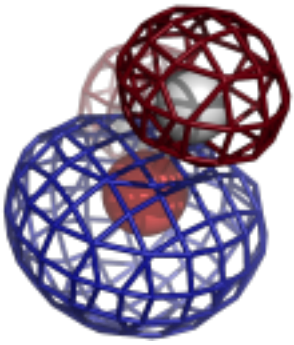
VOLUME NINE



COMPUTATIONAL CRYSTALLOGRAPHY NEWSLETTER

JULY MMXVIII

1.14, CABLAM, VICINAL SS

Table of Contents

- Phenix News 45
- Expert Advice
  - Fitting tips #16 – Vicinal disulfides: Have you seen one of these strange gems? 46
- Short Communications
  - CaBLAM: A C-Alpha Based Low-resolution Annotation Method for secondary structure and validation 51
  - Tools for interpreting cryo-EM maps using models from the PDB 58
- Articles
  - Using the New Program Template 62

Editor

Nigel W. Moriarty, NWMoriarty@LBL.Gov

Phenix News

Announcements

Phenix 1.14 release

The Phenix developers are pleased to announce that version 1.14 of Phenix is now available (build 1.14-3260). Binary installers for Linux, Mac OSX, and Windows platforms are available at the download site.[1] Highlights for this version include new tools and feature enhancements:

Reorganization, updates and addition of cryo-EM tools

- phenix.mtriage - assess map and model quality
- phenix.auto_sharpen - map sharpening
- phenix.map_symmetry - identify symmetry in maps
- phenix.map_box - cut out unique parts of maps
- phenix.combine_focused_maps - combine different maps
- phenix.dock_in_map - automatically place an atomic model into a map
- phenix.map_to_model - automatically build atomic model from a map
- phenix.sequence_from_map - identify sequence from a map
- phenix.real_space_refine - improved refinement of models
- phenix.validation_cryoem - separate tool for comprehensive validation of models and maps
- phenix.cablam_idealization - Tool to automatically fix Cablam outlier

eLBOW

- better support for metals and metal clusters
- added plugin for QM package Orca

[1] http://phenix-online.org/download/

Phaser-2.8.2

- bugfixes
- Phassade substructure search when starting from seed substructure
- fix crash in MR_ATOM
- problems with cumulative intensity distribution for extremely weak data
- improve computation and presentation of data information content

Performance improvements

- NCS search
- Generation of secondary structure restraints
- Clashscore calculation
- AmberPrep is more robust
- Restraints for ARG improved

New Phenix video tutorials

GUI

- New section in main window for cryo-EM tools
- Separate validation GUI for cryo-EM structures
  - Added phenix.map_symmetry
  - Added phenix.dock_in_map
  - Added phenix.map_to_structure_factors
  - Added phenix.combine_focused_maps
  - Added phenix.sequence_from_map

phenix.ligand_identification:

- Added an option to generate ligand library based on sequence and structural homologs of the input pdb model.

Amber

A new command, amber.h_bond_information, has been added to use the Amber H-bond detection code. Once AmberTools is installed, the command will provide the total number of H-bonds into a protonated model.

Expert advice

Fitting Tip #16 – Vicinal disulfides: Have you seen one of these strange gems?

Jane Richardson and Lizbeth Videau

Duke University

What is a vicinal SS?

A vicinal SS is a disulfide between two sequence-adjacent cysteine residues, with the rather startling appearance shown in figure 1 for a 1.75Å-resolution example in 1wd3. Early calculations implied that a vicinal SS could form only with a *cis* peptide (Chandrasekharan 1969) with the first two protein-crystal examples were both fit as *cis*, although undeposited. When both of those cases were shown to actually be *trans*, most studies including the one review (Carugo

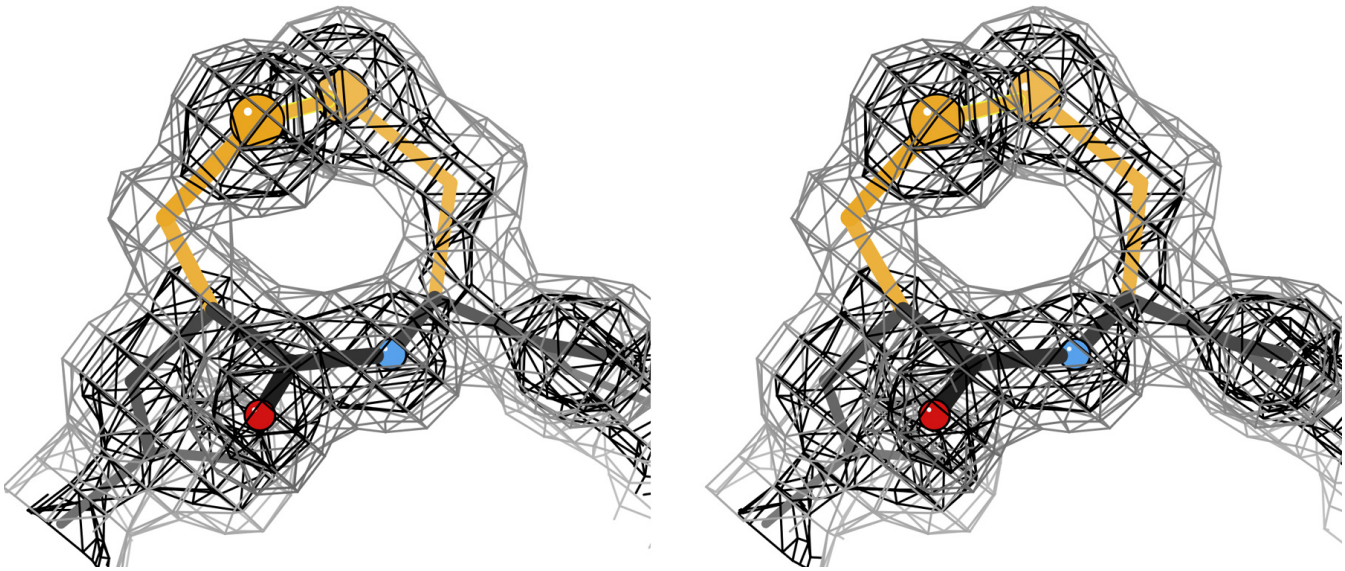


Figure 1: Stereo of a *cis* vicinal disulfide in C-conformation, in its clear $2F_o$-F_c electron density at 1.75Å (1.2σ contours in gray, 3σ in black) Cys 177 from the 1wd3 arabino-furanosidase (Miyanaga 2004).

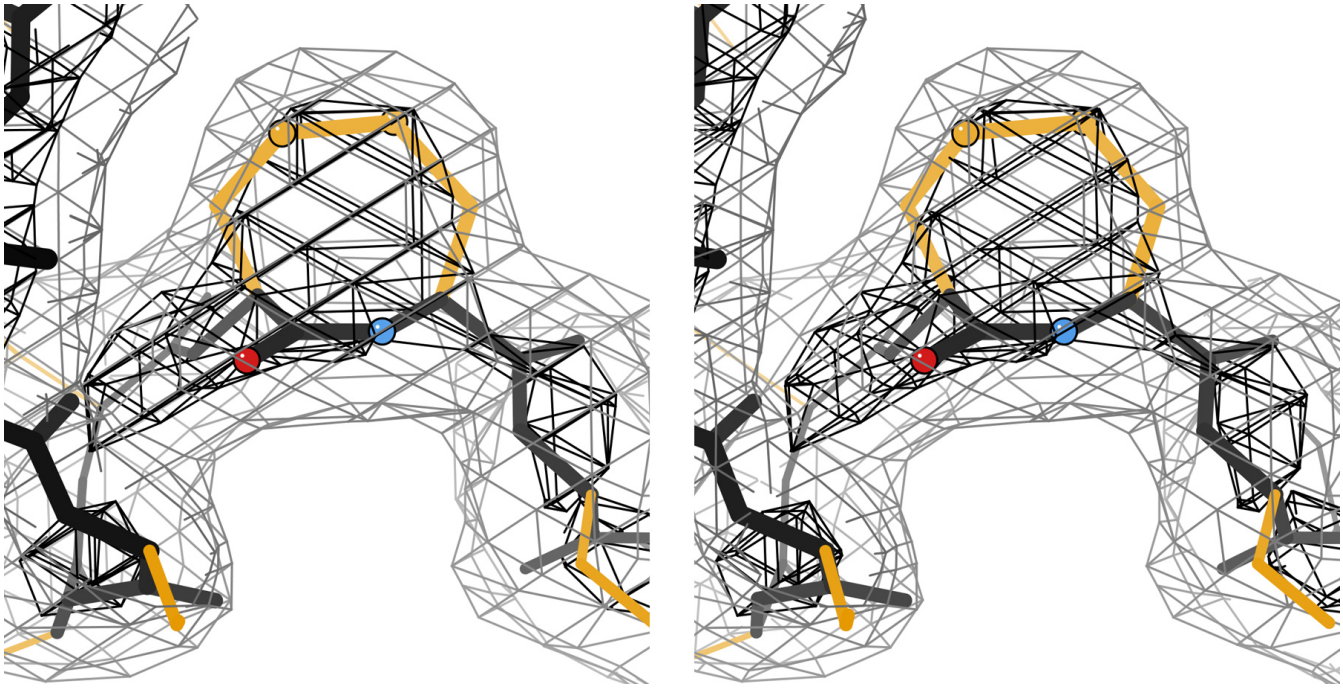


Figure 2: Stereo of a *cis* vicinal disulfide, also in C- conformation, shown as bonded by the electron density at 3.1Å resolution. Cys e 28 from the 4tvp human anti-HIV Fab in complex with env (Pancera 2014). The vicinal SS is in a variable loop of the light chain, near but not at the binding site.

