Home
How to install
Documentation
Programming

Phenix developers documentation

The Phenix developers documentation is aimed towards researchers who contribute tools to Phenix. It might be also of interest to scientists who want to use Phenix or cctbx in scripts.

Topics include the following:

- Description of high level Phenix tools
- How to use the Phenix program template
- Useful jiffys that speed up debugging or analysis
- Programming tips; for example how to profile runtime

Home
How to install
Documentation
Programming

Install and set up Phenix

A prerequisite of using Phenix modules is to install Phenix.

Before you start you will need to install and set up Phenix so that when you type the command `phenix.python` in a terminal window on your computer you get something like this:

```
Python 2.7.15 | packaged by conda-forge | (default, Mar  5 2020, 14:58:04
[GCC Clang 9.0.1 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The >>> is the command prompt that means “type something here”.

When you have Phenix installed, you automatically get tutorial, testing, and demo data files that come with Phenix. We’ll use some of those in this tutorial. Some of these will be in the directory:

```
$PHENIX/modules/phenix_regression/model_building/
```

which contains tests for model-building tools.

The Phenix directory structure

It is helpful to have an idea about the directory structure in Phenix so that you can find Phenix tools that you want to use in your programming. This page describes the overall setup.

The top-level Phenix directory is typically located in the directory referred to by the environmental variable `$PHENIX`. It contains the following files and folders:

- bootstrap.py -- Python program used to build and update
- conda_base -- base for Conda
- dev_env -- for installing git.lfs
- modules -- source programs
- build -- installed programs
- doc -- installed documentation

Since all Phenix code is in the `modules` directory, the location of a particular file is often referred to relative to this modules directory. For example, as `modules/phenix/phenix/programs/mtriage.py`.

Within the modules directory are a big set of directories for different modules of Phenix (including cctbx_project, the computational crystallography project):

```
Plex                    dxtbx                   phenix_dev_doc
PyQuante                eigen                   phenix_examples
amber_adaptbx           elbow                   phenix_html
amber_library           gui_resources           phenix_regression
annlib                  iota                    probe
annlib_adaptbx          king                    pulchra
boost                   ksdssp                  reduce
cbflib                  labelit                 reel
ccp4io                  msgpack-3.1.1           scons
ccp4io_adaptbx          msgpack-3.1.1.tar.gz    scons-3.1.1.zip
cctbx_project           muscle                  solve_resolve
chem_data               opt_resources           suitename
clipper                 phaser                  tntbx
cxi_user                phaser_regression       xia2
dials                   phenix
```

Several directories are of particular interest:

- **cctbx_project**: Most of the source code for low-level programming is in the cctbx_project directory. This code is documented in the cctbx documentation.
- **phenix**: Most of the code for Phenix programming is in the phenix directory (see below)
- **phenix_regression**: Regression tests for phenix tools
- **phenix_examples**: Tutorials on Phenix tools
- **phenix_html**: Documentation on Phenix

The phenix module has the following subdirectories:

- **phenix**: source code for most Phenix tools
- **wxGUI**: no longer used
- **wxGUI2**: source code for Phenix GUI
- **conda_envs**: files that define the conda environments

Often, you will add to existing files or create new files within the existing directory structure. If you start a completely new area of Phenix programming, you would create a new directory structure to hold it. Before creating a new directory structure, it should be discussed among Phenix group members.

The phenix/phenix/ subdirectory is further subdivided into:

```
__init__.py             model_building          rosetta
automation              molrep                  server
autosol                 mptbx                   solve_driver
command_line            pdb_tools               strategies
erraser                 pds                     substructure
file_reader.py          pds_server              tasks
foundation              phenix_info.py          tracking
heavy_search            program_template.py     utilities
hidden                  programs                wizard_scripts
installer               queue                   wizards
ligands                 refinement
loop_lib                regression
```

There are two special subdirectories in the phenix/phenix directory:

- **programs**: program files that interface between the user or the GUI and a program
- **command_line**: short files that specify what program to run when a user types something like `phenix.refine`

Table of contents

Home
How to install
Documentation
Programming
Program template
Runsnake
Line_profiler
C++ fast compilation

Creating a program in Phenix with the Program Template

Learn how to use the Phenix Program Template to write a new Phenix tool.

Introduction
Set up the Phenix environment
Creating the prototype for the tool
Creating a unit-test
Creating the Program Template that runs map_cc
Creating a file in the command_line directory
Running bootstrap to install the program

Introduction

Let's create a new Phenix program. To illustrate the steps involved in this process, we will create an example program that reads in a map and a model and calculates the map-model correlation. In particular, we will use the Program Template (see CCN newsletter article), which standardizes the interfaces between a user or the GUI and a program. This approach provides some consistent functionality for basic tasks, like file and parameter handling. It also helps ensure that the same code is executed regardless of the user interface.

The name of the program will be `map_cc`. While creating the program, we'll set up a unit test.

Here are the steps we are going to follow to create our program. Each step will be discussed in more detail further below.
First we make a prototype and a test that shows that the prototype works:

1. Create a prototype of the program that runs from the command line. The file for the new program will be created in the directory modules/phenix/phenix/model_building/map_cc.py. Normally you will put the code to do real work in a directory that corresponds to the type of work to be done.

2. Create a unit test to make sure that the program works now and that it continues to work later, when we do some modifications to it. The unit test will be put in modules/phenix_regression/model_building/tst_map_cc.py. The directory phenix_regression is a special directory in which any file that starts with "tst" will be considered a test to be run automatically. The subdirectory model_building matches the location of our program (phenix/model_building).

Next we create the high-level files that allow easy communication between the user, the program and the GUI:

3. Create a file modules/phenix/phenix/programs/map_cc.py that is an edited version of the Program Template. This file is going to specify the inputs to our program that does the work and it is going to call our program to do the work
4. Create the file modules/phenix/phenix/command_line/map_cc.py in the command_line directory. This is a special directory where the Phenix installer always looks and finds programs to run.
5. Run `bootstrap.py` in the top-level phenix directory to rebuild Phenix, creating all the necessary relationships so that we can type `phenix.map_cc` on the command-line to run our program.

Then we'll be set and we can run our program with the command `phenix.map_cc`, and the automatic regression testing will test our program using the `tst_map_cc.py` regression test.

When you write your own program, you can follow these same steps, substituting your code and locations for the ones shown here.

Let's go through each step in detail now.

Set up the Phenix environment

We are going refer to files in the Phenix directory, and the name of this directory depends on your setup. However, you can always use the environment variable PHENIX to refer to this top-level directory. First let's make sure your environment is set up. Open a terminal window and type:

```
which phenix
```

This should print out something that ends in "build/bin/phenix". Now let's check and see if the PHENIX environmental variable is set. Type:

```
echo "$PHENIX"
```

This should print out a directory name corresponding to your top-level Phenix directory.

If either of these does not work, please see installation and set up for how to install Phenix and set up the environment.

At the end of this procedure, we use `bootstrap.py` to rebuild your Phenix installation. If you do not have `bootstrap.py`, you can do everything in the directory of your new tool. You just won't be able to type "phenix.map_cc", so you will have to type "phenix.python $PHENIX/modules/phenix/phenix/command_line/map_cc.py" to get the same result.

Creating the prototype for the tool

We create our program as a new file. We put the code that does the real work in a directory that corresponds to the type of work to be done. Our program will use the cctbx DataManager to read in files and have keywords that we are going to set later with the Program Template.

Here is our program:

```
# -*- coding: utf-8 -*-

from __future__ import division, print_function
import sys
from libtbx import group_args

def run(
    map_model_manager= None,
    resolution = None,
    log = sys.stdout):
  '''
  Demo program to calculate map-model correlation

  Parameters:
    map_model_manager:   iotbx.map_model_manager holds map and model
    resolution:          nominal resolution of map
    log:                 output stream

  Returns:
    group_args object containing value of map_model correlation
  '''

  print ("Getting map-model correlation for the map_model_manager:\n %s
     map_model_manager), file = log)
  print ("Resolution will be: %.3f A " %(resolution), file = log)

  # use the function map_model_cc in map_model_manager to calculate the
  cc = map_model_manager.map_model_cc(resolution = resolution)

  print ("Map-model correlation is: %.3f " %(cc), file = log)

  return group_args(
     map_model_cc = cc,
   )
```

```
if __name__ == "__main__":
  from iotbx.data_manager import DataManager
  dm = DataManager()
  args = sys.argv[1:]
  if (len(args)) != 3:  # print how to run program and quit
    print ("Usage: 'phenix.map_cc <model_file> <map_file> <resolution>
  else:
    model_file = args[0]
    map_file = args[1]
    resolution = float(args[2])
    log = sys.stdout
    mmm = dm.get_map_model_manager(model_file=model_file,
      map_files = map_file,
      log = log)
    result = run(map_model_manager = mmm, resolution = resolution)
    print ("Result was: %.3f" %(result.map_model_cc))
```

Put this program in a file called “map_cc.py” in the directory modules/phenix/phenix/model_building/ (where modules is the modules subdirectory in your top-level Phenix directory).

We can run our program right away if we have a map and model. We can get them by setting up a tutorials directory. Get into a working directory in a terminal window (doesn’t matter where) and type:

```
phenix.setup_tutorial tutorial_name=model-building-scripting
```

This creates a subdirectory “model-building-scripting” that contains files we can use. Get into this subdirectory and type:

```
phenix.python $PHENIX/modules/phenix/phenix/model_building/map_cc.py \
 short_model_main_chain_box.pdb short_model_box.ccp4 3
```

This should run our program and produce output ending with:

```
Resolution will be: 3.000 A
Map-model correlation is: 0.742
Result was: 0.742
```

Creating a unit-test

Now we have a program that seems to work. Let’s make a test to make sure it continues to work as we modify it.

Our unit test should be quick and it should test the important aspects of our program. It should give a hard fail if the test does not succeed (i.e., the test should crash on failure with an AssertionError or similar error).

When creating a test, keep in mind that later on, we and other programmers are going to do things that affect our program. We want our test to fail if anyone else does something that causes our program to not work the way we want it to. Our test is our protection against us or anyone else doing something that makes the program fail.

In our example, we can use the data files we just read in to our program as test data, and the result we just got as the expected result. Below is our testing program. Notice that it looks a lot like the code at the bottom of our original program that read in files and ran the “run” method of our program. The imports at the top are typical for a test program and set up functions that we are going to need:

