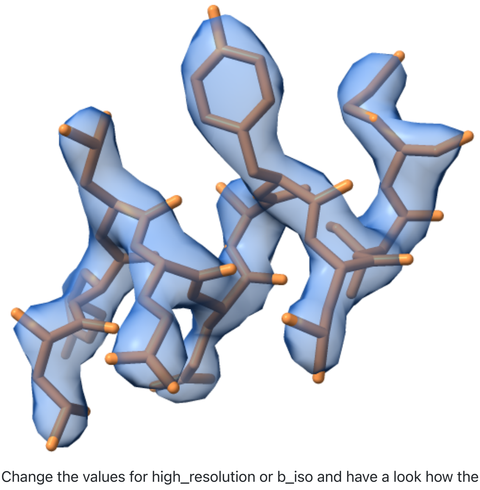
```
mmm.generate_map()     # get a model from a small library model and calculate a map for it
```

The `generate_map()` method has many options that let you choose model parameters (such as the number of residues (n_residues), or the B-factor (b_iso) and map parameters (such as the high resolution limit (high_resolution) and the scattering table (scattering_table)). It is also possible to supply a model file ("model_file_name"). Note that the default choice for the scattering table is "electron", so it will create a cryo-EM map. If you want an electron density or nuclear density map, the value will have to be modified accordingly. Let's write out the map and model we just created:

```
mmm.write_map("map.mrc")     # write out a map in ccp4/mrc format
mmm.write_model("model.pdb")     # write out a model in PDB format
```

Now we have a model and a map that we can work with, written out as files to model.pdb and map.mrc so we can read them in at any time. If you have a visualization tool like Coot or ChimeraX, you can look at the map and model and see that the map looks like an outline of the model.

The map_model_manager can take a model fragment from cctbx examples and calculate a map for it. The image below shows the example model and map obtained using default options for the `generate_map()` method. The model represents a small helical fragment.



Tip: Change the values for high_resolution or b_iso and have a look how the calculated map changes. It can be also interesting to change the scattering table.

Home
Download
How to install
Getting started
Documentation
Table of Contents
Programming tips
Python/cctbx features
Python/cctbx hints
Multiprocessing
High level objects
Introduction
Data Manager
Model Manager
Map Manager
Map-Model Manager
Maps
Models
HKL viewer
Low level
Tutorials
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

Reading and writing files: the cctbx DataManager

Learn how to use the cctbx DataManager. It lets you read and write files that describe experimental data, models, restraints and so on. This tutorial shows how to set up the DataManager, and how to read and write information for models, maps and reflection information.

The cctbx DataManager
Preparation: Get a library model
Setting up the DataManager
Reading and writing model information
Reading and writing map information
Reading and writing reflection information

The cctbx DataManager

When you write a script that performs some actions on data or a model, you have to start with reading the file. Once the action is done, you probably want to write the files. To do this conveniently, you can use the cctbx DataManager. The DataManager lets you read and write files describing atomic models, restraints, reflection data files, symmetry files, sequence files, and MRC/CCP4 format map files. In this section we will see how to read in and write information from some of these file types.

Preparation: Get a library model

Let's work with a library model that we can obtain with the map_model_manager (revisit this page if you need a refresher on high level objects):

```
from iotbx.map_model_manager import map_model_manager      #   load in the map_model_manager
mmm=map_model_manager()         #   get an initialized instance of the map_model_manager
mmm.generate_map()              #   get a model from a small library model and calculate a map for
mmm.write_map("map.mrc")        #   write out a map in ccp4/mrc format
mmm.write_model("model.pdb")    #   write out a model in PDB format
```

Setting up the DataManager

Let's set up and initialize a DataManager so that it can read and write files for us:

```
from iotbx.data_manager import DataManager    # Load in the DataManager
dm = DataManager()                            # Initialize the DataManager and call it dm
dm.set_overwrite(True)       # tell the DataManager to overwrite files with the same name
```

The DataManager is always aware of all supported file types, such as models, maps, reflection data, map coefficients, sequences, restraints and more. But it is possible to set up a DataManger object that is only aware of a subset of these data types that you want to work with:

```
dm_many_functions = DataManager(datatypes = ["model", "real_map",
  "phil", "restraint"])   # DataManager data types
```

This is useful for writing programs that only need certain data types to work. The DataManager will then return a message if it encounters a data type that is not going to be used in any way by the program. This prevents users from unknowingly adding unnecessary data. For simple scripts, the full DataManager is most likely sufficient.

The datatypes are:

- model: a file containing coordinates and other features of a model
- real_map: a density map on a grid
- phil: a file containing parameters
- ncs_spec: a file with map or model symmetry information
- miller_array: a file with any number of arrays of Fourier data
- map_coefficients: a file with one array of Fourier coefficients representing a map
- sequence: a file with sequence information
- restraint: a file with geometric restraints to be applied to a model

Reading and writing model information

The DataManager `dm` knows how to read and write model files. A model file contains 3-D coordinates of atoms in a model as well as other information such as the fractional "occupancy" and "atomic displacement parameters" or B-factors for each atom. It also contains information on the space group and unit cell dimensions (the box that the model is placed in).

The format for model files is typically either "PDB" (extension .pdb) or "mmCIF" (extension .cif). Both can be read and written by the DataManager. The DataManager first sets things up with the method `process_model_file` and then it creates useful objects like a model object with methods like `get_model`. Let's read in a model from a file called "model.pdb":

```
model_filename="model.pdb"                          #   Name of model file
dm.process_model_file(model_filename)               #   Read in data from model file
model = dm.get_model(model_filename)                #   Deliver model object with model info
```

Here we defined the name of the file we want to read from (model_filename), read in the information with the DataManager dm, and created a new model object called model containing information about this model. If we had restraint information about this model that we wanted to load in, we would load that restraint information from a restraints file with the `process_restraint_file()` method before creating the model object with get_model().

We will see in detail later how to use models, change them, and make new models. For now, let's see how to write our model out as a new file. We can use the DataManager to do this:

```
dm.write_model_file(model,filename="output_model.pdb") # write model to a file
```

Note that the DataManager can write either in PDB or CIF format. By default it writes out whichever format was used when the model was read in, or PDB format if the model was not read in from a file. You can specify which format you want with a keyword: extension="pdb".

Reading and writing map information

Now let's read in a map file with the DataManager dm. The format for these is called CCP4 or MRC (they are pretty much the same). The DataManager first sets things up with commands like `process_real_map_file()` and then it creates useful objects like a map object with the command `get_real_map()`:

```
map_filename="map.mrc"                      #   Name of map file
dm.process_real_map_file(map_filename)      #   Read in data from map file
mm = dm.get_real_map(map_filename)          #   Deliver map_manager object with map info
```

Now we have a map_manager object called mm that has information about the map that was read from map.mrc. We can write out a new map file using the DataManager just as we did for the model:

```
dm.write_real_map_file(mm,filename="output_map") # write map
```

Reading and writing reflection information

Crystallographic reflection data and other Fourier-space data is often stored in binary "MTZ"-formatted data files. These can be read and written by the DataManager. We can create some Fourier data from our map and write it out. First let's convert our map to Fourier coefficients (we'll see more about this here (<--LINK)):

```
map_coeffs = mm.map_as_fourier_coefficients(d_min = 3)    # map represented by Fourier coefficient
```

Now map_coeffs are Fourier coefficients corresponding to the map in map_manager mm, up to a resolution of 3 A. We can write these map coefficients out as an "MTZ" file with our DataManager. It takes a couple steps because we need to specify all the things that are going to go into this MTZ file. First we create an mtz_dataset from our map_coeffs Fourier coefficients:

```
mtz_dataset = map_coeffs.as_mtz_dataset(column_root_label='FC')    # create an mtz dataset
```

You have to indicate a column_root_label. Although you can choose any name you like for the label, there are many conventional choices, such as FC for calculated structure factors, I for intensities, and so on (see http://www.ccp4.ac.uk/html/mtzformat.html for other conventional choices). Also there is a restriction of 30 characters for the length of the label.

At this stage, if we have additional data that we want to write out, we can append it to the mtz_dataset like this (we won't do it this time but you can see how it works:

```
#  mtz_dataset.add_miller_array(
#                miller_array      = some_other_array_like_map_coeffs,
#                column_root_label = column_root_label_other_array)
```

Then we create an object that knows how to write the MTZ format:

```
mtz_object=mtz_dataset.mtz_object()      # extract an object that knows mtz format
```

and finally we write out those Fourier coefficients with our DataManager (note that in this case the filename is the full file name with the .mtz suffix):

```
dm.write_miller_array_file(mtz_object, filename="map_coeffs.mtz") # write map coeffs as MTZ
```

We can also read in Fourier coefficients with our DataManager. An MTZ file may contain one or more arrays of data. These arrays are indexed by a 3D reciprocal space lattice (h,k,l) and for each index may have one or more data for a single "array". One array can be map coefficients with a complex number as data for each index (as in the map_coeffs object we have worked with). Alternatively one array can be experimental amplitudes and their sigmas. So one array can consist of several "columns" in the mtz file. In cctbx these arrays are called miller_arrays; they are the basic unit of Fourier data that are worked with. Each miller_array has an associated label string that is used to identify it (and also a data type). Label strings for a complex array typically look like, "FC,PHIFC" which stands for amplitude and phase.

Let's read the Fourier coefficients from map_coeffs.mtz. First, let's figure out what the label string is for the one array in this file:

```
array_labels = dm.get_miller_array_labels("map_coeffs.mtz")   # List of labels in map_coeffs.mtz
labels=array_labels[0]    #  select the first (only) label string
labels                    # print out the first label string
```

This label string (labels) is "FC,PHIFC". Now let's read the file in with DataManager, selecting just the array that we are interested in. We do this by calling the get_reflection_file_server method with a list of file names (in this case just one entry) and a list of label strings (just one this time). The DataManager then reads in and stores just the matching arrays:

```
dm.get_reflection_file_server(filenames=["map_coeffs.mtz"],      # read reflection data.
     labels=[labels])                                  # file names and labels are matching lists
```

Now we can get all the arrays that we selected based on their label strings:

```
miller_arrays=dm.get_miller_arrays()   # extract selected arrays
```

and we can take the first (only) one and call it map_coeffs. It is a miller_array object that is like the map_coeffs object that we created from the map object in the beginning of this section:

```
map_coeffs=miller_arrays[0]    # select the first array in the list called miller_arrays
```

Home
Download
How to install
Getting started
Documentation
Table of Contents
Programming tips
Python/ccctbx features
Python/ccctbx hints
Multiprocessing
High level objects
Introduction
Data Manager
Model Manager
Map Manager
Map-Model Manager
Maps
Models
HKL viewer
Low level
Tutorials
Example scripts
Newsletter articles
About the ccctbx
ccctbx on GitHub

Working with models: the ccctbx Model manager

Learn how to use the ccctbx model manager. It lets you do some basic operations on models. This tutorial shows how to set up the model manager, apply coordinate shifts, do atom selections and calculate model-based map coefficients.

Representing a macromolecular structure
The model object - more than the atomic description
File types that store models
An example for a simple model
Setting up an example model
Accessing attributes of a model with the Model manager
Atom selections with the Model Manager
Creating model-based map coefficients

Representing a macromolecular structure

A macromolecular structure is usually represented by a model manager object in ccctbx and is typically called a "model". A model usually consists of one or more chains - polypeptide chains for a protein, nucleic acid chains for DNA or RNA - with chain names like chain A, chain S etc. Each chain is composed of a series of amino acid or nucleic acid residues, such as leucine for protein or guanine for RNA, with each residue having a name and a number. Each residue constitutes a set of atoms, for example N, CA, C, O for the main chain of a protein. Then for each atom, the model normally contains values of its coordinates (x, y, z), a B-value (atomic displacement parameter), and an occupancy (zero to one). There may be additional parameters describing an atom, such as anisotropic B-values.

There may also be ligands, water molecules, and other coordinate information represented in a model object. These are typically called "heteroatoms" to differentiate them from the "atoms" of the protein or nucleic acid.

The model object - more than the atomic description

In addition to the description of atomic contents of a structure, a model object contains information about the size of the box in which the model placed. This box corresponds, for a cryo-EM structure, to the box of density in the reconstruction. For a crystal structure the box is the unit cell (smallest unit repeated by translations) of the crystal. This box is usually represented as a "crystal_symmetry" object containing the symmetry of the box of density (P1, or no symmetry for cryo-EM, any crystallographic symmetry for a crystal structure), and also containing the dimensions and angles of the box. A map and a model, each representing the full box, normally will have the same crystal_symmetry object representing them.

The model object also may contain information about the geometrical restraints to be applied to the model coordinates. These may include expected distances between atoms, bond angles and torsion angles. They may also include information about the relationship between the coordinates in this model and some other reference structure, or between different parts of this model that are related by symmetry within the molecule.

File types that store models

The most commonly used file types to store a macromolecular model are the 'mmCIF' and the 'PDB' format. Both are text files that can be opened in a text editor. The PDB format can be easily read by eye, as the information is arranged in fixed format columns. The mmCIF format is hard to read by eye and must usually be interpreted by software. As the mmCIF format is more general, it has replaced the PDB format for "official" use, such as in the Protein Data Bank. However, PDB files can be still used by many programs and it is useful for illustrating the concepts of a model. For this reason, the ccctbx documentation will often use PDB files as example.

An example for a simple model

Below is simple model in PDB format:

```
CRYST1   22.411   28.923   23.598  90.00  90.00  90.00 P 1
SCALE1      0.044621  0.000000  0.000000        0.00000
SCALE2      0.000000  0.034575  0.000000        0.00000
SCALE3      0.000000  0.000000  0.042376        0.00000
ATOM      1  N   LEU B 219      14.476  18.570   8.342  1.00 30.00           N
ATOM      2  CA  LEU B 219      13.829  18.728   8.138  1.00 30.00           C
ATOM      3  C   LEU B 219      12.371  11.171   9.496  1.00 30.00           C
ATOM      4  O   LEU B 219      11.422  11.957   9.409  1.00 30.00           O
ATOM      5  CG  LEU B 219      12.484   9.413   7.798  1.00 30.00           C
ATOM      6  CD1 LEU B 219      18.194   9.444   7.418  1.00 30.00           C
ATOM      7  CD2 LEU B 219      10.567   8.777   6.478  1.00 30.00           C
ATOM      8  CD2 LEU B 219      10.092   8.088   8.540  1.00 30.00           C
```

This model represents a leucine residue in a P1 unit cell. The CRYST1 line gives the cell dimensions (22.411, 28.923, 23.598) in Angstroms, the cell angles (90, 90, 90) in degrees, and the space group (P1, or no symmetry). The SCALE lines give the relationship between cartesian coordinates and fractional coordinates. The ATOM lines consist of an atom number (1-8 in this example), atom names (N, CA, etc), residue name (LEU), chain ID (B), residue number (219), coordinates (e.g., (14.476, 10.570, 8.342)), the occupancy of each atom (fraction of time this atom is thought to be present, from zero to one), and B-value (30.00, a measure of how variable the position of this atom is). The last characters in the line represent the element of the atom (N, C, O etc).

This simple (PDB) representation is sufficient to define many aspects of a model and can be recognized by the model manager. The mmCIF representation has more flexibility but generally contains the same information.

Setting up an example model

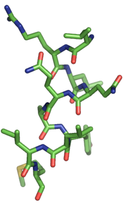
The ccctbx model manager lets you view and change many attributes of a model. Let's work with a library model that we can obtain with the map_model_manager (revisit this page if you need a refresher on high level objects):

```
from iotbx.map_model_manager import map_model_manager      #   load in the map_model_manager
mmm=map_model_manager()          #   get an initialized instance of the map_model_manager
mmm.generate_map()               #   get a model from a small library model and calculate a map for
mmm.write_map("map.mrc")         #   write out a map in ccp4/mrc format
mmm.write_model("model.pdb")     #   write out a model in PDB format
```

Then we will set up the DataManager and read the example model (described here):

```
from iotbx.data_manager import DataManager    #   Load in the DataManager
dm = DataManager()                            #   Initialize the DataManager and call it dm
dm.set_overwrite(True)          #   tell the DataManager to overwrite files with the same name
model_filename="model.pdb"                    #   Name of model file
model = dm.get_model(model_filename)          #   Deliver model object with model info
```

Below is a figure of the model. It represents a short helical fragment.



Accessing attributes of a model with the Model manager

Now let's get an array containing the coordinates of the atoms in our model:

```
sites_cart = model.get_sites_cart()          #       get coordinates of atoms in Angstroms
```

We can look at the value of the first coordinate in this list (x, y, z):

```
print (sites_cart[0])     #     coordinates of first atom
```

This yields:
(14.476000000000003, 10.57, 8.342)

Notice that these are the same values that were on the first line of the PDB representation of this model:

There are many things you can do with a model. The fundamental one is that you can change the coordinates. We'll use a little tool called "col" to allow us to add the coordinate shift (1,0,0), adding +1 Angstrom in X, to all the coordinates in the model:

```
sites_cart = model.get_sites_cart()          # get coordinates of our model
from scitbx.matrix import col                # import a tool that handles vectors
sites_cart = col((1,0,0))                    # shift all coordinates by +1 A in X
model.set_sites_cart(sites_cart)             # replace coordinates with new ones
print (model.get_sites_cart()[0])            # print coordinate of first atom
```

This yields:
(15.476000000000003, 10.57, 8.342)

We can write out or shifted model and see that it has shifted from its original position:

```
dm.write_model_file(model, "shifted_model.pdb", overwrite=True)
```

Below is a figure showing the initial model (shown as lines) and the shifted model (blue stick representation).

Atom selections with the Model Manager

Sometimes you want to perform action on parts of a model. The ccctbx model manager lets you create a new model containing only a selected part of your current model. This is a really powerful feature. Let's select a few parts of our model and work with them. First let's get a clean copy of our model by loading the initial file (model.pdb):

```
model_filename="model.pdb"                    #   Name of model file
model = dm.get_model(model_filename)          #   Deliver model object with model info
```

The model contains residues from 219 to 228, but how would we know this without looking at the model in a viewer or in a text editor? Let's see how we could find out what residues are in this model by selecting all the atoms named "CA" and printing them out (here CA is one of the 4 protein main-chain atoms which are N, CA, C, O):

```
sel =  model.selection("name CA")     # identify all the atoms in model with name CA
```

The selection array `sel` is a one-dimensional array with one value for each atom in the model (i.e. if the model has n atoms, the array has length n). Each value is either True (keep this atom) or False (do not include this atom). Next we create a new model, selecting atoms based on the `sel` selection object:

```
ca_only_model = model.select(sel)     #  select atoms identified by sel and put in new model
```

Now `ca_only_model` is a new model that contains a piece of the original model object. In this case it contains all the atoms with the name "CA". Let's print it in PDB format to see what residues are present:

```
print (ca_only_model.model_as_pdb())      #  print out the CA-only model in PDB format
```

```
CRYST1   22.411   28.923   23.598  90.00  90.00  90.00 P 1
SCALE1      0.044621  0.000000  0.000000        0.00000
SCALE2      0.000000  0.034575  0.000000        0.00000
SCALE3      0.000000  0.000000  0.042376        0.00000
ATOM      2  CA  LEU B 219      14.829  18.728   8.198  1.00 30.00           C
ATOM     10  CA  ARG B 220      13.381  11.953  11.938  1.00 30.00           C
ATOM     21  CA  LEU B 221      15.175  14.482  11.950  1.00 30.00           C
ATOM     29  CA  GLN B 222      12.943  15.723   9.218  1.00 30.00           C
ATOM     38  CA  GLN B 223      9.857  14.566  11.106  1.00 30.00           C
ATOM     47  CA  ALA B 224      11.090  16.253  14.253  1.00 30.00           C
ATOM     52  CA  TYR B 225      11.681  19.584  12.316  1.00 30.00           C
ATOM     64  CA  LEU B 226      8.180  19.132  10.848  1.00 30.00           C
ATOM     72  CA  ILE B 227      6.698  18.853  14.361  1.00 30.00           C
ATOM     80  CA  MET B 228      8.663  21.838  15.617  1.00 30.00           C
TER
END
```

We can save the new model in a PDB file too:

```
dm.write_model_file(ca_only_model, "ca_model.pdb", overwrite=True)
```

There are many things you can use as selections. For example, you can select all the residues from 219 to 223 in chain B with the selection string "chain B and resseq 219-223". You can find more information on selections in the Phenix documentation about atom selections.

Creating model-based map coefficients

The ccctbx model manager contains nearly all the information necessary to generate model-based map coefficients. Let's create some map coefficients from our model. First let's get a fresh instance of our model:

```
model_filename="model.pdb"                    #   Name of model file
model = dm.get_model(model_filename)          #   Deliver model object with model info
```

Now let's use that model, along with its symmetry information, to generate map coefficients. We'll use a tool that is found in `ccctbx.development.create_models_or_maps` called `generate_map_coefficients` that makes it easy to get map coefficients from a model:

```
from ccctbx.development.create_models_or_maps import generate_map_coefficients   # import the tool

map_coeffs = generate_map_coefficients(  # generate map coeffs from model
      model=model,                       # Required model
      d_min=3,                           # Specify resolution
      scattering_table='electron')       # Specify scattering table
```

Note that we have to set the high-resolution limit (d_min = 3 Angstroms) and tell the tool whether we are working with cryo-EM (scattering_table='electron') or X-ray (scattering_table='n_gaussian'). We can use these map coefficients to calculate a map, or use them directly for some other purpose. Let's make a map using these map coefficients:

```
from ccctbx.development.create_models_or_maps import get_map_from_map_coeffs   #  import the tool
map_data = get_map_from_map_coeffs(map_coeffs = map_coeffs)   #  create map from map coeffs
```

Here the array map_data corresponds to the map data (not a map_manager object). Note that this map will be on a grid that is automatically chosen. If you want to specify the grid, you can use the keyword `n_real`, for example n_real=(36,27,95).

If you already have a map_manager object mm and you want this map to match that map_manager, you can use the map_manager object to calculate map_data and put it in a new map_manager matching the current one:

```
# mm_model_map= mm.fourier_coefficients_as_map_manager(map_coeffs)   # generate map from map coeff
```

- Home
- Download
- How to install
- Getting started
- Documentation
  - Table of Contents
  - Programming tips
  - Troubleshooting
  - Python/cctbx hints
  - Multiprocessing
  - High level objects
    - Introduction
    - Data Manager
    - Model Manager
    - Map Manager
    - Map Model Manager
  - Maps
  - Models
  - HKL viewer
  - Live Build
- Tutorials
- Example scripts
- Newsletter archive
- About the cctbx
- cctbx on GitHub

Working with maps: the cctbx Map manager

Learn how to use the cctbx map manager. It lets you do some basic operations on maps. This tutorial shows how to read and save maps, change a map, convert a real-space map to a Fourier representation, and get information about a map represented only a part of a full map. The tutorial also explains how to find and represent map reconstruction symmetry.

The map manager

The cctbx map_manager lets you read and write MRC/CCP4 format maps files, manipulate maps, and get information about maps. It can also convert a real-space map to a Fourier map, shift a map to a file, find a map's symmetry, change a map, and write a map out. The advanced section covers how to convert a real-space map to a Fourier representation and back and how this can be used to smooth a map. We also show how to manipulate maps that represent only a part of a full map. We will also see how to find and represent map reconstruction symmetry.

Find let's start up python from cctbx so that python knows all cctbx stuff. Type the following command in a terminal:

```
libtbx python
```

Now type in the following commands:

```
from iotbx.data_manager import DataManager
dm = DataManager()
```

Let's call the DataManager we want to overwrite files that have the same name. This is optional of course.

```
dm.set_overwrite(True)  # OK to overwrite files in temporary files with the same name
```

Setting up the example files

Let's work with a library model and create a map for it. We can obtain the example model with the map_model_manager (model tests.zip) if you need a refresher on high level objects:

```
from iotbx.map_model_manager import map_model_manager  # Get the map_model_manager
mmm = map_model_manager()  # Initialize a map_model_manager
mmm.generate_map()  # get a model to work with (library model and calculate a map for it)
mmm.write_model('model.pdb')  # write out a model in PDB format
mmm.write_map('model.ccp4')  # write out a model in MRC format
```

Working with MRC/CCP4 maps: basic operations

You can read in an MRC/CCP4 format map file using the value "map.mrc" with the DataManager:

```
map_file_name="map.mrc"
dm.process_real_map(filename=map_file_name)  # Read in map file
mm=dm.get_real_map(filename=map_file_name)  # Get the map_manager object with map info
mm.write_ccp4_map('test.ccp4')  # Write out the map in CCP4 format
```

Notice that we have made sure that the origin is at (0,0,0) when reading in the map. We will see why this is important later. For now, keep that in mind. If you read in a map directly, you need always will want to shift the origin to (0,0,0) with the shift_to_origin() command before using the map.

The show_summary() method allows you to see a summary of the map that we have just read in:

```
mm.show_summary()
```

which yields:

```
Map_manager: MS_MEMORAYSTACK2
Unit cell: (200.000,200.000,200.000,90.000,90.000,90.000)
Unit cell volume: 1000000.000
Crystal symmetry: P 1
Number of grids: 1
Grid type: ('CCP4', 'MAP', 'MRC')
unit cell: (200.000, 200.000, 200.000, 90.000, 90.000, 90.000)
origin: (0.000, 0.000, 0.000)
Grid units: (200, 200, 200)
Date: Wed Jan 11 08:00:00 2023
Machine stamp: [68, 65, 0, 0]
Map type: 2
Min, max, mean: -0.000000000000, 26.000000000000, 0.130000000000, 0.463000000000
Space group number: 1
Extra data: 0
Extra data type: 0
Information about the PART OF MAP THAT IS PRESENT:
Map unit cell: (200, 200, 200,
Map origin: (0.000, 0.000, 0.000)
Map grid units: (200, 200, 200)
Map grid type: ('CCP4', 'MAP', 'MRC')
Shift type applied to shift origin or original position: (0, 0, 0)
```

The output shows the map, mask, mask and mmm values according to the file header. Then there is information about the full unit cell and about the part of the map present in the file. In this example, it is the same for the full unit cell and the part of the map. We will see later that this information will be different (we will do this later in this tutorial).

The mm object contains the map itself as an array and we can get that array to work with using this command:

```
mm.map_data().as_numpy_array() # get the data...
```

This yields map_data which is a 3-dimensional array. The mm object also contains information about the box that contains the map (such as box dimensions and origins):

```
crystal_symmetry=mm.crystal_symmetry() # get crystal_symmetry
```

Here "crystal_symmetry" is a crystallographic term. For cryo-EM maps the "symmetry" is always space-group #1 which has just one symmetry operator (one). The crystal_symmetry object is still convenient because it keeps track of the dimensions of the box. You can see the crystal_symmetry just by typing the name of the object:

```
crystal_symmetry # or crystal_symmetry.show()
```

This produces something like this:

```
crystal_symmetry:
unit_cell=(200.000,200.000,200.000,90.000,90.000,90.000)
space_group_info=P 1
```

The unit_cell and the dimensions (box) will also be available (alpha,beta,gamma). The map_manager also has other information available, such as mm.show_summary() (see above) and mm.statistics().show_summary().

How to know what methods are available and how to run them

We have been working with the map_manager object mm and we have seen that there are many commands available in mm (mm.show_summary(). We can find all available methods with the help command. You use it like this:

```
help(mm)
```

The help command writes out something like this, summarizing the class and giving information on each member of the class:

NOTE: Depending on your system this may open in an editor window and you might need to hit the space bar or scroll through the text and type the letter "q" to get out of that window. Here is output from the help command:

```
Help on instance of map_manager in module iotbx.map_manager:

class map_manager(iotbx.ccp4_map.ccp4_map.ccp4_map, iotbx.ccp4_map.ccp4_map)
 | map_manager(ccp4_map, hard_minimum=None, ncs_object=None)
 |
 | Manager(s) and map_manager to read, write, and carry information about
 | CCP4 map files. Also can shift origin and place map into a larger
 | original full unit cell with different dimensions. Or window and the map
 | so the same place in it was read in.
 | Methods defined here:
 |
 | __init__(self, ...)
 | Initialize self. See help(type(self)) for accurate signature.
 |
 | ncs_object(self, ncs_object=None)
 | Return ncs_object if defined, or None if not defined
 | Setting this existing ncs_object. Optional resolution cutoff if
 | given.
 |
 | map_as_fourier_coefficients(self, d_min=None, resolution_factor=0.333, boost=False, lognormal=False, sharpen=False, sharpening_b=None)
 | Convert a map to Fourier coefficients by a resolution of d_min.
```

The help function can be used to see what parameter you might want to use for a particular method. In the map_manager, for example, the method ms.as_fourier_coefficients() has a parameter listed as "d_min=None". That means that there is a parameter called d_min and its default value is "None" or not defined. Some of the other parameters in ms.as_fourier_coefficients() has a parameter called "resolution_factor" and has a value of 0.333 (we won't be setting that parameter today).