2003) assumed vicinal SS would all be *trans.* It is by now indisputable that the intervening peptide can be either *cis* or *trans*, with *trans* more common – however, both forms are quite rare. Vicinal SS occurrence patterns, conformations and functions were recently reviewed (Richardson 2017) based on quality-filtered reference data and hand-curation of examples.

Most sequence-adjacent Cys residues (not bridged) are in the reduced SH form and only rarely can both Cys ligand the same metal. If they are in the oxidized state, they usually bond to different partners rather than to each other, especially in small SS-rich proteins. Vicinal SS are the exception to all the above rules. Like most rare and energetically unfavorable arrangements, when they do genuinely occur, they are almost always functionally important and thus well worth examining. There are surprisingly few vicinal SS with redox functionality, but they confer stability, bind ligands and gate large conformational changes.

Appearance at various resolutions

At high to medium resolution a vicinal disulfide is very clear and obvious unless there is substantial local disorder. In well-ordered regions, even at 3Å, they can be recognized as a large, dense protrusion from the backbone (see figure 2). However, the evidence can disappear near 4Å (discussed later) or in partially disordered regions. Assigning the detailed conformation is more difficult, of course, since that depends on positions of the carbonyl O and Cβ atoms, which disappear somewhere between 2.5 and 3Å resolution. Even so, trying the four possibilities shown in the next section might suggest a preferred answer. Keep in mind that a vicinal SS is somewhat strained and susceptible to radiation damage, so if the bond is seen to be partially or fully broken, that does not prove it was not originally SS-

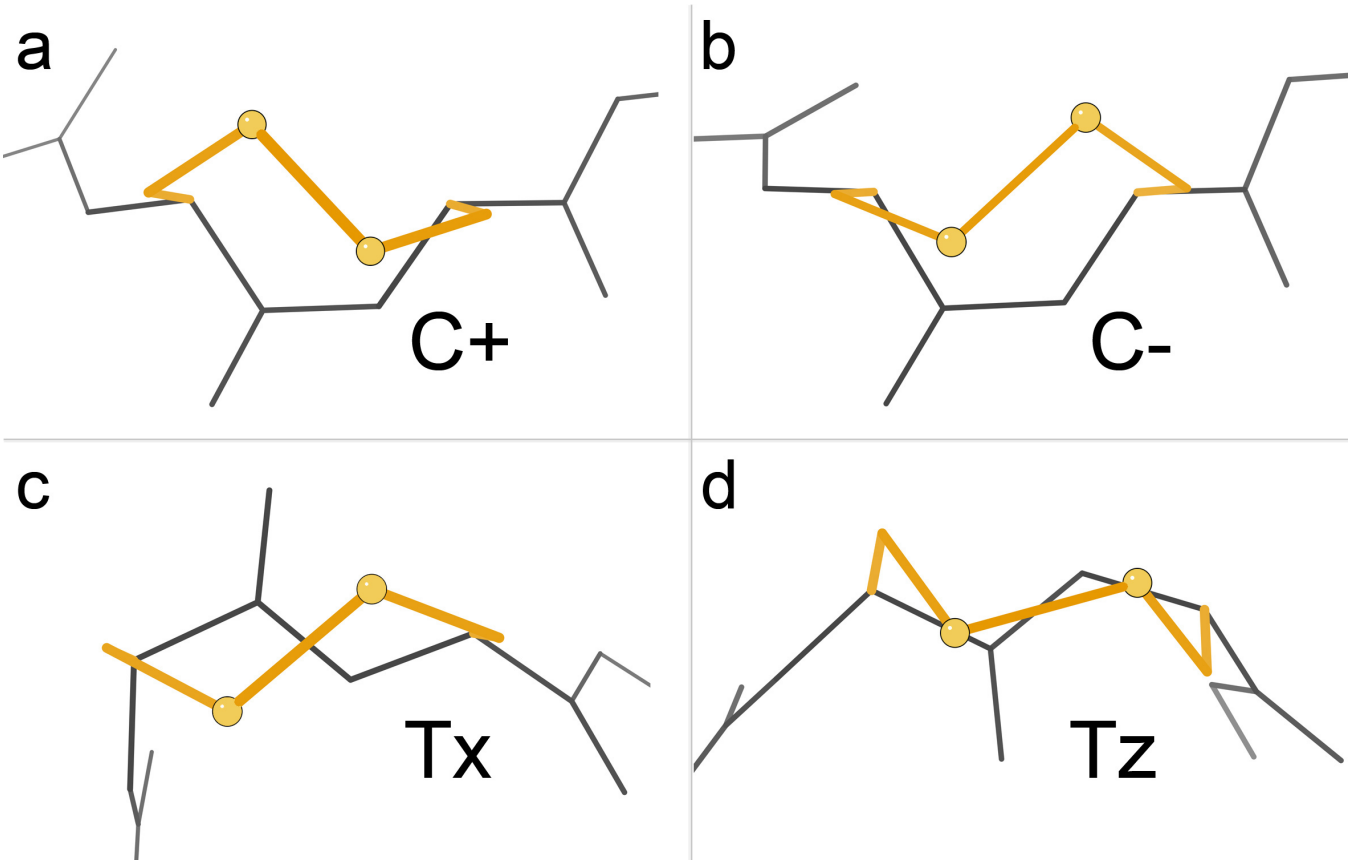


Figure 3: The four conformations of vicinal disulfides seen in reliable examples. a) *Cis* peptide with righthanded χ3, called C+. b) *Cis* with lefthanded χ3, called C-. c) *Trans* and lefthanded, with the SS making an X shape relative to the peptide bond, called Tx. d) *Trans* and lefthanded, with the SS making a Z shape relative to the peptide, called Tz.

bonded (see 3t4m, with Cys-*trans*-Cys191 bonded in chain B, broken in chain E, and probably a mixture in chain D).

Conformations

Only four different conformations are observed in reliable examples, two for *cis* and two for *trans*, as compared in figure 3. For *cis*, disulfide handedness changes between the two conformations, and for *trans*, the peptide orientation changes. The names of the conformations change sign (C+ vs C-) or describe the shape as viewed from above (Tx vs Tz). No cases of dynamic interconversion between *cis* and *trans* are seen, not surprising given the tight, rather strained ring. Specifications of these conformations for use in model building are given in Richardson 2017. These four conformations also match the four distinct cases observed by NMR for dipeptides in solution (Creighton 2001).

Functions

The best-known vicinal disulfide is the Cys-*trans*-Cys of the "C loop" of α subunits in the pentameric nicotinic acetylcholine receptor (nAchR). The vicinal SS is essential both for agonist binding and for the large conformational change that couples that binding to ion channel opening. Figure 4 shows it for the closed, agonist-binding state. In this cryoEM map at 3.7Å one can see it is positioned to touch the ligand, but confirmation of the actual SS bond comes from decades of biochemical and genetic analysis and from higher-resolution structures.