```
# -*- coding: utf-8 -*-
from __future__ import division, print_function

import time, sys, os
from libtbx.test_utils import approx_equal
import libtbx.load_env

def tst_01():

  # Identify where the data are to come from
  data_dir = libtbx.env.under_dist(
    module_name="phenix_regression",
    path="model_building",
    test=os.path.isdir)
```

```
  # Identify the map and model files
  map_file=os.path.join(data_dir,"short_model_box.ccp4")
  model_file=os.path.join(data_dir,"short_model_main_chain_box.pdb")

  # Import that DataManager
  from iotbx.data_manager import DataManager
  dm = DataManager()

  # Set the log file
  log = sys.stdout

  # Read in the map and model and get a map_model_manager
  mmm = dm.get_map_model_manager(model_file=model_file,
      map_files = map_file,
      log = log)

  # Set the resolution
  resolution = 3

  # Import and test the program map_cc.py:
  from phenix.model_building.map_cc import run

  result = run(map_model_manager = mmm, resolution = resolution)
  print ("Result was: %.3f" %(result.map_model_cc))

  # Make sure we got the right answer. Fails if result.map_model_cc doe
  #   match result.map_model_cc within machine precision
  assert approx_equal (result.map_model_cc, 0.7424188591)

  # We are done...if we had the wrong answer it would have crashed abov

if __name__=="__main__":
  t0 = time.time()
  tst_01()
  print("Time: %6.4f"%(time.time()-t0))
  print("OK")
```

Put this test program in $PHENIX/modules/phenix_regression/model_building/tst_map_cc.py. Anything you put there is going to be automatically run as part of the Phenix regression system.

Let's run our test:

```
phenix.python $PHENIX/modules/phenix_regression/model_building/tst_map_
```

This should run and print out "OK" at the end.

Creating the Program Template that runs map_cc

We now have a working program and a working unit test that makes sure the program is ok. Now let's use the Program Template to make it easy for a user to run this program, and also to make it easy to put this program into the Phenix GUI (we are not going to do that here but this does make it easy). Here is our Program Template file, edited to run our map_cc program:

```
from __future__ import absolute_import, division, print_function
from phenix.program_template import ProgramTemplate
from libtbx import group_args

# ==================================================================

class Program(ProgramTemplate):

  description = """
  Demo program to calculate map-model correlation
  Usage: phenix.map_cc model.pdb map.mrc 3

  """

  # Define the data types that will be used
  datatypes = ['model', 'phil', 'real_map']

  # Input parameters

  # Note: include of map_model_phil_str allows specification of
```

```
#   full_map, half_map, another half_map, and model and Program Templ
#   produces input_files.map_model_manager containing these

master_phil_str = """

input_files {
  include scope iotbx.map_model_manager.map_model_phil_str
  }

crystal_info {
  resolution = None
    .type = float
    .help = Nominal resolution of map
    .short_caption = Resolution
  }

"""

# Define how to determine if inputs are ok
def validate(self):

  # Expect exactly one map and one model. Stop if not the case
  self.data_manager.has_real_maps(
    raise_sorry = True,
    expected_n  = 1,
    exact_count = True)
  self.data_manager.has_models(
    raise_sorry = True,
    expected_n  = 1,
    exact_count = True)

# Set any defaults
def set_defaults(self):
  pass


# Run the program
def run(self):

  from phenix.model_building.map_cc import run as get_map_cc
  self.result = get_map_cc(
    map_model_manager = self.data_manager.get_map_model_manager(
       from_phil=True),
    resolution = self.params.crystal_info.resolution,
    log        = self.logger)

# Get the results
def get_results(self):
  return group_args(
    map_model_cc = self.result.map_model_cc,
   )
```

Put this text in the file “modules/phenix/phenix/programs/map_cc.py” and we are almost ready to go.

Creating a file in the command_line directory

We want to be able to type `phenix.map_cc` to run our program. The way we do this is to put a file in a special directory (modules/phenix/phenix/command_line/) that tells the installer to set this up. Let’s make a file in this directory and call it “map_cc.py”. Here are the contents of map_cc.py. It defines what the program is going to be called (phenix.map_cc) and what to run when this program is invoked (the Program unit in phenix/programs/map_cc.py)

```
# LIBTBX_SET_DISPATCHER_NAME phenix.map_cc
from __future__ import absolute_import, division, print_function

from iotbx.cli_parser import run_program
from phenix.programs import map_cc

if __name__ == '__main__':
  run_program(program_class=map_cc.Program)
```

Running bootstrap to install the program

Now we want to run `bootstrap.py` in the top-level phenix directory to rebuild Phenix, creating all the necessary relationships so that we can type `phenix.map_cc` on the command-line to run our program. If you have bootstrap.py in your top-level Phenix directory, you can type:

```
python bootstrap.py build  --builder=phenix --use_conda
```

When it is done, you are ready to go. You will need to open a new terminal window or type “rehash” in an existing one. Then you can type “phenix.map_cc” to run your program. Do it like this after getting in your directory where you put the demo files:

```
phenix.map_cc short_model_main_chain_box.pdb short_model_box.ccp4 resol
```

That should run the program and give you the same result as before.

If you do not have `bootstrap.py` you just won’t be able to type `phenix.map_cc` to run the program. You will have to type “phenix.python $PHENIX/modules/phenix/phenix/command_line/map_cc.py” instead to get the same result.

Home
How to install
Documentation
Table of Contents
Quick reference
Directory structure
Model building
Restraint jiffys
Programming

Phenix/CCTBX quick reference

This section summarizes some commonly-used functions in Phenix and cctbx.

Setup

Copy tutorial data

```
phenix.setup_tutorial tutorial_name=model-building-scripting
```

Set up a DataManager

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

Show all the keywords and summary documentation for a function

```
help(dm)
help(dm.get_map_model_manager)
```

Reading/writing files and creating managers

Get a map_model manager from a map and model

```
mmm_ligand = dm.get_map_model_manager(        # get map_model manager
  model_file="boxed_ligand_map.pdb",   # model file
  map_files="boxed_ligand_map.ccp4")   # map file
```

Read some models

```
model_for_morphing = dm.get_model("short_model_box_for_morph.pdb") # mo
model_for_sequence = dm.get_model("short_model_main_chain_box.pdb") # m
```

Make a copy of a model so that when model is shifted we have the original model

```
original_model_for_morphing = model_for_morphing.deep_copy()  # copy of
```

Read in a model of atp

```
model_atp = dm.get_model("atp.pdb")  # model file
```

Read in a restraints file

```
dm.process_restraint_file("atp.cif")  # Set up restraints
restraints_atp=dm.get_restraint("atp.cif") # get a restraints object
```

Read in map and get map_managers

```
mm = dm.get_real_map("boxed_ligand_map.ccp4")   # map file
mm_for_morphing = dm.get_real_map("short_model_box.ccp4")   # another m
dm.remove_real_map("short_model_box.ccp4")   # remove map from data_man
```

Get a map_model manager from a map only

```
mmm_map_only = dm.get_map_model_manager(        # get map_model manager
    map_files="short_model_box.ccp4")   # map file
```

Set defaults in a map_manager

```
mm.set_wrapping(False)   # do not wrap (do not extend outside supplied
mm.set_experiment_type("xray")   # alternatives are cryo_em or neutron
mm.set_scattering_table("n_gaussian")   # electron/neutron/wk1995/it199
```

Read in reconstruction symmetry ("ncs") and make an ncs object:

```
ncs_d7 = dm.get_ncs_spec("D7.ncs_spec")   # reconstruction symmetry fil
```

Make a map_model_manager

```
map_manager = mm_for_morphing.deep_copy()  # use a separate copy
model = model_for_morphing.deep_copy()    # model, also a separate copy
from iotbx.map_model_manager import map_model_manager # import manager
mmm = map_model_manager(map_manager = map_manager, model = model)   #
```

Make a map_model_manager with reconstruction symmetry

```
map_manager = mm_for_morphing.deep_copy()  # use a separate copy
model = model_for_morphing.deep_copy()    # model, also a separate copy
from iotbx.map_model_manager import map_model_manager # import manager
mmm_ncs = map_model_manager(map_manager = map_manager, model = model, #
 ncs_object = ncs_d7)   # reconstruction symmetry
```

Shift a model to match origin shift in a map_manager

```
model_1 = model.deep_copy()  # copy of a model
map_manager.shift_model_to_match_map(model_1)  # shift model_1 to match
```

Make a copy of a map_model_manager (deep copy where everything is separate from origin)

```
mmm_copy = mmm.deep_copy() # deep copy of a map-model manager
```

Set defaults in a map_model_manager

```
mmm.set_resolution(3)   # set default resolution to 3 A
mmm.set_experiment_type("xray")   # alternatives are cryo_em/neutron
```

Generate a model map using a map_model_manager

```
dm = DataManager()             # Initialize the DataManager and call it
dm.set_overwrite(True)
model = dm.get_model("structure_search_target.pdb")    # read in a mode
model_mmm = dm.get_map_model_manager(  # getting a map_model_manager
    model_file = "structure_search_target.pdb",)   # model to read in
model_mmm.generate_map(d_min=3)        #   generate map to resolution o
```

Converting from one type of manager to another

Get a model_building object from a map_model_manager

```
build = mmm.model_building()   # get model_building object
```

Get a map_model_manager from a model_building object

```
new_mmm = build.as_map_model_manager() # get map-model manager
```

Get a map_manager or model from a model_building object or map_model_manager

```
mm = build.map_manager()   # map_manager from model-building object
model = build.model()      # model from model-building object
mm = mmm.map_manager()     # map_manager from map-model manager
model = mmm.model()        # model from map-model manager
```

Working with map_managers

Tell DataManager to forget any previously-read map from a file, then read in map_manager from that file.

```
for file_name in dm.get_real_map_names():   # list of previously read m
  dm.remove_real_map(file_name)   # forget previous reads
mm = dm.get_real_map("boxed_ligand_map.ccp4")   # read in map file
```

Deep-copy map_manager

```
copy_of_mm = mm.deep_copy()  #  Separate copy of map_data
```

Shift origin of map_manager object to place start of map data that is present at (0, 0, 0). Note: This is done automatically by map_model_manager if one is used.