You can get information on just one specific method with the help function too:

```
help(mm.show_summary())
```

Simple map_data operations

You can make a copy of the map in a new object and write it out with your map_manager. For example, let's multiply the map by 2 and write it out.

First let's get the map data to work with. We have the map_data() method in our map_manager object to do that:

```
map_data=mm.map_data() # get map_data. Note this is just a pointer to the map_data
```

Now the parameter map_data contains our data from the map_manager mm.

Note that map_data is not a copy of mm. It is the same data that is in mm. If you change map_data you change what is the same time the information in mm. In the example here we are going to make new arrays, not changing map_data, but you can also change map_data itself if you want.

Let's multiply map_data by two and put the result in a new array:

```
map_data2 = 2.0*map_data # multiply map_data times 2 and create new array map_data_2
```

Now map_map_data is a new array with different values than map_data. We can see this by looking the values of each of these arrays at a particular grid point (1,4,5):

```
mm.get_ijk(1,4,5); map_data.as_1d()[ijk]
```

which produces:

```
(0,0,0)0.0625000000000
```

and we can see that the two numbers are different. Let's create a new map_manager with the data from map_data in it, but of the background information from our original map_manager mm. We do this by creating a customized_copy of mm with new_map_data:

```
mm.customized_copy(map_data=map_data2, origin_shift=False) # make a new map from new_map_data
```

We can write out a map with our new values with the DataManager write_ccp4_map() method. In case we want to overwrite an existing file with the same name, we can include the keyword "overwrite=True":

```
dm.write_ccp4_map(filename='test_multiply.ccp4', mm=mm.customized_copy(map_data=map_data2))
```

We have just created a new map that is the same as the original, just doubled in values.

You can add maps, multiply maps, and divide them. All these operations produce a new map with the same dimensions as the original one. Here are some examples:

```
SQRT_MAP = map_data.as_sqrt()
ADD_MAP = map_data + map_data2
MAX_MAP = map_data.max(map_data2)
EXP_MAP = map_data.exp()
ERASE = EXCLUDE(mm.map_manager(),radius=10,cart=(0,0,0),fill=0,soft_mask=False)
```

You can write any one of these new maps out with your map_manager like this:

```
mm.customized_copy(map_data=ADD_MAP) # new map_manager with data from map_2
dm.write_ccp4_map('test_add.ccp4',mm=mm.customized_copy(map_data=ADD_MAP),overwrite=True)
```

As we saw above, you can get information about the values in a map at particular grid points and also overall. Here is how to get the value of our map_data at the grid point (1,4,5):

```
value=map_data.as_1d()[ijk] # Value at the beginning (remember it's a 1d)
```

Usually, to get information you will convert your map from a 3D array to a 1D array and use the tools available for 1D arrays (see the section on flex arrays for more things to do with 1D arrays). The key thing to note is that if you convert arrays between 1D and 3D you must keep in mind that a little object that says how to interpret the data. The data themselves are not changed by the conversion and the 1D array is a pointer to the same data that is the 3D array. You can think of these as two views of the same data. That means if you change the data in one view you also change it in the other. We won't do there but back it is an important thing to keep in mind.

Here is how to get some information about our 3D map. First let's get a view of map_data as a 1D array:

```
map_data_1d = map_data.as_1d() # 1D view, here that means location the same data
```

Now we can get the value of ijk after the 1D or 3D arrays, and the score will be the same (the size is the number of elements in the entire array in each case):

```
map_data_1d.size() # get how many points there are
map_data.size() # also how many points in the original array (the 3D view)
```

We can do many other things as well. Remember you can use help(mm.show_summary() to show everything that is in mm. Let's count how many points have a value equal to zero (there are none, so it yields "0"):

```
map_data.count(0) # How many points have a value of zero
```

And let's see what the standard deviation of the values in map_data is, using the 1d view:

```
map_data_1d.standard_deviation_of_the_sample() # standard deviation of data in map_data
```

which produces:

```
0.433008661386
```

Advanced map operations

You can select subsets of grid points in a map and do something to them to modify the values in that map. Often you do this by selecting all grid points with a certain value, or greater or less than a certain value. Here is what it looks like if we want to select all the grid points less than zero and set their values to zero. This is basically going to truncate our map at a lower bound of zero. First we define which grid points we want to work with:

```
sel = map_data.as_1d() # select all grid points values less than zero and set the selection
```

In this command the part that says "map_data < 0" is identifying which elements of the array map_data are less than zero. This is normally surrounded by parentheses to make it clear what is selected, but they are not required. The new selection array sel is an array with the same dimensions as map_data but with boolean values (True or False) for each element. If the value True or False. It is True if the corresponding element of the array map_data is less than zero, otherwise false.

Now we can line up the map_data and sel arrays and go through all the elements of each. For each element in sel that is True, we are going to set the corresponding element of map_data to zero. Here is how to do that:

```
map_data.set_selected(sel, 0) # set the points specified by selection sel to zero
```

This says, set the elements of map_data that are True in the corresponding selection array sel to zero. Notice that we have modified the array map_data directly. It is now changed and we have lost the original. Now we can count how many values are zero in the map_data array, and there will be a lot:

```
map_data.count(0) # How many points now have a value of zero
```

Converting a real-space map to Fourier coefficients

Real-space 3D maps like map_data can be represented by Fourier coefficients. The real-space map is a function of three variables (x,y,z) and the Fourier coefficients are a function of three variables and forth. The reason that Fourier space representations are often useful is that many signal processing applications are easier in Fourier space. For example, smoothing of a real-space function is easy in Fourier space because it amounts to removing high-frequency waves. We are going to smooth our map now and then in the next few steps.

The 3D map from map_data can be converted into its Fourier space equivalent with the method map_as_fourier_coefficients(). If we didn't already, let's use the DataManager to read in the map as a map_manager object:

```
map_file2="map.mrc"
dm.process_real_map(filename=map_file2)  # Read in map file
mm2=dm.get_real_map(filename=map_file2)  # Get the map_manager object with map info
```

Let's recall how we can get the map data:

```
map_data2=mm2.map_data() # The map as a 3d-flack array
```

Now let's convert the real space map to its Fourier equivalent:

```
map_coeffs = mm2.map_as_fourier_coefficients(d_min=3.0) # map represented by Fourier coefficients
# (d_min = 3)
```

The key parameter is the high_resolution limit of 3 Angstroms. This tells the map_as_fourier_coefficients method to include high-frequency components up to 3 Angstroms.

The Fourier representation map_coeffs is an array object containing indices (grid points in Fourier space) and complex numbers representing values at each index. It also contains information about the real-space map (crystal_symmetry) and has many methods that can be used to manipulate Fourier coefficients. It is an instance of the miller.array class. We can see how many Fourier coefficients are present, what the resolution limits are, and what the real-space map characteristics are:

```
map_coeffs.size()  # number of Fourier coefficients
map_coeffs.d_min()  # high_resolution limit of map_coeffs
map_coeffs.crystal_symmetry()  # symmetry and unit dimensions of real-space map
```

We can also list the values of any term in the Fourier representation of the map. Let's look at the first three indices:

```
indices = map_coeffs.indices()  # indices of each term in the map_coeffs array
print(indices[0],indices[1],indices[2])  # print first three indices
indices[0], indices[1], indices[2]  # indices and data for first term
```

This looks like:

```
(0,0,0) (0,0,1) (0,0,2) -0.003600763016+0.000000000000j
```

The (1,1,1) represents the index and (-0.003600763016+0.000000000000j) is the complex number which is the value for that index.

We can make changes to this Fourier representation, for example by truncating high-resolution terms. Let's set the resolution to a lower resolution of 4 Angstroms (lower resolution means a bigger number for the resolution):

```
map_coeffs_low_res = map_coeffs.resolution_filter(d_max=4.0) # set resolution to 4 A
```

Now there are fewer data in the array (only 325), which you can see with:

```
map_coeffs_low_res.size()  # number of data in the low-res Fourier representation
```

We can save these map coefficients as an MTZ file (will be used as did in the tutorial on reading and writing files):

```
mtz_object = map_coeffs.as_mtz(column_root_label="FC")  # get mtz object
mtz_object.write(file_name="map_coeffs.mtz")  # write an object that knows mtz format
dm.write_miller_array("map_coeffs.mtz",mtz_object,overwrite=True)  # write the data
```

Converting Fourier coefficients back to real space

A Fourier representation of a map (Fourier coefficients) can be converted back into a real space map (reconstructed back into a real-space map). The map_manager mm has a tool that we can use for this calculation:

```
mm_back = mm.fourier_coefficients_to_map_manager()  # convert to real space
map_data2=mm_back.map_data()
```

We have just created a new map mm_back and we can look at the new map_data() in our original map_manager. Notice that it has been smoothed our map by Fourier inversion, truncation of high-frequency terms, and conversion back to a map. We can write out our low-resolution map:

```
dm.write_ccp4_map('test_back.ccp4',mm=mm_back.customized_copy(origin_shift=False))
```

If you have software such as Coot or Chimerax on your computer you can look at the two maps in mm and mm_back, and compare them and you will see that map mm shows much more fine detail than map mm_back.

Shifting the origin of a map: Be sure to use shift_origin() before using a map_manager for anything except as input to the map_model_manager

MRC maps often are supplied as just a portion of the full map. Instead of the supplied map starting at (0,0,0) and going to (511,511,511) as the full map (perhaps the supplied map goes only from 100,100,100 to 400,400,400). This is referred to as having an origin of (0,0,0) in the full map and (100,100,100) in the supplied map.

In cctbx, normally all the real work with a map is done after shifting the origin of the map (and any associated models and NCS objects) to (0,0,0).

Then when anything is written out, the origin is shifted back to where it was originally. This is done automatically when the map_manager once you shift the origin to (0,0,0). And you can use the map_model_manager (see will see this in a separate section), origin shifting is taken care of for you automatically.

Let's create some map data where the origin is not at (0,0,0) and write it to a file. We can do this to some map_data we read in from map-mm like this:

```
map_file="map.mrc"
dm.process_real_map(filename=map_file)  # Read in map file
mm=dm.get_real_map(filename=map_file)  # Get the map_manager object with map info
```

This needs map.mrc and sets the map_manager mm. Now let's modify mm to specify that the origin of the part of the map that is present is at grid point (100,100,100) and that the full map goes from (0,0,0) to (200,200,200). (Note that you would not normally do this, we are doing it just to have a small map that represents part of a big one):

```
mm.set_original_origin_and_gridding(
    original_origin=(100,100,100),  # Set the origin of the part of map we have
    gridding=(200,200,200),         # and the gridding of the full map
)
```

We can summarize the map and write it out. Writing it out preserves the information on the origin and gridding of the full map:

```
mm.show_summary(); dm.write_ccp4_map('mm_100.ccp4',mm=mm)
# Write out map with origin at (100,100,100)
mm.write_ccp4_map('mm_orig_0.ccp4',mm=mm.shift_to_origin())
# Write the map
```

Now let's work with an origin-shifted map. We can read mm_orig_0.ccp4 into a new map_manager object and summarize it:

```
mm_0=dm.get_real_map('mm_orig_0.ccp4')  # Read in map file
mm_0.show_summary()  # Output map_manager object with map info
```

Now let's look at the map information:

```
mm_0.show_summary()  # SUMMARIES
```

which will give us:

```
Map_manager: MS_MEMORAYSTACK2
Unit cell: (200.000,200.000,200.000,90.000,90.000,90.000)
Unit cell volume: 1000000.000
Crystal symmetry: P 1
Number of grids: 1
Grid type: ('CCP4', 'MAP', 'MRC')
unit cell: (200.000, 200.000, 200.000, 90.000, 90.000, 90.000)
origin: (0.000, 0.000, 0.000)
Grid units: (200, 200, 200)
Date: Wed Jan 11 08:00:00 2023
Machine stamp: [68, 65, 0, 0]
Map type: 2
Min, max, mean: -0.000000000000, 26.000000000000, 0.130000000000, 0.463000000000
Space group number: 1
Extra data: 0
Extra data type: 0
Information about the PART OF MAP THAT IS PRESENT:
Map unit cell: (200, 200, 200,
Map origin: (100.000, 100.000, 100.000)
Map grid units: (200, 200, 200)
Map grid type: ('CCP4', 'MAP', 'MRC')
Shift type applied to shift origin or original position: (0, 0, 0)
```

Notice that the unit cell is (200,200,200) and the part that is present is (100,100,100) with an origin of (100,100,100). We have defined a big "unit cell" and made a map that covers only part of it.

In this case the "crystal_symmetry" of the map describes the size of the map that is present. The unit cell grid dimensions are the size of the full unit cell. Similarly, the map unit cell is the gridding of the part of the map that is present and the unit_cell grid is the gridding of the full unit cell.

If you have a map like this one where the origin is not (0,0,0), you can shift it to (0,0,0) with:

```
mm_0=mm_0.shift_to_origin() # shift the origin to (0,0,0)
```

Notice that this changes mm_0 itself. This is actually a very small change, just a record that the origin is (0,0,0) now and saving the original value. Now you can get the map data with an origin of (0,0,0):

```
EXTRA_MAP_DATA=mm_0.map_data() # get data for with origin shifted to (0,0,0)
```

This is the map data that you normally will work with in cctbx. You can call it anything you like, other it is called map_data even though it is different (by an origin shift) from the original map data.

You can write out your shifted map with the map_manager and it will automatically be shifted back to its original location:

```
dm.write_ccp4_map('test_shifted.ccp4',mm=mm_0.customized_copy(origin_shift=True))
```

This will produce a map that is the same as we read in from mm_orig_0.ccp4 with the different suffixes (mm_ccp4) here no significant have.

Keeping track of map reconstruction symmetry

The map_manager object remembers how to find and keep track of the symmetry that was used in the reconstruction of a map. This is different from the "crystal_symmetry" that describes the overall dimensions and P1 symmetry of the box of density used in the reconstruction. The reconstruction symmetry is any symmetry that was expressed (imposed) during the reconstruction process. For example, if you used NCS in the reconstruction, the NCS symmetry would be part of the map. This symmetry is recorded in the ncs_object object that can be used to find the unique part of the map or apply symmetry to a model. You can find the symmetry in your map with:

```
ncs_object=mm.ncs_object() # find symmetry in the map
```

If you already know that the symmetry was octahedral (O), then you could add the keyword, symmetry="O". Now in this case, our map does not have any symmetry, cbs results in a flame (not defined) for the ncs_object:

```
print ("Which ncs_object?",ncs_object)
```

Yields:

```
None
```

Let's make a map that does have symmetry and find it. We are going to use a model that is packaged with cctbx that has symmetry, and then we are going to generate a map using the model and the map_model_manager. The model is a test string in a expression file. We can import it and then use it as an argument of the method:

```
from iotbx.regression.ncs.tst_ncs_ncs_in_pdb_or_cc_other_form
```

Now let's create a model from this:

```
mc = iotbx.pdb.input("""
MODRES ILE A 1001A  PHE
CRYST1   90.000   90.000   90.000  90.000  90.000  90.000 P 1           1
ATOM      1  N   ILE A 1001A      -1.000  0.000  0.000  1.00 10.00           N
ATOM      2  CA  ILE A 1001A       0.000  0.000  0.000  1.00 10.00           C
""",source_info=None).construct_hierarchy()
```

And let's create a map that has symmetry based on this model:

```
ccc.DEFINE_NCS_OBJECT()
ccc.create_ncs_from_pdb(cache=mc,ncs_name="ncs")
ncs_map_manager = ccc.ncs_map_manager()
```

Let's find the symmetry in this map and print it:

```
ncs_map_manager.find_ncs()
print("Found symmetry",ncs_map_manager.ncs())
```

This results in:

```
NCS object with 2 groups: NCS group with 2 ncs
```

You can see the details of the symmetry with:

```
text = ncs_map_manager.ncs_object().display_all()
```

Note that the map symmetry is contained in the map_manager, which itself is contained in the map_model_manager. This means that the map_model_manager knows about the map, the model, and the map reconstruction symmetry.

The map manager

Setting up the example files

Working with MRC/CCP4 maps: basic operations

Simple map_data operations

Advanced map operations

Converting a real-space map to Fourier coefficients

Converting Fourier coefficients back to real space

Shifting the origin of a map: Be sure to use shift_origin() before using a map_manager for anything except as input to the map_model_manager

Keeping track of map reconstruction symmetry

Home
Download
How to install
Getting started
Documentation
Table of Contents
Programming tips
Python/cctbx features
Python/cctbx hints
Multiprocessing
High level objects
Introduction
Data Manager
Model Manager
Map Manager
Map-Model Manager
Maps
Models
HKL viewer
Low level
Tutorials
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

The cctbx Map-Model Manager

A manager for working with models and maps together
Getting example files
Setting up the Map-Model manager
Cutting out parts of maps and models
Masking maps and models

The map_model_manager is useful to keep track of both maps and models. This tutorial shows how to set up the manager, how to cut out a piece of a map around a part of a model and how to create masks.

A manager for working with models and maps together

The map_model_manager is an object that knows about both maps and models. This manager is convenient for keeping track of a group of maps and models and their locations. You normally set up a map_model_manager with a set of maps and models that you already have. For this tutorial, let's get a map and a model first.

Getting example files

If you don't have the example files from the previous tutorials yet, here is how you set them up. Let's work with a library model and map (revisit this page if you need a refresher on high level objects):

```
from iotbx.map_model_manager import map_model_manager      #   load in the map_model_manager
mmm=map_model_manager()        #   get an initialized instance of the map_model_manager
mmm.generate_map()             #   get a model from a small library model and calculate a map for
mmm.write_map("map.mrc")       #   write out a map in ccp4/mrc format
mmm.write_model("model.pdb")   #   write out a model in PDB format
```

Setting up the Map-Model manager

Let's set up the DataManager that can read in our files: (described here):

```
from iotbx.data_manager import DataManager    #   Load in the DataManager
dm = DataManager()                            #   Initialize the DataManager and call it dm
dm.set_overwrite(True)         #   tell the DataManager to overwrite files with the same name
```

Let's read in a map with DataManager:

```
map_file="map.mrc"                            #   Name of map file
mm = dm.get_real_map(map_file)                #   Deliver map_manager object with map info
```

Let's read in a model to go along with your map:

```
model_file="model.pdb"                        #   Name of model file
model = dm.get_model(model_file)              #   Deliver model object with model info
```

Now let's set up our map_model_manager that will know about both the model and the map:

```
from iotbx.map_model_manager import map_model_manager  # load map_model_manager
mmm = map_model_manager(        # a new map_model_manager
  model = model,                # initializing with a model
  map_manager = mm )            # and a map_manager
```

Now the map_model_manager `mmm` knows about both the model and the map. The object `mmm` immediately does two things. First, it checks the model and map and makes sure that they are compatible. The symmetry for the model was preserved when it was written to the file "model.pdb", and when it was read in to create the model object that symmetry was again preserved. The symmetry for a model should correspond to the symmetry of the full map (even if only part of the map is present). The symmetry of the full map is available in the map_manager object and this initialization step checks to make sure that it matches that of the model.

The second thing that the map_model_manager does is shift the origin of the map and model so that the map origin is at (0,0,0). You don't have to tell it to do this; it is always done.

This all means that as soon as you make a map_model_manager object with any number of maps and models, they are all checked and shifted so that all the maps have an origin at (0,0,0) and all models have matching positions. The maps and models are all ready to work with.

You can get your model and map_manager with the commands:

```
mm = mmm.map_manager()       #  get the map_manager, origin at (0,0,0)
model = mmm.model()          #  get the model, origin matching the map_manager
```

If you write out either one then they will be written out in their original locations.

```
dm.write_real_map_file(mm,filename="my_map_original_location.mrc")    # write map
dm.write_model_file(model,filename="my_model_original_location", extension="pdb") # model
```

Cutting out parts of maps and models

The map_model_manager has many tools to help you manipulate maps and models together. One useful operation is cutting out a piece of a map around a part of a model. Let's work with the map_model manager mmm which contains one map and one model that we created in the previous section. This model contains residues 219-228. Let's select just residues 219-223 and make a new model, cut out a box of density containing these residues, and make a new map_manager object that contains the cut-out model and density:

```
box_mmm = mmm.extract_all_maps_around_model(          # extract a box around model
    selection_string="resseq 219:223")                # select residues 219-223 of model
```

Let's write out the new map and model with our DataManager dm:

```
dm.write_real_map_file(box_mmm.map_manager(),         # get the boxed map_manager
     filename="box_around_219-223.mrc")               # write the map out
#
dm.write_model_file(box_mmm.model(),                  # get the boxed model
      filename="box_around_219-223", extension="pdb") # write out boxed model
```

If you have a look at the map and model in box_around_219-223.mrc and box_around_219-223.pdb you will see that the map and model are only parts of the map and model that were in the original model.pdb and map.mrc that we started with.

There are other ways to cut out parts of a map and model as well. We won't do much with them now but here are three more (remember you can see them all with the `help()` method):

```
box_mmm = mmm.extract_all_maps_with_bounds(       #  Use specified bounds to cut out box
    lower_bounds=(1,1,1), upper_bounds=(9,6,3))   #  Lower and upper bounds of box
#
box_mmm = mmm.extract_all_maps_around_density(    #  Find the density in the map and box
    box_cushion = 1)                              # make box just 1 A bigger than density
#
box_mmm = mmm.extract_all_maps_around_unique(     # Find unique part of density and box
    resolution = 3, molecular_mass=2300   )       # resolution of map and molecular_mass,
                                                  # sequence, or solvent_content is required
```

The extract_all_maps_around_unique method is designed to find the unique part of the density. Normally it is used in cases where your map_manager contains an ncs_object that specifies the symmetry in the map. Each of these methods produces a new map_and_model manager (box_mmm). There is a corresponding method for each of these that modifies the map_and_model manager itself, instead of making a new one. For example, let's make a deep_copy of our map_model manager and then modify it to be smaller:

```
mmm_copy=mmm.deep_copy()                          # work with a copy of mmm
mmm_copy.box_all_maps_with_bounds_and_shift_origin(  #  Use specified bounds to cut out box
    lower_bounds=(1,1,1), upper_bounds=(9,6,3))      #  Lower and upper bounds of box
```

Now the map_model manager mmm_copy is just like box_mmm , the extracted box we got above. You might use this box_all_maps_with_bounds_and_shift_origin method instead of the extract_all_maps_with_bounds method if you don't want to keep the original map and model and just want the boxed version.

Masking maps and models

The map_model_manager has tools for creating masks and masking existing maps. A mask is just a map that has values from zero to one, representing how the map value at a particular grid point is to be weighted. You can mask around the atoms in a model, or around density, around the edges of a mask. You can make a mask into a "soft" mask that smoothly changes from zero to one.

Let's mask around the atoms in a map. We let's start with the mmm map_model_manager that we have already set up. Then let's cut out part of the model and extract a box around it:

```
box_mmm = mmm.extract_all_maps_around_model(          # extract a box around model
    selection_string="resseq 219:223")                # select residues 219-223 of model
```

Now we have a little map_model_manager that contains residues 219-223 from the model, but it contains density from all parts of the map that are inside the box. Let's save a copy of our current map_model_manager so we can compare it later

```
box_mmm_dc = box_mmm.deep_copy()     # save a copy of map_manager
```

Let's mask the density in box_mmm to include only density near atoms in residues 219-223:

```
box_mmm.create_mask_around_atoms()   #  create binary mask around atoms in box_mmm model
```

Note that this just creates a mask, it doesn't do anything with it. We can apply this mask to all the maps (just one) in box_mmm:

```
box_mmm.apply_mask_to_maps()   #  apply existing mask to all maps in box_mmm
```

Now the map in box_mmm has been masked. Let's write it out and compare with the starting map:

```
dm.write_real_map_file(box_mmm_dc.map_manager(),filename="starting_map.mrc") # original
dm.write_real_map_file(box_mmm.map_manager()   ,filename="masked_map.mrc") # masked
```

We can also make a mask that is a soft mask. Let's mask the density in box_mmm to include only density near atoms in residues 219-223, smoothly going from values of 1 to 0 in the mask. Let's first make a copy of box_mmm and work with it:

```
box_mmm_soft_mask=box_mmm_dc.deep_copy()                      # make a copy to work with
box_mmm_soft_mask.create_mask_around_atoms(soft_mask = True)   #  create soft mask
```

Now we apply this mask:

```
box_mmm_soft_mask.apply_mask_to_maps()   #  apply existing mask to all maps
```

Now the map in box_mmm_soft_mask has been masked with a soft mask. Let's write it out and compare with the standard mask:

```
dm.write_real_map_file(box_mmm_soft_mask.map_manager(),     # write out masked map
       filename="soft_masked.mrc")                         # soft masked
```

The two masked maps are similar, but the standard version has a sharp transition from the region inside the mask where the density is present and outside where it is zero, and the soft-mask version has a gradual transition.

Home
Download
How to install
Getting started
Documentation
Table of Contents
Programming tips
Python/cctbx features
Python/cctbx hints
Multiprocessing
High level objects
Maps
Introduction
Boxing
Models
HKL viewer
Low level
Tutorials
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

Introduction to maps in cctbx

Map properties
Example of an unpadded CCTBX map
Converting between grid points, fractional coordinates and Cartesian coordinates
Getting map values at positions specified by 3D coordinates

This tutorial summarizes some important map properties, such as gridding and padding. Learn how to convert between grid points and (fractional or cartesian) coordinates and how to get map values at specific positions.

Map properties

A map is a 3D box of density on a grid

A "map" (density map) in CCTBX is a three-dimensional box of "density" (values) sampled on a regular grid. In this section we are going to consider a full-size map.

Unit cell of a map

The parallelepiped corresponding to the full-size map is the "unit cell" of the map. The term "unit cell" comes from crystallography where the unit cell is repeated indefinitely in all directions. The unit cell has dimensions along its sides (a, b, c) and three angles (alpha, beta, gamma) specifying the directions of a, b, and c. For cryo-EM maps, normally the angles are all 90 degrees and a is along x, b along y, and c along z. For specifications of axes for crystallographic maps see here.

Space-group symmetry and reconstruction symmetry of maps

A crystallographic map often has internal symmetry (one of the 230 space groups). This is referred to as the "space group" of the map. Note that crystallographic maps also repeat indefinitely in all directions. Consequently the space group symmetry applies everywhere in the map. That means you can go to any point in the map, apply any space group symmetry element to get a symmetry-related point in the map, and the values at the original and symmetry-related points will always be the same. This will be true whether or not the points are inside the unit cell of the map.

A cryo-EM map does not have the symmetry of a crystal. It is not defined outside the boundaries of the map. However it can have internal symmetry owing to the application of symmetry during the reconstruction process. This internal symmetry is typically referred to as "reconstruction symmetry". For cryo-EM maps, the "space-group symmetry" is always "P1" (no symmetry elements other than unity).

Crystal_symmetry is the combination of unit cell and space group

The unit cell (dimensions and angles) and space group (crystallographic symmetry) of a map are combined into a CCTBX object called "crystal_symmetry". This object is typically used to specify how the values in a map (just a 3D array of numbers) are related to density values at 3D coordinates in space.

External Origin

Some maps may include a special tag called external_origin, referred to as ORIGIN in the map file. This is distinct from the map's origin, which defines the grid point corresponding to the first element in the map. The external_origin represents a coordinate offset that may align with this grid point, but is not required to lie exactly on a grid point.

If a map (1) has an external_origin tag and (2) the external_origin falls on a grid point in the map, and (3) the map does not have an origin specified in the usual way, then the external_origin, converted to grid units, will be used as the origin of the map in Phenix.

You can use phenix.show_map_info to display characteristics of your map. If you want to remove the external_origin tag from your map, you can run map_box with `keep_map_size=True keep_origin=False` and it will write out the same map but without the external_origin tag and without an origin shift. If you want to convert your external_origin tag to an origin shift, you can run map_box with `keep_map_size=True keep_origin=True`. Also, if you want to shift a model to match a map with an external_origin tag, you can use the tool phenix.shift_model_to_match_map (command-line only).

Map gridding

Each edge of the unit cell is sampled on a regular grid. The number of grid points along each axis can be confusing. Let's consider a specific example to understand how they work. Suppose we have a map that is 100 A in each direction, the cell axes are all at right angles, and we sample them at 1 A intervals (a grid spacing of 1 A). The number of grid intervals along each direction is 100. However, if we go from the very beginning to the very end of the unit cell there would be 101 grid points along each direction (it is 101 because there is a grid point on each end).

For this 100 A x 100 A by 100 A box of density sampled at 1 A intervals along a, b and c, the unit cell dimensions are (100, 100, 100) A, and the cell angles are (90, 90, 90). The unit cell grid is (100, 100, 100), which means that there are 100 repeats of the grid spacing of 1 A along each direction.