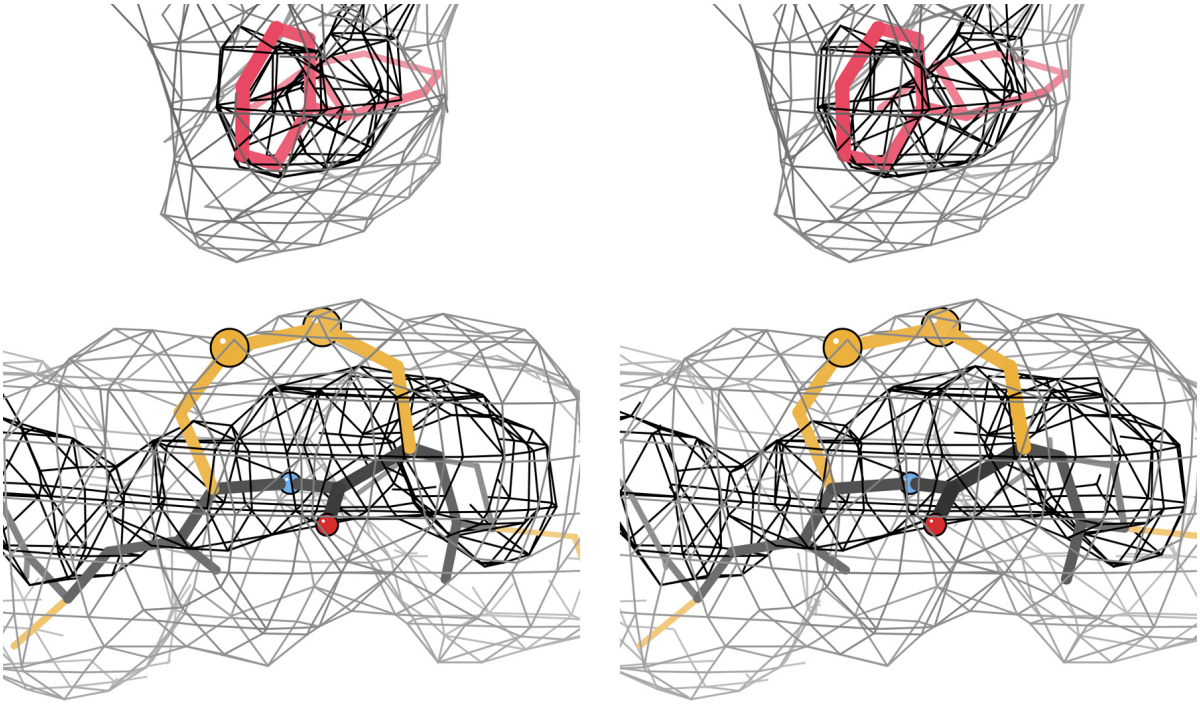


Figure 4: Stereo of the *trans* vicinal disulfide of nAchR at 3.7Å. Cys e 28 from the 6cnj 2α3β nicotinic acetylcholine receptor (Walsh 2018).

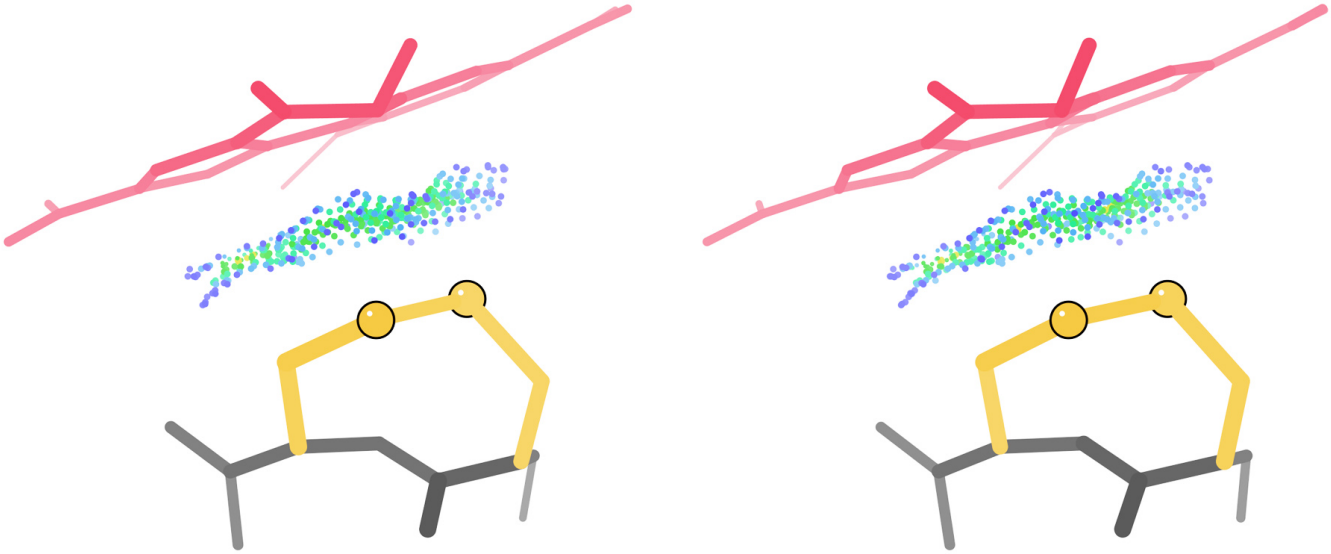


Figure 5: Stereo of a vicinal SS contributing to specific binding of a ring ligand. Cys 104 and PQQ (pyrroloquinoline quinone) from the 2.2Å 2ad6 methanol dehydrogenase (Li 2011). All-atom contacts for favorable van der Waals interaction between SS and PQQ are shown by green and blue dot surfaces.

Other vicinal SS that control large conformational changes occur in von Willebrand blood clotting factor (3gxb) where the SS stays bonded as in nAchR, and in mercuric reductase (1zk7) and the human transglutaminase 2 implicated in celiac disease, where the change involves reduction of the SS bond (2q3z).

Many examples seem to primarily add stability by tight, buried contacts of protein structure around the large lump of the vicinal SS, for example in transferrin-binding

proteins (3hoL), a cytokine receptor (4nn5), and a viral-envelope ribonuclease (4dvk).

Perhaps the most widespread but unexpected function of vicinal disulfides is to provide specific binding of the undecorated face of a sugar or other ring ligand. This occurs in at least six unrelated protein families with distinct folds: nAchR (2qc1), ConA-type lectins (1wd3), small Greek-key antiparallel β (1k12), β-propeller-5 (1gyh, 3d61), β-propeller-8 (2ad6), and TIM barrel (2fhf). The flat, rigidly held C-S-S face of the vicinal SS makes extensive steric contact with the ligand ring, thus selecting for an undecorated position with only H, in concert with H-bonding groups that can select for a position with an OH group. Figure 5 shows the all-atom contact dots of such a case, for Cys-*trans*-Cys104 helping bind the PQQ cofactor in the 2ad6 methanol dehydrogenase at 2.2Å (Li 2011). Functional details about more vicinal SS examples can be found in Richardson 2017.

The bottom line

Vicinal disulfides are extremely rare; so don't model one unless you're sure it's right. But if you do see one, it's very likely to be of functional importance for stability, or for control of conformational change, or for specific binding of the undecorated face of a ring ligand.

References:

Carugo O, Cemazar M, Zahariev S, Hudaky I, Gaspari,Z, Perczel,A, Pongor S (2003) Vicinal disulfide turns, *Protein Engin,* **16**: 637-639

Chandrasekharan R, Balasubramanian R (1969) Stereochemical studies of cyclic peptides. VI. Energy calculations of the cyclic disulphide cysteinyl-cysteine, *Biochim Biophys Acta* **188**: 1-9

Creighton CJ, Reynolds CH, Lee DHS, Leo GC., Reitz AB (2001) Conformational analysis of the eight-membered ring of the oxidized cysteinyl-cysteine unit implicated in nicotinic acetylcholine receptor ligand recognition, *J Am Chem Soc.***123**: 12664-12669

Li J, Gan J-H, Mathews FS, Xia Z-X (2011) The enzymatic reaction-induced configuration of the prosthetic group PQQ of methanol dehydrogenase, *Biochem Biophys Res Commun* **406**: 621-626 [2ad6]

Miyanaga, A., Koseki, T., Matsuzawa, H., Wakagi, T., Shoun, H. & Fushinobu, S. (2004). Crystal structure of a family 54 alpha-L-arabinofuranosidase reveals a novel carbohydrate-binding module that can bind arabinose, *J Biol Chem* **279**: 44907-44914 [1wd3]

Pancera M, Zhou T, Druz A, Georgiev IS, Soto C, Gorman J, Huang J, Acharya P, Chuang G-Y, Ofek G, Stewart-Jones GBE, Stuckey J, Bailer RT, Joyce MG, Louder MK, Tumba N, Yang Y, Zhang B, Cohen MS, Haynes BF, Mascola JR, Morris L, Munro JB, Blanchard SC, Mothes W, Connors M, Kwong PD (2014) Structure and immune recognition of trimeric pre-fusion HIV-1 Env, *Nature* **514**: 455-461 [4tvp]

Richardson JS, Videau LL, Williams CJ, Richardson DC (2017) Broad analysis of vicinal disulfides: Occurrences, conformations with *cis* or with *trans* peptides, and functional roles including sugar binding, *J Molec Biol* **429**: 1321-1335

Walsh RM, Roh SH, Gharpure A, Morales-Perez CL, Teng J, Hibbs RE (2018) Structural principles of distinct assemblies of the human alpha 4 beta 2 nicotinic receptor, *Nature* **557**: 261-265 [6cnj]

FAQ

How do I model a partially broken disulphide?