```
mm.shift_origin() # shift origin to (0,0,0)
```

Shift origin of map_manager object back where it was originally

```
mm1 = copy_of_mm.deep_copy()   # map manager to work with
mm1.shift_origin_to_match_original()  # put origin back where it was
```

List all methods available for a map_manager

```
dir(mm)  # list all the methods available for mm
```

Set resolution, experiment_type, scattering_table

```
mm.set_resolution(3)   # nominal resolution is 3 A;
mm.set_experiment_type("xray")  # alternatives are cryo_em/neutron
mm.set_scattering_table("n_gaussian")   # electron/neutron/wk1995/it199
```

Get origin shift in A (shift of position since map was read in)

```
shift_cart = mm.shift_cart() #   shift since start
```

Get position of original origin in grid units (same as how much to shift map to overlay original)

```
origin_shift_grid_units = mm.origin_shift_grid_units #  origin pos
```

Get gridding of working map (the part that is present)

```
n_real = mm.map_data().all()  #  gridding of working map
```

Get wrapping of map (wrapped means infinite, repeating with gridding of map. Map must be full size (not boxed)

```
wrapping = mm.wrapping()  #  wrapping of working map
```

Determine if map is full size, if map is zero-based, if map is consistent with wrapping

```
is_full_size = mm.is_full_size()  #  is map full size (not boxed)
is_zero_based = mm.origin_is_zero()  #  is zero-based
is_consistent_with_wrapping = mm.is_consistent_with_wrapping() # ok to
```

Determine if map_manager is similar to another one (same gridding, origin)

```
is_similar = mm.is_similar(mm1)  # map_managers are similar
```

Get map_data (flex array) from map_manager object

```
map_data = mm.map_data()  #  map_data
```

Customized copy of map_manager object with new data

```
some_map_data = mm.map_data() + 5 #  some map_data
new_mm = mm.customized_copy(map_data = some_map_data)  # new map manage
```

Set values of data in existing map_manager or zero out a map_manager

```
new_mm.set_map_data(map_data = map_data)  # new_mm data is now map_data
                # Note:  change new_mm and you are also changing map_da
new_mm.initialize_map_data()  # set values of map_data in new_mm to zer
```

Get full size version of map (zeros outside existing region)

```
full_size = mm.as_full_size_map()  # full size version padded with zero
```

Resample map on new grid (new grid must be multiple of existing if origin_shift_grid_units is not (0,0,0)

```
n_real = mm.map_data().all()  # current gridding
new_n_real = [n_real[0],n_real[1],n_real[2] * 2]  # double gridding alo
resampled = mm.resample_on_different_grid(new_n_real)  # resample
```

Get correlation to another map

```
cc = mm.cc_to_other_map_manager(new_mm) # map correlation to other map
```

Resolution filter, gaussian filter, binary filter:

```
low_res_map =  mm.resolution_filter(10)  # 10 A map
gaussian_blur = mm.gaussian_filter(1) # blur with 1 A Gaussian
binary_filtered = mm.binary_filter(threshold = 0.5)  # set value to 1 i
                                # average of 27 in box around this poin
```

Create binary mask around density, around edges, around atoms (does not apply mask, just creates it)

```
mm.create_mask_around_density(resolution = 3) # mask around density
mm.create_mask_around_edges(soft_mask_radius = 3) # mask around edges
model = dm.get_model("structure_search_target.pdb")  # get a model
mm.create_mask_around_atoms(model = model)  # radius=max(resolution,3)
```

Make the mask a soft mask

```
mm.soft_mask()  #  make the mask a soft mask. Default radius is resolut
```

Apply mask to working map or get mask as a map_manager

```
mm.apply_mask()  # apply the mask to working map (changes working map)
mask_as_map_manager = mm.get_mask_as_map_manager()   # mask as a map ma
```

Get map values at coordinates

```
sites_cart = model.get_sites_cart()  #  some coordinates
density_values = mm.density_at_sites_cart(sites_cart)  # density at coo
```

Normalize the map (set mean to zero and sd to 1)

```
mm.set_mean_zero_sd_one()  #  normalize map
```

Find reconstruction symmetry in map

```
mm.find_map_symmetry() #  find map symmetry
ncs_obj = mm.ncs_object()  # get the map symmetry as an ncs object
```

Find highest grid points in map and return as coordinates. Note: sites are relative to origin of map

```
sites_cart = mm.find_n_highest_grid_points_as_sites_cart() # highest po
```

Find n_atoms grid points in map in high density separated by dist_min

```
sites_cart = mm.trace_atoms_in_map(dist_min = 2, n_atoms = 31) # separa
                              # sites in density
```

Get map as Fourier coefficients and back. Note: origin stays at (0,0,0) throughout.

```
map_coeffs = mm.map_as_fourier_coefficients(d_min = 3)  # map coeffs to
new_mm = mm.fourier_coefficients_as_map_manager(map_coeffs) # new map_m
```

Working with map_data arrays (3D arrays)

Deep-copy map_data

```
copy_of_map_data = map_data.deep_copy()  #  Separate copy of map_data
```

Size of map_data

```
print ("Size of almost anything is anything.size(): %s" %(map_data.size
```

Pointer-only copy of map_data

```
pointer_to_map_data = map_data  #  just an alias to map_data
```

Get or set value of map_data at indices i,j,k

```
value = map_data[3,4,5] # get value of map_data at indices 3,4,5
map_data[3,4,5] = value # set value of map_data at indices 3,4,5
```

Select all map_data > 0. Note: selection array has same shape as map_data

```
sel = (map_data > 0)  #  array with True at all grid points with map_da
```

Set all selected values to value. Note: if map_data is a 3D array, sel must also be a 3D array

```
map_data.set_selected(sel, value) # set values where sel is True to val
```

Work with 3D map as a 1D array (Note: the data in the arrays are identical; change one and the other changes too

```
map_data_1d = map_data.as_1d()  # 1-D array containing the data in map_
```

Empty 1D array

```
from scitbx.array_family import flex   # import flex
new_array = flex.double(map_data_1d.size(), 0) # new array of zeroes
```

Convert a 1D array to 3D

```
acc = map_data.accessor()  # get the accessor for our 3d map data
new_array.reshape(acc)     # change new_array shape to match map_data
```

Working with 1-D versions of map_data arrays (flex.double)

Duplicate (deep copy, new numbers)

```
dc =  map_data_1d.deep_copy()    # copy of map_data_1d
```

Multiply, divide, add, check result

```
dc = dc * 3   # multiply all values in dc by 3 and make it the new dc
dc *= 3   # multiply all values in dc by 3 and make it the new dc
new_array = dc + (0.5 * dc ) * (dc)  # each value in new_array is
                                     #  value in dc + 0.5 *value in dc
new_array_2 = dc + 0.5 * flex.pow2(dc) # same result
assert new_array == new_array_2      # assert all values in 2 arrays ma
```

Get the minimum, maximum, and mean

```
m = map_data_1d.min_max_mean()
print ("Min, max, mean: (%.3f, %.3f, %.3f" %(m.min, m.max, m.mean))
```

Get the standard deviation

```
sd = map_data_1d.standard_deviation_of_the_sample()
```

Working with model objects

Get coordinates (orthogonal Angstrom units, flex array) from model object

```
sites_cart = model.get_sites_cart()  # coordinates in model. Order as P
```

Get coordinates (fractional, flex array) from model object

```
sites_frac = model.get_sites_frac()  # fractional coordinates
```

Convert fractional to orthogonal A and back

```
sites_frac = model.crystal_symmetry().unit_cell().fractionalize(sites_c
sites_cart = model.crystal_symmetry().unit_cell().orthogonalize(sites_f
```

Print out first 10 coordinates

```
for x,y,z in sites_cart[:10]:                  # go through sites_cart
  print ("SITE: (%.3f, %.3f, %.3f)" %(x,y,z))   # print them out
```

Select part of a model

```
model = dm.get_model("structure_search_target.pdb")   # read in model
part_of_model = model.apply_selection_string('resseq 6:25')  # select p
```

Separately create a selection (a flex.bool array) and then apply it

```
sel = model.selection('resseq 6:25')    # get a selection array
part_of_model = model.select(sel)       # apply the selection
```

Working with coordinate arrays (3-D, flex.vec3_double)

Get the minimum x,y,z or maximum x,y,z

```
min_xyz = sites_cart.min() # lowest value of x, of y, and of z (3 numbe
max_xyz = sites_cart.max() # highest value of x, of y, and of z (3 numb
```

Get the distances of a set of coordinates from the origin

```
distances = sites_cart.norms()  # distances from origin
```

Get the distances between 2 sets of coordinates

```
sites_cart_2 = sites_cart.reversed()  # just something different
diffs = (sites_cart - sites_cart_2)  # difference vectors
distances = diffs.norms()  # distances between sites_cart and sites_car
```

Get rms length of a set of coordinates

```
rms_length = sites_cart.rms_length()  # rms length
```

Get rms difference between two sets of coordinates

```
rms_difference = sites_cart.rms_difference(sites_cart_2)  # rms diff
```

Select all the sites where x > 10 and add 3 to their y values.

```
x = sites_cart.parts()[0]     # x values for each coordinate
y = sites_cart.parts()[1]     # y values
z = sites_cart.parts()[2]     # z values
sel = (x > 10)   #   all sites where x value is > 10

sel_y = y.select(sel)  # selected y values
new_y = y.set_selected(sel,sel_y + 3) # add 3 to selected y values
                              # Note: sel_y.size() == sel.count(True)
from scitbx.array_family import flex
new_sites_cart = flex.vec3_double(x, new_y, z) #  new sites with new y
```

Working with ncs (reconstruction symmetry) objects

Read in reconstruction symmetry ("ncs") and make an ncs object:

```
ncs_d7 = dm.get_ncs_spec("D7.ncs_spec")   # reconstruction symmetry fil
```

Print out ncs information

```
ncs_d7.display_all()  #  print out ncs
```

Get ncs from a model

```
ncs_d7_model = dm.get_model("D7.pdb")   # model with D7 symmetry
ncs_d7_model.search_for_ncs()  # find NCS in this model
ncs_spec_object = ncs_d7_model.get_ncs_obj().get_ncs_info_as_spec() # n
print (ncs_spec_object.display_all()) #   print out summary
```

Find ncs from a map and apply to a model using the model_building tool