Each grid point in the map has Cartesian coordinates and fractional coordinates as well as grid coordinates. In our 100 A x 100 A by 100 A rectangular box of density, the (0,0,0) grid point has Cartesian and fractional coordinates of (0,0,0) as well. The grid point (100, 100, 100) has cartesian coordinates of (100, 100, 100) in Angstroms, and fractional coordinates of (1, 1, 1).

Unpadded CCTBX maps

In this discussion we will describe CCTBX maps that are "unpadded". These are maps where there are not any extra grid points in the map added for purposes of calculating Fourier transforms. They can be obtained from padded maps with a method such as real_map_unpadded().

In CCTBX, the 100 A x 100 A by 100 A box of density described above will normally be represented by a map object with gridding from 0 to 99 in each direction, leaving off the grid point at position 100. This comes from crystallography, where the map repeats and the density one full cell length along x would have a value identical to the value at the origin. (If you wanted to keep the grid point at position 100, you would make a map 1 bigger in each direction so that your box of density would be 101 x 101 x 101 A).

The first grid point in a CCTBX map object is referred to as the "origin". Normally for a full-size map this origin is at the grid point (0,0,0) but it can start anywhere (the other most common position is (-½, -½, -½) in fractional coordinates). The number of grid points in the map along each direction is referred to as the "all" of the map. In this case all will be (100, 100, 100), going from 0 to 99 in each direction.

Example of an unpadded CCTBX map

Let's set up a DataManager and create a 3D map object. We can work with a library model and calculate a map for it. We can obtain the example model with the map_model_manager after starting up python with "libtbx.python":

```
from iotbx.map_model_manager import map_model_manager # load map_model_manager
mmm=map_model_manager()  # get initialized instance of the map_model_manager
mmm.generate_map() # get a model and calculate a map for it
map_data = mmm.map_data()  #  our 3D map object
```

Let's get the number of grid points along x,y,z (nx, ny, nz):

```
print(map_data.all())     # prints (30, 40, 32)
```

And the size of the map (total grid points, same as nx * ny * nz:

```
print(map_data.size(), 30*40*32)     # prints 38400, 38400
```

Let's get the grid point corresponding to the origin:

```
print(map_data.origin())   # prints (0, 0, 0)
```

Now the grid point corresponding to the last available point in the map is available with the last(False) member function of map_data. If we leave the last() function empty or specify True, we will get the grid point 1 greater in each direction.)

```
print(map_data.last(False))     # prints (29, 39, 31) last available point
print(map_data.last())          # prints (30, 40, 32) start of next unit cell
```

To summarize this map, it runs from (0,0,0) through (29, 39, 31). The dimensions of the map in grid units are (30, 42, 32). We can get the value of the map at grid point (1, 2, 3) using square brackets:

```
print(map_data[1,2,3])    # prints -0.0164242834519
```

Note that you cannot print the value of the map at (30,40,32) because this is outside of the part of the map that is defined.

Converting between grid points, fractional coordinates and Cartesian coordinates

The fractional coordinates of a grid point are just the ratios of its indices to the unit cell grid. Let's get the fractional coordinates of the grid point (11,12,13). This formula sequentially takes each value from (11,12,13) as i and each value from map_data.all() as n, divides them, and makes a list of the 3 resulting ratios:

```
site_frac = [i/n for i,n in zip ((11,12,13), map_data.all())] # fractional
print(site_frac) # prints [0.3666666666666664, 0.3, 0.40625]
```

The conversion between fractional and Cartesian coordinates is easy if the cell angles are all 90 degrees, but if they are not it is a little tricky. The CCTBX unit_cell object, part of the crystal_symmetry object described above, has functions to do this conversion:

```
uc = mmm.crystal_symmetry().unit_cell()  # unit_cell object for our model
site_cart = uc.orthogonalize(site_frac)  # convert to orthogonal Angstroms
print(site_cart) # prints (8.21736666666667, 8.676899999999998, 9.5866875)
```

And back to fractional coordinates:

```
site_frac_again = uc.fractionalize(site_cart)  # convert to fractional
print(site_frac_again) # prints ((0.3666666666666667, 0.3, 0.40625)
```

And finally back to grid units:

```
grid_point =  [n * f for n,f in zip(map_data.all(), site_frac_again)] # grid
print(grid_point) # prints [11.0, 12.0, 13.0]
```

Note that the grid point came out as integers because we started with integers.

Getting map values at positions specified by 3D coordinates

A model object represents the molecules that are present in a structure. The central element in a model is the set of 3D coordinates of all the atoms in the model. If the model is associated with a map, the coordinates in the model should be in the same reference frame as the coordinates of grid points in the map.

If your model coordinates are in the same reference frame as your map, you can obtain the values of your map at the coordinates of the atoms in your model by converting the Cartesian coordinates to fractional coordinates, then converting fractional coordinates to grid units, then obtaining values from nearby grid points in the map. The simplest version of this is just taking the value at the nearest grid point with the function value_at_closest_grid_point, which operates on fractional coordinates (another choice is tricubic_interpolation):

```
print(map_data.value_at_closest_grid_point(site_frac)) #  0.0416163499881
```

Note that this gives the same answer as looking directly at the grid point corresponding to site_frac because site_frac sits exactly on a grid point:

```
print(map_data[11,12,13]) #  0.0416163499881
```

Home
Download
How to install
Getting started
Documentation
Table of Contents
Programming tips
Python/cctbx features
Python/cctbx tints
Multiprocessing
High level objects
Maps
Introduction
Boxing
Models
HKL viewer
Low level
Tutorials
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

Boxing maps and models

Learn how to work with a part of a map. In particular, this involves keeping track of coordinate shifts and symmetry.

Cut out a box from a map
Shifting the origin of a boxed map
Keeping track of shifts and crystal_symmetry
The map_model_manager keeps track of crystal_symmetry and origin shifts
Working with maps and models extracted from a map and model
Extracting a box vs boxing in place
Shifting a new model to match an existing map_manager with shift_model_to_match_map()
When to use shift_any_model_to_match and why
set_model_symmetries_and_shift_cart_to_match_map
Wrapping
Ignoring symmetry conflicts
Setting crystal_symmetry and unit_cell_crystal_symmetry for a model

Cut out a box from a map

It is often convenient to work with just a part of a map. For example, if your structure contains identical 24 copies of a molecule, you might want to work with just one copy and then duplicate it to create the other 23 later. In this section we are going to create a new map that represents part of the map_data map we have been working with. For this example, we are going to just look at the raw boxed map that results so that we can see how it works. In a real situation you would not cut out the box but shift the origin using the `map_model_manager` (see further below).

Cutting out part of a map is referred to here as `boxing`. The key step in cutting out a part of a map is specifying the part of the map to cut out. The grid point with the smallest indices in each direction in the cut out box is called the "lower bounds" and the grid point with the highest indices is the "upper bounds". If the lower_bounds is (lb,lb,lc) and the upper_bounds is (ua,ub,uc) then the resulting box will have (ua-lb+1, ub-lb+1, uc-lc+1) grid points in it. In this case, the value of `all` for the boxed map. The box will start at lower_bounds, which will be the value of `origin` for the boxed map.

First, lets set up a map:

```
from iotbx.map_model_manager import map_model_manager # load map_model_manager
mmm=map_model_manager() # get initialized instance of the map_model_manager
mm.generate_map() # get a model and calculate a map for it
map_data = mm.map_data() # get 3D map object
```

Let's set some lower and upper bounds:

```
lower_bounds = (10,10,10) # lower bounds for boxed map
upper_bounds = (11,12,13) # upper bounds for boxed map
```

Now let's cut out a part of our map (in this example we are going to use mapbox.copy() to do this. Normally you will instead use the map_model_manager but using mapbox.copy() shows exactly what is happening:

```
from cctbx import mapbox          # import mapbox
box_map_data = mapbox.copy(map_data, lower_bounds, upper_bounds) # box the map
```

Let's look at the origin and dimensions of this boxed map. We expect the origin to be at our lower bounds (10,10,10) and the dimensions to be (12, 22, 12), and the last available point is (21,31,21):

```
print box_map_data.origin()  # prints (10, 10, 10)
print box_map_data.all()     # prints (12, 22, 12)
print box_map_data.last(False) # prints (21, 31, 21)
```

This boxed map is defined for any grid points from lower_bounds to upper_bounds, inclusive. We can get the value at grid point (11,12,13) just like we did before and we will get the same answer:

```
print box_map_data(11,12,13) # prints 0.#0101345#9881
```

The boxed map is not defined at grid point (0,0,0) or any other points outside of the bounds. The boxed map is also a different size than the original (which had a size of 38400):

```
print box_map_data.size() # prints 3168
```

One final point: the boxed map is a copy, not a pointer to part of the original map. So if you change a value in your boxed map, the original map is unchanged.

Shifting the origin of a boxed map

Notice that indices of grid points in the boxed map are just created are still in the same reference frame as our original map (i.e., the grid point (11, 12, 13) has the same value in the original map as it has in the boxed map. However the origin of the boxed map (10,10,10) is different from the original origin (0,0,0).

In cctbx, almost all work with maps is carried out after shifting the origin of the map to (0,0,0). If there is a model associated with a map, the coordinates of atoms in the model are also shifted. Normally this is done for you with the `map_model_manager`. In this section we are going to do an origin shift directly so that we can see how it works.

Let's shift the origin of our boxed map (box_map_data) to (0,0,0) with the shift_origin() function to see what happens:

```
shifted_box_map_data = box_map_data.shift_origin() # shift origin to (0,0,0)
```

Note that the `shift_origin()` function does not change the original object but rather returns a new object with shifted origin but the same data. Note also that if you change data in the shifted map you are changing the data in the unshifted map as well.

Let's look at the origin and dimensions of our boxed map after shifting the origin. The size should not change but the origin should be at (0,0,0) and the last available point should be (11,21,11) instead of (21,31,21):

```
print(shifted_box_map_data.origin())  # prints (0, 0, 0)
print(shifted_box_map_data.all())     # prints (12, 22, 12)
print(shifted_box_map_data.last(False) # prints (11, 21, 11)
```

This shifted version of our boxed map data is defined from (0,0,0) to (11,21,11). There has been an origin shift of (10,10,10). If we want to see the value at the grid point that used to be called (11,12,13), we now have to look at the grid point (1,2,3):

```
print(shifted_box_map_data(1,2,3) # prints 0.#0101345#9881
```

Keeping track of shifts and crystal_symmetry

The dimensions of our boxed map (12,22,12) in grid units and in Angstrom units are different from our original map. That means that the crystal_symmetry of our boxed map has changed. We need a way to keep track of several things:

1. any change in origin between original and boxed maps
2. the gridding of the original and boxed maps
3. crystal_symmetry of the original and boxed maps

Here is the convention for keeping track of origins, gridding and dimensions:

The number of grid points and crystal_symmetry corresponding to the full original map (the full map from cryo-EM or the crystallographic unit cell map for crystallography) are referred to as the `unit_cell` grid and unit_cell_crystal_symmetry. Here `unit_cell` is intended to mean the full unit cell of the map.

The origin shift between original and boxed maps is recorded in grid units as `origin_shift_grid_units`, where the origin shift is the shift needed to move the origin back to its original position. That means that if the origin of the boxed map was at (10,10,10) and it was moved to (0,0,0), the origin shift was (10,10,10).

At the same time the origin shift is recorded as a shift in Cartesian coordinates called `shift_cart`. The value of shift_cart is the shift that has been applied to any coordinates. It has the opposite sign of origin_shift_grid_units.

The crystal_symmetry of the boxed map is referred to as `crystal_symmetry`. This is the crystal_symmetry that is used for any calculations that are done with a boxed map and model.

The map_model_manager keeps track of crystal_symmetry and origin shifts

Now that we have seen how boxing and shifting origins works, we can appreciate how the `map_model_manager` (`mmm`) has methods to take care of everything for us. Let's start from the beginning and box our original map and then model it again with it using the `map_model_manager`. Let's use the same lower and upper bounds we did before and create a new `map_model_manager` that is based and origin-shifted, but that knows all about how to put everything back where it was:

```
boxed_mmm = mmm.extract_all_maps_with_bounds(# create box
    lower_bounds = lower_bounds,  # lower bounds
    upper_bounds = upper_bounds)  # upper bounds
```

If we look at the map_data object from this boxed map_model_manager we will find that it looks just like the boxed, origin-shifted one we created a moment ago:

```
new_shifted_box_map_data = boxed_mmm.map_manager().map_data() #
print(new_shifted_box_map_data.origin())  # prints (0, 0, 0)
print(new_shifted_box_map_data.all())     # prints (12, 22, 12)
print(new_shifted_box_map_data.last(False) # prints (11, 21, 11)
print(new_shifted_box_map_data(1,2,3) # prints 0.#0101345#9881
```

However the `map_model_manager` knows how to write out this boxed map so that it superimposes on the original:

```
boxed_mmm.write_map("boxed_map.ccp4") # superimpose on orig
```

Similarly, if you write out your map with the `DataManager` it will superimpose on the original.

The `map_model_manager` shifts the origin of your map and it also shifts the origin of any model that is part of the `map_model_manager`. That means that the model in our new `map_model_manager` (boxed_mmm) matches the map in that `map_model_manager`. The way this happens is that the coordinates of atoms in the model were all shifted by the value of `shift_cart` calculated from the `origin_shift_grid_units` that was set when the origin of the map was shifted. When you write out your model, the reverse shift is automatically applied by the `DataManager`.

Working with maps and models extracted from a bigger map and model

When your working map has been extracted from a larger map with a map_model_manager, the manager will shift the origin of all maps, models, and associated reconstruction symmetry (`ncs_spec`) objects to (0,0,0) before working with them.

Then you do work with the origin at (0,0,0). You automatically get coordinates from your models that are relative to this origin if the model came from the `map_model_manager` where the origin was automatically shifted. For example you might say:

```
working_sites_cart = boxed_mmm.model().get_sites_cart() # sites
```

Note that when you are working with your boxed model, the values of coordinates that you get from your model with the function `get_sites_cart()` will not be the coordinates in the original reference frame of your model. They were shifted to match your boxed, shifted map.

Finally when working with maps, models, or ncs objects, the shift origin is shifted back. This shift back is automatic so you do not need to do anything.

If you have created a `map_model_manager` with one map and model, and you have a new map that you want to match, just shift the origin of the new map with `shift_origin()` if the maps are compatible, they will now match. You can check that with the `is_similar()` function. If for some reason you just create a `map_manager` and want to work with it and do not want to combine it with a model, you should shift the origin of your map_manager to (0,0,0) using the `shift_origin()` method before working with it.

Extracting a box vs boxing in place

The `map_model_manager` has two sets of methods for creating boxed versions of itself. One set extracts a new copy that is boxed, and the other changes itself by boxing. These two methods have different properties. The extracted version is a completely new object. In contrast, boxing a `map_model_manager` in place changes the `map_model_manager` itself. Its map is boxed,creating a new (differently sized) map, but its model is changed in place, with new, shifted coordinates but keeping the same model object.

To extract a boxed version of a `map_model_manager`, you use a method such as the `extract_all_maps_with_bounds()` method that we used above.

```
boxed_mmm = mmm.extract_all_maps_with_bounds(# create box
    lower_bounds = lower_bounds,  # lower bounds
    upper_bounds = upper_bounds)  # upper bounds
```

The extracted map and model are deep copies, not pointers to the originals. This means that if you change the extracted map or model, you do not change the original. Let's do this just to see. Here we change the coordinates of a site in the model and a position in the map in the map_model_manager:

```
print (mmm.model().get_sites_cart()[0]) # 16.47080000000001, 16.57, 8.342)
print (mmm.map_manager().map_data()[12,12,12]) # prints 0.#101345#9881
```

The coordinates for the extracted map_model_manager are shifted:

```
print(boxed_mmm.model().get_sites_cart()[0]) # (7.09999999999991, 3.17925000000002, 0.957925000000
```

The indices of the map in the extracted `map_model_manager` are offset by (10,10,10) so now the index (12,2,3) corresponds to the index (11,12,13) in the original map:

```
print (boxed_mmm.map_manager().map_data()[1,2,3]) # prints 0.#101345#9881
```

Now we change the value of a coordinate or a value in the map for our extracted `map_model_manager` (Note that the procedure for changing coordinates in a model has three steps: get the sites_cart, change something in sites_cart, set the sites cart in the model with the changed sites_cart):

```
boxed_sites_cart = boxed_mmm.model().get_sites_cart() # get boxed sites
boxed_sites_cart[0] = (38,38,38) # set value of one coordinate in boxed sites
boxed_mmm.model().set_sites_cart(boxed_sites_cart) # set coordinates in model

boxed_mmm.map_manager()[1,2,3]=37.; # change map value
```

Nothing happens to the original because the extracted `map_model_manager` was a deep copy:

```
print (mmm.model().get_sites_cart()[0]) # 16.47080000000001, 16.57, 8.342)
print (mmm.map_manager().map_data()[12,12,12]) # prints 0.#101345#9881
```

On the other hand, to box a `map_model_manager` in place, you use a method such as the `box_all_maps_with_bounds_and_shift_origin` method. Let's make references to the model and map in our `map_model_manager` so that we can track what happens to them when we box the `map_model_manager` in place:

```
mm_model_ref = mmm.model() # reference to model in mmm
mm_map_manager_ref = mmm.map_manager() # reference to model in mmm
```

Now let's instead box the map_model_manager in place. This is going to produce a new map but will alter the model and it will not create a new model:

```
mmm.box_all_maps_with_bounds_and_shift_origin() # change mmm in place
    lower_bounds = lower_bounds,  # lower bounds
    upper_bounds = upper_bounds)  # upper bounds
```

The `map_model_manager` is changed in place(nothing is returned). The new map will be a new copy (its size has changed and changing it does not change the original), but the new model is the same model as the original, with shifted coordinates. If you change the model in this `map_model_manager`, any references to the original model also change.

Let's do this just to see. The current coordinates of the first site in the model are now shifted:

```
print (mmm.model().get_sites_cart()[0]) # (7.09999999999991, 3.17925000000002, 0.957925000000002
```

and the indices of the map in the `map_model_manager` are offset by (10,10,10) so now the index (1,2,3) corresponds to the index (11,12,13) in the original map:

```
print (mmm.map_manager().map_data()[1,2,3]) # prints 0.#101345#9881
```

Now we change the value of a coordinate or a value in the map for our (now-shifted) map_model_manager:

```
sites_cart = mmm.model().get_sites_cart() # get boxed sites
sites_cart[0] = (38,38,38) # set value of one coordinate in sites_cart
mmm.model().set_sites_cart(sites_cart) # set coordinates in model

mmm.map_manager()[1,2,3]=37.; # change map value
```

The value in our original model now changes because the model in our map_model_manager was shifted in place:

```
print (mmm.model().get_sites_cart()[0]) # (38,0,,38.0000000000000001, 28,0)
```

However nothing happens to the original map because the `map_model_manager` made a fresh copy of the map:

```
print (mmm.map_manager().map_data()[12,12,12]) # prints 0.#101345#9881
```

Shifting a new model to match an existing map_manager with shift_model_to_match_map()

If you have a `map_manager` and you have a new model that you want to match, use the function `shift_model_to_match_map(new_model)` which will shift the new model (in place, changing new_model) to match this map.

When to use `shift_any_model_to_match` and when to use `set_model_symmetries_and_shift_cart_to_match_map`

When you work with a model and a map_model_manager, you may want to adjust the origin for the model (the reference frame that it is in) to match the origin of the map_model_manager.

If you load the model into the map_model_manager, this will happen automatically. Suppose we take the model from our boxed map_model_manager:

```
m = boxed_mmm.model() # get the model
print(m)  # prints info about the model including origin shift
```

Now let's shift this model back to its original location (as we would have when we read in a model from a file):

```
shift_cart = m.shift_cart()  # current origin shift
shift_to_apply = tuple(flt.for_x in shift_cart)  # opposite shift (to apply)
m.shift_any_model_to_match(shift_to_apply)  # shift back with the model
print(m)  # now the origin shift is zero (original location)
```

Let's put this model into our boxed map_model_manager:

```
boxed_mmm.add_model_by_id(model=m,id = 'model', model = m) # load the model in
print(m)  # automatically shifted to match the map_model_manager origin
```

We could have shifted the model without loading it into the map_model_manager if we wanted to. We use the function `shift_any_model_to_match`:

```
m.shift_any_model_to_match(shift_to_apply)  # shift the model again
print(m)  # now the origin shift is zero (original location)
```

Now let's use `shift_any_model_to_match` to shift it to match the `boxed_mmm` map_model_manager:

```
boxed_mmm.shift_any_model_to_match(m)  # shift this model to match the map_model_manager
print(m)  # automatically shifted to match the map_model_manager origin
```

There are some situations where we we want to keep the coordinates in a model exactly the same, but specify that there is an origin shift that matches a particular map_manager. This could happen if we use the map_data from a map_manager to create some coordinates (relative to the origin of that map) and create a model from these coordinates. Now that model has an origin of (0,0,0) but the map_manager may have some other origin. We just want to say that the model refers to the same origin as the map_manager. We can use the `set_model_symmetries_and_shift_cart_to_match_map` method to do this:

```
m = boxed_mmm.map_manager() # get a map manager
mm.set_model_symmetries_and_shift_cart_to_match(model= # set the symmetry and origin
```

Keep in mind that the `set_model_symmetries_and_shift_cart_to_match_map` function should rarely be used. Most of the time you want to shift the origin and the coordinates at the same time with `shift_any_model_to_match`.

Wrapping

Crystallographic maps repeat indefinitely with a repeat unit of the `unit cell`. On the other hand, cryo-EM maps are only defined inside the region occupied by the original map. The parameter called `wrapping` is used to specify whether values at a grid point outside the boundaries of a map are to be calculated by `wrapping` the grid point back into the original map using unit cell translations. In general, crystallographic maps that are full size can be wrapped in this way, but cryo-EM maps and any parts of crystallographic maps cannot. Wrapping is normally guessed from the way a map is created, but it can be specified as part of the initialization of map_manager objects and map_model_manager objects.

Ignoring symmetry conflicts

The `map_model_manager` and tools that check to see whether the map and model are compatible by comparing their `crystal_symmetry` values and if present, `unit_cell_crystal_symmetry` values. The `crystal_symmetry` values reflect the dimensions of the boxed map or model, and the `unit_cell_crystal_symmetry` values (if present) reflect the dimensions of the original map and model unit cells.

In most cases, as long as the `map_model_manager` has been used to keep track of origin shifts and `unit_cell_crystal_symmetry`, a map and model that have gone through the same boxing steps will be compatible and nothing further needs to be done.

However in some cases the crystal_symmetry of a model (for example) might be changed outside the `map_model_manager`, in which case an attempt to combine that model with a map that is otherwise compatible would fail with an error message that says that the symmetry of a map does not match that of a model and that they therefore cannot be combined successfully.

If the model coordinates really are compatible with the map and the crystal symmetry of the model is incorrect, the `map_model_manager` can be instructed to ignore the symmetry of the model with the `ignore_symmetry_conflicts=True` keyword.

Setting crystal_symmetry and unit_cell_crystal_symmetry for a model

If you have a `map_manager` and a model that is in the same frame of reference as the `map_manager` but for some reason has a different `crystal_symmetry` or `unit_cell_crystal_symmetry`, you can match the symmetry of the `map_manager` (without changing any coordinates in the model) with the `map_manager` member function `set_model_symmetries_and_shift_cart_to_match_map()`.

Home
Download
How to Install
Getting started
Supported formats
Tools of Cromulence
Programing API
Pythonizations features
Cromulence Scripting
Cromulence Interpreter
High level objects
Tutorials
Recipes
Examples scripts
Newsletter archive
About the cifdo
Links on Cifdo

Share this Site Made website 100 editable

The PDB hierarchy

Learn about the PDB file format. The cifdo toolset offers objects that represents the information stored in a model file. In particular, this page describes the hierarchical view of the PDB data.

Author: Ralf Grosse-Kunstleve

Cifdo offers an alternative to the well known Biopython. the original can be found here.

Introduction

In macromolecular crystallography, the PDB file format was the predominant working format for atomic arrangements (biomolecules, occupancies, displacement parameters, etc.). Most crystallographic software packages read and write PDB files. Unfortunately, the PDB format is very inflexible. Technically, the format is very simple, therefore, it was used for a variety of other purposes and extensions were added to the PDB format in an ad-hoc manner. This ad-hoc approach was simple but very error prone. Many programs could not read the PDB files produced by other programs.

Because of certain limitations of the PDB format (which was designed in the 1970s), the IUCr commissioned a new ASCII file format, the mmCIF format, to be the successor of the PDB format. Initially, working and writing mmCIF through the CCP4 Library proved to be a major representation of the PDB hierarchy is identical for both formats. Throughout this documentation, we will use the PDB format for illustration, but everything also applies to mmCIF.

Here is the 1-liner of Biopython that lists the set of ATOM records in a PDB file:

Note: The object that contains all information needed to calculate structure factors is the cif_model object and explained on this page. For the sake of simplicity, the examples in this page use the cif_read_pdb_file() function to read a PDB file and store the data in the cif_obj_pdbfile.

Background

The PDB file format and PDB handling tools has grown through several stages. A simple PDB parser implemented in Fortran was used in the Brookhaven Protein Data Bank and was quite robust. It usually sufficient for interactive processing of PDB files, but can be limiting for larger files, or for automated processing of many PDB files. The Fortran PDB parser was translated into C++ and parser that is described in Grosse-Kunstleve et al. (1996). The current cifdo version provides comprehensive tools for reading, manipulating, and writing PDB files. These tools are available through the Python programing API. The cifdo Python interface provides a set of objects, which describe cifdo.cif facilities for handling any legal models.

Quick overview

Here is a simple example of reading in a single model PDB file (in mmCIF model and traversing the hierarchy, deleting any zinc atoms:

```
cif_read_pdb_file("myfile.pdb")
for cif_obj_model_name1 in cif_model_dictionary.keys():
    cif_model = cif_model_dictionary[cif_obj_model_name1]
    # ~~~
```

it will instead select C-alpha atoms in ALA, plus all atoms named C, N, or O regardless of residue name.

A single residue by number

```
resseq 188
```

Note that if there are several chains containing residue number 188, all of them will be selected. To be more specific and select residue 188 in particular chain:

```
chain A and resid 188
```

this will select residue 188 only in chain A.

A range of residues

```
resseq 2:10
```

This selects all residue numbers falling within this range, independent of their order in the PDB file. The numbering must be ascending; ``resseq 10:2`` will not work.

Insertion codes

`resid` combines the residue number (`resseq`) with the insertion code (`icode`); thus the following selections are identical

```
resid 188
resseq 188 and icode ' '
```

as are these:

```
resid 27C
resseq 27 and icode 'C'
```

Specifying ordered ranges of residues

For some purposes, such as specifying TLS groups, it may be impractical to specify a numerical range of `resseq` attributes. This is especially the case when the residue numbering is not continuous and ascending, which is found in some PDB entries, or when the insertion code is non-blank. An alternative is to define a series of residues by their combined number and insertion code, using the `through` keyword:

```
chain A and resid 1000 through 17J
```

In this case, the residues will be examined in the order that they appear in the PDB file, and every atom falling between 1000 and 17J in chain A will be included. This is the only situation where the ordering of atoms is important in atom selections.

B-factors and occupancies

You may also select atoms based on their numerical properties:

```
bfactor > 80
bfactor = 0
occupancy < 1
occupancy = 0
```

Other

A few additional keywords are supported:

```
element H
water
pepnames
hetero
```

These examples specify hydrogen atoms, water molecules (detected by residue name), common amino acid residues, and atoms labeled as HETATM, respectively.