Use the phil parameter:

disulfide_bond_exclusions_selection_string="chain A and resseq 34 and name SG and altloc A"

It works in `phenix.refine, phenix.real_space_refine, phenix.dymanics, phenix.geometry_minimization`, and more. There are GUI fields of these commands in "All parameters" or "model interpretation parameters" where a search will find the interface required. There is a video tutorial "Changing custom parameters in phenix.refine" @

http://phenix-online.org/documentation/reference/tutorial_channel.html

CaBLAM: A C-Alpha Based Low-resolution Annotation Method for secondary structure and validation

Christopher J. Williams, David C. Richardson, and Jane S. Richardson
Department of Biochemistry, Duke University, Durham, NC 27710

Correspondence email: Christopher.sci.williams@gmail.com or dcrjsr@kinemage.biochem.duke.edu

Introduction

The intent of this piece is to provide documentation of the CaBLAM validation, including its methods, its parameter space and its accessibility within *Phenix*. More detail is provided here than could be included in the Williams 2018 MolProbity paper; still further background and detail can be found in the Williams 2015 PhD thesis.

CaBLAM stands for C-Alpha Based Low-resolution Annotation Method. It is a validation system designed to describe and validate protein backbone using Cα geometry and relative peptide plane orientations. CaBLAM is most valuable as a validation at resolutions and in regions where the backbone Cα trace can be reasonably determined from the electron density, but the positions of backbone carbonyl oxygens cannot.

CaBLAM measures

CaBLAM describes protein backbone using various 2- and 3-dimensional parameter spaces constructed from 4 geometric measures. These measures are two Cα pseudodihedrals, μ_{in} and μ_{out}; the Cα virtual angle; and a dihedral relating adjacent peptide planes, ν.

The two Cα pseudodihedrals, or virtual dihedrals, (figure 1) are used in all of CaBLAM's parameter spaces. μ_{in} is defined using the atom positions $C\alpha_{i-2}$, $C\alpha_{i-1}$, $C\alpha_i$, $C\alpha_{i+1}$. μ_{out} is defined using the atom positions $C\alpha_{i-1}$, $C\alpha_i$, $C\alpha_{i+1}$, $C\alpha_{i+2}$. The combined calculation of μ_{in} and μ_{out} requires five residues in sequence from $C\alpha_{i-2}$ to $C\alpha_{i+2}$, making CaBLAM validation undefined within two residues of chain termini and breaks.

The Cα virtual angle is defined in the conventional manner, using the atom positions $C\alpha_{i-1}$, $C\alpha_i$, $C\alpha_{i+1}$.

The ν dihedral (figure 2) requires the construction of two pseudo-atom points. The point X_{i-1} is constructed on the line from $C\alpha_i$ to $C\alpha_{i-1}$, at the point closest to O_{i-1}. The point X_i is constructed on the line from $C\alpha_i$ to $C\alpha_{i+1}$, at the point closest to O_i. The ν dihedral is then defined using the positions O_{i-1}, X_{i-1}, X_i, O_i.

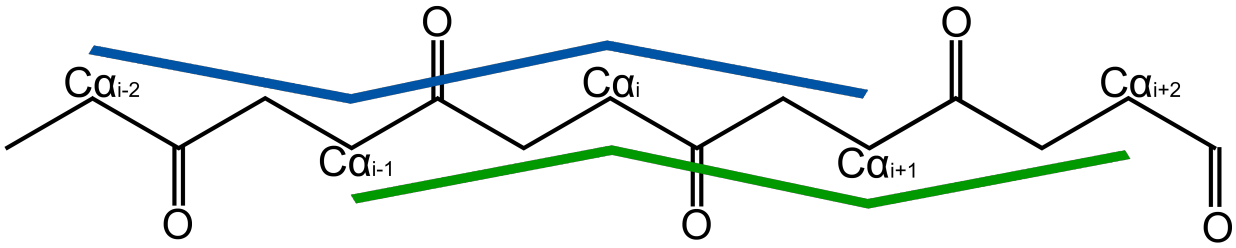


Figure 1: μ_{in} (blue) and μ_{out} (green) pseudodihedrals describe Cα trace of protein backbone.

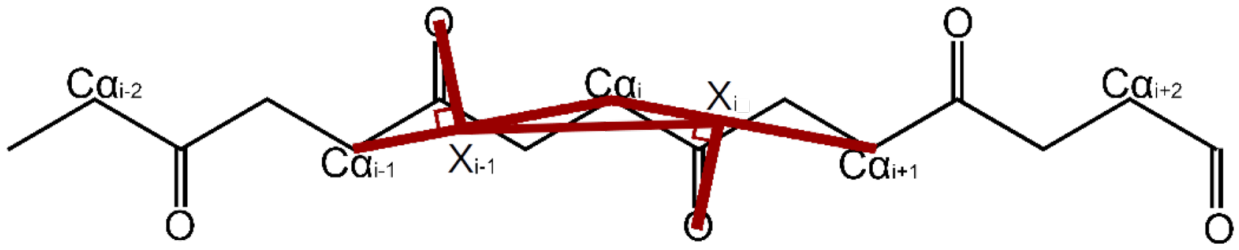


Figure 2: ν pseudodihedral describes torsion relation between adjacent peptide planes across the Cα of the residue of interest.

CaBLAM parameter spaces

Three parameter spaces are constructed from these parameters. In all three parameter spaces, μ_{in} is the x-axis and μ_{out} is the y-axis. The main CaBLAM parameter space uses ν as the z-axis. This space is used to identify CaBLAM outliers. A second, Cα-only space uses the Cα virtual angle as the z-axis. This space is used to identify Cα geometry outliers. The third and final parameter space is two-dimensional, consisting of only μ_{in} and μ_{out}. This two-dimensional space is used to identify secondary structure elements.

Contour levels for these parameter spaces were set using data from the Top8000 quality-filtered database. The $\mu_{in}/\mu_{out}/\nu$ space uses two-tiered cutoffs of 1% for CaBLAM outliers and 5% for CaBLAM disfavored, similar to the outlier and allowed cutoffs in Ramachandran space. The μ_{in}/μ_{out}/Cα-virtual-angle space uses a single cutoff at 0.5% for Cα geometry outliers. The 2D μ_{in}/μ_{out} space uses a cutoff of 0.1% for identifying alpha and 3_{10} helix, and a cutoff of 0.01% for identifying beta strand. The secondary structure behavior is more cleanly defined in CaBLAM space than in Ramachandran space, resulting in steeper edges for the secondary structure distributions and allowing these lower cutoffs without introducing significant false positives.

The geography of the CaBLAM parameter spaces is easiest to understand starting with the 2D μ_{in}/μ_{out} space and its secondary structure contours (figure 3). These representations place 0° at the center of each axis, with -180° at the

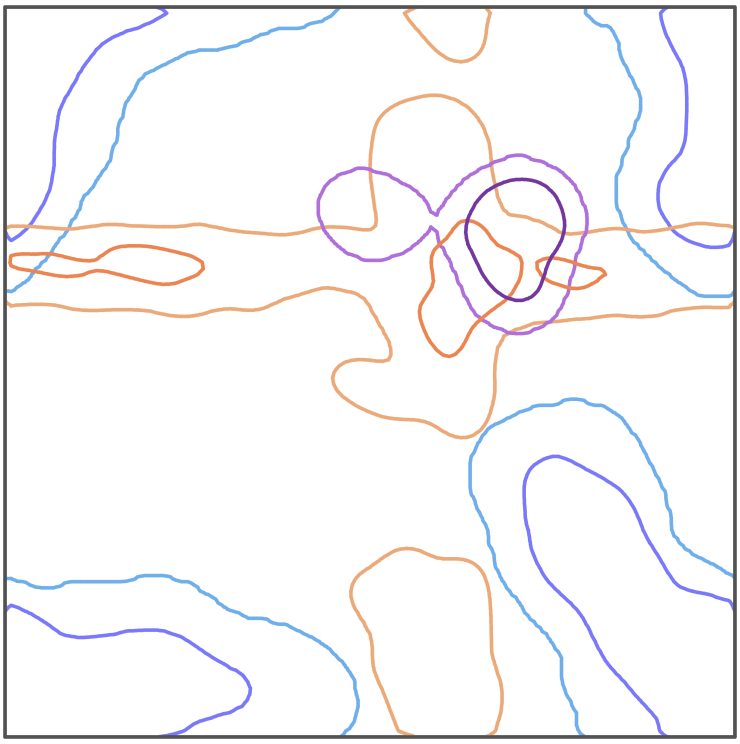


Figure 3: The 2D μ_{in}/μ_{out} space used for secondary structure identification. Cutoffs for alpha helix are shown in orange, 3_{10} helix in purple, and beta strand in blue.