```
mmm = dm.get_map_model_manager(model_file="D7.pdb",   # model
   map_files="D7.ccp4")    #map
build=mmm.model_building()    # get a model-building object
build.map_manager().find_map_symmetry(symmetry = "D7")  # find symmetry
build.map_manager().ncs_object().display_all() # summary
model_no_ncs = build.model().apply_selection_string("chain A") #
print (model_no_ncs)#  make sure we got one chain
model_ncs = build.apply_ncs(model = model_no_ncs) # apply ncs to model_
print (model_ncs)  # now we have 14 chains
```

Phenix high-level tools for model-building

Introduction to high-level tools for model-building in Phenix. Learn when to use automatic model-building and when to use the library modules in scripts.

Phenix has easy-to-use tools for carrying out basic model-building tasks such as building a model into density, fitting loops, fitting ligands, and rebuilding segments of a model. A set of high-level tools allow you to write simple Python scripts that can do all these things. You can use these scripts to interactively build or modify a model or to develop your own protocol for model-building or improvement. This interface to model building is designed for both developers and users of Phenix who want to script model-building tasks.

Here are model-building tasks that you can script:

- Build: build a model into a region of density
- Fit_ligand: fit a flexible ligand into density
- Fit_loop: fit a loop between to chain ends
- Replace_segment: replace a part of a chain
- Insert_loop: insert a fitted loop or segment into working model
- Combine_models: take the best parts of several models and combine them into a single model
- Morph: adjust a model to match density
- Sequence_from_map: align sequence or guess sequence from map and main-chain model
- Refine: carry out simple real-space refinement

For most of these tasks there are several methods you can use to carry out the task. You can try them all in parallel and choose the result that gives the best fit at the end.

All of these tools use high-level cctbx objects such as the map_model_manager to read, write, and manipulate the maps and models. For an introduction to these high-level cctbx objects see the cctbx overview with tutorials.

This part of the documentation describes how to use these model-building tools and provides simple scripts to illustrate the procedures.

The high-level model-building tools are designed for local model-building

The high-level model-building tools that are described here are intended to be used on a small part of a density map. You can use them on any size map, but the most efficient way to use them on a big map is to cut out a box of density for the part of the map that you are interested in and then to use these tools on that box of density. You can use the map_model_manager to conveniently cut out (box a part of a map suitable for model-building. You can also use the split_into_boxes and merge_boxes tools to do this. This will be demonstrated below as well.

When to use fully automatic model-building and when to use high-level tools for model-building

If you just want to use Phenix to build a model, the easiest thing to do is to use one of the fully automatic model-building tools, such as map_to_model (for cryo-EM) or autobuild (for crystallography). These tools carry out all the steps in model-building and give you a partial or complete model. You have some control over the process, but only a little.

On the other hand, if you have something specific that you want to do, such as to fix a particular segment of a model, or to fit a particular gap in your model, the high-level model-building tools are designed to allow you to carry out that specific task, examine the results, and choose what to do next.

Additionally, if you want to design your own procedure for model building or improvement, you can use the high-level tools to carry out individual steps in your procedure.

Read in a map and model

The very first thing we need to do for model building is to read in experimental data (maps, reflections) and a model. Here, we learn how to read in existing files from the Phenix regression directory.

When you have Phenix installed, you automatically get tutorial, testing, and demo data files that come with Phenix. We'll use some of those in this tutorial. Let's make a new directory to work in and put the files we need there. In a terminal window type:

```
phenix.setup_tutorial tutorial_name=model_building_scripting
```

This creates the directory model-building-scripting. Let's change our working directory to this directory:

```
cd model_building_scripting
```

It is pretty easy to set up the high-level model-building tools. As an example, let's read in a map and working model and get ready to fit a ligand into the map.

Type `phenix.python` to start up Python with the Phenix environment all set up:

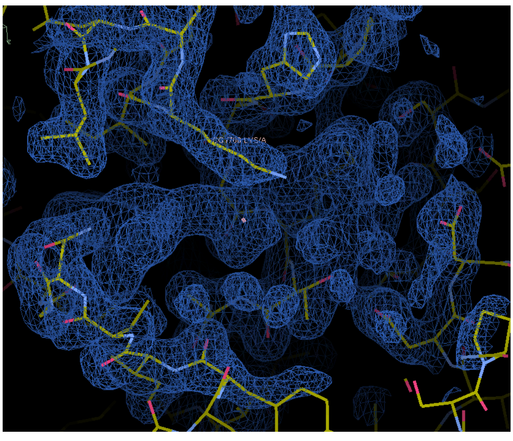
```
phenix.python
```

Now let's set up a DataManager that knows how to read and write files. Paste these commands in to your terminal window after starting up with `phenix.python` (Note: everything after the hash mark is a comment and is ignored, and also note that you are usually not allowed to have spaces before the start of a command):

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

The DataManager has many useful functions for reading and writing files. Here we are going to use a function that can read in a map and a model that we specify and that creates a map_model_manager from the map and model:

The map and model with unmodeled ligand density are in the in the model-building regression directory. Note the unmodelled density in the center of the screenshot.



Let's read the map and the model in and create a map_model_manager. Just cut and paste this command into your terminal window:

```
mmm = dm.get_map_model_manager(        # getting a map_model_manager
  model_file="boxed_ligand_map.pdb",   # model file
  map_files="boxed_ligand_map.ccp4")   # map file
```

We can summarize the map and model just by typing the word `mmm`:

```
mmm     # summarize the map_model_manager
```

This will produce a short summary of the model-manager (unit cell, space group, number of chains and residues, coordinate shift from its original location), and of the map_manager (original file name, unit cell, space group, gridding of the unit cell, part of the unit cell that is present, and coordinate shift from its original location):

```
Map_model_manager:

Model manager
Unit cell: (115.986, 115.986, 44.151, 90, 90, 120) Space group: P 6 (No
Chains: 1 Residues 247
Working coordinate shift (-6.95916000366212, -21.696513667819186, -27.1

map_manager:
Map manager (from $build_phenix/modules/phenix_regression/model_buildin
Unit cell: (115.986, 115.986, 44.151, 90, 90, 120) Space group: P 6 (No
Unit-cell grid: (250, 250, 96), (present: (66, 67, 35)), origin shift (
Working coordinate shift (-6.95916000366212, -21.696513667819186, -27.1
```

First let's tell the map_model_manager `mmm` about the resolution of the map. This is used later in fitting the ligand and in model-building.

```
mmm.set_resolution(3.0)    # working (nominal) resolution of map
```

Let's also tell the map_model_manager mmm about the experiment type (cryo_em or xray). This information can be used later in refinement.

```
mmm.set_experiment_type('xray')   # define experiment type
```

Now we are ready to work with this map_model_manager containing a map and model. Let's save a copy so we can come back to this place any time:

```
mmm_saved = mmm.deep_copy()   # save a copy of the map_model_manager
```

Home
How to install
Documentation
Table of Contents
Quick reference
Directory structure
Model building
Intro
Reading files
Fitting ligands
Build a model
Loop Fitting
Replace a segment
Morphing
Assign a sequence
Restraint jiffys
Programming

Building a model into density

Learn how to build a model into some density and how to automatically extend the newly built chain.

Setting up example data

First, let's set up the example data (described in more detail here):

Get the files from the Phenix regression directory and change into the new folder:

```
phenix.setup_tutorial tutorial_name=model_building_scripting
cd model_building_scripting
```

Type `phenix.python` to start up Python with the Phenix environment all set up:

```
phenix.python
```

Set up the high level objects, so we can start our task:

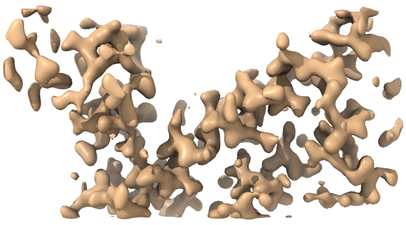
```
from iotbx.data_manager import DataManager    # Load in the DataManager
dm = DataManager()                # Initialize the DataManager and call it
dm.set_overwrite(True)            # Overwrite files with the same name
```

Building a model

Let's build a model into some density in a map using the build model-building tool. We can read in a map file from the model-building regression directory (no model this time):

```
mmm = dm.get_map_model_manager(  model_file = None,       # getting a ma
    map_files="short_model_box.ccp4")   # map file
```

Below is an image of the map:



Let's also tell the map_model_manager mmm about the resolution and experiment type (cryo_em or xray).

```
mmm.set_resolution(3.0)    # working (nominal) resolution of map
mmm.set_experiment_type('xray')   # define experiment type
```

Let's get a model_building object:

```
build = mmm.model_building()  # get a model-building object
```

...and set the number of processors to use (whatever number you have):

```
build.set_defaults(nproc = 4)   # use 4 processors
```

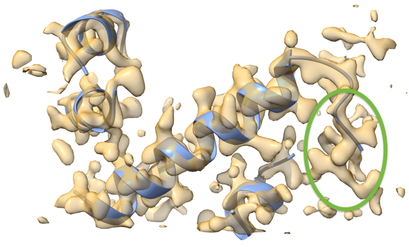
And now let's build into this map:

```
build.build()    # build a model into current map
```

We can write out the resulting model and the map that was used:

```
dm.write_model_file(build.model(), 'built_model.pdb')  # Working model
dm.write_real_map_file(build.map_manager(), 'build_map.ccp4')# Working
```

Have a look at this model and map in Coot or Chimera. You can see that three helices were built but there is still more to be built. In particular, the helix starting at residue 1 of chain U (green circle) appears to be suitable for extension in the N-terminal direction. Let's do this next.



Extending a chain

Let's extend one of the chains in the model we just built with the build tool. We can use the extend_reverse() method to create an extension in the N-terminal direction starting at residue 1 of chain U (it won't insert it yet).