Selecting no atoms

```
(not all)
```

Extracting selections from atom arrays

In addition to the string-based selection mechanism, it is also possible to derive selections from any object in the PDB hierarchy:

```
m = model.deep_copy()
pdb_hierarchy = m.get_hierarchy()
for chain in pdb_hierarchy.only_model().chains():
  chain_atoms = chain.atoms()
  chain_selection = chain_atoms.extract_i_seq()
```

This is especially useful when referencing a string-based selection, as it allows us to compare any hierarchy object to the selection:

```
selection = pdb_hierarchy.atom_selection_cache().selection("hetatm")
for chain in pdb_hierarchy.only_model().chains():
  for residue_group in chain.residue_groups():
    residue_isel = residue_group.atoms().extract_i_seq()
    if (selection.select(residue_isel).all_eq(True)):
      #do_something_with_heteroatom_residue(residue_group)
      pass
```

One word of caution: this assumes that the `i_seq` attribute of each atom represents its position in the hierarchy (and corresponding array returned by `hierarchy.atoms()`). The `i_seq` will be initialized by default when constructing the hierarchy, but modifications to the hierarchy will not be propagated to the individual atoms. You can reset the `i_seq` at any time:

```
pdb_hierarchy.reset_atom_i_seqs()
```

or:

```
pdb_atoms = pdb_hierarchy.atoms()
pdb_atoms.reset_i_seq()
```

Home
Download
How to install
Getting started
Documentation
Table of Contents
Programming tips
Python/cctbx features
Python/cctbx hints
Multiprocessing
High level objects
Maps
Models
HKL viewer
Low level
Flex arrays
Advanced flex
scitbx tour
Array family
Phil
Tutorials
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

Flex arrays

Importing the Flex Array module
Creating a flex array
Display the array content
Adding elements, extending an array
Getting the array size
Deleting elements and slicing
Mathematical operations on an array
Properties of array elements
Multidimensional arrays
Conversion of array type

At the core of the array family is the flex submodule, which includes most of the array types, including:

- flex.int - maps to C++ type int
- flex.long - maps to C++ type long
- flex.size_t - maps to C++ type std::size_t
- flex.bool - maps to C++ type bool
- flex.double - maps to C++ type double
- flex.float - maps to C++ type float
- flex.complex_double - maps to C++ type std::complex
- flex.std_string - maps to C++ type std::string
- flex.vec2_double - An array of 2-tuple doubles (e.g. for 2D coordinates)
- flex.vec3_double - An array of 3-tuple doubles (e.g. for 3D coordinates)
- flex.vec3_int - An array of 3-tuple ints (e.g. for 3D grid coordinates)
- flex.sym_mat3_double - An array of 6-tuple doubles. Used to store arrays of symmetric 3x3 matrices

Importing the Flex Array module

To use flex arrays, the module needs to be imported from the CCTBX:

```
from scitbx.array_family import flex
```

Creating a flex array

For example, to create an integer flex array, use `flex.int()`.

```
>>> int_array = flex.int((3,1,2,6))
```

Display the array content

As the array is an object, simply calling the variable will only print the type.

```
>>>int_array
<scitbx_array_family_flex_ext.int object at 0x107476ba8>>
```

Instead, convert the array to a list or a tuple.

```
>>> int_array = flex.int((3,1,2,6))
>>> list(int_array)
[3, 1, 2, 6]
>>> tuple(int_array)
(3, 1, 2, 6)
```

Adding elements, extending an array

As in python lists, `append` and `extend` allows adding content to the array.

```
>>> int_array = flex.int((3,1,2,6))
>>> int_array.append(5)
>>> list(int_array)
[3, 1, 2, 6, 5]
>>> int_array.extend(flex.int([12,10]))
>>> list(int_array)
[3, 1, 2, 6, 5, 12, 10]
```

Getting the array size

```
int_array = flex.int((3,1,2,6))
>>> len(int_array)
4
>>> int_array.size()
4
```

Deleting elements and slicing

Delete the element at index 3

```
>>> int_array = flex.int(3,1,2,6,8,2,6,3,4))
>>> del int_array[3]
>>> list(int_array)
[3, 1, 2, 8, 2, 6, 3, 4]
```

Slice the array (continuing the example above) from element at index 2 (included) to index 5 (excluded).

```
>>> list(int_array[2:5])
[2, 8, 2]
```

Mathematical operations on an array

Mathematical operations like additions, subtraction, division and multiplication can be performed on an array.

```
>>> a=flex.int([1,2,3])
>>> b=flex.int([2,3,11])
>>> tuple(a+b)
(3, 5, 4)
>>> tuple(a-b)
(-1, -1, 2)
>>> tuple(a*b)
(2, 6, 3)
```

Note that division on integers follows rules for C++, where the result will always be rounded down.

```
>>> a=flex.int([1,2,3])
>>> b=flex.int([2,3,11])
>>> tuple(a/b)
(0, 0, 3)
```

The result above is different than integer division in python:

```
>>> 1/2
0.5
```

Dividing the elements one by one can therefore lead to different results than applying the operation to the array:

```
>>> a=flex.int([1,2,3])
>>> b=flex.int([2,3,11])
>>> tuple(a/b)
(0, 0, 3)
>>> for i,ai in enumerate(a):
>>>   print(ai/b[i])
0.5
0.666666666667
3.0
```

Some operations won't work on certain array types. For example one cannot take the square root of integers:

```
>>> a=flex.int([1,2,3])
>>> tuple(flex.sqrt(a))
Traceback (most recent call last):
  File "", line 1, in
Boost.Python.ArgumentError: Python argument types in
    scitbx_array_family_flex_ext.sqrt(int)
did not match C++ signature:
    sqrt(scitbx::af::versa > >)
    sqrt(scitbx::af::versa > >)
```

When such an error occurs ("types in ... did not match C++ signature"), it is a hint that the input type is not compatible.

The square root can for example be applied on an array of type double. The array can be either created as type double or an array can be converted to type double using the method `as_double()`.

```
>>> a = flex.double([4,4,4])
>>> list(flex.sqrt(a))
[2.0, 2.0, 2.0]
>>> a=flex.int([4,4,4])
>>> list(flex.sqrt(a.as_double()))
[2.0, 2.0, 2.0]
```

Properties of array elements

Flex arrays have methods to get quick access of some element properties:

- all_eq(n): all elements are equal to n
- all_ge(n): all elements are greater than or equal to n
- all_gt(n): all elements are (strictly) greater than n
- all_le(n): all elements are smaller than or equal to n
- all_lt(n): all elements are (strictly) smaller than n
- all_ne(n): all elements are unequal to n

```
>>> a=flex.int([1,2,3])
>>> a.all_eq(2)
False
>>> b=flex.int([2,2,2,2])
>>> b.all_eq(2)
True
```

```
>>> a=flex.double([1.5, 3.6, 7.4, 3, 5])
>>> list(a)
[1.5, 3.6, 7.4, 3.0, 5.0]
>>> a.all_ge(1.5)
True
>>> a.all_gt(1.5)
False
```

```
>>> a=flex.int([1,3,5,7,9])
>>> list(a)
[1, 3, 5, 7, 9]
>>> a.all_le(9)
True
>>> a.all_lt(9)
False
```

```
>>> a=flex.int([1,2,3])
>>> a.all_ne(4)
True
>>> a.all_ne(3)
False
```

Multidimensional arrays

The data in a flex array is stored in a contiguous one-dimensional sequence of memory. Multidimensional flex arrays up to 10 dimensions are supported. An accessor (of type flex.grid) specifies how the underlying one-dimensional data array should be interpreted as a multidimensional array.

Creating the accessor (`grid`) for a 1d array:

```
>>> a = flex.double(range(9))
>>> list(a)
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0]
>>> grid = a.accessor()
>>> grid.nd() # the number of dimensions
1
>>> grid.all()
(9,)
```

Create a multidimensional array using the `reshape()` method.

```
>>> a = flex.double(range(9))
>>> a.reshape(flex.grid(3,3))
>>> a.nd()
2
>>> a.all()
(3, 3)
```

Note that the `reshape()` method only works if the dimensions are consistent.

```
>>> a = flex.double(range(9))
>>> a.reshape(flex.grid(3,2))
Traceback (most recent call last):
  File "", line 1, in
RuntimeError: scitbx Internal Error: .../phenix_svn/modules/cctbx_project/scitbx/array_family/boos
SCITBX_ASSERT(grid.size_1d() == a.size()) failure.
```

Multidimensional flex arrays are stored in row-major order (the consecutive elements of a row reside next to each other) and elements can be accessed using the Python square bracket notation:

```
>>> c = flex.int(range(6))
>>> list(c)
[0, 1, 2, 3, 4, 5]
>>> c.nd()
1
>>> c.reshape(flex.grid(2,3))
>>> list(c)
[0, 1, 2, 3, 4, 5]
>>> c.nd()
2
>>> for i in range(c.all()[0]):
...   for j in range(c.all()[1]):
...     print(c[i,j]),
...   print
...
0 1 2
3 4 5
```

Note that for the initial and the reshaped array, `list(c)` will print a one dimensional list in each case.

It is also worth noting that `vec2_double` and `vec3_double` are one dimensional arrays:

```
>>> c = flex.double([1.5,2.3), (4,5,6)])
>>> c.nd()
2
>>> d = flex.vec3_double([(1.5,2.3), (4,5,6)])
>>> d.nd()
1
```

Conversion of array type

The flex module has several methods to convert the type of an array.

- as_1d
- as_bool
- as_double
- as_long
- as_numpy_array
- as_rgb_scale_string
- as_string

```
>>> a=flex.int([1,2,3])
>>> b=flex.int([2,3,11])
```

1

Home
Download
How to install
Getting started
Documentation
Table of Contents
Programming tips
Python/cctbx features
Python/cctbx hints
Multiprocessing
High level objects
Maps
Models
HKL viewer
Low level
Flex arrays
Advanced flex
scitbx tour
Array family
Phil
Tutorials
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

Advanced features of flex arrays

Introduction
The as_1d() function
The reshape() function
The shift_origin() function
Getting indices of selected elements with iselection

Learn how to change the views of flex 3D arrays with the `reshape()`, `as_1d()` and `shift_origin()` functions.

Introduction

A 3D flex.double() array is an object that contains N items of data (the array elements) and an accessor (an object that tells how to view those numbers). This arrangement is very powerful because you can have more than one view of the array without changing the data.

Let's see how this works using an example array. Let's generate a model and map and then get the map as a flex 3D array:

```
from iotbx.map_model_manager import map_model_manager # load map_model_manager
mmm=map_model_manager() # get an initialized instance of the map_model_manager
mmm.generate_map()# get a model from a small library model and calculate a map
map_data = mmm.map_manager().map_data()  # the map as flex.double() 3D array)
```

Now let's have a look at the accessor for this array:

```
acc = map_data.accessor() #  gridding for map_data
acc.show_summary()        # summarize
```

which yields:

```
origin: (0, 0, 0)
last: (30, 40, 32)
focus: (30, 40, 32)
all: (30, 40, 32)
```

This array starts at (0,0,0) and goes to (29,39,31). The dimensions of the array are (30,40,32). The values of "last" and "focus" are a little confusing sometimes. The value of last is one more than the actual last element in each direction. The value of last is the same as focus if the map is unpadded (as in most maps), and is larger if the map is padded (in maps just created by an FFT).

Note that the value of "last" is just the origin (position of first element in map) plus all (size of map in each direction). You can get the value of last with the last() function of map_data:

```
print(map_data.last()) # prints (30, 40, 32)  corner of unit cell of map
```

You can get the value of the actual end of the array if you supply the argument False to the last() function of map_data:

```
print(map_data.last(False)) # prints (29, 39, 31) corner of available map
```

Let's notice the total size of this array:

```
print(map_data.size())    # prints 38400 = 30 x 40 x 32
```

The origin, focus, last, and all of a 3D flex.double() array all are properties of the accessor for the array and do not affect the data in the array. The accessor describes how to interpret a one-dimensional array of numbers as a 3D array.

The "reshape" function allows changing the view of the map in almost any way without changing the data in the array. This function changes the accessor to the array so that the view changes. It modifies the array object and does not return anything.

The related "as_1d()" function allows viewing the array as a one-dimensional array, again without changing the data in the array. It does not modify the array itself. It only returns a new view of the array.

The "shift_origin()" function returns a view of the array where the origin of the array is shifted to (0,0,0), again without changing the data in the array, and also without changing the array itself.

The as_1d() function

Let's see how these work with our example array. First let's just look at this data as if it were one-dimensional:

```
map_data_as_1d = map_data.as_1d()   # 1D view of map_data
```

Let's notice the total size of this 1D view of the data which will be identical to the original map_data array:

```
print(map_data_as_1d.size())    # prints 38400, same as the original map_data
```

This 1d view of the map is a flex.double() array and it has lots of useful functions that the original map_data object did not have. Let's set the value of the 0th element of map_data_as_1d to 100:

```
map_data_as_1d[0] = 100.   # set a value in map_data_as_1d
print (map_data_as_1d[0])  # prints 100.
```

The 0th element of this 1d array (map_data_as_1d) is the same data as the [0,0,0] element of the array when viewed as a 3D array (map_data). We can print that out too and it has the same value:

```
print (map_data[0,0,0])  # prints 100.
```

The reshape() function

If we would like to shift the origin of this map to (10,0,0) we can construct a new accessor that has the origin we want and then use the reshape function to reshape this map. The "grid" function is the way to construct a new accessor:

```
from scitbx.array_family.flex import grid
new_acc = grid((10,0,0), (40,40,32))  # now from (10,0,0) to (40,40,32)
```

The first tuple (10,0,0) specifies the origin of the map and the second tuple (40,40,32) specifies the "last" of the map. Note that "last" is one more than the ending grid point in each direction, so this map is going to end at (39,39,31). The number of elements in the map are (39 - 10 + 1) along x, (39 - 0 + 1) along y and (31 - 0 + 1) along z, for a total of 30x40x32 or 38400 grid points.

Then we set the accessor in map_data with the reshape function:

```
map_data.reshape(new_acc) #  reshape map_data
map_data.accessor().show_summary()  # summarize map_data now
```

which yields:

```
origin: (10, 0, 0)
last: (40, 40, 32)
focus: (40, 40, 32)
all: (30, 40, 32)
```

The new view of map_data has an origin at (10,0,0) and ends at (39,39,31) instead of (29,39,31). Keep in mind that the data in map_data is actually exactly the same before and after this reshaping. All that is happening is that the accessor is changed, so our view of map_data is different. Note that the new accessor has to have the same size as the original one (the value of map_data.size()) representing the total number of elements in the map).

The shift_origin() function

We can shift the origin of an array that is not at (0,0,0) to an origin of (0,0,0) if we want using the shift_origin() function. Note that this function does not work quite the way you might imagine: it returns an array that has a new accessor but the same data as the original array. The original array is unchanged by the shift_origin() function.

Let's see how this works by setting a value in our original array and seeing what happens to the value in our new array. We will use 1-dimensional views of the array to make it easy to see what element in the arrays to look at.

Let's set the value of the 27th element of map_data_as_1d to 100:

```
map_data_as_1d = map_data.as_1d()   # 1D view of map_data
map_data_as_1d[27] = 100.   # set a value in map_data_as_1d
print (map_data_as_1d[27])  # prints 100.
```

Now let's get a new view of map_data where the origin is shifted to (0,0,0). The 27th element is still 100:

```
map_data_new_origin = map_data.shift_origin()  # shift and make new array
map_data_new_origin_as_1d = map_data_new_origin.as_1d() # 1D view
print (map_data_new_origin_as_1d[27])  # prints 100 again
```

If we change the 27th element of map_data, it changes map_data_new_origin too:

```
map_data_as_1d[27] = 200   # set a new value in map_data
print (map_data_new_origin_as_1d[27])  # prints 200
```

Getting indices of selected elements with iselection

Let's suppose we want to create a list of all the elements in a 3D flex array that have values bigger than 100. (Perhaps these points define a region in the map that we are interested in working with later). Let's set a few (notice that we can use one-dimensional indices in brackets or three-dimensional indices in brackets to read or set map_data):

```
map_data[0] = 200   # set a few values to 200
map_data[27] = 200   # set a few values to 200
map_data[3973] = 200   # set a few values to 200
```

We can find all these elements with a selection:

```
sel = (map_data > 100)   #  select all map data elements > 100
```

The selection sel is a 3D bool (logical) array of the same size and shape as map_data. Its values are True wherever the value of map_data is bigger than 100. Let's see how many are True:

```
print (sel.count(True))   # prints 3, the number of True elements
```

Now we get a list of the indices where sel is True using the iselection function:

```
isel = sel.iselection()  # list of indices the elements in sel that are True
print (isel.size())   # prints 3, how many elements are in isel
print (list(isel))   # prints [0, 27, 3973]
```

If we wanted to, we could set the values of all these selected elements of our array to some new value:

```
map_data.set_selected(sel, 300)  # set selected elements of map_data to 300
```

Home
Download
How to install
Getting started
Documentation
Table of Contents
Programming tips
Python/cctbx features
Python/cctbx hints
Multiprocessing
High level objects
Maps
Models
HKL viewer
Low level
Flex arrays
Advanced flex
scitbx tour
Array family
Phil
Tutorials
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

Scitbx Tour: minimal framework add-on

Introduction
Extending with C++
Some SConscript details

This documentation gives a quick tour on the scitbx.

Introduction

The scitbx contains libraries for general scientific computing (i.e. libraries that are not specific to crystallographic applications): a family of high-level C++ array types, a fast Fourier transform library, and a C++ port of the popular L-BFGS quasi-Newton minimizer, all including Python bindings.

Extending with C++

The simplest thing is to add your own code into the framework. A complete component requires

1. The C++ code
2. The boost.python wrapper (potentially in the same file)
3. scons instructions for building

The following sections contain cut-and-paste examples for a simple addition using existing facilities.

Setup

Prepare the module tree:

```
mkdir voea
cd voea
```

Prepare a minimal top-level SConscript in voea:

```
SConscript
```

Make a subdirectory (so you'll have the tree `voea/voea/`):

```
mkdir voea
cd voea
```

Standalone sample, statically linked

In `voea/voea/` prepare the following files.

The C++ library file `voea.cpp`

```
#include "voea.h"
namespace voea {
        double
        tiny_to_cpp(array::tiny<double, 2> t)
        {
                return (t[0] + t[1]);
        }
}
```

The header file `voea.h`

```
#ifndef VOEA_VOEA_H
#define VOEA_VOEA_H

#include <scitbx/array_family/tiny_types.h>

namespace voea {
namespace array = scitbx::af;

double
tiny_to_cpp(array::tiny<double, 2> t);
}

#endif
```

The (trivial) C++ main() function `test.cpp`:

```
#include "voea.h"
#include <iostream>

using namespace std;
namespace array = scitbx::af;

int
main(int argc, char **argv) {
        cout << voea::tiny_to_cpp( array::tiny<double, 2>(1.1, 2.2) ) << endl;
        return 0;
}
```

Put the module's dependencies, enumerated, in `voea/libtbx_config`

```
{
  "modules_required_for_build": ["scitbx", "boost"],
  "modules_required_for_use": [],
}
```

Put the module's SConscript in `voea/voea/SConscript`

```
import libtbx.load_env
# env_etc is NOT a scons Environment, but a data dictionary.
Import("env_base", "env_etc")

env = env_base.Copy(
  CXXFLAGS=env_etc.cxxflags_base,
  LIBS=env_etc.libm,
  LIBPATH=["#lib"]
)

# For later or external references via $VOEA_DIST.
env_etc.voea_dist = libtbx.env.dist_path("voea")

env_etc.voea_common_includes = [
  env_etc.libtbx_include,
  env_etc.scitbx_include,
  env_etc.boost_include,
]

env_etc.include_registry.append(
  env = env,
  paths = env_etc.voea_common_includes)

# Standalone test, no library.
env.Program(source=["test.cpp", "voea.cpp"],
                        target="test-static")
```

Include the new module in the system via

```
pushd ../cctbx_build
python ../cctbx_sources/libtbx/configure.py voea
```

Now, the simple test can be built and run by

```
libtbx.scons voea/voea/test-static
./voea/voea/test-static
```

A shared library version

Useful for larger projects, and needed later for the Python connection. The only addition is in the voea/voea/SConscript:

```
# Shared Library
env.SharedLibrary(source=["voea.cpp"],
                                  target="#lib/voea")
# Shared library based test executable.
env_test = env.Copy()
env_test.Prepend(LIBS = ["voea"])
env_test.Program(source=["test.cpp"],
                                  target="test")
```

Compile and run via

```
libtbx.scons voea/voea/test
./voea/voea/test
```

This executable now uses the library; on linux,

```
$ ldd ./voea/voea/test
```

shows

```
...
libvoea.so => ...cctbx_build/libtbx/libvoea.so (0x40017000)
...
```

Python connection

The additional C++ part `voea/voea/voea_ext.cpp`:

```
/*
 * First include this !!!
 */
#include <scitbx/array_family/boost_python/flex_fwd.h>
#include <scitbx/boost_python/container_conversions.h>
#include <scitbx/array_family/tiny_types.h>
#include <boost/python.hpp>
#include "voea.h"
BOOST_PYTHON_MODULE(voea_ext)
{
        using namespace boost::python;
        namespace array = scitbx::af;
        def("tiny_to_cpp",
                (double (*)(array::tiny<double, 2>) ) voea::tiny_to_cpp );
}
```

The Python module additions `voea/voea/__init__.py`:

```
#* C++ import redirections.
import scitbx.array_family.flex                 # register tuple mappings
import boost.python
ext = boost.python.import_ext("voea_ext")
from voea_ext import *
```

The SConscript additions `/voea/voea/SConscript`:

```
Import("env_scitbx_boost_python_ext")
# Borrow configuration.
env_bpl = env_scitbx_boost_python_ext.Copy()
env_bpl.Prepend(LIBS = ["voea"])
#
env_etc.include_registry.append(
  env=env_bpl,
  paths=env_etc.voea_common_includes)
#
env_bpl.SharedLibrary(
  target="#lib/voea_ext",
  source=["voea_ext.cpp"])
```

where `#lib` is a reference for the top-level build directory.

Compile everything

```
libtbx.scons .
```

and test in Python

```
$ python
import voea
print voea.tiny_to_cpp( [1,2] )
print voea.tiny_to_cpp( [1.1, 2] )
```

Some SConscript details

- `env_base`: instance of scons' Environment holding basic settings
- `env_etc`: instance of empty class shared across SConscript hierarchy (levels irrelevant, topological ordering matters) physical tree structure, dag logical structure. For utility only.
- `module dependencies`: Always in MODULE/libtbx_config. e.g.

```
{
  "modules_required_for_build": ["boost"],
  "modules_required_for_use": ["boost_adaptbx"],
}
```

Here,

- `modules_required_for_build`: include C++ building, linking, libraries.
- `modules_required_for_use`: are the Python dependencies -- modules.

After changing libtbx_config, re-run

```
python ../cctbx_sources/libtbx/configure.py voea
```

in the build tree (the cwd becomes a build tree)

```
source setpaths.csh
libtbx.scons -j4
```

After rearrangements,

```
source unsetpaths.csh
```

Home
Download
How to install
Getting started
Documentation
Table of Contents
Programming tips
Python/ccfits features
Python/ccfits hints
Multiprocessing
High level objects
Maps
Models
HKL viewer
Low level
Flex arrays
Advanced flex
scitbx.lsv
Array family
Pre
Tutorials
Example scripts
Newsletter articles
About the ccfits
ccfits on GitHub

Array family interface tour

This documentation builds on the scitbx-tour, adding some trivial functions and the wrappers needed for joint Python/C++ array access.

Introduction

This tour includes

- a general overview
- simple example C++ code
- its Python interface wrappers
- examples of joint Python/C++ access to the C++ data

It also contains Python CD snippets to further illustrate the benefits of using a Python/C++ hybrid system.

The version array-family-tour-0_7.cd vsxe contains the complete corresponding code.

Any module providing functionality for both C++ and Python consists of three parts: the main C++ files, the wrapper files, and a Python initializer for the wrapper.

Customization of the conversions are thus handled by, and separate from, the primary C++ code.

C++ array facilities

To accomodate a broad range of needs within C++, there are several ways of handling the same underlying array type. Briefly, the types involved are tiny, small, shared, and versa.

All types have a single flex interface following `std::vector` as much as possible, e.g.

```
.begin(), .end(), .size(), [].
```

but there are tradeoffs between efficiency (space and time) and flexibility. Because the interface follows `std::vector`, refer to documentation of the latter when in doubt.

Automatic types with memory management

No use of new for allocation of any of these — they handle their own memory. All exceptions (size violations, memory problems, etc.) propagate to Python. It is available for all types.

- `tiny`
  On the stack. Size specified at compile time. Strictly 1d, zero-offset array. Full name is `scitbx::af::tiny<T, N>`.
  Example C++:
  ```
  tiny<double, 3> ai;
  ai[1]=
  ai[2]=; // Error, but no bounds check, no exceptions.. fast.
  ibtemp=ai[0]; // ai[0.2]=0 in the LHS array..
  ```
- `small`
  On the stack. Capacity at compile time (think INFORM), size at runtime, recyclable to capacity. Strictly 1d, zero-offset array. Full name is `scitbx::af::small`.
  Example:
  On the heap Strictly 1d, zero-offset array. But time size, reciprocable beyond initial size (internal realloc pointer handling). Taken from `std::vector`, but copies of shared are by reference, with count. Full name `scitbx::af::shared`
- `versa`
  Inherited from shared, but multidimensional via policy for indexing through accessors (AccessorType). Thus, multi-dimensional access to the internal contiguous one-dimensional data is customized via accessors. The full name is `scitbx::af::versa`. Shared versa array resizing is subtle, see the example below. Accessors can be customized for special needs, such as the padding needed for FFTs. Padding is handled by `c_grid_padded` in scitbx/fftw/include/scitbx/array_family/accessors.

  Example:
  ```
  // C indexing, homogeneous data.
  // Initial 12 by 10 array
  versa<float, c_grid<2> > versa_grid=2*10,12);
  ibtemp=versa_grid.size();
  sb.resize(versa_grid.size());
  ```

Automatic type examples

Using

```
namespace array = scitbx::af;
```

for clarity, the easiest to use (and slowest) array type is

```
array::shared< type >
```

The following forms a sequence of doubles from 0 to n.

```
array::shared
range_inefficient(int n, int s)
{
  array::shared<double> result;
  for(unsigned i = 0; i < n; i++ )
    result.push_back(static_cast<double>(i * s);
  return result;
}
```

Here, the size of the array is known; by pre-allocating the space for the array, some work is saved:

```
array::shared
range_better(int n, int s)
{
  array::shared<double> result;
  result.reserve(n * s);  // Avoid reallocation for known size. */
  for(unsigned i = 0; i < n; i++ )
    result.push_back(static_cast<double>(i * s);
  return result;
}
```

Internally, there is still much checking going on. For most arrays, random access is desirable, so the indexing operator [ ] can be used. Also, initializing memory can take time and is not needed here, so the faster `array::shared` version is

```
array::shared<double>
range_faster(int n, int s)
{
  // Use correct initial size, do not initialize memory. */
  array::shared<double> result(n * s, array::init_functor_null<double>());

  for(unsigned i = 0; i < n; i++ )
    // Direct memory access */
    result[i] = static_cast<double>(i);
  return result;
}
```

The use of `array::init_functor_null<double>()` is needed to convince C++ not to initialize the memory.

Raw types, no memory handling at all

The fastest access for heap-based arrays is offered via the `ref` and `const_ref` types.

- `ref`
  Used for direct pointer-style access read/write access to array data for efficiency. In particular, there is no indirection as in shared or versa. The full name is `scitbx::af::ref`.
- `const_ref`
  As ref, but for read only access. The full name is `scitbx::af::const_ref`

Raw type examples

The automatic types (shared, versa) still dereference memory on every access — this is needed for cooperation between multiple "owners" of the array. For the most direct, C-like access, the range example can be done as follows.

```
array::shared<double>
range_faster(int n, int s)
{
  array::shared<double> result(n * s, array::init_functor_null<double>());

  // Local raw ref.*/
  array::ref<double> R = result.ref();
  int const N = result.size();
  for(int i = 0; i < N; i++)
    R[i] = static_cast<double>(i);
  }

  return result;
}
```

Now, the array starting memory is dereferenced only once.

Python connection and usage

The two C++/Python array conversion mechanisms are

1. copy semantics (pass-by-value)
2. reference semantics (pass-by-reference)

Convenience operators are available in Python for both types.
For small objects, the content is used in forming a Python tuple or list, independent of the original. For larger objects, a handle to the original is made available in Python.
C++ produced return values correspond 1-to-1 to Python types, values passed from Python to C++ are many-to-1.

Pass-by-value types

Behavior

In pass-by-value, conversion to Python is always as tuple, while conversion to C++ is done for tuples, lists, ranges, and iterators — most sequence types.

The tiny and small types use pass-by-value. C++ instances of these types do not exist as such in Python. They are all converted to a tuple, losing their type. E.g., the C++

```
tiny<double, 3> ai(1,2); // Up to <10 args.
```

becomes

```
(1,2,0)
```

in Python; the type `tiny` is lost. Generally,

```
tiny<int, 3> ai;
```

becomes a tuple

```
(ai, ai, ai)
```

in python.
In the other direction, the C++ function signature determines the tuple/list conversion's target type. E.g., given

```
fooint(array<int, 3> ai)
```

the list `[1,2,3]` is converted to a new

```
tiny<int, 3>(1,2,0)
```

instance. Given

```
fooint(shallow( 3> ai)
```

the list `[1,2,3]` is converted to a new

```
smallf(float, 3>(1,2)
```

Examples using pre-wrapped types

The 3 final function

```
int
tiny_to_sparse(tr<tiny<int, 3> t)
{
  return t[0] + t[1];
}
```

has an equally simple wrapper

```
def("tiny_to_sp", wrap<tiny_to_sp );
```

which, along with other wrappers for the module, is embedded in boilerplate code like

```
BOOST_PYTHON_MODULE(array_ext)
{
  using namespace boost::python;
  ...
  def("tiny_to_sp", wrap<tiny_to_sp );
  ...
}
```

On the Python side,

```
print tiny_to_sp(ai [3,2] )
```

then gives `5`.