bottom/left and +180° at the top/right. As in the Ramachandran plot, the edges of this space wrap to the other side. The colored outlines in figure 3 show the cutoff contour levels for alpha helix (orange), 3_{10} helix (purple), and beta strand (blue). Linear, elongated structures like beta strand are centered in the corners around ±180°. A completely *cis* Cα conformation – that is, both μ dihedrals at 0° – would fall at the very center of this space. Alpha helix is elongated and twisted slightly from *cis* to permit its repetitive structure, and so the center of the alpha helix distribution is somewhat up and to the right of center. 3_{10} helix requires further elongation of the Cα trace to permit its tighter repeat pattern, and so its center falls further from *cis* than the alpha helix. Right-handed conformations like alpha and 3_{10} helix result in positive μ dihedral values, so these motifs are centered up and right from *cis*. Left-handed alpha would appear in the lower left.

These secondary structure contours also show transitional structures, most clearly evident in the alpha helix contours. The center of the alpha helix distribution has two long arms, a continuous arm across μ_{in} and a discontinuous arm along μ_{out}. These arms represent residues with one helix-like μ and one helix-unlike μ, that is, residues in transition between helix and another structure type such as N- and C-caps. The CaBLAM parameter space's ability to represent transitional structures is both a benefit of the system and a challenge that must be managed when interpreting its results.

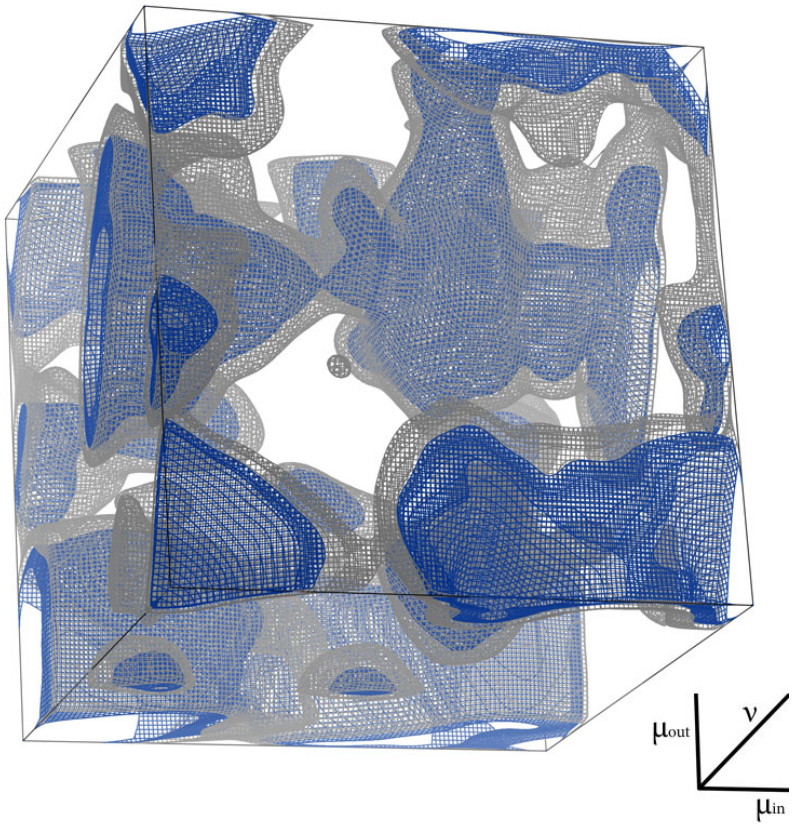


Figure 4: The 3D $\mu_{in}/\mu_{out}/\nu$ space used to identify CaBLAM outliers. The 10% contour is shown in blue and the "disfavored" 5% contour in gray.

The 3D $\mu_{in}/\mu_{out}/\nu$ space (figure 4) is dominated by the common secondary structure elements, which are further differentiated by their distribution in the ν dimension. In beta structure, carbonyl oxygens alternate direction along the strand, near 180° from each other, so beta strand residues are clustered in the corners of this space. In alpha helix, successive carbonyl vectors are only about 60° away from *cis*, so alpha helix residues cluster just above the center of the ν axis. Some new clusters become evident in the full 3D space. The most interesting of

these are the clusters at the very center of each μ/ν face of the parameter space cube, seen most clearly on the μ_{in}/ν face at the bottom of figure 4. These clusters are where beta bulges fall, having both ν and one μ near *cis*. The distance between the beta strand clusters in the corners and the beta bulge clusters on the faces demonstrates the surprising distance that can occur between related structures in the CaBLAM parameter space.

The 3D μ_{in}/μ_{out}/Cα-virtual-angle space (figure 5) is much more compressed, because the Cα virtual angle for good data samples only a range of about 70°-160°. Secondary structures are not distinctly visible in this distribution, although beta strands are extended and thus have larger Cα virtual angles than helices. At the contour level (0.5%) used for identifying outliers, virtually all μ_{in}/μ_{out} combinations are permitted. Therefore, most Cα geometry outliers are assumed to be problems with the Cα virtual angle. Inspection of structures supports this assumption, as most Cα geometry outliers involve obviously too-extended or too-constricted Cα virtual angles.

As in Ramachandran analysis, proline residues have a significantly more restricted distribution than the general case and glycine residues have a significantly more permissive distribution. Proline and glycine therefore each have their own sets of 3D contours for use in validation. Further subcategories of residue types are not currently defined for CaBLAM.

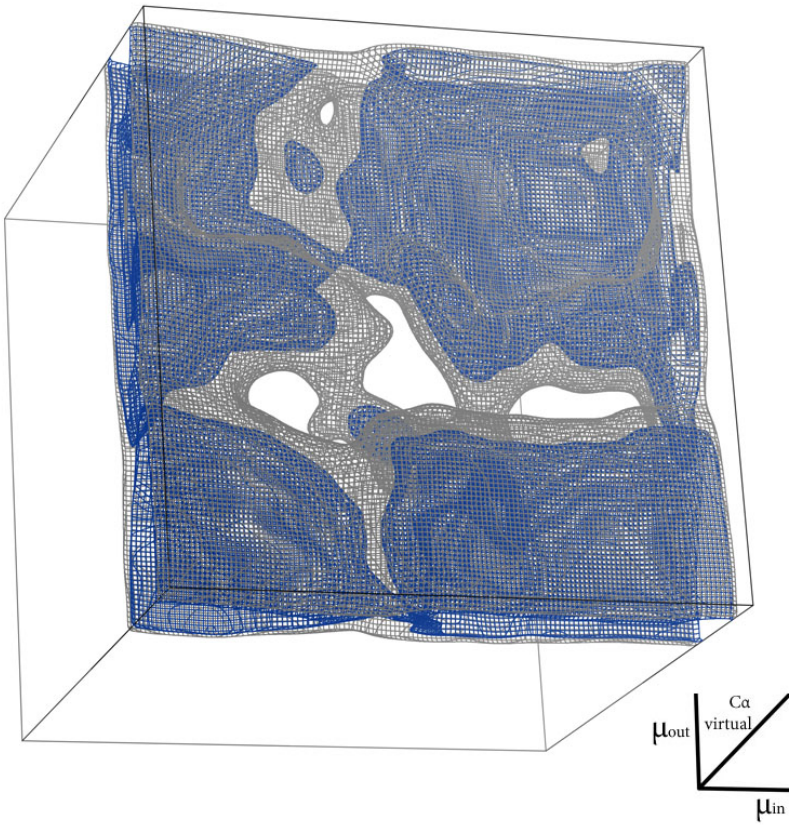


Figure 5: 3D μ_{in}/μ_{out}/Cα virtual angle space used to identify Cα geometry outliers. The 5% contour is shown in blue and the 1% contour in gray.