```
build.extend_reverse(chain_id = "U", first_resno_after_extend = 1)  # e
```

We can get just the extension if we want with the command:

```
result = build.get_best_result()   # get the best extension result obje
insertion_model = result.model     # the extension as a model
```

Let's insert the extension:

```
build.insert_extension()   # insert best extension result object
```

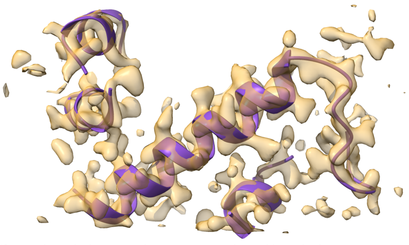
We can refine the model with the inserted extension:

```
build.refine()   # insert best extension result object
```

And write out the resulting model and the map that was used:

```
dm.write_model_file(build.model(), 'extended_model.pdb')  # Working mod
dm.write_real_map_file(build.map_manager(), 'extended_map.ccp4')# Worki
```

This results in more residues at the N-terminal end.



If some do not look very good you can remove them with a pair of commands like this:

```
new_model = build.model().apply_selection_string(    # apply a selectio
   'not (chain U and resseq 1:1)'  )  # remove residue 1 from chain U
build.set_model(new_model)            # replace existing model with new
```

You can write out your new model:

```
dm.write_model_file(build.model(), 'extended_model_trimmed.pdb')  # new
```

If you want, you can get a map_model_manager back from the build model-building object with:

```
new_mmm = build.as_map_model_manager()  # return a map_model_manager ob
```

In this example we extended backwards from the N-terminus of the model. You can also extend forwards using the command `build.extend`.

Also in these examples we built a protein model. You can build and extend an RNA or DNA model as well. You usually specify the chain type before building with:

```
build.set_defaults(chain_type = 'RNA')
```

or a similar command for DNA or PROTEIN (the default is PROTEIN). The chain extensions just keep the same chain type as the part that is already present.

Home
How to install
Documentation
Table of Contents
Quick reference
Directory structure
Model building
Intro
Reading files
Fitting ligands
Build a model
Loop Fitting
Replace a segment
Morphing
Assign a sequence
Restraint jiffys
Programming

Morphing a model

Learn how to adjust a model by morphing it to match a map. Refinement can further improve the model-to-map fit.

Setting up example data

First, let's set up the example data (described in more detail here):

Get the files from the Phenix regression directory and change into the new folder:

```
phenix.setup_tutorial tutorial_name=model-building-scripting
cd model-building-scripting
```

Type `phenix.python` to start up Python with the Phenix environment all set up:

```
phenix.python
```

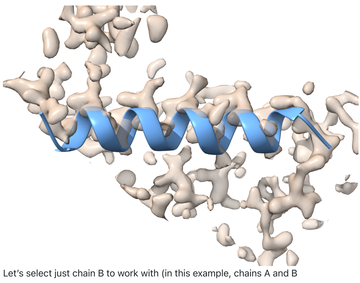
Set up the high level objects, so we can start our task:

```
from iotbx.data_manager import DataManager    # Load in the DataManager
dm = DataManager()            # Initialize the DataManager and call it
dm.set_overwrite(True)        # Overwrite files with the same name
mmm = dm.get_map_model_manager(       # getting a map_model_manager
  model_file="short_model_box_for_morph.pdb",   # model file
  map_files="short_model_box.ccp4")   # map file
```

Morphing a model

Let's adjust a model by morphing it to match the density in a map. Morphing basically distorts a model in a smooth way, so that locally the model does not change much, but parts of the model that are far apart along the chain can move relative to each other.

Our starting model has two chains (A and B). The B chain is offset from its correct position by about 1.5 Å.



Let's select just chain B to work with (in this example, chains A and B overlap and so it is best to work with just one).

```
chainB = mmm.model().apply_selection_string(    # apply a selection
  'chain B'  )  # chain B
```

And now we can replace the model in mmm with chainB (adding a model with model_id of 'model' replaces the existing model):

```
mmm.add_model_by_id(chainB, model_id = 'model')  # replace model in mmm
```

Let's fill in the resolution and experiment type and get a model_building object:

```
mmm.set_resolution(3)                  # resolution is 3 A
mmm.set_experiment_type('cryo_em')     # it is a cryo-EM map
build = mmm.model_building()           # get a model-building object
```

We can write out the model and look at it with Coot, comparing it with the map:

```
dm.write_model_file(build.model(), "short_model_box_for_morph.pdb") # m
dm.write_real_map_file(build.map_manager(), "short_model_box.ccp4") # m
```

Let's save the coordinates of the atoms in our model so we can compare them to their positions after morphing:

```
chainB = build.model().deep_copy()   # save chain B
starting_coords_chain_B = chainB.get_sites_cart()    # get coordinates
```

Now let's morph the model. We'll use the selection method 'by_segment' to choose how to split up the model when morphing. This choice means split up by chains, and also split any chains that are broken:

```
build.set_defaults(debug=True)      # debugging run
build.morph(default_selection_method='by_segment')   # morph model
```

We can see how much each chain has moved. The model in build has been updated so we get the coordinates of the atoms in this working model:

```
chainB_morphed = build.model()  # chain B
final_coords_chain_B = chainB_morphed.get_sites_cart()    # get coordin
```

The rmsd between starting and final chains A and B are then:

```
rms_B = final_coords_chain_B.rms_difference(starting_coords_chain_B)  #
```

You can print this out:

```
rms_B   # print out rms value
```

Which yields something like:

```
1.5801081738194553
```

We can compare the map-model correlations of the original and morphed models:

```
cc_before = mmm.map_model_cc(model = chainB)  # map-model cc for chain
cc_after = mmm.map_model_cc(model = chainB_morphed)  # map-model cc aft
cc_before, cc_after   # cc before and after morphing
```

Which yields something like:

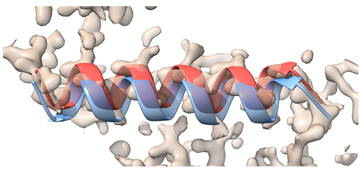
```
(0.25513080677867195, 0.8180704813656688)
```

Indicating that the map-model correlation is much higher after morphing.

Let's write out the morphed model to 'morphed_model.pdb' and compare it in Coot with the original in "short_model_box_for_morph.pdb":

```
dm.write_model_file(build.model(), 'morphed_model.pdb')  # write out mo
```

The morphed model (red) is moved compared to the initial model (transparent blue), and it now fits to the density.



Refining a model

Let's run a simple version of real-space refinement to improve the morphed model (see section above). This option is just a simplified version of phenix.real_space_refine that is suitable for quick improvement of a model while you are in the middle of model-building. For serious refinement you will want to use the standalone phenix.real_space_refine tool.

Let's refine our morphed model from the previous section. This is pretty easy. You type:

```
build.refine()     # refine working model against working map
```

and wait a minute or two and now you can write out the refined model:

```
dm.write_model_file(build.model(), 'refined_model.pdb')  # refined
```

We can get the map-model correlation now:

```
cc_refined = mmm.map_model_cc(model = build.model())  # refined cc
```

And print it out:

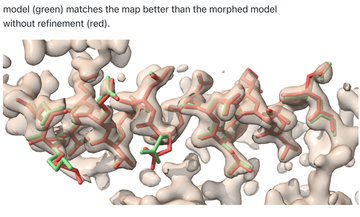
```
cc_refined  # print out refined cc
```

Which gives something like:

```
0.8237540703832011
```

Which is just a tiny bit better than the morphed model.

Have a look at morphed_model.pdb and refined_model.pdb along with the map in short_model_box.ccp4 and you will see that the refined model (green) matches the map better than the morphed model without refinement (red).



Fitting a loop

Learn how to fill in a gap in a model with the fit_loop tool.

Setting up example data

First, let's set up the example data (described in more detail here):

Get the files from the Phenix regression directory and change into the new folder:

```
phenix.setup_tutorial tutorial_name=model-building-scripting
cd model-building-scripting
```

Type `phenix.python` to start up Python with the Phenix environment all set up:

```
phenix.python
```

Set up the high level objects, so we can start our task:

```
from iotbx.data_manager import DataManager    # Load in the DataManager
dm = DataManager()             # Initialize the DataManager and call it
dm.set_overwrite(True)         # Overwrite files with the same name
mmm = dm.get_map_model_manager(        # getting a map_model_manager
  model_file="short_model_main_chain_box.pdb",   # model file
  map_files="short_model_box.ccp4")   # map file
```

Fitting a loop

We can fill in the resolution and experiment type and get a model_building object:

```
mmm.set_resolution(3)                 # resolution is 3 A
mmm.set_experiment_type('cryo_em')    # it is a cryo-EM map
build = mmm.model_building()          # get a model-building object
```

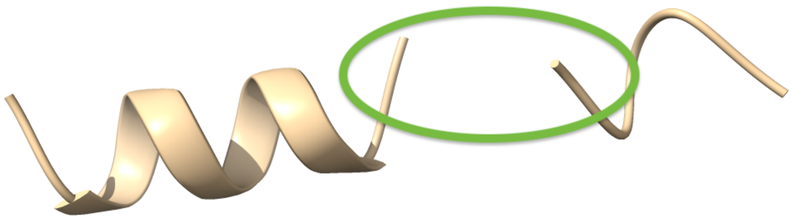
Let's remove some residues in this model to create a model that needs a loop (we don't have to do this, but it makes this example clearer):

```
new_model = build.model().apply_selection_string(    # apply a selectio
   ' not (chain A and resseq 516:519)'  )  # remove residues 516-519 ch
build.set_model(new_model)   # replace existing model with new_model
```

You can read about how to specify selections in the Phenix selection syntax documentation. The `apply_selection_string` method is part of a model object and it applies the selection you specify and returns a new model based on that selection.

Let's write out the working model:

```
dm.write_model_file(build.model(), 'model_without_loop.pdb')  # new mod
```



We can choose which methods to use in loop fitting. Let's use resolve loop fitting:

```
build.set_defaults(fit_loop_methods=['resolve'])    #choose just resolv
```

And let's fit a loop. If were are already residues where the loop is to go (between residues 515 and 520) they would be removed before the loop is built:

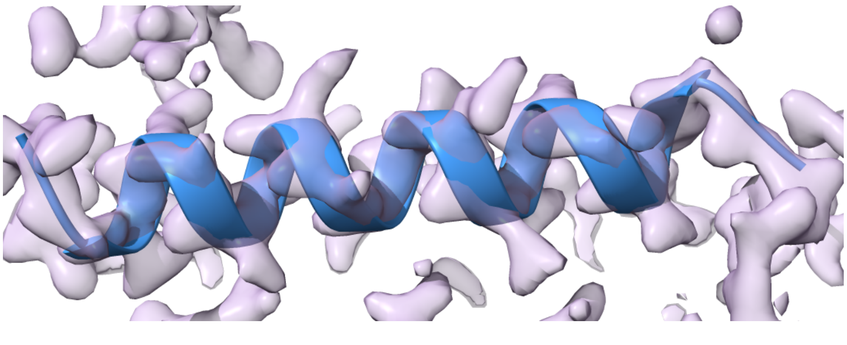
```
fit_loop_model=build.fit_loop(   #  fit a loop
     chain_id="A",               # chain ID
     loop_sequence = 'RVLD',        # Sequence of residues in loop
     last_resno_before_loop=515,    #  Residue number just before loop
     first_resno_after_loop=520,)    #  Residue number just after loop
```

Let's insert the loop:

```
build.insert_fitted_loop()    # insert the loop
```

Finally let's write out the new model:

```
dm.write_model_file(build.model(), 'fitted_loop_model.pdb')  # new mode
```



Replacing a segment of a model

Replace a part of a model by swapping in a segment from a similar model and learn how to combine models to create a composite model.

Setting up example data

First, let's set up the example data (described in more detail here):

Get the files from the Phenix regression directory and change into the new folder:

```
phenix.setup_tutorial tutorial_name=model-building-scripting
cd model-building-scripting
```

Type `phenix.python` to start up Python with the Phenix environment all set up:

```
phenix.python
```

Set up the high level objects, so we can start our task:

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

```
model = dm.get_model("structure_search_target.pdb")    # read in a mode
mmm = dm.get_map_model_manager(  # getting a map_model_manager
    model_file = "structure_search_target.pdb",)
```

Replacing a segment

Let's replace a part of a model by finding a similar model in the PDB and swapping in the segment from that model. This procedure is very similar to fitting a loop, but we are going to use a segment right from the PDB to fill in our loop instead of trying to build it from scratch.

The map_model_manager mmm knows how to generate a model-based map from a model. Let's create a map at a resolution of 3 Å and specify that this is a cryo-EM map:

```
mmm.generate_map(d_min=3)       #   generate map to resolution of 3 A
mmm.set_experiment_type('cryo_em')  # this is a cryo-EM map
```

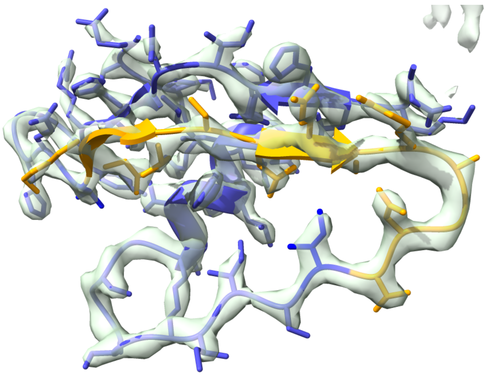
Now we are ready to get a model_building object and to specify that we want to use the 'structure_search' method to replace a segment. Just to see how it works, we'll turn on debugging (prints more output too) and set the thoroughness of the run to 'quick':

```
build = mmm.model_building()          # get a model-building object
build.set_defaults(                   # set some defaults
     thoroughness='quick',            #  quick run
     debug=True,)                     # turn on debugging output
```

We can save and write out the starting model before we modify it:

```
model_before_replace = build.model().deep_copy()   #  save model
dm.write_model_file(build.model(), 'model_before_replace.pdb')# startin
```

We will replace residues 22-34 (show in orange in the figure below).



We can now run the segment replacement:

```
replace_segment_model=build.replace_segment(    # replace a segment
     chain_id="A",                       #  chain to work on
     last_resno_before_replace=21,       # just before replace
     first_resno_after_replace=35,       # just after replace
     refine_cycles=1,)                   # cycles of refinement
```

And we can insert the new segment, get a new map_model_manager, and write out the results:

```
build.insert_fitted_loop()         # inserts loops or segments
mam=build.as_map_model_manager()   # get a map_model_manager to write o
mam.write_model('model_with_replacement.pdb') #  the model
mam.write_map('map_for_replacement.ccp4')  #   the map
```

Have a look at the starting model (model_before_replace.pdb), the model with a replacement segment (model_with_replacement.pdb), and the map (map_for_replacement.ccp4).

Combining Models

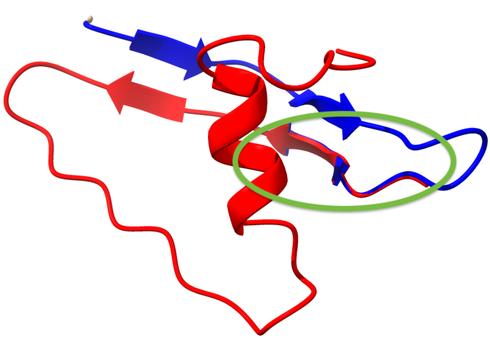
Let's combine several models and create a new model that has the best parts of each. We can use two models that we just worked with (build.model() and model_before_replace) along with the map we just used. The working model_building object contains model_with_replacement.pdb and the working map.

Just to see how it works, let's create two overlapping models: residues 6-25 from the starting model, model_before_replace, and residues 20 - 60 from the model with a segment replaced (in build.model()):

```
model_1 = model_before_replace.apply_selection_string(    # apply a sel
   'resseq 6:25'  )  # keep only residues 6 through 25
```

```
model_2 = build.model().apply_selection_string(    # apply a selection
   'resseq 20:60'  )  # keep only residues 20 through 60
```

The overlapping selections for the two models are shown below; model_1 is blue, model_2 is red, the overlappting segment is encircled.



We can make sure this worked:

```
model_1.get_hierarchy().overall_counts().n_residues  #  residues in mod
model_2.get_hierarchy().overall_counts().n_residues  #  residues in mod
```

Now we can combine the overlapping models with:

```
new_model = build.combine_models(          # combine models
    model_list = [model_1, model_2]) # models to combine
```

We can write out the new combined model with:

```
dm.write_model_file(new_model, 'combined_model.pdb')   # write out new
```

Have a look at model_before_replace.pdb, model_with_replacement.pdb, and combined_model.pdb. The combined model has residue 20 from model_1 and residue 30 from model_2.

Assign a sequence to a model

Learn how to to assign a sequence to a model or to guess the sequence from the density.

Setting up example data

First, let's set up the example data (described in more detail here):

Get the files from the Phenix regression directory and change into the new folder:

```
phenix.setup_tutorial tutorial_name=model-building-scripting
cd model-building-scripting
```

Type `phenix.python` to start up Python with the Phenix environment all set up:

```
phenix.python
```

Set up the high level objects, so we can start our task:

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

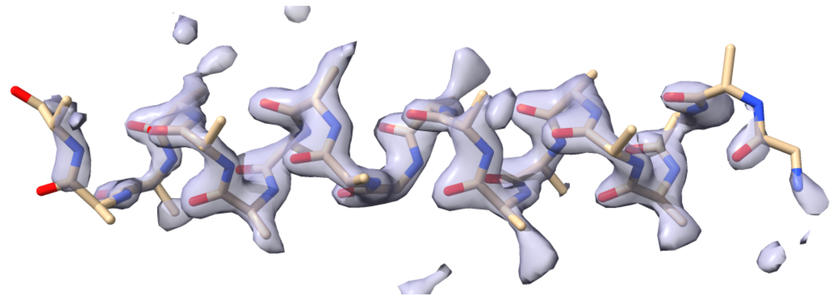
Assigning a sequence

The model_building object has a tool that examines the density in a map at the positions of each side-chain in a model and tries to figure out what side chain belongs there. This can be used to assign a sequence to a model or to guess the sequence from the density. Let's use it to assign a sequence to a model.

Use the DataManager to read in a model and map from our working directory:

```
mmm = dm.get_map_model_manager(        # getting a map_model_manager
  model_file="short_model_main_chain_box.pdb",   # model file
  map_files="short_model_box.ccp4")   # map file
```

The model consists of alanine residues:



We need one extra file this time, a sequence file. Let's read our sequence as an object from seq.dat:

```
seq_object=dm.get_sequence("seq.dat")     # read sequence object from s
```

Our sequence as text is available from the sequence object:

```
sequence_as_text = seq_object[0].sequence         # get simple text seq
```

We can set the resolution and experiment type and get a model_building object:

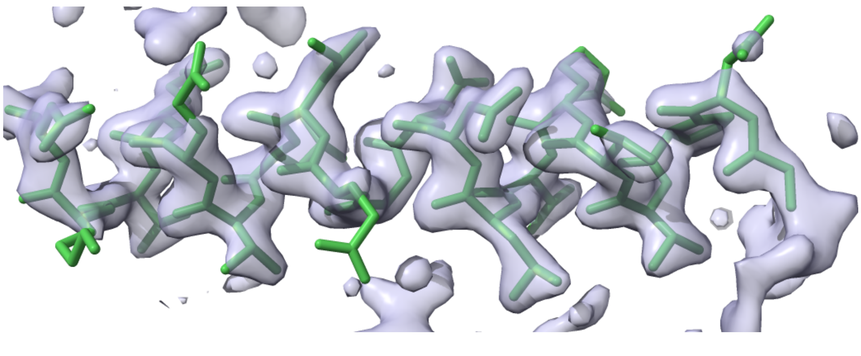
```
mmm.set_resolution(3.0)    # working (nominal) resolution of map
mmm.set_experiment_type('xray')   # define experiment type
build = mmm.model_building()  # get a model-building object
```

Now we are all set. We can just say:

```
build.sequence_from_map(sequence=sequence_as_text)   # get sequence fro
```

...and our working model will be replaced by a model with sequence assigned (if we are lucky). We can write out the new model and compare it with the original in Coot or Chimera along with the map in short_model_box.ccp4:

```
dm.write_model_file(build.model(), 'sequence_assigned.pdb') # write out
```



Restraint jiffys

This page summarizes some helpful command line tools to find and display restraints.

Finding ligand related CIF files

Phenix has three databases of CIF files related to ligands:

- Chemical Components
- Monomer Library
- Geostandard (GeoStd)

The first, Chemical Components (CC), is obtained from the Protein Data Bank (PDB). It contains information about all entities in the PDB including protein, nucleic acids and ligands. The information includes bonding but no other internal coordinates, so these are not restraints files. The second, Monomer Library, is a curated copy of the Monomer Library containing restraints for many common ligands. The third, GeoStd, contains all protein entities (amino acids, modified amino acids), all glycans, many nucleic acids and ligands. This is a curated set of restraints that is maintained as the "best" possible.

In some cases, restraints are be required to run a tool, for example when we want to create the `geometry_restraints_manager()`. It is then useful to determine whether there is a restraints file in the Phenix distribution and if so where. Otherwise, it would be necessary to get restraints, for example with `ReadySet`. The following command returns the contents of the restraints file and the path for the molecule `hoh` (water molecule):

```
elbow.where_is_that_cif_file hoh
```

The command `elbow.where_is_that_cif_file` is followed by the 3-letter code of a ligand or component. The output is:

```
Looking for ligand code hoh

------------------------------------------------------------------
  CIF file
    /Users/nigel/phenix/conda/modules/chem_data/geostd/h/data_HOH
------------------------------------------------------------------

#
data_comp_list
loop_
_chem_comp.id
_chem_comp.three_letter_code
_chem_comp.name
_chem_comp.group
_chem_comp.number_atoms_all
_chem_comp.number_atoms_nh
_chem_comp.desc_level
_chem_comp.initial_date
_chem_comp.modified_date
_chem_comp.source
HOH   .   'water     ' solvent   3   1  .   2012-10-17  2013-01-1
:
```

According to the output, the restraints file for hoh is in GeoStd.

If the file is not in the GeoStd, the code looks in the Monomer Library and returns it it is found. This scenario occurs for example in the case of GTP. If the component is not found in the Monomer Library, the output displays the contents of the Chemical Components file. An example for this is the component 000.