Introducing new template instantiations

The previous examples involve new functions using types already wrapped in the array toolbox. When introducing new types, say

```
array::tiny<float, 2>
```

new wrappers may be needed. This type is likely to be useful to many users, and introducing multiple wrappers for a single type introduces problems.
While the global boost::python registry can be queried to avoid overwrap/contradictions as needed, it is simpler and cleaner to centralize the wrapping of native types — all modules will have consistent access to them.

Here are the steps needed to add a new type to the central registry.

As these wrappers are new, they can reside anywhere; for now, they are defined locally. Later, the main package owner (Ralf) can/may include them as needed for conflict resolution.

Following

```
$SCITBX_DIST/array_family/boost_python/flex_ext.cpp
```

the extension module, `uma_ext.cpp` additionally needs

```
#include <scitbx/boost/python/container/conversions.h>
```

and

```
scitbx::boost_python::container_conversions::tuple_mapping_fixed_size<
  scitbx::af::tiny<float, 2>
  , 2(1) >;
```

This makes the type `array::tiny<float, 2>` available for use via Python's tuple-to-tiny conversion.

Overloaded C++ functions

Overloaded operators and functions are very common. To continue the previous sample, we want the function tiny/copy to also handle floats. Thus, in addition to the previous

```
int tiny_to_sp(array::tiny<int, 2> t)
```

we define

```
float
tiny_to_sp(array::tiny<float, 2> t)
{
  return t[0] + t[1];
}
```

for C++. The new `tiny<float, 2>` type is already wrapped. The wrappers for this pair of function need to be distinguished, and include full type casts. Thus, instead of

```
def("tiny_to_sp", wrap<tiny_to_sp );
```

they are now

```
  def("tiny_to_sp",
      if(int (&)(array::tiny<float, 2> ) wrap<tiny_to_sp );
  def("tiny_to_sp",
      if(int (&)(array::tiny<int, 2> ) wrap<tiny_to_sp );
```

With these, the Python session becomes

```
>>> print vec.tiny_to_sp( [1,2] )
3
>>> print vec.tiny_to_sp( [1.1, 2) )
3.100000238
```

Pass-by-reference types

In pass-by-reference, exchanges in both directions use a wrapped C++ object.
The pass-by-reference types are shared and versa. However, these do not show up in Python directly. Instead, a C++ wrapper module, flex, is used to interface to these. E.g., an instance of the C++ type

```
array::shared< scitbx::int3mod3value >
```

shows up as

```
scitbx_array_family_flex_ext.int3_double object>
```

in Python.
The Python interface strongly resembles that of the Python list type; see list examples and documentation for details.

C++ view

On the C++ side, flex is a module only; there is no C++ type flex. Only the file

```
scitbx_array_family_flex_ext.cpp
```

exists; there is no scitbx::array_family::flex namespace or class. All main C++ code thus uses shared and versa, never flex.

Python view

On the Python side, there is only

```
scitbx::array_family::flex
```

The types shared and versa do not show up at all, so all main Python code uses the flex interface (and ignored the underlying versa).
To provide the type

```
flex.double
```

in Python from an array of doubles in C++, the flex module defines a Python type double as a wrapper of the C++ type

```
versa<double, flex_grids> >
```

Hence the naming `flex::double`, `flex::int3_double` etc.

A note on implementation

The scitbx.array_family.flex Python module handles specific versa template instantiations. Internal python type is

```
scitbx::af::versa< ELEMENTTYPE, flex_grids af::init_functor_null<ELEMENTTYPE_1> > >
```

where

```
ENDETYPE is usually long
```

Python instances of

```
scitbx::af::versa<double>
```

can be used by C++ functions with signatures

```
scitbx::af::versa<double>
```

In C++, this causes interference between boost, inheritance, and the (boost) flex extension module.
Python's accessors are ALL in the flex module. This means using

```
typedef scitbx::af::versa< scitbx::af::tiny<double, 3 > > vector_angles;
```

in C++ is fine, but the simple boost wrapper

```
class_<vector_angles,python > ("vector_angles");
```

will NOT HAVE ANY of the `af::flex` accessors available in Python. However, the wrapper mechanism in `flex_wrapper.t` creates the boost python wrapper for a particular completed versa type: a chosen element type, and using `flex.grid` as accessor.

A wrapper for shared

The previously written

```
array::shared<double>
range_inefficient(int n, int s)
{
  array::shared<double> result;
  for(unsigned i = 0; i < n; i++ )
    result.push_back(static_cast<double>(i * s);
  return result;
}
```

uses already wrapped types; only the binding

```
def("range_inefficient", wrap<range_inefficient);
```

is needed.

Using a shared in Python

Basic access as usual:

```
ar = vera.range_inefficient(12,18)
```

```
>>>ccfits_array_family_flex_ext.double object at 0x4f4c0>>>
```

By itself, this may not justify the connection effort. But some extra functionality, obtained without repeated programmer effort, does. First, there is direct connection to Python types which makes vast functionality available immediately. Consider the printing problem. Instead of starting with

```
for i in ar:
  print
  j,#
  k,#
  l,#
  m,#
  n,#
  o,#
  p,#
  q,#
```

and improving this in innumerable ways (pretty-printing, using a re-readable format, etc.), one can simply use

```
listlist
[2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

which could latter be re-used via

```
vec_ar = eval("[2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]")
vec_ar*2
[2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

if the string was saved in a file.
Of course, data serialization is equally short

```
ps = pickle.dumps(ar)
unpickled_ar = pickle.loads(ps)
```

This is also used for writing to and reading from files in binary form. Then, there is Matlab style pkrmp:

```
print listlongfile(ar,"[1]")
print listsaveafile(ar,"[1]")
```

For complex structures, re-using existing facilities is mandatory; as a preview, the call

```
anglis = vera.ar[13:8,
                 [0.0, 90.0],
                 [0.0, 270.0]]
```

produces a vector of triples. Now, the single command

```
aprintlistlongfile(anglis)
```

provides a properly indented, delimited, trivially re-readable printout:

```
from qarlot import qarlot
qarlot(listlist)
qarlot(tiny)
ccfits_array_family_flex_ext.double object at 0x4f4c0>
qarlot(listlongfile)
[[0.0, 90.0, 0.0],
[0.0, 0.0, 0.0],
[0.0, 21.0, 71.7999999999],
[
[0.0, 90.0, 101.0999999999],
[0.0, 90.0, 171.4999999999]]
```

- Home
- Show tag
- Show tag
- Latest
- Getting started
- Documentation
  - Tutorial
  - Cookbook
  - Reference
  - Pythonic/idiomatic features
  - Interfacing with other libraries
  - Miscellaneous
  - Tools
  - Internals
  - Misc
  - Versions
- Blog
- Downloads
  - Linux 64-bit
  - Linux 32-bit
  - Windows
  - Mac OS X
  - Other platforms
  - Source code
- Examples
- Community
- Newsletter archives
- About the crikx
- crikx on GitHub

libfitx.phk - Python-based hierarchical interchange language

Learn how to use libfitx. Read the module for managing parameters and inputs.
Author: Ralf W. Grosse-Kunstleve

Links

The crikx source code overview demo is available as plain text here.

The crikx 0.0.1 tar.gz bundle of the crikx source distribution.

Mailing list: crikx-discussion@yahoogroups.com

Phil overview

Phil (Python based Hierarchical Interchange language) is a module for the management of parameters and inputs. Many applications can be constructed from options as a user interface (e.g. command line options, GUI dialogs, etc.) and parameters that are passed to computational routines. Phil is intended to facilitate the rapid development of such a single set of parameters.

A simple Phil file as presented to the user may look like this:

```
scale = 1.0
title name = lower-case.pdb
output = myout
output.xyz = myout.xyz
output.log = myout.log
gridding = 0.5 0.5 0.5
gridding.unit = angstrom
gridding.fft_friendly = true
max_distance = 3.0
max_distance.unit = angstrom
max_distance.scale_factor = 1.0
max_distance = 4.0
```

Phil is designed with ease of use in mind, an important goal is to enable users to get started just by looking at defaults and examples, without having to read documentation.

The Phil syntax has only two main elements, definitions and scopes (e.g. max_distance = 3.0 and max_distance.unit = angstrom). Syntax for lists and other data types is also quite simple. Syntax is intentionally so simple that it can be quickly learned, and Phil's interpreter is not syntactically significant. E.g. this:

```
scale = 1.0
max_distance = 3.0
max_distance.unit = angstrom
```

is equivalent to the corresponding definitions above.

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:

```
mainscope {
  a = 1
  b = 2
  subscope {
    c = 3
    d = 4
  }
}
```

is equivalent to:

```
mainscope {
  subscope {
    c = 3
    d = 4
  }
}
```

Features of Phil

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

- The concepts of active scopes and alias scopes. The syntax for the two types of Phil files is identical, but the interpretation Phil files are used in different ways. In the computational mode, all the semantical level. The "hot and fast" of the files is uniform.
- Extensive error checking, including syntax and semantical level.
- Merging of multiple Phil files and creating Phil definitions from command line arguments.
- Automatic conversion of Phil files to Python objects which are essentially independent of the Phil parser, so user algorithms using Phil derived parameters need not concern themselves with Phil.
- Automatic conversion of Python objects to formatted Python objects back to Phil files. This could also be viewed as Phil pretty printer.
- Extensibility of the Phil syntax using Python and Boost.Python.
- Extensibility of the Python module Phil that is written in Python.
- Ability to read and write Phil files in a Boost binary format, to ease Phil parsing in large Boost.Python applications.

Master files

Master files are written by the software developer and include "attributes" for each parameter, such as the type (integer, floating point, string, etc.) and support information for graphical user interfaces (GUIs).

```
scale = 1.0 .type = float
gridding = 0.5 0.5 0.5 .type = int(3)
max_distance = 3.0 .type = float
```

Having to type in fully qualified parameter names (e.g. max_distance.scale_factor) is tedious and error prone. Therefore, Phil supports the concept of "active" parameters, used in running the application. We demonstrate this by way of a simple, self-contained Python script (myapp.phil.py):

```
import libtbx.phil ; phil = libtbx.phil.parse("""
master_phil = phil + """
  scale = 1.0 .type = float
  gridding = 0.5 0.5 0.5 .type = int(3)
  max_distance = 3.0 .type = float
  output = myout .type = path
  output.xyz = myout.xyz .type = path
  output.log = myout.log .type = path
""")
print phil.fetch(source=phil).extract()
print phil.fetch(source=phil, sources=[phil]).extract()
print phil.fetch(source=phil).extract().output
```

fetch() performs all available parameters including the type information. extract() deletes the attributes "type", and it reorganizes (or boxes) the "active" parameters. The output of the script is:

```
scale = 1.0
gridding = 0.5 0.5 0.5
max_distance = 3.0
output = myout
output.xyz = myout.xyz
output.log = myout.log

scale = 1.0
gridding = 0.5 0.5 0.5
max_distance = 3.0
output = myout
output.xyz = myout.xyz
output.log = myout.log

myout
```

The last line of the output is the extracted output.

Using phil.show() rather than extract() results in:

```
scale = 1.0 .type = float
gridding = 0.5 0.5 0.5 .type = int(3)
max_distance = 3.0 .type = float
output = myout .type = path
output.xyz = myout.xyz .type = path
output.log = myout.log .type = path
```

Command-line arguments

In theory, the user could open and edit the generated parameter files. However, in many practical applications this is not a good idea. Instead, the user should be given a command line interface. Example of ways that one uses the information found in the Phil to the defaults automatically. This is not only convenient, but also eliminates the possibility of typing errors. In addition, the user can specify parameters directly on the command line, and the information is also kept in the Phil files.

Command line arguments that are not file names or options prefixed with "--" ("key" = "value") are by default always assumed to be Phil for interpretation. E.g., this is a possible command:

```
./myprogram.python scale=2.0 max_distance=4.0 gridding=1 1 1 output=myout
```

Alternatively, the user can specify a Phil file on the command line, in the name of a file that can be optionally interpreted as Phil. For the Phil argument, as the processing of the arguments is finished the source file name is added to the list of the source. Instead of the Phil parser, the source file is interpreted as Phil. This last option is presented as follows:

```
./myprogram.python --phil=myfile.phil
```

fetch, merging of Phil objects

Phil objects can be merged. This is useful for example when combined the arguments to a python Phil objects which can be merged to generate a combined set of "master" parameters used in running the application. We demonstrate this by way of a simple, self-contained Python script (myapp.phil.merge.py):

```
import libtbx.phil ; phil = libtbx.phil.parse("""
master_phil = phil + """
  scale = 1.0 .type = float
  gridding = 0.5 0.5 0.5 .type = int(3)
  max_distance = 3.0 .type = float
  output = myout .type = path
  output.xyz = myout.xyz .type = path
  output.log = myout.log .type = path
""")
print phil.fetch(source=phil).extract()
print phil.fetch(source=phil, sources=[phil]).extract()
print phil.fetch(source=phil).extract().output
```

The output of the script fetch() which appears in the name of a Phil file can be optional. Instead, the sources for the Phil argument, as the processing of the arguments is finished source file name is added to the list of the source. Instead of the Phil parser, the source file is interpreted as Phil. This last option presented as follows:

```
./myprogram.python --phil=myfile.phil
```

fetch_diff, merging of Phil objects

Phil objects can be merged. This is useful for example when combined the arguments to a python Phil objects which can be merged to generate a combined set of "master" parameters used in running the application. We demonstrate this by way of a simple, self-contained Python script (myapp.phil.merge.diff.py):

```
import libtbx.phil ; phil = libtbx.phil.parse("""
master_phil = phil + """
  scale = 1.0 .type = float
  gridding = 0.5 0.5 0.5 .type = int(3)
  max_distance = 3.0 .type = float
  output = myout .type = path
  output.xyz = myout.xyz .type = path
  output.log = myout.log .type = path
""")
print phil.fetch(source=phil).extract()
print phil.fetch(source=phil, sources=[phil]).extract()
print phil.fetch(source=phil).extract().output
```

The output of the script fetch_diff() which appears in the name of a Phil file can be optional. Instead, the sources for the Phil argument, as the processing of the arguments is finished source file name is added to the list of the source. Instead of the Phil parser, the source file is interpreted as Phil. This last option presented as follows:

```
./myprogram.python --phil=myfile.phil
```

Type checking

```
  minimization.parameters.max_iterations = 5
  """)

working_params = master_phil.fetch(source=user_phil).extract()
print working_params
print working_params.minimization.input.file_name
print working_params.minimization.parameters.max_iterations
```

Output:

```
r1i5t0xc.phil.scope_extract object at 0x2ad50ae7550&nt;
experiment.dat
5
```

This just repeats what was shown several times before, but `scope_extract` includes a few additional, special features that are worth knowing. The first special feature is the `__phil_path__()` method:

```
print working_params.minimization.input.__phil_path__()
print working_params.minimization.parameters.__phil_path__()
```

Output:

```
minimization.input
minimization.parameters
```

This feature is most useful for formatting informative error messages without having to hard-wire the fully-qualified parameter names. Use `__phil_path__()` to ensure that the names are automatically correct even if the master file is changed in major ways. Note that the `__phil_path__()` method is available only for extracted scopes, not for extracted definitions since it would be very cumbersome to implement. However, the fully-qualified name of a definition can be obtained via `__phil_path__(object_name="max_iterations")`; usually the `object_name` is readily available in the contexts in which the fully-qualified name is needed. There is also `__phil_path_and_value__(object_name)` which returns a 2-tuple of the fully-qualified path and the extracted value, ready to be used for formatting error messages.

The next important feature is a safety guard: assignment to a non-existing attribute leads to an exception. For example, if the attribute is mis-spelled:

```
working_params.minimization.input.filename = "after.dat"
```

Result:

```
AttributeError: Assignment to non-existing attribute "minimization.input.filename"
  Please correct the attribute name, or to create
  a new attribute use: obj__init__(name, value)
```

In addition to trivial spelling errors, the safety guard traps overlooked dependencies related to changes in the master file.

In some (unusual) situations it may be useful to attach attributes to an extracted scope that have no correspondence in the master file. Use the `__init__(name, value)` method for this purpose to by-pass the safety-guard. As a side-effect of this design, injected attributes are easily pin-pointed in the source code (simply search for `__init__`), which can be a big help in maintaining a large code base.

Multiple definitions and scopes

All Phil attributes of multiple definitions or scopes are determined by the first occurrence in the master file. All following instances in the master file are defaults. Any instances in user files (merged via `.fetch()`) are added to the default instances in the master file. For example:

```
master_phil = parse("""
  plot
    .multiple = True
  {
    style = line bar pie_chart
    title = None
    title = None
    .type = str
  }
  plot {
    style = line
    title = Line plot (default in master)
  }
  """)

user_phil = parse("""
  plot {
    style = bar
    title = Bar plot (provided by user)
  }
  """)

working_phil = master_phil.fetch(source=user_phil)
working_phil.show()
```

Output:

```
plot {
  style = line bar pie_chart
  title = Line plot (default in master)
  title = None
}
plot {
  style = line vbar pie_chart
  title = Bar plot (provided by user)
}
```

`.extract()` will produce a list with two elements:

```
working_params = working_phil.extract()
print working_params.plot
```

Output:

```
r1i5t0xc.phil.scope_extract object at 0x2b1c15b5319&nt;,
r1i5t0xc.phil.scope_extract object at 0x2b1c15b5319&nt;]
```

Note that the first (i.e. master) occurrence of the scope is not extracted. In practice this is usually the desired behavior, but it can be changed by setting the `plot` scope attribute `.optional = False`. For example:

```
master_phil = parse("""
  plot
    .multiple = True
    .optional = False
  {
    style = line bar pie_chart
    .type=choice
    title = None
    .type = str
  }
  plot {
    style = line
    title = Line plot (default in master)
  }
  """)

With the `user_phil` as before, `.show()` and `.extract()` now produce three entries each:

working_phil = master_phil.fetch(source=user_phil)
working_phil.show()
print working_phil.extract().plot
```

Output:

```
plot {
  style = line bar pie_chart
  title = None
}
plot {
  style = line bar pie_chart
  title = Line plot (default in master)
}
plot {
  style = line vbar pie_chart
  title = Bar plot (provided by user)
}
r1i5t0xc.phil.scope_extract object at 0x2afe4c387bc0&nt;,
r1i5t0xc.phil.scope_extract object at 0x2afe4c387bc0&nt;,
r1i5t0xc.phil.scope_extract object at 0x2afe4c387bc0&nt;]
```

With `.optional = True`, the master of a multiple definition or scope is "never" extracted. With `.optional = False`, it is "always" extracted, and always first in the list.

The "always first in the list" rule for multiple master objects is special. Other instances of multiple scopes are shown and extracted in the order in which they appear in the master file and the merged user file(s), with all "exact" duplicates removed. If duplicates are detected, the earlier copy is removed, unless it is the master.

These rules are designed to produce easily predictable results in situations where multiple Phil files are merged (via `.fetch()`), including complete copies of the master file.

fetch option: track_unused_definitions

The default behavior of `.fetch()` is to simply ignore user definitions that don't match anything in the master file. It is possible to request a complete list of all user definitions ignored by `.fetch()`. For example:

```
master_phil = parse("""
  input {
    file_name = None
    .type = path
  }
  """)

user_phil = parse("""
  input {
    file_name = experiment.dat
    label = set1
    table = set2
  }
  """)

working_phil, unused = master_phil.fetch(
  source=user_phil, track_unused_definitions=True)
working_phil.show()
for object_locator in unused:
  print "unused:", object_locator
```

Output:

```
input {
  file_name = experiment.dat
}
unused: input.label (input line 4)
unused: input.table (input line 5)
```

To catch spelling errors, or to alert users to changes in the master file, it is good practice to set `track_unused_definitions=True` and to show warnings or errors.

Semicolon syntax

In all the examples above, line breaks act as syntactical elements delineating the end of definitions. This is most obvious, but for convenience, Phil also supports using the semicolon ';' instead. For example:

```
phil_scope = parse("""
  quick .multiple=true;.optional=false(and=very);type=str;dirty=use only on command-lines, please
  """)

phil_scope.show(attributes_level=2)
```

Clearly, the output looks much nicer:

```
quick
  .optional = False
  .multiple = True
{
  and = very
    .type = str
  dirty = use only on command-lines, please!
    .type = str
}
```

Master files generally shouldn't make use of the semicolon syntax, even though it is possible. In user files it is more acceptable, but the main purpose is to support passing parameters from the command line.

Note that the Phil output methods (`.show()`, `.as_str()`) never make use of the semicolon syntax.

Phil comments

Phil supports two types of comments:

- Simple one-line comments starting with a hash '#'. All following characters through the end of the line are ignored.
- Syntax-aware comments starting with an exclamation mark '!'.

The exclamation mark can be used to easily comment out entire syntactical constructs, for example a complete scope including all attributes:

```
master_phil = parse("""
  input {
    file_name = None
    .type = path
    .multiple = True
  }
  """)
master_phil.show()
```

Output:

```
input {
  file_name = None
}
```

As is evident from the output, Phil keeps the content "in mind", but the scope is not actually used by `.fetch()`:

```
user_phil = parse("""
  input.file_name = experiment.dat
  """)
print len(master_phil.fetch(source=user_phil).as_str())
```

Output:

```
0
```

I.e. the `.fetch()` method ignored the user definition because the corresponding master is commented out.

Quotes and backslash

Similar to Python, Phil supports single quotes, double quotes, and triple quotes (three single or three double quotes). Unlike Python, quotes can often be omitted, and single quotes and double quotes have different semantics, similar to that of Unix shells: '$' variables are expanded if embedded in double quotes, but not if embedded in single quotes. See the variable substitution section above for examples.

The backslash can be used in the usual way (Python, Unix shells) to "escape" line breaks, quotes, and a second backslash.

For convenience, a line starting with quotes is automatically treated as a continuation of a definition on the previous line(s). The trailing backslash on the previous line may be omitted.

The exact rules for quoting and backslash escapes are intricate. A significant effort was made to mimic the familiar behavior of Python and Unix shells where possible, but nested constructs of quotes and backslashes are still prone to cause surprises. In unusual situations, probably the fastest method to obtain the desired result is trial and error (as opposed to studying the intricate rules).

scope.show(attributes_level=3)

In this document, the `scope.show()` method is used extensively in the examples. With the defaults for the method parameters, it only shows the Phil scope or definition names and and associated values. It is also possible to include some or all Phil scope or definition attributes in the `.show()` output, as directed by the `attributes_level` parameter:

```
attributes_level=0: shows only names and values
                    1: also shows the .help attribute
                    2: shows all attributes which are not None
                    3: shows all attributes
```

`scope.show(attributes_level=2)` can be used to pretty-print master files without any loss of information. `attributes_level=3` is useful to obtain a full listing of all available attributes, but all information is preserved with the usually much less verbose `attributes_level=2`. This is illustrated by the following example:

```
master_phil = parse("""
  minimization {
    input
      .help = "File names and data labels."
      .multiple = True
    {
      file_name = None
        .type = path
      label = None
        .help = "A unique substring of the data label is sufficient."
        .type = str
    }
  }
  """)

for attributes_level in range(4):
  master_phil.show(attributes_level=attributes_level)
```

Output with `attributes_level=0` (the default):

```
minimization {
  input {
    file_name = None
    label = None
  }
}
```

Output with `attributes_level=1`:

```
minimization {
  input
    .help = "File names and data labels."
  {
    file_name = None
    label = None
      .help = "A unique substring of the data label is sufficient."
  }
}
```

Output with `attributes_level=2`:

```
minimization {
  input
    .help = "File names and data labels."
    .multiple = True
  {
    file_name = None
      .type = path
    label = None
      .help = "A unique substring of the data label is sufficient."
      .type = str
  }
}
```

Output with `attributes_level=3`:

```
minimization
  .style = None
  .help = None
  .extract = None
  .short_caption = None
  .optional = None
  .call = None
  .multiple = None
  .sequential_format = None
  .disable_add = None
  .disable_delete = None
  .expert_level = None
{
  input
    .style = None
    .help = "File names and data labels."
    .caption = None
    .short_caption = None
    .optional = None
    .call = None
    .multiple = True
    .sequential_format = None
    .disable_add = None
    .disable_delete = None
    .expert_level = None
  {
    file_name = None
      .style = None
      .caption = None
      .short_caption = None
      .optional = None
      .type = path
      .multiple = None
      .input_size = None
      .expert_level = None
    label = None
      .help = "A unique substring of the data label is sufficient."
      .caption = None
      .short_caption = None
      .optional = None
      .type = str
      .multiple = None
      .input_size = None
      .expert_level = None
  }
}
```

Home
Download
How to install
Getting started
Documentation
Tutorials
Table of contents
HKLViewer
Melk 2019
Rigid Body
cctbx tour
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

Tutorials

Beginner

- **MX model/data tutorial** (Computing Forum ECM 2019 in Melk)
  This tutorial explains how to read a model file and do some basic operations on it (truncate to polyalanine). The next part introduces miller arrays. The last part explains how to compute R-factors.

- **Rigid Body Refinement**
  This code example implements the core algorithms used in rigid body refinement

- **cctbx.xfel tutorials wiki**
  These tutorials provide instructions for indexing and integrating still diffraction images recorded at the LCLS. The tutorials also cover tools to efficiently leverage the LCLS computing cluster to process the thousands to millions of diffraction images that can be recorded in a short time. Finally, processing approaches for data collection at non-LCLS sources are also provided.

Advanced

- **Tour of the cctbx**
  Tour of some cctbx features. This tutorial requires a solid background in crystallography.

Home
Download
How to install
Getting started
Documentation
Tutorials
Table of contents
HKLViewer
Melk 2019
Rigid Body
cctbx tour
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

CCTBX tutorial Crystallographic computing school, Melk, 08/2019

Script reading a model file

This basic script illustrates how to read in a model file.

Files

- tutorial_1_v0.py
- 1aba_pieces.pdb

Getting started:

Open the file `tutorial_1_v0.py` in your source code editor.
Most of this script is so called boilerplate code, i.e. code that in some form or shape is found in most Python scripts.

```
from __future__ import division, print_function
import sys
import mmtbx.model
import iotbx.pdb

def run(args):
  if (len(args) != 1):
    raise RuntimeError("Please specify one pdb file name.")
  model_filename = args[0]
  pdb_inp = iotbx.pdb.input(file_name=model_filename)
  model = mmtbx.model.manager(model_input = pdb_inp)

if (__name__ == "__main__"):
  run(sys.argv[1:])
```

At the beginning of the script are import statements, which import the modules needed for a task.
The bottom lines represent best practice. They enable the script to be imported and used from other Python scripts.
The first two lines of the run() function are a minimalistic - but often sufficient - way to give users a hint how to use the script. It works both for someone reading the source code of the script, and a user running the script without arguments.

Task Execute the script by typing `python tutorial_1_v0.py`.

The output is:

```
Traceback (most recent call last):
  File "tutorial_1_v0.py", line 15, in
    run(sys.argv[1:])
  File "tutorial_1_v0.py", line 8, in run
    raise RuntimeError("Please specify one pdb file name.")
RuntimeError: Please specify one pdb file name.
```

In addition to showing the error message, Python shows exactly where the error originates. This is often extremely helpful.

The important part of the script is in these two lines:

```
pdb_inp = iotbx.pdb.input(file_name = model_filename)
model = mmtbx.model.manager(model_input = pdb_inp)
```

The first line creates the class for reading a model from a file or string. It is the main input method for both PDB and mmCIF files; it will automatically determine the actual format and return the appropriate data type.
The second line creates the model class, which serves as container for model information to be passed between library functions. For example, it selects, copies and updates model information, stores libraries, and ensures consistency between lower level objects.

Use print and help()

While analyzing a script, insert `print()` statements and run the script to find out more about the objects. 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.
For example, let's try to use the `show` method of the `model.composition()` object, which produces output that is useful to give the user a quick overview of what is in the PDB file. Add the following lines to your script.

```
composition_obj = model.composition()
print(help(composition_obj))
```

The `help()` method calls the built-in Python help system. It lets you read the docstring and get an idea of what attributes and methods a class might have. (Press `q` to exit the help mode)

Task Run the script by using the Python `help` method for the `composition` class.

Show output

We can see from the `help` output that the `composition` class has two methods ('show' and 'result') and that the 'show' method requires setting the parameter 'log'. Set it to `sys.stdout`.

```
model.composition().show(log=sys.stdout)
```

Task Use the method `composition.show()` and run the script with the file `1aba_pieces.pdb`.