Interpretation, and assembly of secondary structure

When CaBLAM validation is run, each residue is scored against the 3D CaBLAM contours, the 3D Cα contours, and the 2D contours for alpha helix, 3_{10} helix, and beta strand. Residues that fall below the cutoffs in the 3D CaBLAM or Cα spaces are marked as outliers. Residues that fall above the cutoffs for secondary structure become candidates for assembly into secondary structure elements. A candidate residue is identified as beta strand if that residue and both the preceding and succeeding

residues all pass the beta strand cutoff. Helices can transition between alpha and 3_{10}, so a candidate residue is identified as alpha helix if that residue passes the alpha helix cutoff, and both the preceding and succeeding residues all pass either the alpha or 3_{10} helix cutoff. A candidate residue is identified as 3_{10} helix if it passes the 3_{10} helix cutoff and it scores higher for 3_{10} than for alpha plus at least one of the adjacent residues also passes the 3_{10} helix cutoff. If all of these conditions are met, identification as 3_{10} will override identification as alpha.

Adjacent residues that share the same secondary structure identification are assembled into secondary structures: alpha helix, 3_{10} helix and beta strand. Individual beta strands are not currently assembled into beta sheets because proper registration of strands is challenging in structures where hydrogen bonding is not reliable.

Accessing CaBLAM in *Phenix*

CaBLAM is accessible through the commandline as `phenix.cablam`. The commandline accepts a single PDB or mmCIF file. The default output is a text validation of each residue in the structure, plus a summary of the overall structure statistics. Other outputs can be accessed through the `output=` flag, the most significant of which are `output=kin` for printing CaBLAM's kinemage markup and `output=records` for printing ksdssp-style HELIX and SHEET records.

Internally, CaBLAM is structured similarly to our other validation scripts like ramalyze and rotalyze with a validation class named `cablamalyze` that accepts a model hierarchy object plus some other arguments for controlling amount and destination of output (`sys.stdout`, by default). The `cablamalyze` object contains a list, `cablamalyze.results`, of validations for each residue. In a result object, the CaBLAM geometry parameters can be found in `result.measures`, the contour scores in each parameter space in `result.scores` and the assessment of whether that residue is an outlier and/or secondary structure in `result.feedback`. The `cablamalyze` object contains functions to return various structure-level summary statistics such as available from the parent class `percent_outliers()` function. Another class function, `as_secondary_structure()`, will assemble CaBLAM validations into secondary structure elements and return a *Phenix*-compatible secondary structure annotation object.

Kinemage markup

CaBLAM provides three forms of visual markup for validation kinemages. The first two (figure 6) mark outlier and disfavored residues in CaBLAM space. These markups

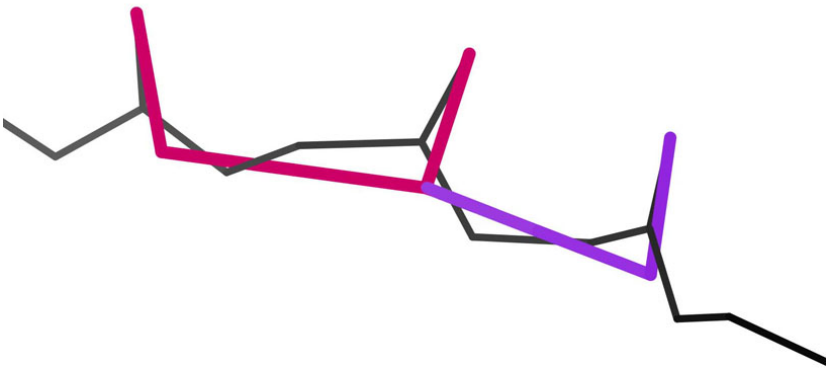


Figure 6: Kinemage markup of an outlier (pink) or disfavored (purple) residue traces the ν dihedral of that residue.

trace the ν dihedral for the outlier or disfavored residue, as that is the geometry most likely to be in error. Disfavored residues (bottom 5% of reference-data protein behavior) are marked in purple and outlier residues (bottom 1% of reference-data protein behavior) in hot pink.

The third markup (figure 7) shows Cα geometry outliers in the μ_{in}/μ_{out}/Cα-virtual-angle space. The Cα geometry markup is red and follows the Cα virtual angle that is both the measure unique to the Cα geometry validation and the parameter most likely to be in error in a Cα geometry outlier.

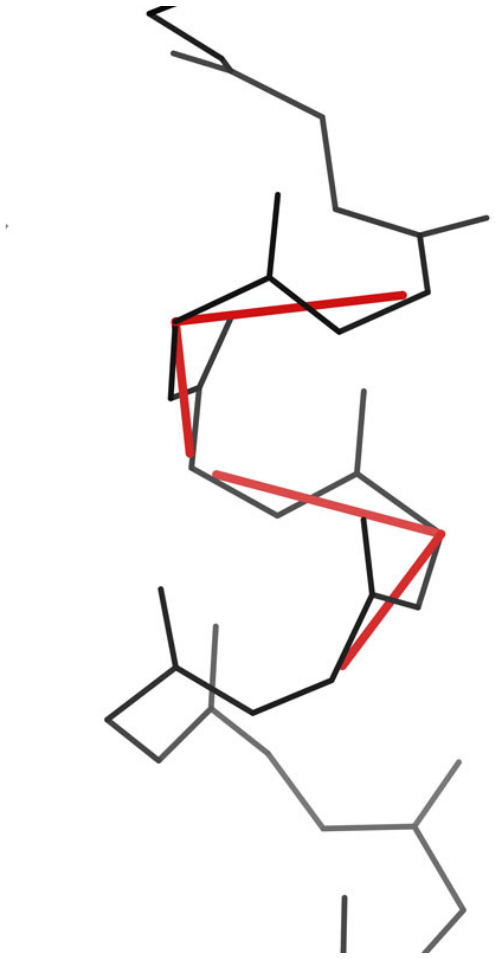


Figure 7: Kinemage markup of Cα geometry outliers follows the Cα virtual angle.

Secondary structure annotations made by CaBLAM are not presented with visual markup as such. However, selecting any vertex of the CaBLAM or Cα geometry markup will display text that includes the secondary structure scores for that residue and KiNG can draw secondary structure ribbons based on HELIX and SHEET records generated by CaBLAM. The secondary structure annotations are designed to be conservative with high confidence and few false positives.

Notes on best usage and interpretation

CaBLAM is bootstrapping up from a minimal form of information – the Cα trace – that is relatively reliable even in poor models. This makes CaBLAM invaluable in validating structures that other methods cannot reliably assess. However, above a certain level of structure quality, more sensitive and precise validations are likely to provide more value and nuance. In MolProbity, we currently approximate this level of structure quality with a resolution cutoff of 2.5Å, and CaBLAM validation is automatically enabled for structures at this resolution or worse.

CaBLAM was designed to address a difficult problem – providing useful validation for structures with minimal reliable information. As a result, some care must be taken in interpreting its feedback. CaBLAM uses two cutoffs to provide some nuance to its validation. The 1% (outlier) cutoff misses some clear modeling problems, while the 5% (disfavored) cutoff includes some motifs that are clearly real-but-rare. No single cutoff adequately separates good structure from bad in CaBLAM space, so context becomes vital.

CaBLAM is overly sensitive in loop regions due to their high variability, so disfavored validations, and sometimes even outliers, in loops should be regarded only as general areas for improvement. However, regular regions, such as those in which CaBLAM has identified secondary structure, disfavored validations do represent significant departures from expected protein behavior and should be taken as serious guides for improvement.

The most generally useful model-rebuilding guidance to take from CaBLAM validation is to:

1) build ideal secondary structure where annotated, and
2) try rotating peptides at one side or the other of a CaBLAM outlier to optimize H-bonding while avoiding steric clashes and other kinds of outliers.

References

Williams CJ, (2015) Using C-alpha geometry to describe protein secondary structure and motifs, Duke University PhD dissertation, 248 pages.

Williams CJ, Hintze BJ, Headd JJ, Moriarty NW, Chen VB, Jain S, Prisant MG, Lewis SM, Videau LL, Keedy DA, Deis LN, Arendall WB III, Verma V, Snoeyink JS, Adams PD, Lovell SC, Richardson JS, Richardson DC (2018) MolProbity: More and better reference data for improved all-atom structure validation, *Protein Science* **27**:293-315.