```
elbow.where_is_that_cif_file 000
```

The output for this command is:

```
Looking for ligand code 000

------------------------------------------------------------------
  CIF file
    /Users/nigel/phenix/conda/modules/chem_data/chemical_componer
------------------------------------------------------------------

data_000
#
_chem_comp.id                                    000
_chem_comp.name                                  "methyl hydroger
_chem_comp.type                                  NON-POLYMER
_chem_comp.pdbx_type                             ATOMP
_chem_comp.formula                               "C2 H4 O3"
_chem_comp.mon_nstd_parent_comp_id               ?
_chem_comp.pdbx_synonyms                         ?
_chem_comp.pdbx_formal_charge                    0
_chem_comp.pdbx_initial_date                     2010-04-27
_chem_comp.pdbx_modified_date                    2011-06-04
_chem_comp.pdbx_ambiguous_flag                   N
_chem_comp.pdbx_release_status                   REL
_chem_comp.pdbx_replaced_by                      ?
```

Options

Should you wish to see the Chemical Components file for an entity in the restraints databases use the option `cc`.

```
elbow.where_is_that_cif_file hoh cc
```

The output of this command is:

```
Looking for ligand code hoh

------------------------------------------------------------------
  CIF file
    /Users/nigel/phenix/conda/modules/chem_data/chemical_componer
------------------------------------------------------------------

data_HOH
#
_chem_comp.id                                    HOH
_chem_comp.name                                  WATER
_chem_comp.type                                  NON-POLYMER
_chem_comp.pdbx_type                             HETAS
_chem_comp.formula                               "H2 O"
_chem_comp.mon_nstd_parent_comp_id               ?
_chem_comp.pdbx_synonyms                         ?
_chem_comp.pdbx_formal_charge                    0
_chem_comp.pdbx_initial_date                     1999-07-08
_chem_comp.pdbx_modified_date                    2011-06-04
_chem_comp.pdbx_ambiguous_flag                   N
_chem_comp.pdbx_release_status                   REL
_chem_comp.pdbx_replaced_by                      ?
```

Note the different locations compared to the first example. Also, this cif file only has coordinates.

Reading .geo files

Example

Perform these commands to follow along.

```
phenix.fetch_pdb 1yjp --mtz
phenix.refine 1yjp.{pdb,mtz} write_final_geo=True
```

These commands will download a model from the PDB and then refine it.

Get RMSD/Z

To get the RMSD and RMSZ values for the initial model broken into all residues, residues without alt. loc. and residues with alt. loc.

```
elbow.refine_geo_display 1yjp_refine_001.geo
```

The output of the command is:

```
RMS results for model

RMS(D/Z)
             d-all          d-none         d-alt          z-all
  bond       0.015(   59)   0.015(   59)   0.000(    0)   0.932(   59)
  angle      1.354(   79)   1.354(   79)   0.000(    0)   0.825(   79)
  dihedral  18.725(   34)  18.725(   34)   0.000(    0)   1.170(   34)
```

The overall bond RMSD is 0.015Å while the overall bond RMSZ is 0.93. We can do the same for the .geo obtained after refinement:

```
elbow.refine_geo_display 1yjp_refine_001_final.geo
```

```
RMS results for model

RMS(D/Z)
             d-all          d-none         d-alt          z-all
  bond       0.006(   59)   0.006(   59)   0.000(    0)   0.315(   59)
  angle      0.844(   79)   0.844(   79)   0.000(    0)   0.447(   79)
  dihedral   3.051(   34)   3.051(   34)   0.000(    0)   0.645(   34)
```

The output shows that after refinement, the RMSD and RMSZ are reduced to 0.06Å and 0.32, respectively.

Displaying restraints of a subset of atoms

The output for a geo file of a large molecule can be overwhelming. If it is desired to look at restraints of a particular residue (such as a ligand), it is handy to apply the command on a subset. The syntax does not follow the standard Phenix atom selection syntax, but it is straightforward. For example, the selection string `GLY` will show restraints for all glycine residues in the model. The output also shows the RMSD/Z for the selection. The selection is therefore particular useful to display information about ligands.

```
elbow.refine_geo_display 1yjp_refine_001_final.geo GLY
```

The output is:

```
RMS results for selection

RMS(D/Z)
             d-all          d-none         d-alt          z-all
  bond       0.004(    4)   0.004(    4)   0.000(    0)   0.235(
  angle      1.901(    5)   1.901(    5)   0.000(    0)   0.679(
  dihedral   4.330(    1)   4.330(    1)   0.000(    0)   0.866(

----- bonds ----- bonds ----- bonds ----- bonds ----- bonds -----
   1 " CA  GLY A   1 " - " C   GLY A   1 " [ 1.516, 1.508,  0.008
   2 " N   GLY A   1 " - " CA  GLY A   1 " [ 1.451, 1.453, -0.002
   3 " C   GLY A   1 " - " N   ASN A   2 " [ 1.330, 1.329,  0.001
   4 " C   GLY A   1 " - " O   GLY A   1 " [ 1.231, 1.230,  0.001
---- angles ---- angles ---- angles ---- angles ---- angles ----
   1 " N   GLY A   1 " - " CA  GLY A   1 " - " C   GLY A   1 " [
   2 " CA  GLY A   1 " - " C   GLY A   1 " - " N   ASN A   2 " [
   3 " CA  GLY A   1 " - " C   GLY A   1 " - " O   GLY A   1 " [
   4 " C   GLY A   1 " - " N   ASN A   2 " - " CA  ASN A   2 " [
   5 " O   GLY A   1 " - " C   GLY A   1 " - " N   ASN A   2 " [
---- dihedrals ---- dihedrals ---- dihedrals ---- dihedrals ----
   1 " CA  GLY A   1 " - " C   GLY A   1 " - " N   ASN A   2 " -
---- chirals ---- chirals ---- chirals ---- chirals ---- chirals
```

To get just the information for the ASN with residue number "3" use

```
elbow.refine_geo_display 1yjp_refine_001_final.geo “ASN 3”
```

```
RMS results for selection

RMS(D/Z)
             d-all          d-none         d-alt          z-all
  bond       0.004(    9)   0.004(    9)   0.000(    0)   0.248(
  angle      0.664(   14)   0.664(   14)   0.000(    0)   0.419(
  dihedral   5.645(    6)   5.645(    6)   0.000(    0)   1.139(

----- bonds ----- bonds ----- bonds ----- bonds ----- bonds ----- b
   1 " C   ASN A   3 " - " N   GLN A   4 " [ 1.331, 1.337, -0.006,
   2 " CA  ASN A   3 " - " CB  ASN A   3 " [ 1.532, 1.526,  0.006,
   3 " CG  ASN A   3 " - " ND2 ASN A   3 " [ 1.328, 1.323,  0.005,
   4 " CG  ASN A   3 " - " OD1 ASN A   3 " [ 1.231, 1.235, -0.004,
   5 " CB  ASN A   3 " - " CG  ASN A   3 " [ 1.516, 1.521, -0.005,
   6 " N   ASN A   3 " - " CA  ASN A   3 " [ 1.456, 1.454,  0.002,
   7 " C   ASN A   3 " - " O   ASN A   3 " [ 1.236, 1.237, -0.002,
   8 " CA  ASN A   3 " - " C   ASN A   3 " [ 1.521, 1.523, -0.002,
   9 " C   ASN A   2 " - " N   ASN A   3 " [ 1.331, 1.329,  0.001,
…
```

Two “selections” will show those restraints that involve both. Note that the selection `4` will select the residue number (resseq).

```
elbow.refine_geo_display 1yjp_refine_001_final.geo “ASN 3” 4
```

```
----- bonds ----- bonds ----- bonds ----- bonds ----- bonds ----- bonds
   1 " C   ASN A   3 " - " N   GLN A   4 " [ 1.331, 1.337, -0.006, 1.29
---- angles ---- angles ---- angles ---- angles ---- angles ---- angles
   1 " C   ASN A   3 " - " N   GLN A   4 " - " CA  GLN A   4 " [ 122.77
   2 " O   ASN A   3 " - " C   ASN A   3 " - " N   GLN A   4 " [ 123.30
   3 " CA  ASN A   3 " - " C   ASN A   3 " - " N   GLN A   4 " [ 116.29
---- dihedrals ---- dihedrals ---- dihedrals ---- dihedrals ---- dihedr
   1 " CA  ASN A   3 " - " C   ASN A   3 " - " N   GLN A   4 " - " CA
```

Advanced “selections” of subsets

To get the information for all connections use the exclamation mark `!`.

```
elbow.refine_geo_display 1yjp_refine_001_final.geo “ASN 3” !
```

```
 ----- bonds ----- bonds ----- bonds ----- bonds ----- bonds ----- bon
   1 " C   ASN A   3 " - " N   GLN A   4 " [ 1.331, 1.337, -0.006, 1.29
   2 " C   ASN A   2 " - " N   ASN A   3 " [ 1.331, 1.329,  0.001, 1.34
---- angles ---- angles ---- angles ---- angles ---- angles ---- angles
   1 " C   ASN A   3 " - " N   GLN A   4 " - " CA  GLN A   4 " [ 122.77
   2 " O   ASN A   2 " - " C   ASN A   2 " - " N   ASN A   3 " [ 122.86
   3 " O   ASN A   3 " - " C   ASN A   3 " - " N   GLN A   4 " [ 123.30
   4 " C   ASN A   2 " - " N   ASN A   3 " - " CA  ASN A   3 " [ 122.99
   5 " CA  ASN A   2 " - " C   ASN A   2 " - " N   ASN A   3 " [ 115.51
   6 " CA  ASN A   3 " - " C   ASN A   3 " - " N   GLN A   4 " [ 116.29
---- dihedrals ---- dihedrals ---- dihedrals ---- dihedrals ---- dihedr
   1 " CA  ASN A   3 " - " C   ASN A   3 " - " N   GLN A   4 " - " CA
   2 " CA  ASN A   2 " - " C   ASN A   2 " - " N   ASN A   3 " - " CA
```

To see the worst restraints use `top=3`.