The output is:

```
Number of:
  all atoms      : 48
  H or D atoms   : 0
  chains         : 2
  a.a. residues  : 5
  nucleotides    : 0
  water          : 4
  other (ligands): 0
Ligands: None
```

Home
Download
How to install
Getting started
Documentation
Tutorials
Table of contents
HKLViewer
Melk 2019
Rigid Body
cctbx tour
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

Rigid Body Refinement

Overview
Use print and help(!)
The rigid_body constructor
Conversion from Euler angles to a rotation matrix
Computation of moved sites
Computation of gradients
Unit tests
References

Overview

This code example implements the core algorithms used in rigid body refinement:

- Computation of moved sites given three Euler angles and a three-dimensional linear translation vector (six degrees of freedom total).
- Summation of gradients w.r.t. Cartesian coordinates, to obtain gradients w.r.t. the Euler angles and the linear translation vector.

The script consists of two parts of roughly equal size. The first part (just 80 lines) implements a rigid_body class with this interface:

```
class rigid_body(object):
  def __init__(self, sites):
  def rotation_matrix(self):
  def center_of_mass_moved(self):
  def sites_moved(self):
  def ea_gradients(self, energy_cart_function):
```

Here `ea` in `ea_gradients()` is for "Euler angles". The rigid body parameters are stored as `rigid_body.ea` and `rigid_body.lt`.

The second part of the script (less than 70 lines) is a unit test, to automatically check the analytical gradient calculations in the first part via finite differences.

To keep the example concise, the target function is a simple least-squares function restraining the sites to the original positions. In a real application, e.g. refinement against x-ray data with other cctbx facilities, the computation of the gradients w.r.t. the Cartesian coordinates of the atomic sites is far more complex, in particular if space group symmetry is involved. However, the computation of the gradients w.r.t. the Euler angles and the linear translation of the rigid body as implemented in this script is completely general. I.e., any refinement program that already includes computation of the gradients w.r.t. the Cartesian coordinates can be easily extended to also support rigid body refinement.

The unit_cell_refinement.py example shows how the target function results (functional and gradients) can be used in combination with the general purpose quasi-Newton LBFGS minimizer to iteratively update the Euler angle and linear translation parameters.

Use print and help(!)

Before you study the example script, download and install the cctbx. This will enable you to run the script. While analyzing the script, insert print statements and run the script to find out more about the objects. 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.

If you don't know what an object is: *thing* is a pretty good approximation.

The `rigid_body` constructor

The "constructor" (called "__init__" method in Python) of the `rigid_body` class is:

```
def __init__(self, sites):
  self.sites_orig = sites
  self.center_of_mass_orig = matrix.col(sites.mean())
  self.lt = matrix.col((0,0,0))
  self.ea = matrix.col((0,0,0))
```

The only argument is an array of `sites`, i.e. the current coordinates of the points ("atoms") in the rigid body. To see the list of coordinates, insert this print statement:

```
print list(sites)
```

This will show a list of Python tuples:

```
[(10.949, 12.815, 15.189),
 (10.484999999999999, 13.954000000000001, 15.917),
 (10.779, 15.262, 15.227)]
```

The `self.sites_orig = sites` assignment keeps a reference to this array for use in other methods (also known as "member functions" in some other languages). In view of repeated use in other methods, the center of mass is computed via `sites.mean()`. `sites` is expected to be a one-dimensional C++ array of type `scitbx.array_family.flex.vec3_double`. This is not formalized in the interface, but implied by the use of certain methods, in this case `.mean()`. This approach is known colloquially as "duck typing". Any other type with compatible methods could be used instead. As far as Python is concerned: if it walks like a duck and quacks like a duck, it is a duck.

The `scitbx.matrix` module implements many commonly used matrix algorithms, e.g. the matrix product, dot product, cross product, matrix inversion etc. The `rigid_body` constructor uses the `scitbx.matrix.col` (column) type to store the center of mass, the Euler angles, and the linear translation vector. It is highly recommended to spend a few minutes reading the `matrix/__init__.py` script. Since it is quite short and mostly implemented in pure Python it is largely self-explanatory.

Conversion from Euler angles to a rotation matrix

The `rigid_body.rotation_matrix()` method passes the Euler angles to the `euler_xyz_matrix()` function further up in the script:

```
def rotation_matrix(self):
  return euler_xyz_matrix(ea=self.ea)
```

The relevant code is:

```
angle_scale = math.pi / 2

def euler_xyz_matrix(ea):
  """
  Mathematica code:
    rx = {{1, 0, 0}, {0, cx, -sx}, {0, sx, cx}}
    ry = {{cy, 0, sy}, {0, 1, 0}, {-sy, 0, cy}}
    rz = {{cz, -sz, 0}, {sz, cz, 0}, {0, 0, 1}}
    rx.ry.rz
  """
  sin, cos = math.sin, math.cos
  cx = cos(ea[0] * angle_scale)
  sx = sin(ea[0] * angle_scale)
  cy = cos(ea[1] * angle_scale)
  sy = sin(ea[1] * angle_scale)
  cz = cos(ea[2] * angle_scale)
  sz = sin(ea[2] * angle_scale)
  return (
```

In general, there are 12 different conventions for Euler angles. In addition to these, the rotation can be defined in two ways: a counterclockwise rotation of the basis system (equivalent to a clockwise rotation of vectors w.r.t. that basis system) or a counterclockwise rotation of vectors (equivalent to a clockwise rotation of the basis system). The function above implements the "xyz" convention with counterclockwise rotation of vectors. The "xyz" convention is advantageous for rigid-body refinement since the Gimbal lock problem occurs only if the rotation around y is plus or minus 90 degrees, a situation that is very unlikely to be reached since the convergence radius of rigid body refinement is typically much smaller.

The Mathematica code for generating the rotation matrix is included as a Python "docstring". From this it is easy to see exactly how the rotations are defined, and how to reproduce the matrix.

The `angle_scale` was introduced as a way to balance scale of the angular parameters compared to the linear translation parameters. For the LBFGS minimizer to work optimally, it is helpful to balance the relative scale of all parameters. The exact `angle_scale` value is not too critical, as long as it is in the right order of magnitude. The value used in the script is the result of a few empirical tests.

For completeness it is noted that we could have used `scitbx.math.euler_angles.xyz_matrix` instead, which is implemented in C++. However, for clarity, and to allow experimenting with the `angle_scale` value, the function is re-implemented in the example script.

Computation of moved sites

These two methods are concerned with the computation of the moved sites, given `self.rotation_matrix()` as introduced above and the linear translation `self.lt`:

```
def center_of_mass_moved(self):
  return self.center_of_mass_orig + self.lt

def sites_moved(self):
  return \
    self.rotation_matrix() \
    * (self.sites_orig - self.center_of_mass_orig) \
    + self.center_of_mass_moved()
```

It will be helpful to insert print statements to inspect the objects involved. `self.center_of_mass_orig` + `self.lt` uses the `scitbx.matrix` implementation for element-wise addition of the three vector components. `self.sites_orig - self.center_of_mass_orig` uses the overloaded binary minus operator of the `flex.vec3_double` C++ array type. The multiplication with `self.rotation_matrix()` and the final addition of `self.center_of_mass_moved()` are again C++ operations. Therefore, the code will work efficiently even for a large number of sites. Since all operations involving large arrays are performed in C++, we can use Python's concise syntax without incurring a significant performance penalty, compared to a more arcane pure C++ implementation.

Computation of gradients

The gradient calculation code is the most complex part of the script (as usual):

```
def ea_gradients(self, energy_cart_function):
  sites_moved = self.sites_moved()
  energy_cart = energy_cart_function(
    nodes=sites_moved, homes=self.sites_orig)
  ne_f = newton_euler_f(
    sites=sites_moved,
    pivot=self.center_of_mass_moved(),
    d_potential_energy_d_site=energy_cart.gradients())
  f = list(ne_f)
  C = matrix.sqr(euler_xyz_ea_d_ea_omega_f(ied_frame_matrix(
    ea=self.ea).transpose())
  return list(angle_scale * C * matrix.col(f[:3])) + f[-3:]
```

The first action in this method is to compute `sites_moved` as explained above. The next step is to call an external function object `energy_cart_function`. In the "duck typing" spirit (see above), the requirements for `energy_cart_function` are implied by the implementation. The implementation of the trivial least-squared restraints to the original sites serves as an example:

```
class energy_cart(object):

  def __init__(self, nodes, homes):
    assert nodes.size() == homes.size()
    self.nodes = nodes
    self.homes = homes

  def functional(self):
    return flex.sum(self.nodes-self.homes).dot())

  def gradients(self):
    return 2*(self.nodes-self.homes)
```

Again, it will be helpful to insert print statement to inspect the types involved. `nodes` and `homes` are `flex.vec3_double` arrays. Therefore the calculations in `.functional()` and `.gradients()` are fast C++ array operations.

Any function object with the same interface could be used instead. I.e. the `rigid_body` code could be used unmodified in refinement against x-ray data. All application-specific calculations will be concentrated in the (much more complex) `energy_cart` equivalent.

The transformation of the gradients w.r.t. Cartesian coordinates to gradients w.r.t. the six rigid body parameters consists of two parts and follows the procedure described by Schwieters & Clore (2001). The first part is a subset of the "Newton-Euler" equations for the motion of a rigid body. The Cartesian gradients are summed to obtain the total translational gradients (linear force changing the linear velocity in the context of dynamics) and the total angular gradients (angular force changing the angular velocity):

```
def newton_euler_f(sites, pivot, d_potential_energy_d_site):
  "Schwieters & Clore (2001) equations 24"
  sum_grads = matrix.col((0,0,0))
  sum_moments = matrix.col((0,0,0))
  for site_grad in zip(sites, d_potential_energy_d_site):
    grad = -matrix.col(grad)
    sum_grads += grad
    sum_moments += (matrix.col(site) - pivot).cross(grad)
  return matrix.col(sum_moments.elems + sum_grads.elems)
```

In the case of a freely moving rigid body, the `pivot` is the center of mass. (In the context of dynamics, any other pivot implies external forces.)

The implementation of the summations is rather inefficient since the loop over the (potentially large number of) sites is in Python. While sum_grads for the translational gradients could be obtained as with the C++ array operation `flex.sum(d_potential_energy_d_site)`, there is no existing C++ array version of the more complex sum for calculating the angular gradients. For clarity, to keep the function uniform, both results are computed in Python. For actual applications, the entire function should be re-implemented in C++.

Given the total angular gradients from the solution of the Newton-Euler equations, it is just a matter of transforming these values into the frame of reference for the Euler angles. This step is described in Goldstein (2002), towards the end of section 4.9. The formula for the Euler angle xyz convention is given in the appendix of Goldstein (2002):

```
def euler_xyz_ea_d_ea_omega_f(ied_frame_matrix(ea):
  "Goldstein (A14.xyz) with sinus sign reversed"
  sin, cos = math.sin, math.cos
  cx = cos(ea[0] * angle_scale)
  sx = sin(ea[0] * angle_scale)
  cy = cos(ea[1] * angle_scale)
  sy = sin(ea[1] * angle_scale)
  return (
    1,  0,   sy,
    0, cx, -cy*sx,
    0, sx,  cx*cy)
```

Since Goldstein (2002) defines the Euler angles as a rotation of the basis system, the sign of the sinus terms is reversed in the function above.

The transformation of the angular velocity to time derivatives of the Euler angles is, evidently, a simple linear transformation. Luckily, from this it follows that the transformation of the gradients is simply given by the transpose of the same matrix (Murshudov et al., 1999; very end of appendix B). This leads to the last three lines of the `ea_gradients()` implementation. The preceding line simply reverses the sign of the Newton-Euler forces; this could be avoided by changing newton_euler_f(), but we prefer to keep the implementation compatible with the Schwieters & Clore (2001) equations to avoid confusion.

Unit tests

The nested functions inside the `exercise()` function are unit tests to ensure the correctness of the gradient code discussed above.

For the idea behind the finite difference tests, please refer back to the Unit Cell Refinement tutorial. With this background, the test code should be easy to follow. The only slightly unobvious fragments are in `incr_position()` function, e.g.:

```
v = list(rb.ea)
v[i] += delta
rb.ea = matrix.col(v)
```

The detour through a Python list is necessary because the scitbx.matrix objects (in this case rb.ea) are understood to be immutable. This is not actually fully enforced to keep the implementations simple, but is true for many objects in the cctbx. The mentation is to avoid surprises due to "side effects" caused by changing objects in place. Of course, the `rb.ea = matrix.col(v)` assignment is doing just that, which tells us that it is sometimes impractical to be pure about the "no in-place operations" idea, taking runtime and memory consumption considerations into account. However, our practical experience strongly suggests it is important to apply the "no in-place operations" idea in most cases, and that it is a good excuse for the three lines of gymnastics above.

Typical (good) unit tests should produce no output. However, in this special case we chose to simply print the analytical and finite-difference gradients for visual inspection. In most other cctbx unit tests, the following would be used instead (to be inserted in `show_gradients()`):

```
from libtbx.test_utils import approx_equal
assert approx_equal(an, fd)
```

This replaces the visual inspection with an automatic test. If included an routine testing script, it ensures that the results do not change as a side-effect of development work on the underlying libraries. This form of automatic testing is the only practical approach to quality assurance in an ever-growing and evolving library, in particular if there are many developers.

References

- Goldstein, H. (2002). Classical Mechanics, 3rd edition.
- Murshudov, G.N., Vagin, A.A., Lebedev, A., Wilson, K.S., Dodson, E.J. (1999). Acta Cryst. D55, 247-255.
- Schwieters, C.D., Clore, G.M. (2001). J. Mag. Res., 152(2), 288–302.

Home
Download
How to install
Getting started
Documentation
Tutorials
Table of contents
HKL2Image
Mab 2D
Rigid Body
Scitbx Set
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

1. The basics
2. At the very bottom
3. scitbx.array_family.flex
4. cctbx.array_family.flex
5. A balancing act
6. Hybrid systems
7. Building bridges
8. Thinking hybrid

Tour of the cctbx

1. The basics in the box

If you go to the beach you will find massive amounts of a material known to crystallographers as quartz, which in this case is just a fancy word for sand. As an example here is the cctbx way of playing in the sandbox:

```
from libtbx import easy
from libtbx import sys\*sys\*
from cctbx.array_family import flex

quartz_structure = sys.stdin\*sys.stdin\*
  space_group_info=sgtbx\*crystal.space_group_info(symbol="P6222"),
  scatterers=flex.scatterer([sgtbx\*xray.scatterer(
    site=(x,0.0,0.0),
    u=0.0,
    occupancy=1.0,
    scattering_type="Si",
    flags=sgtbx\*xray.scattering_type_flags(
      fixed_site=True,
      fixed_u=True,
      fixed_occupancy=True,
      fixed_scattering_type=True,
      use_u_iso=True,
      use_u_aniso=False,
      grid_site=0,
      grid_u=0,
      grid_occupancy=0,
      grid_scattering_type=0,
      site_sym="x,-x+y,-z+1/3",
      u_iso=0.0,
      u_aniso=(0,0,0,0,0,0),
      occ=1.0))
  for x in [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]\*0.5\*0.5\*0.5
```

Home
Download
How to install
Getting started
Documentation
Tutorials
Example scripts
Table of contents
Random model+density
Compare SS
R-factors
Ideal SS
LBFGS (no curvature)
LBFGS (with curvature)
Newsletter articles
About the cctbx
cctbx on GitHub

Example scripts

The scripts below illustrate how to use the cctbx.

Beginner

- **Generate a random model and compute its density map**
  This example script shows how to generate a small centrosymmetric unit cell in P-1 with 15 atoms, and how to calculate phases and amplitudes to reconstruct the electron density.

- **Generate ideal helices and strands**
  This script shows how to generate idealized helices and strands from a sequence.

Advanced

- **Compare secondary structure annotations numerically**
  Several methods exist that determine the secondary structure of macromolecules. This script shows how to automatically determine secondary structure using several methods with cctbx tools and how to numerically compare the result.

Home
Download
How to install
Getting started
Documentation
Tutorials
Example scripts
Table of contents
Random model+density
Compare SS
R-factors
Ideal SS
LBFGS (no curvature)
LBFGS (with curvature)
Newsletter articles
About the cctbx
cctbx on GitHub

Create a random model and calculate its density

Required imports
Create a random structure and save as PDB
Create a map from a PDB file

This example script shows how to generate a small centrosymmetric unit cell in P-1 with 15 atoms, and how to calculate phases and amplitudes to reconstruct the electron density.

Required imports

This script requires the following imports:

```
from __future__ import absolute_import, division, print_function
import iotbx.pdb
import iotbx.mrcfile
import mmtbx.model
import mmtbx.real_space
from scitbx.array_family import flex
from cctbx.development import random_structure
from cctbx import sgtbx
from cctbx import maptbx
```

Create a random structure and save as PDB

Use the `random_structure()` method from `cctbx.development` to generate a random structure that has:

- spcae group P-1
- unit cell dimensions a=10, b=20, c=30, alpha=70, beta=80, gamma=120
- 15 carbon atoms

```
# Create random structure
xrs = random_structure.xray_structure(
  space_group_info = sgtbx.space_group_info("P-1"),
  elements         = ["C"]*15,
  unit_cell        = (10, 20, 30, 50, 60, 80))
```

Create a `model` object from this structure:

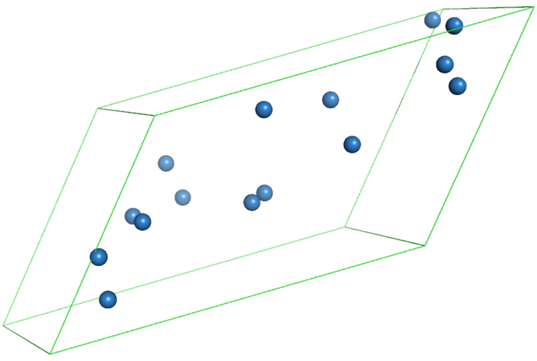
```
# Create model object
model = mmtbx.model.manager.from_sites_cart(
  sites_cart       = xrs.sites_cart(),
  crystal_symmetry = xrs.crystal_symmetry(),
  resname          = 'DUM')
```

`from_sites_cart()` is a convenience function to create a model object from a list of cartesian coordinates. The default is to use atom name CA, scatterer C, occ 1, b_iso 30, resname GLY, residue numbers starting with 1. Here, the residue name is set to 'DUM', which stands for dummy residue. Note that even if we specify different elements in `random_structure()`, we will get CA atoms, unless the `from_sites_cart()` method is supplied with a list of atom names and scatterer names.

Save the model as PDB file:

```
# Write it into PDB file
with open("model.pdb","w") as fo:
  fo.write(model.model_as_pdb())
```

Here is an example for a model that was created using this procedure (your model will look different, as the procedue will place the atoms at random places):



Create a map from a PDB file

Create a model object from a file.

```
  # Read the model file
pdb_inp = iotbx.pdb.input(file_name = "model.pdb")
model = mmtbx.model.manager(model_input = pdb_inp)
xrs = model.get_xray_structure()
```

Calculate structure factors from the model using a high resolution limit of 2 Angstrom. This will give a Fourier map:

```
# Calculate structure factors at given resolution.
f_calc = xrs.structure_factors(d_min = 2.0).f_calc()
```

For book-keeping: save the Fourier map as MTZ file:

```
# Write them down as MTZ file
mtz_dataset = f_calc.as_mtz_dataset(column_root_label="F-calc")
mtz_object = mtz_dataset.mtz_object()
mtz_object.write(file_name = "f_calc.mtz")
```

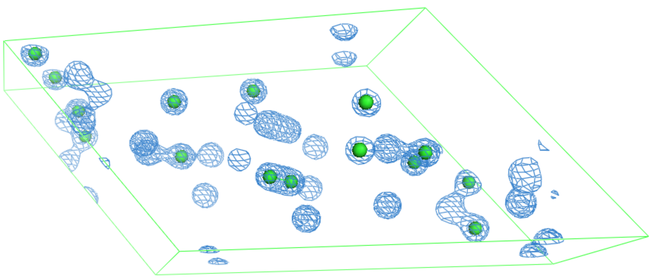
Perform a fast-Fourier-transform (FFT) to convert the structure factors into a real map:

```
# Convert Fcalc into real map (just do FFT)
fft_map = f_calc.fft_map(resolution_factor=1./4)
fft_map.apply_sigma_scaling()
map_data = fft_map.real_map_unpadded()
```

Write the map data into an MRC file:

```
# Write real Fourier map into MRC file
iotbx.mrcfile.write_ccp4_map(
  file_name   = "fourier_map.mrc",
  unit_cell   = f_calc.unit_cell(),
  space_group = f_calc.crystal_symmetry().space_group(),
  map_data    = map_data.as_double(),
  labels      = flex.std_string(["Some text"]))
```

The figure below shows the density overlayed to the model. Note that there are density peaks that are (seemingly) without model. This is because the density was calculated for the unit cell, which is P-1 (centrosymmetric). The spheres only represent the atoms in the asymmetric unit.



Another option is to calculate the exact map. This requires first the computation of the grid (crystal_gridding).

```
# Calculate exact map and write it down
crystal_gridding = maptbx.crystal_gridding(
  unit_cell        = xrs.unit_cell(),
  space_group_info = xrs.space_group_info(),
  symmetry_flags   = maptbx.use_space_group_symmetry,
  step             = 0.3)
m = mmtbx.real_space.sampled_model_density(
  xray_structure = xrs,
  n_real         = crystal_gridding.n_real())
map_data = m.data()
```

Save the exact map in the file exact_map.mrc

```
iotbx.mrcfile.write_ccp4_map(
  file_name   = "exact_map.mrc",
  unit_cell   = f_calc.unit_cell(),
  space_group = f_calc.crystal_symmetry().space_group(),
  map_data    = map_data.as_double(),
  labels      = flex.std_string(["Some text"]))
```

Home
Download
How to install
Getting started
Documentation
Tutorials
Example scripts
Table of contents
Random model+density
Compare SS
R-factors
Ideal SS
LBFGS (no curvature)
LBFGS (with curvature)
Newsletter articles
About the cctbx
cctbx on GitHub

Compare secondary structure annotations numerically

Code
Overview
Imports
run() function
Helper function get_ss()
Helper function match_score()
Example output

There are several methods to determine the secondary structure of macromolecules. This script shows how to automatically determine secondary structure using several methods with cctbx tools and how to numerically compare the result.

Author: Pavel V. Afonine

Code

The following script reads in a model file and numerically compares the secondary structure annotation:

```
from __future__ import absolute_import, division, print_function
import sys
import mmtbx.secondary_structure
from scitbx.array_family import flex
from libtbx.utils import null_out
from iotbx.data_manager import DataManager

def match_score(x,y):
  assert x.size() == y.size()
  match_cntr = 0
  for x_,y_ in zip(x,y):
    if(x_==y_): match_cntr+=1
  return match_cntr/x.size()

def get_ss(hierarchy,
           sec_str_from_pdb_file=None,
           method="ksdssp",
           use_recs=False):
  if(use_recs): params = None
  else:
    params = mmtbx.secondary_structure.manager.get_default_ss_params()
    params.secondary_structure.protein.search_method=method
    params = params.secondary_structure
  ssm = mmtbx.secondary_structure.manager(
    pdb_hierarchy           = hierarchy,
    sec_str_from_pdb_file = sec_str_from_pdb_file,
    params                  = params,
    log                     = null_out())
  alpha = ssm.helix_selection()
  beta  = ssm.beta_selection()
  assert alpha.size() == beta.size() == hierarchy.atoms().size()
  annotation_vector = flex.double(hierarchy.atoms().size(), 0)
  annotation_vector.set_selected(alpha, 1)
  annotation_vector.set_selected(beta, 2)
  return annotation_vector

def run(args):
  dm = DataManager()                      #   Initialize the DataManager and call it dm
  dm.set_overwrite(True)                  #   tell the DataManager to overwrite files with the same
  model_filename = args[0]                #   Name of model file
  model = dm.get_model(model_filename)  #   Deliver model object with model info
  pdb_hierarchy = model.get_hierarchy() #   Get hierarchy object
  sec_str_from_pdb_file = model.get_ss_annotation()
  # get secodary structure annotation vector from HELIX/SHEET records (file header)
  print('Running secondary structure annotation...')
  v1 = get_ss(
    hierarchy               = pdb_hierarchy,
    sec_str_from_pdb_file = sec_str_from_pdb_file)
  # get secodary structure annotation vector from method CA atoms
  v2 = get_ss(hierarchy = pdb_hierarchy, method = "from_ca")
  # secodary structure annotation vector from KSDSSP
  v3 = get_ss(hierarchy = pdb_hierarchy, method = "ksdssp")
  #
  print()
  print("CC REMARK vs from_ca:", flex.linear_correlation(x = v1, y = v2).coefficient())
  print("CC REMARK vs ksdssp:", flex.linear_correlation(x = v1, y = v3).coefficient())
  print("CC from_ca vs ksdssp:", flex.linear_correlation(x = v3, y = v2).coefficient())
  print()
  print("match REMARK vs from_ca:", match_score(x = v1, y = v2))
  print("match REMARK vs ksdssp:", match_score(x = v1, y = v3))
  print("match from_ca vs ksdssp:", match_score(x = v3, y = v2))

if __name__ == '__main__':
  run(args=sys.argv[1:])
```

Overview

The script consists of several parts:

Imports: At the beginning of the script are import statements. They are used to import the modules needed for a task.

Main function: The `run()` function is used to read the input file, to call the helper functions and to print the result to standard output.

Helper functions: `match_score()` and `get_ss()`: To avoid duplicating code, some part of the calculations are put into functions. The `get_ss()` function runs automatic secondary structure annotation, then returns a 1dimensional array representing the annotation. This vector is a linear array of length equal to the number of atoms in the model. It is initialized with zeros. Using the information from the SS annotation, the array elements are set to a non-zero number for helices and to a different number for sheets.

The bottom lines represent best practice. They enable the script to be imported and used from other Python scripts.

Imports

The script requires the following imports:

```
from __future__ import absolute_import, division, print_function
import sys
import mmtbx.secondary_structure
from scitbx.array_family import flex
from libtbx.utils import null_out
from iotbx.data_manager import DataManager
```

run() function

In the beginning of the `run()` function, we read in a PDB file using the DataManager and create a `model` object as well as a `hierarchy` object:

```
dm = DataManager()                      #   Initialize the DataManager and call it dm
dm.set_overwrite(True)                  #   tell the DataManager to overwrite files with
model_filename = args[0]                #   Name of model file
model = dm.get_model(model_filename)  #   Deliver model object with model info
pdb_hierarchy = model.get_hierarchy() #   Get hierarchy object
```

Model files from the PDB usually have records with secondary structure annotation. Let's get the records from `model`:

```
sec_str_from_pdb_file = model.get_ss_annotation()
```

Then we get the annotation vectors from the helper function `get_ss()`, using three different methods:

```
v1 = get_ss(
    hierarchy               = pdb_hierarchy,
    sec_str_from_pdb_file = sec_str_from_pdb_file)
v2 = get_ss(hierarchy = pdb_hierarchy, method = "from_ca")
v3 = get_ss(hierarchy = pdb_hierarchy, method = "ksdssp")
```

v1 represents the annotation vector based on the records in the model file. v2 is obtained from the annotation method "from_ca", which uses CA positions to find out if a residue belongs to secondary structure element. The annotation vector v3 relies on the the KSDSSP method.

The last lines print the results. With the annotation vectors, we can calculatie correlation coefficients using the method `flex.linear_correlation(x = v1, y = v2).coefficient()`. We can also calculate the percentage of similarity of the vectors, which is done in the `match_score()` function.

```
print("match REMARK vs from_ca:", match_score(x = v1, y = v2))
```

Helper function get_ss()

In the `get_ss()` function, we calculate the annotation vectors. To do this, we first obtain the default parameters for the secondary structure module:

```
params = mmtbx.secondary_structure.manager.get_default_ss_params()
```

The `params` object is a container for the parameters used in the `mmtbx.secondary_structure.manager`. As we want to get the annotation via different methods, we change the parameter `params.secondary_structure.protein.search_method`.

Then we get the secondary structure manager object:

```
ssm = mmtbx.secondary_structure.manager(
  pdb_hierarchy           = hierarchy,
  sec_str_from_pdb_file = sec_str_from_pdb_file,
  params                  = params,
  log                     = null_out())
```

We get the selections for helices and sheets with the `helix_selection()` and `beta_selection()` methods:

```
alpha = ssm.helix_selection()
beta  = ssm.beta_selection()
```

These selections are flex bool arrays. The length of the array corresponds to the number of atoms. We then make an annotation vector that has the same length. Its values are 1 if the atom is part of a helix, 2 if the atom is part of a sheet and 0 if the atom is not part of a secondary structure element.

```
annotation_vector = flex.double(hierarchy.atoms().size(), 0)
annotation_vector.set_selected(alpha, 1)
annotation_vector.set_selected(beta, 2)
```

We can then use this vector to compare the annotations numerically, as done with `flex.linear_correlation(x = v1, y = v2).coefficient()` and the `match_score()` function.