Tools for interpreting cryo-EM maps using models from the PDB

Tom Terwilliger

Los Alamos National Laboratory, Los Alamos NM 87545
New Mexico Consortium, 100 Entrada Dr, Los Alamos, NM 87544

Phenix now has a set of tools for finding the symmetry in a cryo-EM map, cutting out the unique part of a map and docking a model into a map. These tools now make it easier for you to interpret a cryo-EM map using existing models from the PDB.

Finding symmetry in a map with *phenix.map_symmetry*

The *phenix.map_symmetry* tool is designed to find the reconstruction symmetry used to create a cryo-EM map and to write out a file containing the symmetry operators. The tool has an internal database of common symmetries and it quickly checks to see which ones match the map that you supply. It makes the assumption that the principal axes of symmetry often match the *a,b,c* axes of your map. For helical symmetry, the tool assumes that the helix is along the *z*-axis.

You can run the map symmetry tool with a simple command such as:

```
phenix.map_symmetry emd_8750.map symmetry=D7
```

This will look for D7 symmetry (7-fold symmetry about *z* and 2-fold symmetry about *x* or *y*) and it will report back the symmetry operators that match the map. You can leave off the `symmetry=D7` keyword and the map symmetry tool will look for all types of symmetry.

The symmetry operators that you find with *phenix.map_symmetry* can be useful later if you want to create a complete molecule from one component chain. You might want to do this if you work on a single chain, for example. You can create the entire molecule with:

```
phenix.apply_ncs edited_chain_A.pdb symmetry_from_map.ncs_spec
```

This command will apply each symmetry operator in *symmetry_from_map.ncs_spec* to the chain or chains in edited_chain_A.pdb and create a new model with all these chains. New chain ID will be created for the new chains.

Cutting out the unique part of a map

The *phenix.map_box* tool is a multi-purpose tool that allows you to make a new smaller cryo-EM map by cutting out a part of your cryo-EM map. A new option for the *phenix.map_box* tool is to automatically find and cut out the unique part of your map. All you need to supply is your map, the resolution, the molecular mass of the contents of your map and the symmetry or a symmetry file with your symmetry operators.

The way this works is the *phenix.map_box* tool tries to find a compact part of your map that, including your symmetry operators, represents the whole map. This is done by first finding all the regions in your map that are above a certain contour level, then finding which of these are

duplicated by the map symmetry and choosing a set of regions that is unique and compact. The method is described in Terwilliger et al., 2018.

Figure 1 shows an example of applying the map-box tool in this way. The map is the deposited cryo-EM map of groEL (EMD entry 8750) and the command used is:

```
phenix.map_box emd_8750.map extract_unique=true resolution=4 \
molecular_mass=1000000 symmetry=d7
```

Figure 1 shows the entire map in purple the extracted part of the map in yellow. The extracted part very closely matches chain G of the model for this map (pictured in Fig. 1; PDB entry 5w0s).

Docking a model into a cryo-EM map

You can dock a model into a cryo-EM map using the new *Phenix* tool, *phenix.dock_in_map*. This tool finds the translation and rotation that best matches your model to the map. If you have

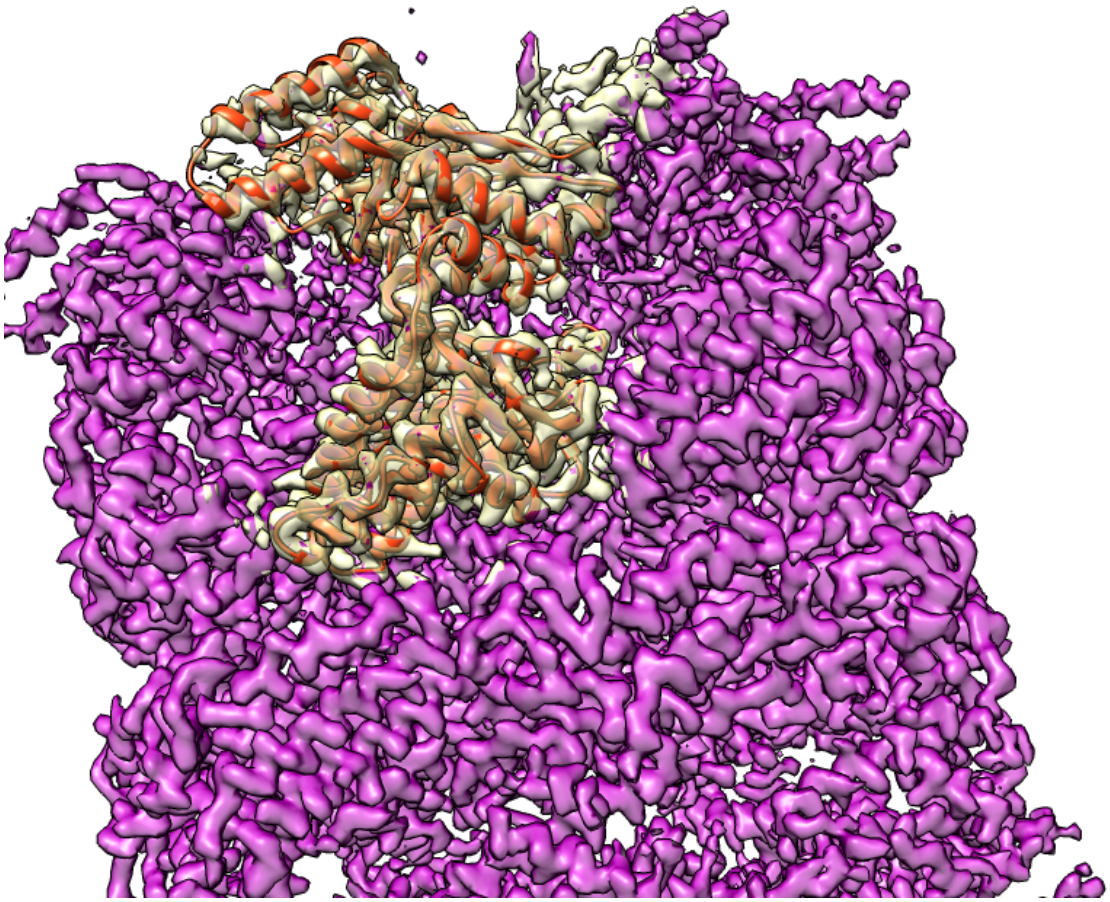


Figure 1

more than one model, or want to fit more than one copy of your model in the map, you can do those things as well.

The *phenix.dock_in_map* tool often can find the location of your model very quickly because it first uses low-resolution representations of your model and the map to find the placement and orientation of the model. Then if the placement is satisfactory, real-space rigid-body refinement is carried out using the full resolution of the map to optimize the placement of the model.

If you are placing more than one model, the density for all previously placed models is first removed from the map, then a search is carried out for the next model to be placed. This can allow you to construct a complex molecule from its parts.

Figure 2 shows how you can dock a model of groEL for chain A from the PDB entry 1ss8 into the deposited full cryo-EM map shown in purple in Fig. 1. The command used is:

```
phenix.dock_in_map 1ss8_A.pdb emd_8750.map resolution=4 nproc=4 \
pdb_out=placed_model_from_emd_8750.pdb
```