```
elbow.refine_geo_display 1yjp_refine_001_final.geo “ASN 3” top=3
```

```
----- bonds ----- bonds ----- bonds ----- bonds ----- bonds ----- bonds
   1 " CG  GLN A   4 " - " CD  GLN A   4 " [ 1.516, 1.537, -0.021, 2.50
   2 " CA  ASN A   6 " - " CB  ASN A   6 " [ 1.527, 1.513,  0.014, 1.75
   3 " CB  ASN A   6 " - " CG  ASN A   6 " [ 1.516, 1.497,  0.019, 2.50
---- angles ---- angles ---- angles ---- angles ---- angles ---- angles
   1 " N   GLY A   1 " - " CA  GLY A   1 " - " C   GLY A   1 " [ 113.30
   2 " N   ASN A   3 " - " CA  ASN A   3 " - " C   ASN A   3 " [ 108.90
   3 " CB  GLN A   4 " - " CG  GLN A   4 " - " CD  GLN A   4 " [ 112.60
---- dihedrals ---- dihedrals ---- dihedrals ---- dihedrals ---- dihedr
   1 " CA  ASN A   3 " - " C   ASN A   3 " - " N   GLN A   4 " - " CA
   2 " CA  ASN A   2 " - " C   ASN A   2 " - " N   ASN A   3 " - " CA
   3 " CA  GLN A   4 " - " C   GLN A   4 " - " N   GLN A   5 " - " CA
---- chirals ---- chirals ---- chirals ---- chirals ---- chirals ---- c
   1 " CA  ASN A   6 " - " N   ASN A   6 " - " C   ASN A   6 " - " CB
   2 " CA  GLN A   5 " - " N   GLN A   5 " - " C   GLN A   5 " - " CB
   3 " CA  TYR A   7 " - " N   TYR A   7 " - " C   TYR A   7 " - " CB
```

Hydrogen atoms

You can use the option `only_hydrogens=True` to see just the information of the hydrogen atoms. Note that RMSD/Z is zero for the bonds (in our example, H atoms are placed at ideal distances).

```
phenix.reduce 1yjp.pdb > 1yjp_h.pdb
phenix.pdb_interpretation 1yjp_h.pdb write_geo=True
elbow.refine_geo_display 1yjp_h.pdb.geo "ASN 3" only_hydrogens=True
```

The above commands will place H atoms with reduce, then the new file will be run by `pdb_interpretation` and finally, the restraints information is displayed for the H atoms only.

```
RMS results for selection

RMS(D/Z)
             d-all          d-none         d-alt          z-all
  bond       0.000(    6)   0.000(    6)   0.000(    0)   0.000(
  angle      1.575(   14)   1.575(   14)   0.000(    0)   0.525(
  dihedral   0.900(    1)   0.900(    1)   0.000(    0)   0.030(

----- bonds ----- bonds ----- bonds ----- bonds ----- bonds -----
   1 " CA  ASN A   3 " - " HA  ASN A   3 " [ 0.970, 0.970,  0.000
   2 " ND2 ASN A   3 " - "HD21 ASN A   3 " [ 0.860, 0.860, -0.000
   3 " N   ASN A   3 " - " H   ASN A   3 " [ 0.860, 0.860, -0.000
   4 " CB  ASN A   3 " - " HB2 ASN A   3 " [ 0.970, 0.970, -0.000
   5 " ND2 ASN A   3 " - "HD22 ASN A   3 " [ 0.860, 0.860,  0.000
   6 " CB  ASN A   3 " - " HB3 ASN A   3 " [ 0.970, 0.970, -0.000
---- angles ---- angles ---- angles ---- angles ---- angles ----
   1 " N   ASN A   3 " - " CA  ASN A   3 " - " HA  ASN A   3 " [
   2 " HB2 ASN A   3 " - " CB  ASN A   3 " - " HB3 ASN A   3 " [
   3 " C   ASN A   3 " - " CA  ASN A   3 " - " HA  ASN A   3 " [
   4 " CB  ASN A   3 " - " CA  ASN A   3 " - " HA  ASN A   3 " [
   5 " C   ASN A   3 " - " N   GLN A   4 " - " H   GLN A   4 " [
   6 " CG  ASN A   3 " - " CB  ASN A   3 " - " HB2 ASN A   3 " [
```

Other features

To see information about non-bonded atoms use `show_nonbond=True`.

```
elbow.refine_geo_display 1yjp_refine_001_final.geo “ASN 3” show_nonbond
```

```
- nonbonded - nonbonded - nonbonded - nonbonded - nonbonded - nonbonded
  1 " OD1 ASN A   3 " - " ND2 ASN A   3 " ['2.730', '2.520', 'x,y+1,z'
  2 " ND2 ASN A   3 " - " OD1 ASN A   3 " ['2.730', '2.520', 'x,y-1,z'
  3 " O   ASN A   2 " - " CA  ASN A   3 " ['2.801', '2.776']
  4 " O   ASN A   2 " - " N   ASN A   3 " ['2.838', '2.520', 'x,y+1,z'
  5 " N   ASN A   3 " - " O   ASN A   2 " ['2.838', '2.520', 'x,y-1,z'
  6 " O   ASN A   3 " - " CA  GLN A   4 " ['2.844', '2.776']
  7 " ND2 ASN A   3 " - " OE1 GLN A   5 " ['2.845', '2.520']
  8 " N   ASN A   3 " - " O   ASN A   3 " ['2.904', '2.496']
```

Home
How to install
Documentation
Programming
Program template
Runsnake
Line_profiler
C++ fast compilation

Install and use Runsnake

It is a good idea to profile code to speed up runtime.

Install and set up RunSnakeRun

First, we create a separate conda environment for runsnake. We need to use Intel architecture since wxpython 4.0.7 is not available for arm, and runsnake does not work with newer wxpython (https://github.com/mcfletch/runsnakerun/issues/3), Py3.8 version is required for older wxpython.

1. Make sure you have conda installed

Install conda if necessary. For Apple Silicon, you need native conda, which you can get here: https://github.com/conda-forge/miniforge/releases

You can check if you have conda installed by simply typing `conda` in the terminal.
You can get the location of the `base_environment` for conda (not to be confused with the `conda_base` directory of your phenix installation!) with this command:

```
conda info | grep base
```

The output is something like this:

```
       base environment : /Users/dcliebschner/Software/miniforge3  (wri
```

So the base_environment diretory is `/Users/dcliebschner/Software/miniforge3` .

2. Install Runsnake

Execute these commands to install runsnake. You can do it anywhere.`snake_x86` is the name of the environment, so you can choose another name if you prefer.

```
CONDA_SUBDIR=osx-64 conda create -n snake_x86 python=3.8
conda activate snake_x86
CONDA_SUBDIR=osx-64 conda install wxpython=4.0.7
pip install runsnakerun
```

Note: For x86_64 conda architecture one should be able to omit `CONDA_SUBDIR=osx-64` part of the commands and get the same result.

3. Set up an alias

It is a good idea to use an alias. `your_base_environment_here` is the location of the base_environment for conda (see step 1 above).

```
alias runsnake='your_base_environment_here/envs/snake_x86/bin/pythonw y
```

Profiling a script

```
libtbx.python -m cProfile -o prof.out myscript.py my_model.pdb
```

`myscript.py` is the code that is being profiled. This script may or may not have inputs. In this example the script has the input `my_model.pdb`.

The alias runs the `runsnake.py` script that is installed in phenix/base.

Profiling with the ProgramTemplate

Use the `--profile` command line flag. If the ProgramTemplate is called by the "run_program" function in `iotbx/cli_parser.py` (i.e. you have the basic boilerplate code in your file in the command_line directory), you will get a `profile.out` file.

```
mmtbx.hydrogenate 1kyc.pdb --profile
```

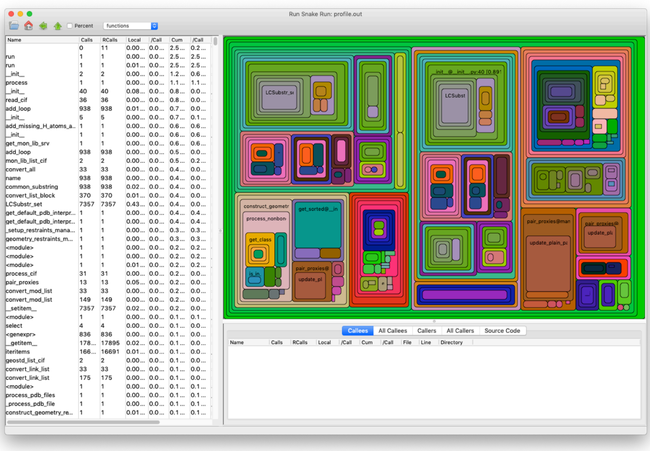
The file is hardcoded to "profile.out" so that the parser does not get confused (e.g. there is a default filename when no argument is provided, but "--profile model.pdb" should not dump the stats into model.pdb).
Note that if you customize how you run your program, then you would need to handle the profiling manually.

Looking at the profiling output

After profiling you can view the file with (if you set up an alias, see above):

```
runsnake prof.out
```

In the output of the above command, you will see many squares of different size. The size is proportional to runtime. In the square you can find what it is exactly (function name). When you hover with mouse over a square, the same functions are highlighted as well, so it is easy to see if something is run from different places.



You can also use a flag, so every command outputs a runsnake profile:

```
setenv LIBTBX__CPROFILE_FLAG__ 1
```

C++ fast partial compilation

Learn how to quickly debug and compile C++ code.

Often when editing C++ code in cctbx_project one needs to test and fix errors multiple times. It is not efficient to use bootstrap for compiling because it is very slow. There are two ways to that make compilation faster:

Method 1

1. Go to build directory.
2. run `libtbx.scons`. This command is equal to the `build` flag for bootstrap.
   This command will check all the directories and compile and link modified files only. If there is a compilation error, the build will fail, so you'll need to debug.
3. The compilation commands are printed to stdout, so they can be copy-pasted and executed when editing the C++ file(s).

Method 2

1. Go to build directory.
2. run `libtbx.configure --only iotbx`.
   This command tells libtbx.scons to build only a specific module, e.g. iotbx
3. Then libtbx.scons will run only iotbx build. It is important to switch the libtbx.scons behavior back when debugging is finished. To do this, run
   `libtbx.configure solve_resolve phaser phenix chem_data`