Helper function match_score()

The `match_score()` function calculates how many atoms have the same annotation and then divides the result by the number of atoms. This gives therefore a percentage of how many atoms of the model have the same secondary structure annotation.

```
assert x.size() == y.size()
match_cntr = 0
for x_,y_ in zip(x,y):
  if(x_==y_): match_cntr+=1
return match_cntr/x.size()
```

Note the assertion in the first line, which makes sure that the arrays x and y have the same length.

Example output

You can run the script via the command line:

```
python script.py my_model.pdb
```

For example, if you run the script with model 1us0, the output will be the following (you can conveniently fetch a PDB model with "phenix.fetch_pdb 1us0"):

```
CC REMARK vs from_ca: 0.803945216619
CC REMARK vs ksdssp: 0.869800857769
CC from_ca vs ksdssp: 0.727252951902

match REMARK vs from_ca: 0.91220440337
match REMARK vs ksdssp: 0.910009785268
match from_ca vs ksdssp: 0.864908942647
```

You can get some more information about the annotation if you remove the parameter `log = null_out()` in the generation of the SS manager:

```
ssm = mmtbx.secondary_structure.manager(
  pdb_hierarchy           = hierarchy,
  sec_str_from_pdb_file = sec_str_from_pdb_file,
  params                  = params)
```

(The default for log is the standard output.)

Then we'll get the additional output:

```
    Secondary structure from input PDB file:
      14 helices and 2 sheets defined
      46.5% alpha, 8.6% beta
      0 base pairs and 0 stacking pairs defined.
  running find_ss_from_ca liberal...
    Secondary structure from input PDB file:
      12 helices and 3 sheets defined
      47.5% alpha, 13.7% beta
      0 base pairs and 0 stacking pairs defined.
  running ksdssp...
    Secondary structure from input PDB file:
      14 helices and 3 sheets defined
      39.5% alpha, 7.6% beta
      0 base pairs and 0 stacking pairs defined.
```

Home
Download
How to install
Getting started
Documentation
Tutorials
Example scripts
Table of contents
Random model+density
Compare SS
R-factors
Ideal SS
LBFGS (no curvature)
LBFGS (with curvature)
Newsletter articles
About the cctbx
cctbx on GitHub

Calculate R-factors for a crystal structure

Required imports
Get the model object
Read the mtz file and get the miller array
Calculate R-factors

This example script shows how to calculate R-factors with a model and a mtz file.

Required imports

This script requires the following imports:

```
from __future__ import absolute_import, division, print_function
import iotbx.pdb
import mmtbx.model
import mmtbx.f_model
from iotbx import reflection_file_reader
```

To calculate R-factors we need a model (coordinates, b-factors, atom-type, occupancy, etc.) to obtain calculated structure factors and data (structure factor amplitudes or intensities, R-free-flags for Rfree). The following sections explain how to obtain the model and data objects from the input files.

If you don't have examples files at hand, you can use files that come with the phenix installation:

```
import os
import libtbx.load_env
pdb_file = libtbx.env.find_in_repositories(
  relative_path="phenix_regression/pdb/1yjp_h.pdb",
  test=os.path.isfile)
mtz_file = libtbx.env.find_in_repositories(
  relative_path="phenix_regression/reflection_files/1yjp.mtz",
  test=os.path.isfile)
assert (not None in [pdb_file, mtz_file])
```

Get the model object

First, let's create a model object from the input file `pdb_file`.

```
# Read in the model file and create model object
pdb_inp = iotbx.pdb.input(file_name = pdb_file)
model = mmtbx.model.manager(model_input = pdb_inp)
```

Read the mtz file and get the miller array

We need data to calculate R-factors. First we'll read the mtz file, then we get an appropriate Miller array. In this example, we know that the miller arrays have the names `FOBS,SIGFOBS` and `R-free-flags` in the mtz file.

```
# Get miller arrays for data and Rfree flags
miller_arrays = reflection_file_reader.any_reflection_file(file_name =
  mtz_file).as_miller_arrays()
for ma in miller_arrays:
  print(ma.info().label_string())
  if(ma.info().label_string()=="FOBS_X,SIGFOBS_X"):
    f_obs = ma
  if(ma.info().label_string()=="R-free-flags"):
    r_free_flags = ma
```

Now we have to remove reflections that don't have an equivalent in the other array, in other words, both arrays need to have the same number of reflections. We also bring the r_free_flags array into a boolean format suitable for further calculations.

```
# Obtain a common set of reflections
f_obs, r_free_flags = f_obs.common_sets(r_free_flags)
r_free_flags = r_free_flags.array(data = r_free_flags.data()==0)
```

To doublecheck, let's print the number of reflections that are "free" and "work", respectively:

```
print(r_free_flags.data().count(True), r_free_flags.data().count(False))
```

Calculate R-factors

The following lines of code yield the `f_model` object that combines the data and model:

```
fmodel = mmtbx.f_model.manager(
  f_obs          = f_obs,
  r_free_flags   = r_free_flags,
  xray_structure = model.get_xray_structure())
```

You can see that we did not pass the model object itself to `f_model`. The reason is that `model` contains way more infomation than necessary to calculate structure factors, for example it may carry restraint information. To calculate R-factors, this is not needed. Instead, the minimal structural information is contained in the `xray_structure` object.

We need one more step before calculating R-factors: The calculated and observed structure factor amplitudes are usually not on a common scale and low resolution structure factors need to be adjusted. To address this, the `update_all_scales()` method will calculate scale factors and add a bulk solvent model.

```
fmodel.update_all_scales()
```

Finally, we print the results:

```
print("r_work=%6.4f r_free=%6.4f"%(fmodel.r_work(), fmodel.r_free()))
fmodel.show(show_header=False, show_approx=False)
```

Home
Download
How to install
Getting started
Documentation
Tutorials
Example scripts
Table of contents
Random model+density
Compare SS
R-factors
Ideal SS
LBFGS (no curvature)
LBFGS (with curvature)
Newsletter articles
About the cctbx
cctbx on GitHub

Generate ideal helices and strands

Introduction
Code
Example

This script shows how to generate idealized helices and strands from a sequence.

Introduction

For certain purposes, such as writing tests for cctbx, it is useful to generate a model of idealized secondary structure, such as helices and beta strands.

Code

The script needs the following imports:

```
from __future__ import absolute_import, division, print_function
from mmtbx.secondary_structure.build import ss_idealization as ssb
```

The first line makes sure that the script runs with both Python2 and Python3.

The second import from the `mmtbx/secondary_structure` module makes available the function used to generate ideal secondary structure.

Then we create an ideal helix from a sequence, here we use the example sequence "ILMKARNDWYV":

```
ph_helix = ssb.secondary_structure_from_sequence(ssb.alpha_helix_str,"ILMKARNDWYV")
```

The `secondary_structure_from_sequence` function creates a new `pdb_hierarchy` object. The first argument is a string in PDB format (pdb_str) that has the geometry of the secondary structure. The following strings are available in the code; in parenthesis is the name of the variable in the code:

- Alpha helix (`alpha_helix_str`)
- 3-10 helix (`a310_helix_str`)
- Pi-helix (`pi_helix_str`)
- Beta strand (`beta_pdb_str`)

The second argument is the sequence, in form of a string of one-letter codes. It can be an explicit sequence, such as "ILMKARNDWYV", or in form of `"A"*20` for 20 alanine residues.

Once we have the new `pdb_hierarchy` for the idealized helix, we can save it as PDB file:

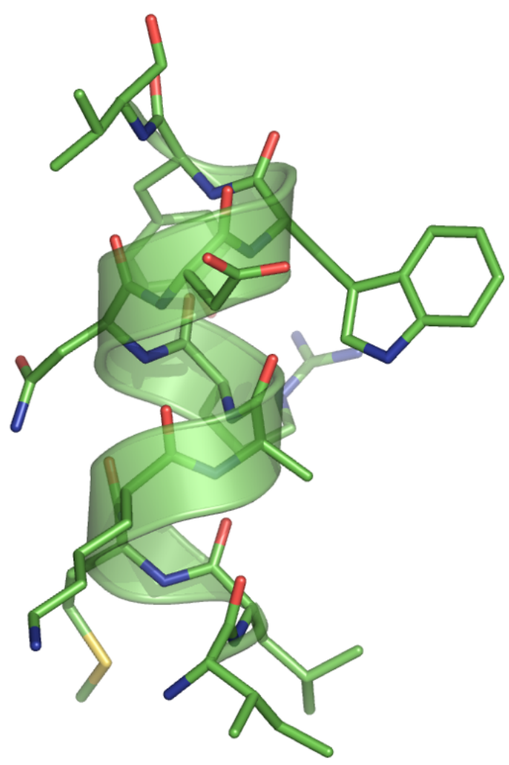
```
ph_helix.write_pdb_file(file_name="m-helix.pdb")
```

Similarly, we can create an ideal beta strand:

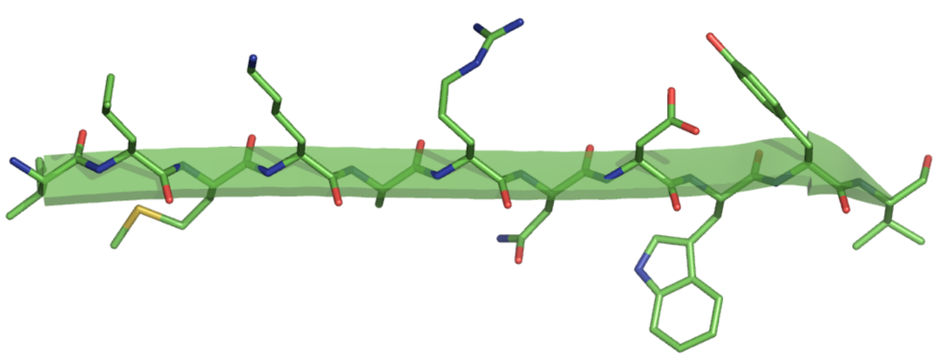
```
ph_strand = ssb.secondary_structure_from_sequence(ssb.beta_pdb_str,"ILMKARNDWYV")
ph_strand.write_pdb_file(file_name="m-strand.pdb")
```

Example

For the sequence used in our example ("ILMKARNDWYV"), we get the ideal alpha helix below.



For the same sequence, we get the follwing strand. Note the peptide N-H and C=O groups pointing to the top and the bottom.



Home
Download
How to install
Getting started
Documentation
Tutorials
Example scripts
Table of contents
Random model-density
Compare SS
R-factors
Ideal SS
LBFGS (no curvature)
LBFGS (with curvature)
Newsletter articles
About the ccztx
ccztx on GitHub

LBFGS minimizer without curvatures

This script shows how to use the LBFGS minimizer without curvatures (gradients only), with and without bounds.

Authors: Pavel V. Afonine, Dorothee Liebschner

The LBFGS algorithm
The Rosenbrock function
Code
Overview
The class rosenbrock()
The run() function
Minimization without bounds
Minimization with bounds
Output

The LBFGS algorithm

The limited-memory Broyden–Fletcher–Goldfarb–Shanno (LBFGS) algorithm is an optimization method to find the minimum of a function f(x), with x being a vector in $\mathbb{R}^n$ and f(x) a differentiable function.

The basic idea of minimization algorithms is to find iteratively the minimum of a function $f(x)$. For example, the gradient descent method uses, at a random initial point $f(x_0)$ and iteratively takes steps that are proportional to the negative gradient of $f(x)$ at the current point. As this is quite inefficient (a lot of iterations are required to find the minimum), the second order behavior (second derivatives = curvatures) can be used in addition. This is the Newton method, where the iterative scheme is given by:

$$x_{k+1} = x_k - [H(x_k)]^{-1} \nabla f(x_k) \tag{1}$$

The Newton method is quite efficient, but it is computational expensive to calculate the inverse of the Hessian matrix. This is where the BFGS algorithm kicks in: Instead of explicitly calculating the second derivatives, it uses approximations of the Hessian matrix to converge faster. At each iteration k+1, an approximate inverse second derivative is calculated using quantities from previous steps (the first derivative from the previous two iterations k and k-1):

$$x_{k+1} = x_k - f'(x_k) \frac{x_k - x_{k-1}}{f'(x_k) - f'(x_k - 1)} \tag{2}$$

The L-BFGS algorithm is an approximation of the BFGS procedure. It uses a limited amount of computer memory.
L-BFGS does not use boundaries as it was developed to solve unconstrained optimization problems. However, there is an extension of L-BFGS, called L-BFGS-B (Zhu et al., 1997), that can handle simple bounds. Both options are available in the ccztx.

Generally, the gradient based optimizers have a larger convergence radius but have a harder time to approach the minimum.
Second derivative based minimizers have much smaller convergence radius, but if you are close enough to the minimum, they can get there much more efficiently.
Therefore it is best to use both: when you are far from minimum, use gradient based minimizers, and once you are close enough switch to use 2nd derivatives. However, this option is not available in the ccztx.

Further reading:

- Introduction to BFGS.
- David M. Himmelblau, Applied Nonlinear Programming, 1972 (out of print, but can be found second-hand)

The Rosenbrock function

The Rosenbrock function is a test function for evaluate the performance of optimization algorithms, such as convergence rate, precision and robustness.
Simple functions like this are useful to debug and test new algorithms as they are fast to implement and to execute. If the algorithm does not perform well on this standard problem, it indicates that it most likely won't work well on real life problems.
It is defined by:

$$f(x, y) = (a - x)^2 + b(y - x^2)^2 \tag{3}$$

The partial derivatives are:

$$\frac{\partial f(x, y)}{\partial x} = -2a + 2x - 4bxy + 4bx^3 \tag{4}$$

$$\frac{\partial f(x, y)}{\partial y} = 2by - 2bx^2 \tag{5}$$

To find the global minimum, we set the partial derivatives to 0. Equation (5) gives the condition $y = x^2$. Inserting this into equation (4) gives the coordinates of the global minimum: $x = a$, $y = a^2$.

Code

The code below will compute the minimum of the Rosenbrock function with parameters a = 20 and b = 10: $f(x, y) = (20 - x)^2 + 10(y - x^2)^2$.

```
from __future__ import division, print_function
from libtbx.utils import Sorry, raise_from
from libtbx import adopt_init_args
import scitbx.lbfgs ;
from scitbx import lbfgs

class rosenbrock(object):
  def __init__(self, a, b ):
    adopt_init_args(self, locals())
    assert self.a.size() == 2

  def update(self, x):
    self.x = x
    assert self.x.size() == 2

  def target(self):
    return (self.a-self.x[0])**2+self.b*(self.x[1]-self.x[0]**2)**2

  def gradients(self):
    g1 = 2*(self.x[0]-self.a) + 4*self.b*(self.x[0]**3-self.x[0]*self.x[1])
    g2 = 2*self.b*(self.x[1]-self.x[0]**2)
    return flex.double([g1,g2])

def lbfgs_run(target_evaluator, use_bounds, lower_bound, upper_bound):
  minimizer = lbfgs.minimizer(
    m      = 5,
    l      = lower_bound, # lower bound
    u      = upper_bound, # upper bound
    ndd    = flex.int(target_evaluator.x, use_bounds)) # flag to apply both bounds
  minimizer.error = None
  try:
    iter1 = 0
    while 1:
      iter1 += 1
      x, f, g = target_evaluator()
      have_request = minimizer.process(x, f, g)
      if(have_request):
        f, g = f_and_g = minimizer.requests_f_and_g()
      else:
        assert not minimizer.requests_f_and_g()
      if(minimizer.is_terminated()): break
  except RuntimeError as e:
    minimizer.error = str(e)
  minimizer.n_calls = iter1
  return minimizer

class minimizer_bounds(object):

  def __init__(self,
               calculator,
               use_bounds,
               lower_bound,
               upper_bound,
               initial_values):
    adopt_init_args(self, locals())
    self.x = initial_values
    self.n = self.x.size()

  def run(self):
    self.minimizer = lbfgs_run(
      target_evaluator=self,
      use_bounds=self.use_bounds,
      lower_bound = self.lower_bound,
      upper_bound = self.upper_bound)
    self()
    return self

  def __call__(self):
    self.calculator.update(x = self.x)
    self.f = self.calculator.target()
    self.g = self.calculator.gradients()
    return self.x, self.f, self.g

class minimizer_unbounds(object):
  pass
  def __init__(self, max_iterations, calculator):
    adopt_init_args(self, locals())
    self.x = self.calculator.x
    self.minimizer = scitbx.lbfgs.run(
      target_evaluator=self,
      termination_parameters=lbfgs.termination_parameters(
        max_iterations=max_iterations))

  def compute_functional_and_gradients(self):
    self.calculator.update(x = self.x)
    t = self.calculator.target()
    g = self.calculator.gradients()
    return t,g

def run():
  # Instantiate rosenbrock class
  calculator = rosenbrock(a = 20, b = 10, x = flex.double([8,8]))

  print('Run L-BFGS (no boundaries)')
  m_unbound = minimizer_unbounds(max_iterations=100, calculator=calculator)
  print('\"Minimum: ", list(m_unbound.x))
  print()
  print('Run L-BFGS-B with boundaries')
  m_bound = minimizer_bounds(
    calculator     = calculator,
    use_bounds     = 2,
    lower_bound    = flex.double([-10000,-10000]),
    upper_bound    = flex.double([10000,10000]),
    initial_values = flex.double([8,8]).run().x)
  print('\"Minimum: ", list(m_bound.x)

if (__name__ == "__main__"):
  run()
```

Overview

The script consists of several parts:

Imports: At the beginning of the script are import statements. They are used to import the modules needed for a task.

Main function: The `run()` function is used to call the minimizer and to print the result to standard output.

class Rosenbrock: The minimizers use this class to compute values and gradients of the Rosenbrock function.

class minimizer_unbound: Class to perform minimization of the Rosenbrock function, without bounds.

class minimizer_bound: Class to perform minimization of the Rosenbrock function, without bounds.

function lbfgs_run Helper function to run the minimizer with bounds. The code could be part of the class but it helps keeping tasks apart by moving it to a separate function.

The class rosenbrock()

The minimizer needs a calculator object that represents the function to be minimized. In our example, this is the class `rosenbrock()`. The class is initialized in the `run()` function with the two parameters a and b and a starting value $x = (0, 0)$.

```
calculator = rosenbrock(a = 20, b = 10, x = flex.double([8,8]))
```

Note that x is a two-dimensional array that represents the vector (x,y) in equation (3); so x[0] = x and x[1] = y. If we translate the notation in the code to the notation in the equation:
The `update()` method assigns the vector (x,y) as attribute x of the rosenbrock class. We get the current value of x from the minimizer and it is used to calculate the next value of the minimization procedure. After enough iterations, this value should correspond to the minimum of the function.
The `target()` method returns the value of the rosenbrock function in eqs. (3) calculated at (x,y).
The `gradients()` method returns the array of partial derivatives (equations 4 and 5).

The run() function

The `run()` function instantiates the `rosenbrock()` class with parameters a=20 and b=10:

```
calculator = rosenbrock(a = 20, b = 10, x = flex.double([8,8]))
```

Then it creates the `minimizer_unbounds()` class:

```
m_unbound = minimizer_unbounds(max_iterations=100, calculator=calculator)
```

and the `minimizer_bounds()` class:

```
m_bound = minimizer_bounds(
  calculator     = calculator,
  use_bounds     = 2,
  lower_bound    = flex.double([-10000,-10000]),
  upper_bound    = flex.double([10000,10000]),
  initial_values = flex.double([8,8]).run().x)
```

In both cases, this will launch the minimization. For both computations, we print the result to standard output.

Minimization without bounds

For minimization without bounds, we will use the minimizer from `scitbx.lbfgs`.
First, we initialize the minimizer: We set the starting value (x,y) to the initial value supplied to the rosenbrock class.

```
self.x = self.calculator.x
```

In our example, this is $x = (0, 0)$.

Then, we start the minimizer:

```
self.minimizer = scitbx.lbfgs.run(
  target_evaluator=self,
  termination_parameters=scitbx.lbfgs.termination_parameters(
    max_iterations=max_iterations))
```

As the name suggests, the `scitbx.lbfgs.termination_parameters` object supplies parameters relevant for decision-making to terminate the minimization. The full set of termination parameters can be found in scitbx/lbfgs/__init__.py. In our example, we only change one parameter, the maximum number of iterations, which we set to 100 when we instantiated the `minimizer_unbounds()` class in the `run()` function.

To perform L-BFGS minimization, we need to calculate the value and the derivative of our function at a point (x,y). this is done in the `compute_functional_and_gradients()` method:

```
def compute_functional_and_gradients(self):
  self.calculator.update(x = self.x)
  t = self.calculator.target()
  g = self.calculator.gradients()
  return t,g
```

This is all we need for minimization without boundaries. The attribute x of the `minimizer_unbounds()` object will give us the minimum of the Rosenbrock function. Let's print the value in the `run()` function:

```
print("\"Minimum: ", list(m_unbound.x)
```

Minimization with bounds

Sometimes, the variables of the function to be minimized are not expected to exceed or be inferior to certain values. Then we can use this information and apply boundaries to the minimization procedure. This means we will use the L-BFGS-B algorithm, which is in scitbx:

```
from scitbx import lbfgs ;
```

The lbfgsb procedure needs to be constructed a bit differently than the lbfgs minimizer, but the principle is the same.
First, we construct the `minimizer_bounds()` class in the `run()` function:

```
m_bound = minimizer_bounds(
  calculator     = calculator,
  use_bounds     = 2,
  lower_bound    = flex.double([-10000,-10000]),
  upper_bound    = flex.double([10000,10000]),
  initial_values = flex.double([8,8]).run()
```

By providing the parameters for `use_bounds`, `lower_bound` and `upper_bound`, we apply a minimum value of -10,000 and a maximum value of 10,000 for both x and y. Note that we could have chosen different values for each variable.

In the class, the minimizer is first initialized. As in the previous example (minimizer_unbound), the starting value is set to $x = (0, 0)$.

```
def __init__(self,
             calculator,
             use_bounds,
             lower_bound,
             upper_bound,
             initial_values):
  adopt_init_args(self, locals())
  self.x = initial_values
  self.n = self.x.size()
```

The `run()` method calls our helper function `lbfgs_run()`.

```
def run(self):
  self.minimizer = lbfgs_run(
    target_evaluator=self,
    use_bounds=self.use_bounds,
    lower_bound = self.lower_bound,
    upper_bound = self.upper_bound)
  self()
  return self
```

The `call()` method calculates the function value and the derivatives, a bit like in the `compute_functional_and_gradients()` method of the unbound minimizer.

```
def __call__(self):
  self.calculator.update(x = self.x)
  self.f = self.calculator.target()
  self.g = self.calculator.gradients()
  return self.x, self.f, self.g
```

Let's look at the `lbfgs_run()` function. We need to define this additional function because the lbfgsb minimizer does not take an evaluator class as input. Instead, we have to provide the variables, its function value and the gradient explicitly:

```
have_request = minimizer.process(x, f, g)
```

The rest of the code in the try-except part handles errors and makes sure the minimizer stops when appropriate, according to termination parameters.

Output

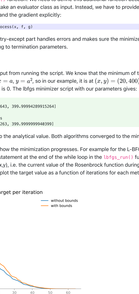
Let's have a look at the output from running the script. We know that the minimum of the Rosenbrock function is at $x = a$, $y = a^2$, so in our example, it is at $(x, y) = (20, 400)$. The value of the function at this point is 0. The lbfgs minimizer script with our parameters gives:

```
Run L-BFGS (no boundaries)
  Minimum:  [19.999999250664414, 399.999942899112641]

Run L-BFGS-B with boundaries
  Minimum:  [19.999999995281263, 399.999999994812991]
```

The values are very close to the analytical value. Both algorithms converged to the minimum.

We can also quickly check how the minimization progresses. For example for the L-BFGS-B method, by adding a print statement at the end of the while loop in the `lbfgs_run()` function, we get the current values of f(x,y), i.e. the current value of the Rosenbrock function during the minimization. We can then plot the target value as a function of iterations for each method:



In our example, the L-BFGS minimizer without bounds converged after 59 iterations, while the L-BFGS-B minimizer finished after 64 iterations. For both methods, the minimization progresses in a similar fashion.

Home
Download
How to install
Getting started
Documentation
Tutorials
Example scripts
Table of contents
Random model+density
Compare SS
R-factors
Ideal SS
LBFGS (no curvature)
LBFGS (with curvature)
Newsletter articles
About the ccttx
ccttx on GitHub

LBFGS minimzer with curvatures

This script shows how to use the LBFGS minimizer with curvatures.
Authors: Pavel V. Afonine, Dorothee Liebschner

See the LBFGS script without curvature for some information about the LBFGS algorighm.

The Rosenbrock function

To illustrate the use of the minimizer, we will use the Rosenbrock function (see also LBFGS script without curvature).
It is defined by:

$$f(x,y) = (a-x)^2 + b(y-x^2)^2 \tag{1}$$

We will use the coefficients a=1 and b=100:

$$f(x,y) = (1-x)^2 + 100(y-x^2)^2 \tag{2}$$

The partial first derivatives are:

$$\frac{\partial f(x,y)}{\partial x} = -2(1-x) + 400(-xy+x^3) \tag{3}$$

$$\frac{\partial f(x,y)}{\partial y} = 200(y-x^2) \tag{4}$$

The partial second derivatives are (the diagonal elements of the Hessian matrix):

$$\frac{\partial^2 f(x,y)}{\partial x^2} = 2 + 400(-y+3x^2) \tag{5}$$

$$\frac{\partial^2 f(x,y)}{\partial y^2} = 200 \tag{6}$$

The global minimum is where the partial derivatives are equal to zero. For the Rosenbrock function with coefficients a=1 and b=100, this is at the coordinates: $x = 1, y = 1$. So the LBFGS minimizer should find coordinates equal (or very close) to this point. The function value of the Rosenbrock function at (1,1) is equal to 0.

Code

```
from __future__ import division, print_function
from scitbx import lbfgs as scitbx_lbfgs
from scitbx.array_family import flex
from libtbx import adopt_init_args

# Rosenbrock's function, gradients and curvatures

def target(x,y):
  return (1-x)**2+100*(y-x**2)**2

def grad_x(x,y):
  return -2*(1-x) + 400*(-x*y+x**3)

def grad_y(x,y):
  return 2*100*(y-x**2)

def curv_xx(x,y):
  return 2 + 400*(-y+3*x**2)

def curv_yy(x,y):
  return 200


def lbfgs_run(target_evaluator,
              min_iterations=0,
              max_iterations=None,
              traditional_convergence_test=True,
              use_curvatures=False):
  ext = scitbx_lbfgs.ext
  minimizer = ext.minimizer(target_evaluator.n)
  minimizer.error = None
  is_converged = ext.traditional_convergence_test(target_evaluator.n)
  try:
    icall = 0
    requests_f_and_g = True
    requests_diag = use_curvatures
    while 1:
      if (requests_f_and_g):
        icall += 1
      x, f, g, d = target_evaluator(
        requests_f_and_g=requests_f_and_g,
        requests_diag=requests_diag)
      if (use_curvatures):
        if (d is None): d = flex.double(x.size())
        have_request = minimizer.run(x, f, g, d)
      else:
        have_request = minimizer.run(x, f, g)
      if (have_request):
        requests_f_and_g = minimizer.requests_f_and_g()
        requests_diag = minimizer.requests_diag()
        continue
      assert not minimizer.requests_f_and_g()
      assert not minimizer.requests_diag()
      if (minimizer.iter() >= min_iterations and is_converged(x, g)): break
      if (max_iterations is not None and minimizer.iter() >= max_iterations):
        break
      if (use_curvatures):
        have_request = minimizer.run(x, f, g, d)
      else:
        have_request = minimizer.run(x, f, g)
      if (not have_request): break
      requests_f_and_g = minimizer.requests_f_and_g()
      requests_diag = minimizer.requests_diag()
  except RuntimeError as e:
    minimizer.error = str(e)
  minimizer.n_calls = icall
  return minimizer


class minimizer:

  def __init__(self, xx=-3, yy=-4, min_iterations=0, max_iterations=10000):
    adopt_init_args(self, locals())
    self.x = flex.double([xx, yy])
    self.n = self.x.size()

  def run(self, use_curvatures=False):
    self.minimizer = lbfgs_run(
      target_evaluator=self,
      min_iterations=self.min_iterations,
      max_iterations=self.max_iterations,
      use_curvatures=use_curvatures)
    self(requests_f_and_g=True, requests_diag=False)
    return self

  def __call__(self, requests_f_and_g, requests_diag):
    self.xx, self.yy = self.x
    if (not requests_f_and_g and not requests_diag):
      requests_f_and_g = True
      requests_diag = True
    if (requests_f_and_g):
      self.f = target(self.xx,self.yy)
      self.g = flex.double(
        (grad_x(self.xx, self.yy),
         grad_y(self.xx, self.yy)))
      self.d = None
    if (requests_diag):
      self.d = flex.double(
        (curv_xx(self.xx, self.yy),
         curv_yy(self.xx, self.yy)))
      assert self.d.all_ne(0)
      self.d = 1 / self.d
    return self.x, self.f, self.g, self.d

def run():
  for use_curvatures in (False, True):
    print("use_curvatures:", use_curvatures)
    m = minimizer().run(use_curvatures=use_curvatures)
    print(tuple(m.x), "final")
    if (abs(m.x[0]-1) > 1.e-4 or abs(m.x[1]-1) > 1.e-4):
      print(tuple(m.x), "failure, use_curvatures="+str(use_curvatures))
    print("iter,exception:", m.minimizer.iter(), m.minimizer.error)
    print("n_calls:", m.minimizer.n_calls)
    print()
    assert m.minimizer.n_calls == m.minimizer.nfun()

if (__name__ == "__main__"):
  run()
```

Overview

The script consists of several parts:

Imports: At the beginning of the script are import statements. They are used to import the modules needed for our task.

Rosenbrock function: `target` returns the value of the Rosenbrock function, `grad_x` and `grad_y` return the partial derivatives, `curve_xx` and `curve_yy` return the diagonal elements of the Hessian matrix (matrix of partial second derivatives).

Main function: The `run()` function is used to call the minimizer and to print the result to standard output.

Class minimizer: Class to perform minimization of the Rosenbrock function.

Function lbfgs_run: Helper function to run the scitbx minimizer.

The run() function

The `run()` function exectues the `run()` method of the `minimizer()` class, once with curvatures and once without curvatures:

```
m = minimizer().run(use_curvatures=use_curvatures)
```

This will launch the minimization. For both computations, we print some results to standard output.

The minimizer class

The `minimizer` class has three methods: `__init__()`, `run()` and `__call__()`.

The `__init__()` method initializes our class with the starting point (-3, -4) (`xx` and `yy`), and sets the number of minimum and maximum iterations to 0 and 10,000, respectively.
`x` is a two-dimensional array that represents the x and y coordinates, so x[0] = -3 and x[1] = -4 are the (arbitrarily chosen) starting points of the minimization task.
`self.n` represents the length of the array `x`.

The `run()` method starts the L-BFGS minimizer:

```
self.minimizer = lbfgs_run(
  target_evaluator=self,
  min_iterations=self.min_iterations,
  max_iterations=self.max_iterations,
  use_curvatures=use_curvatures)
```

Then it executes the `__call__()` method:

```
self(requests_f_and_g=True, requests_diag=False)
```

The `__call__()` method allows an instance of the minimzer class to behave like a function, i.e. it can be called like a function; `minimizer(requests_f_and_g=True, requests_diag=False)` is then a shorthand for `minimizer.__call__(requests_f_and_g=True, requests_diag=False)`. Depending on the values of the boolean parameters `requests_f_and_g` and `requests_diag`, the `__call__()` method will update the target and the gradients or the diagonal elements of the Hessian matrix (second derivatives). Note that it will actually calculate the inverse of the diagonal elements (the Newton method uses the inverse of the Hessian matrix and the LBFGS algorithm approximates the Hessian matrix):

```
assert self.d.all_ne(0)
self.d = 1 / self.d
```

(As we are calculating the inverse, we first assert that the diagonal elements are different from zero.)

The lbgfs_run function

Here, we import the lbfgs module from scitbx:

```
ext = scitbx_lbfgs.ext
```

Then we instantiate the minimizer from lbfgs with the dimension of our function (in our case: n=2).

```
minimizer = ext.minimizer(target_evaluator.n)
```

Then we get a function to check if the minimization converged. Its arguments are the function value and the gradients.

```
is_converged = ext.traditional_convergence_test(target_evaluator.n)
```

Now we can start our minimization loop. We get the values of the booleans `requests_f_and_g` and `requests_diag` from the scitbx minimizer, but the starting values are True and the value of use_curvatures.

We get the coordinates, the function value, the gradients and the curvatures from our minimizer class.

```
x, f, g, d = target_evaluator(
  requests_f_and_g=requests_f_and_g,
  requests_diag=requests_diag)
```

Then we run the scitbx minimizer with the coordinates, the function value and the gradients as input parameters. If we are using curvatures, we also supply the second derivatives. The minimizer will return requests for function values and the gradients and/or for the curvatures:

```
requests_f_and_g = minimizer.requests_f_and_g()
requests_diag = minimizer.requests_diag()
```

If the minimizer requests values, we continue in the loop. Otherwise, we check if we meet the convergence criteria. Finally, the function returns the scitbx minimizer object.

Output

The `run()` function has some output. It will print the final coordinates, the number of iterations in the scitbx minimizer and an exception message, if present.

```
use_curvatures: False
(0.99999983082815378, 0.99999968299645546) final
iter,exception: 27 None
n_calls: 34

use_curvatures: True
(1.0000002135819253, 1.0000004068037093) final
iter,exception: 43 None
n_calls: 60
```

Both methods converged to the minimum at (1,1).

The Rosenbrock function
Code
Overview
The run() function
The minimizer class
The lbgfs_run function
Output

Home
Download
How to install
Getting started
Documentation
Table of Contents
Programming algorithms
Pythonicities features
Pythonicities syntax
Pythonicities tips
High level objects
Maps
Vectors
HKLviewer
Live report
Analysis
Example scripts
Interactive shells
About the crilles
crilles on GitHub

HKLviewer

Learn how to visualize the reflections of your diffraction dataset.
Author: Roland D. Geffner

Introduction

Once you have imported diffracted data sets to unmerged pickle binary files that may hereafter be visualized. The HKLviewer is available from a CCTBX installation. In the command line, type `crilles.hklview` in a system shell. Interactively double click the file `cctbx.hklview` in your `cctbx_project/crilles` folder. Alternatively, start the shell, then type `cctbx.hklview` to start the HKLviewer.

If you find HKLviewer useful in your work please cite Computational Crystallography Newsletter (2021) 12, 151-161 in your publication.

Bug reports or queries

Please submit bug reports or queries to rdg@duke.edu or on crilles@lists.lbl.gov. For suggested enhancements to the HKLviewer feel free to create a pull request on https://github.com/cctbx/cctbx_project.

Opening a file

Data files can be opened from the "File | Open" menu item. Alternatively the filename can be specified as an argument on the command line where the CCTBX environment has been invoked.

```
cctbx.hklview 3ofc.mtz
```

The types of reflections that can be loaded are: .mtz, .hkl, .sf and .sca files (supporting the Shelx and Jana formats) and .pickle files.

Display reflection data

Once a file has been opened the reflection data set can be displayed by double clicking the row with its corresponding label in the open left table. The viewer will display reflections in 3 dimensions according to their reciprocal space coordinates (h,k,l vectors) and with respect to the current viewer orientation. Use the Left and the Right mouse button of the sphere which rotates depends on the table active.

Display uncertainties (sigma)

If a dataset contains uncertainties (sigma) in addition to diffraction data, one is identified as "I/SIGI". Also this can be displayed with a right click on the respective row in the upper left table. Reflections with sigma values larger than the current "Sigma cutoff" value in the menu that has been set displays the data in the viewer with the colour mapping and the size of the reflection spheres now being mapped according to sigma values in the dataset.

Colour mapping

The reflections are coloured according to the colour gradient maps of the Python module matplotlib. Currently there are 57 different colour gradient maps to choose from. The current map is the viewer's colour of displayed dataset. The left listing up the colour gradient maps listing as shown below:

Colour gradient maps in an array of 128 values that usually ranks smoothly between adjacent array elements. The mapping between a data value, x, and a colour gradient map is done with a power law:

$$f(x) = (x - x_{min})^{\gamma}/(x_{max} - x_{min})^{\gamma}$$

where x_{min} and x_{max} are the minimum and maximum data values found in the data set. The power exponent γ can be selected by the user in the menu. The function $f(x)$ returns the index of the colour gradient map chosen by the user. With a power exponent $\gamma = 1$ the function $f(x)$ returns a value of $f(x_{min}) = 0$ for the start of the colour gradient array will be interpolated. If $\gamma > 1$ it will return values closer to the end of the colour gradient array. Reflections with data values closer to the maximum will be coloured with a shade of red. If $\gamma < 1$ the opposite holds true, the figure of merits are now viewed with a finer mapping between colours and the data values. In a should be set to $\gamma = 1$ for a uniform mapping. Choose γ to select the colour gradient for the resolution zones mapping for long lasting the chosen colour gradient map.

Displaying map coefficients

For data with map coefficients in phases only the phase value (radian) 360 is used for the colour mapping. The map coefficients are displayed as reflections with the phase value as the colour and as one of the array equals the colour at the beginning of the array. Averaged the phase value of the array equals the colour at the end of the array. Reflections with phase values closer to the phase 0 will be coloured in red. Reflections with phase values closer to the phase 2π will be coloured in red.

Clicking on a desired colour gradient in the Colour gradient maps dialog will instruct the HKLviewer to use the colour mapping for the displayed data or sigma values. The choice is used for the current viewer and for the current displayed dataset. Clicking on the desired colour gradient chosen for, say, a dataset of amplitudes will cause any other dataset of amplitudes to use the same colour mapping.

Expansion

Processed reflection data usually are stored with only the reflections that are unique under the symmetry of the crystal. Reflections that are related by symmetry to the unique reflections other symmetry operations have been applied. The current of the recorded reflections are for convenience stored in the same assigned of the sphere of all reflections. For the P1 space group this is the case. For other space groups the reflections are stored in the asymmetric unit as the minimum number of reflections necessary for a complete set in a given resolution.P1 is the default space group. The HKLviewer can be instructed to expand the reflections to a space group the assigns of the assigns of reflections will be local smaller and shaped like a prismy with sharp edges.

Some situations where it is preferable to inspect reflections data in P1 and Friedel pairs of data including Friedel mates. This is done in the HKLviewer by ticking the "Expand reflections" and "Expand Friedel mates" in the "Show" menu. The reflections will be expanded and displayed accordingly. If the reflection data file does contain reflections that are not unique under symmetry only the sphere of P1 expansion will be available.

It is also possible to display "Missing reflections". These are the reflections that would have been in the data set if it had been collected in P1 or Friedel mates if the data set had been lost during measurement, due to a radiation damage of the crystal or a shading beamstop. The HKLviewer can be instructed to display only the missing reflections for the data set (P1 expanded or/or including Friedel pairs).

Slicing reflections

It is often useful to look at a slice through the displayed reflections. It is most informative to find reflections that are in the plane of the zero axis. In the HKLviewer this is done by ticking the "Slicing" tab pane pushed to the viewer that displays reflections of the reflections by the axis. This can be useful when scattering reflections governed by a mirror operator where the axis of the mirror is perpendicular to the zero axis. Reflections in the plane of the zero axis. Below are reflections of the data set 2DF8 displayed with a slice through the zero axis in the HKL viewer. The scattering axis of the screw is a distance of 0.5 from the origin.

Reflections can also be shown as just a single plane of reflections restricted to the ones where one of the axes is 0 or near a fixed value as shown below.

Sizing the displayed reflections

The size of each displayed reflections is a function of its associated data value. Similar to colour mapping of data values a function is employed that produces a radius, r, dependent on the data value, x, and a power scale factor, γ:

$$r(x) = (x - x_{min})^{\gamma}/(x_{max} - x_{min})^{\gamma}$$

where x_{min} and x_{max} are the minimum and maximum data values found in the data set. The power scale factor can also be selected by the user or be determined automatically.

For data sets of amplitudes or intensities where there often are more orders of magnitude between the strongest and the weakest reflections it is preferable to use a power scale factor smaller than the power scale factor 0 (other ways of reflections the same γ or 1). In Bragdon ends up actually recorded. Alternatively, to get a better view of the strength of the data or automatic scaling of the reflections to be displayed in the viewer, the reflections can be scaled so that the most that the strongest reflections. Selecting Lowest scale factor of 1 for amplitudes or intensity data is rarely useful as typically the data set is dominated by a handful of very strong data reflections. Setting a high scale factor may aid in illustrating the presence of strong anomalous, or RIS-modulation in a data set.

Binning

A data set can be sorted into bins according to resolution and if more data sets are present in the data set, the data sets can be sorted into separate bins. The number of bins can be set in the menu dialogue. Let us see from the spin crystal in a resolution of 20 - 7Å. The HKLviewer will attempt to divide the currently displayed data into bins having about the same number of reflections in each bin. The bin number can be set by selecting a predefined threshold for a particular bin now under the "Bin value" column.

In the figure below 8 bins of absolute have been created for the P3-P3DR1 dataset.

Changing bin thresholds

If one wishes to change the bin threshold for 4th bin from 1.21 to 1.35 Å this is done by entering the new value in the dialogue box.

Entering return on the keyboard processes the new value and computes new number of reflections in the bin and adjusted sizes.

The reflections in the bin can be visualisation by deselecting reflections in all other bins in the specify column.

Data can also be binned against other data values by selecting the desired data value for binning from the drop-down list.

Below is the data set P3-DDR1 binned according to the sigma found in the data set but only displaying the bins for the data set P3-DDR1 with the Lots F(h). This is done by unticking the checkboxes for the bad sets in the specify column.

Data can also be made semi transparent by selecting a value between 0 and 1 in the opacity column. This may be useful when dissecting sections that are shorter than the radius of the sphere of reflections. Setting a low opacity value for a set of reflections will make them appear as a cloud causing artifacts of object interpretation disappear.Alternatively, if opacity is set below 0.5 reflections will not respond to hovering or clicking with the mouse.

Vectors

Use the Vectors tab series vectors can be displayed in reciprocal space. These are the rotation vectors used by the space group, any HKLO screw specified in the file header as well as vectors defined by the user either in reciprocal or in real space fractional coordinates. In the HKLviewer vectors can be displayed in the viewer. Vectors defined with (0,0,0) as their origin or only real and imaginary part listed in the table.

User defined vectors

In addition to rotation axes given by the space group it is possible to explicitly enter a rotation vector in the "Vectors" tab pane. Rotation vectors are specified in the "Vectors" tab pane. Select "rotation" column of the table of rotation operators or vectors with a name of, for example, "rotation_1".

HKLviewer will parse the operator where the name begins with "rotation" if and only if the row in the table will become visual only and the name specified in the show column will have "h,k,l" in the name, or there is the number of Vectors leads the operators to achieve the identity operator.

Rotation operators should be commensurate with the space group. However, some tetragonal crystals have a screw axis of 4 in the space group but the crystal is in fact a 2 screw axis. Alternatively, some crystals belong only one crystal in which the symmetry related reflections fit in a given reflections data set. In the case of the 2DF8 data set the space group is P4 but the crystal is in fact a 2 screw rotation about a screw has been made in fact and reflections have been expanded to P1 and Friedel mates. This is useful when inspecting how related reflections are discovered in the Friedel mates.

It is also possible to enter a reciprocal space vector. To do this first overwrite the "new name" field in the "Vector" column with a known name, say "myVector1". Then enter the reciprocal space vector in the same row under the "as hkl" column.

When the HKLviewer parses the coordinates the row will become rows only and the coordinates encapsulated by brackets.

Real space fractional vectors may also be entered in the same manner as for reciprocal space vectors but with the "as xyz" column.

The unit cells for real space and reciprocal space can also be shown for the sake of locating the HKLO vector on rotation operations. In the unit cell Below the 2DF8 data set displayed with the redspace unit cell, the R-Hob rotation axis, the myVec1 and the myVec2 vectors defined above.

Making a new dataset

A new dataset can be created from one or two existing dataset that are present in the file loaded. The new dataset can be created by selecting the "Make new dataset" button in the "Show" menu. Select in the Mtz file dialogue box known as crilles.hklview.scripts.data.mtz. The crilles.hklview.scripts module provides a list API of functions for crystallography which makes many crystallographic computational tasks simple as the crilles.scripts module.

To get started, right click a dataset in the opened table in the HKLviewer which you want to use for the new dataset. Select "Make new dataset" in the menu. Type "help()" in the "Command" field. Hit the "Execute and another dataset" menu item. This invokes the "Execute new dataset" dialog.

In the textbox a python expression that defines the new dataset, newarray, can be entered. Below that, a label for the dataset should also be entered. Pressing the "Compute new reflection dataset" button then parses and executes the python expression. If succesful the new dataset with the specified column label will appear at the bottom of the table of existing datasets and automatically be displayed in the viewer. For convenience newarray is initialised to a copy of array1 (here representing "I,SIGI").

Variables

The following variables are available for a python expression submitted in the "Create a new reflection dataset" dialog:

Variable or member function	Type	Description
newarray	cctbx.miller.array	The new dataset that is computed from the python script
array1 or array2	cctbx.miller.array	array1 is the variable name of dataset that was right-clicked in the upperleft table of dataset. array2 is the variable name of the dataset that optionally can be selected from the dropdown list
newarray._data, array1.data() or array2.data()	flex.double, flex,complex, flex.int or flex.hendrickson_lattman	Observed data values, figure of merits, map coefficients, Rfree or other column values present in the reflection file.
newarray._sigmas, array1.sigmas() or array2.sigmas()	flex.double or None if not provided	Observed sigma values associated with the data values.
array1.indices() or array2.indices()	flex.miller_index	HKL indices of each reflection. If array2 has been selected for the computation as well, the indices will be the common subset of indices of both datasets.
dres	flex.double	Resolution of each HKL index

The dres variable is provided as a shorthand for `array1.unit_cell().d(array1.indices())`.

Examples of creating a new dataset

When creating a new dataset copies, the objective is to define `newarray` as a new `cctbx.miller.array` object or to assign its attributes `newarray._data` and `newarray._sigmas` to flex arrays computed in the script. The `newarray` is initalised to the dataset that was right-clicked for invoking the "Create a new reflection dataset" dialog. So the `newarray.indices` attribute needs not specifying. The `array1` and `array2`, of the selected datasets are available for scripting in the "Create a new reflection dataset" dialog. The data(), sigmas() and indices() member functions of `array1` and `array2` can then be used for composing the new dataset. A few examples are given in the table below:

Objective	Python script to be entered
Calculate normalized structure factors, E, from amplitude, F, or intensity, I, dataset.	newarray = array1.normalize()
Calculate I/Sigl for an intensity dataset.	dat = array1.data() / array1.sigmas() newarray._data = dat newarray._sigmas = None newarray.set_observation_type(None)
Multiply data values in one dataset with the squareroot of data values in another dataset	dat = array1.data() * flex.sqrt(array2.data()) newarray._data = dat newarray._sigmas = None newarray.set_observation_type(None)
French-Wilson massage intensity data values.	newarray = array1.french_wilson()

The important thing to bear in mind is that the newarray._data and optionally newarray._sigmas should be of type flex arrays and not scalar values, list or tuples. Many mathematical functions have been overloaded for flex arrays. So when computing the squareroot, the absolute value or raising all elements of a flex double array to a certain power use flex.sqrt(array1.data()) etc. Avoid using a for-loop or list comprehension over all elements in a flex array in the python script.

Further API details

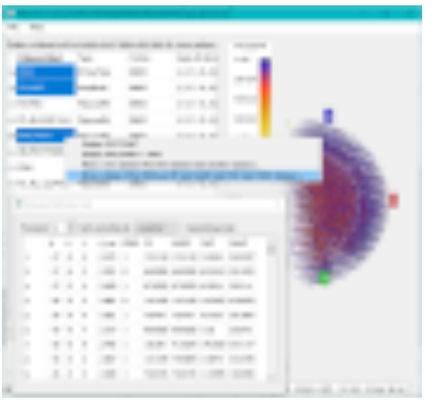
There are many more functions available from the cctbx.miller.array class. Details about flex arrays can be found here and here.

Saving datasets

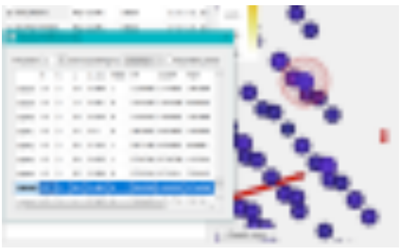
Once new datasets have been created it is possible to save these from the File menu by clicking the "Save Reflection File" item. Supported file formats are MTZ and CIF. If a reflection file contains unmerged data with multiple values for one or more indices each dataset will be saved in separate CIF files. Currently it is not possible to save unmerged data in the MTZ format.

Table of datasets

With a right click on the upper-left table of datasets in the reflection data file individual datasets can be tabulated from the context menu. First highlight the datasets of interest. Then click on "Show a table of ...". This brings up a window with tabulated data as it is stored in the reflection data file as well as the ones that may have been created from scripting.



The table can be sorted according to the values of any available dataset, to the resolution. Double clicking a row in the table will cause the viewer to zoom in on that particular reflection with a red mesh



Similarly will a right-click on a displayed reflection in the viewer retrieve and highlight that reflections data value in the table.
The tabulated datasets will remain unchanged when the displayed reflections are transformed by expanding to Friedel mates, P1 or lower symmetry spacegroups. This also means that double clicking a symmetry copy generated from expanding the dataset to P1 or Friedel mates will only retrieve the original HKL index and its data value in the table.

Home
Download
How to install
Getting started
Documentation
Tutorials
Table of contents
HKLViewer
Melk 2019
Rigid Body
cctbx tour
Example scripts
Newsletter articles
About the cctbx
cctbx on GitHub

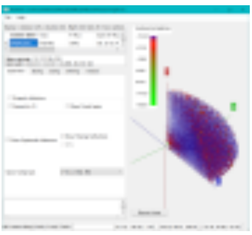
HKLViewer tutorial

Display I/SigI of a dataset
Computing I/SigI
Inspecting TNCS modulation in a dataset
Inspecting a twin law

This tutorial shows some use cases of the HKLviewer.

Display I/SigI of a dataset

Reflections with I/SigI > 2 are considered to be good data for structure solution. Therefore, this cutoff is often used to determine the resolution limit. Here we examine a problematic dataset to show that for a given threshold of I/SigI a matching resolution cutoff does not necessarily exist. Open the dataset merged.mtz in the HKLviewer. The IMEAN,SIGIMEAN reflections are shown by double clicking the row with that label in the upper left table.

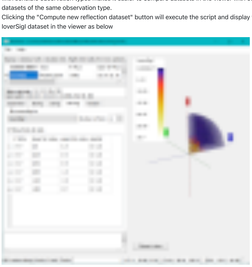


By default, the colour mapping for each reflection is according to values of the intensity. The next section explains how to color according to I/sigI values.

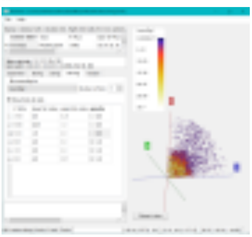
Computing I/SigI

Right click the row with the "IMEAN,SIGIMEAN" label and from the context menu select "Make a new dataset from this dataset and another dataset".
In the "Create New Reflection data" dialog enter the lines of python code as given below into the textbox and "loverSigI" as the formula and the label for the new data respectively.

The first statement `newarray._data = flex.abs(array1.data())/array1.sigmas()` computes the I/SigI of all data values but with the absolute values of the intensities since there appear to be a significant number of negative intensities.
For convenience `newarray` is initialised to a copy of the original array (here representing "IMEAN,SIGIMEAN").
For I/SigI data we are not interested in the sigma values themselves. So we discard the sigmas array with the second statement `newarray._sigmas = None`.
The third statement `newarray.set_observation_type(None)` is optional. It tells CCTBX to deduce which type of data newarray will contain. In the case of I/SigI it will be plain real numbers. It could have been more specialised types such as map coefficients or X-ray observations. Since the colour mapping and sizing schemes are specific to each observation type, datasets classified with sensible observation types makes it easier to compare datasets in the viewer with other datasets of the same observation type.
Clicking the "Compute new reflection dataset" button will execute the script and display the loverSigI dataset in the viewer as below



The "Binning" Tab lets you choose the number of bins and the thresholds. By default data is binned according to resolution. But here we have binned against the newly created "loverSigI" column and set the number of bins to 6.
HKLviewer will try to put an equal number of reflections into each bin and then compute corresponding bin thresholds. For this exercise, we want the bin thresholds at defined I/SigI values: 4.0, 3.0, 2.0 and 1.0.
Do this by entering these new bin thresholds by typing the numbers 4,3,2,1 in row 6, 5, 4 and 3 in the "lower bin value" column.
Then untick the "Opacity" checkboxes of the first three bins to hide reflections with I/SigI less than 2.



The first column of the "Binning" shows the number of reflections in each bin. The relatively small total number of reflections with I/SigI > 2 suggests that it could be difficult to solve the structure for this data set. This is also visible in the Viewer on the right, which shows much fewer spheres than the figure above where none of the bins have been excluded.

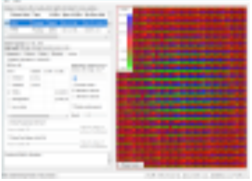
Inspecting TNCS modulation in a dataset

tNCS or pseudo-symmetry is a crystal pathology where two or more molecules in the unit cell are aligned in a way to cause interference in the diffraction pattern. It shows as periodic modulation of the diffraction pattern and can cause problems for structure solution unless identified and taken into account by the software. tNCS vectors are real space vectors and can be determined before structure solution attempts. The dataset 6C5F (available from the PDB) has tNCS which is detected by programs such as `phenix.xtriage`. It determined the pseudo-symmetry vector to be (0.333, -0.334, 0.040) in fractional coordinates. Follow the steps below:

- Open the data file in the HKLviewer.
- Double click the "F_meas_au,F_meas_sigma_au" label (or "FP,SIGFP" if loading an mtz file) to display the data.
- On the Expansion tab, expand the reflections to a sphere.
- On the sizing tab, tick the "User defined power scaling" checkbox and enter a power factor of 2 to obtain values that are roughly proportional to intensities (the file from the PDB does not provide the original intensity data).
- On the Vectors tab, enter the coordinates for the tNCS vector found by xtriage (0.333, -0.334, 0.040) on the last line by overwriting the "new vector" label to "xtrageTNCS" and entering the coordinates in the same row under the "as abc" column. Pressing return will submit the vector to HKLviewer.
- Tick the xtrageTNCS vector checkbox as well as the "Orient vector" checkbox and the radio button "Parallel to screen".
- Either rotate the reflections manually on the slider control or tick the "Rotate continuously" checkbox.

This will show the modulations in the data as ripples in the reflections in planes perpendicular to the tNCS vector. It can be useful to tick the "Show Real Space Unit cell" checkbox to see where the tNCS vector points within the unit cell.

Phasertng, which is currently under development, stores a data column, TEPS, with the correction factors, epsilon, revealing the tNCS modulation of the intensities. The coordinates for the tNCS vector is embedded in the mtz file when the mtz file has been processed with phasertng. This is illustrated below where the reflections have been sliced with a clip plane width of 8. On the sizing tab we have set the power factor to 0 and the linear scaling factor to 0.8. The modulation is clear from the alternating layers of red, green and blue layers of reflections.



Inspecting a twin law

The phenix tutorial with the file, porin.mtz, illustrates twinning. The twin operator found by phenix.xtriage is -h-k,k,-l.
Follow the steps below:

- Load the file in the HKLviewer and display the dataset "F-obs,SIGF-obs".
- On the Vectors tab, enter the twin operator into the table of vectors; first by overwriting the last row with the "new vector" string with "twin". Then by entering the string "-h-k,k,-l" in the adjacent rotation column and pressing return for the HKLviewer to accept the new twin operator. It turns out to be a 2-fold rotation so the HKLviewer will prefix the name with 2-fold.
- Press the "Orient vector" checkbox and the radio button "Normal to screen".
- On the Expansion tab, expand the reflections to P1 and Friedel mates.
- On the Slicing tab, tick the "Slice with a clip plane" checkbox and set "Distance from Origin" and "Clip plane width" to 4 and 1 respectively.
- Show a "Tabulated Reflection data" window by right clicking the column label "F-obs,SIGF-obs" in the upper left table and selecting "Show a table..." from the menu.

To inspect data values of reflections related by the twin operator right-click a reflection in the view. It will bring up two tooltips with the hkl indices at the end of two vectors from the reciprocal space origin pointing to the twin reflections. Due to the clip plane, the vectors may not be fully visible. Double click one of the reflections to bring up the standard tooltip with all other values represented by this reflection (dres, F-obs, SIGF-obs, R-free-flags). Right-clicking a reflection will highlight the data value in the "Tabulated Reflection data" window. The HKL indices in the table are the original from file. Since the twin operator is not part of the spacegroup the hkl index in the tooltip of a twin related reflection may differ from the HKL index generated by P1 or Friedel mates expansion highlighted in the Tabulated Reflection Data.