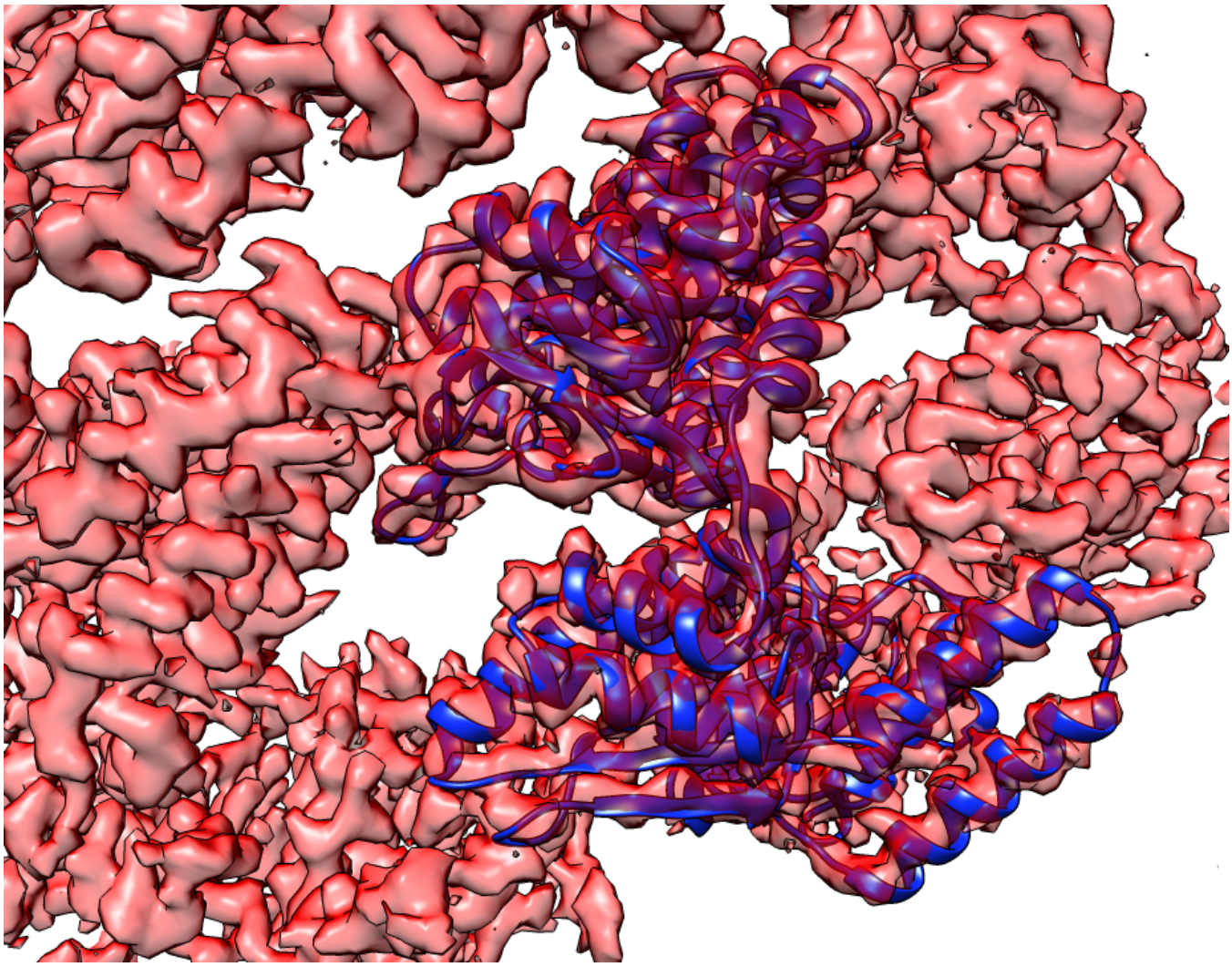


Figure 2

In figure 2 you can see the density for the groEL map in pink and the docked model for 1ss8 chain A in blue.

You can do all these things yourself in a few minutes using the instructions and data in the tutorial called "groel_dock_refine".

Reference:

Terwilliger, T.C., Adams, P.D., Afonine, P.V., Sobolev, O.V.(2018). "Map segmentation, automated model-building and their application to the Cryo-EM Model Challenge" BioRxiv doi: https://doi.org/10.1101/310268

Using the New Program Template

Billy K. Poon

Molecular Biophysics and Integrated Bioimaging Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA

Correspondence email: BKPoon@lbl.gov

Introduction

To help unify the command-line and graphical interfaces for CCTBX-based programs (e.g. Phenix), a new approach based on a program template is introduced. This approach provides some consistent functionality for basic tasks, like file and parameter handling, as well as helps ensure that the same exact code is executed regardless of the user interface.

At a high level, this new approach is based on the Model-View-Controller (MVC) design pattern, probably first introduced by Trygve Reenskaug at Xerox PARC in the late 1970's [1, 2]. Generally, the end user interacts with the View, which can be the command-line terminal or a graphical user interface (GUI). The Controller serves as a translation layer that can convert the user interactions into something the Model uses for the actual work and can convert the output from the Model into something displayed by the View that is understandable by the end user. In this new approach, the Model is the core library functionality of CCTBX that developers use for calculations, the View is the command-line or GUI that end users interact with, and the Controller is composed of several new classes that more clearly define the boundary between the user and the underlying scientific code. These classes are the DataManager that keeps track of the mapping between user-provided data files and the resulting CCTBX data structures that developers manipulate, the ProgramTemplate that explicitly defines a series of steps that a program goes through, and the CCTBXParser that provides a consistent command-line interface (resoectively?). Figure 1 summarizes the relationships between the MVC and the new classes.

Generally, the CCTBXParser takes command-line input from the user to construct a `DataManager` object that contains the input data (e.g. files) and the regular PHIL scope

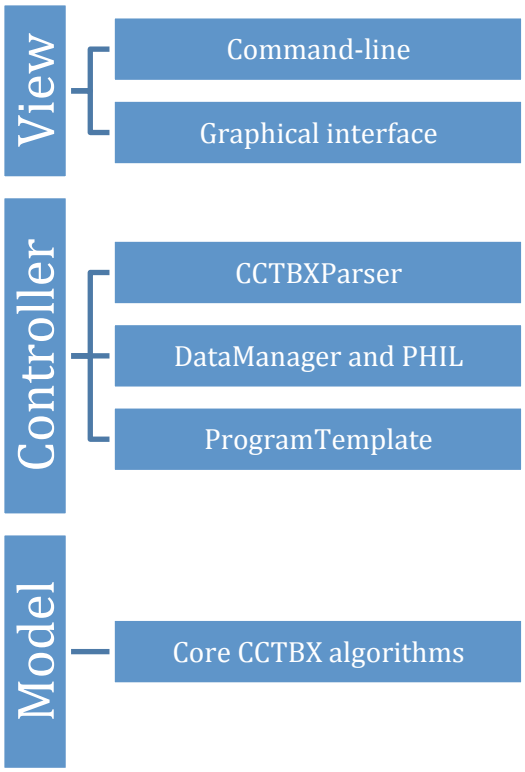


Figure 1: Model-View-Controller design pattern and new classes. User input from the command-line is translated by the CCTBXParser into DataManager and PHIL objects, which are used to create a program object based on the ProgramTemplate. The program then calls core CCTBX functions to do work before reporting output back to the user.

extract object that contains program parameters (e.g. resolution=2.0). These two objects are then used to construct an instance of a program based on the ProgramTemplate since they fully define a job. Once instantiated, standard function names defined in the ProgramTemplate are called to perform the actual work and display output to the terminal. A new graphical interface is under development that utilizes the DataManager and ProgramTemplate classes and will be the graphical equivalent to the CCTBXParser.

New Classes:

To help familiarize developers with this new approach, there is a description of each new class below with some code samples. These code samples are part of a non-trivial program (EMRinger) that was reorganized to adopt this approach. Figures 2 and 3 show the full source code for the program class (mmtbx/programs/emringer.py) and a minimal version of the command-line tool (mmtbx/command_line/emringer.py), respectively.

ProgramTemplate (libtbx/program_template.py):

The ProgramTemplate class should be the parent class for future CCTBX programs. A program is narrowly defined as any tool that is made available to the user. That is, a user provides some form of input, either files and/or parameters, and the program returns some sort of output for the user. To perform this function, the program is broken up into discrete stages, listed below with their associated function names.

1. Initialization (__init__) – This is the normal constructor for a class. The required inputs include a DataManager object and a PHIL extract object. These two objects define the input data and input parameters, respectively. The constructor is predefined in the ProgramTemplate class and generally should not be overwritten by children classes. There are optional parameters for the master PHIL (master_phil) and a logging object (logger), but they are automatically filled in by CCTBXParser (and the future GUI). The constructor basically just sets the arguments to attributes of the instance (self.data_manager, self.master_phil, self.params, self.logger).
2. Custom initialization (custom_init) – This is an optional function that is called after the normal constructor to do any custom initialization. Generally, this is not required because the initialization step should be lightweight and not do any actual work.
3. Input validation (validate) – This step is required and validates the input provided by the user. Since the DataManager object and PHIL object are stored in self.data_manager and self.params, respectively, the developer can check if there is sufficient information to proceed. For example, in Figure 2, the validate step is highlighted pale blue.

   The EMRinger program takes a model and a map and calculates a score for determining the correct sidechain position [3, 4]. To do this, a model and some form of map (real map or map coefficients) are required. The DataManager object is used to check for the existence of necessary data (self.data_manager.has_models, self.data_manager.has_real_maps, and self.data_manager.has_map_coefficients), but also checks that only one kind of map is provided. The DataManager contains some basic functions for these checks and will be described in greater detail later. There are no checks for parameters because the default values are fine, but if some parameter is required, the check should be done here. For example, if a map resolution

```
from __future__ import division, print_function
try:
  from phenix.program_template import ProgramTemplate
except ImportError:
  from libtbx.program_template import ProgramTemplate
import os
import libtbx.phil
from libtbx.utils import Sorry
from libtbx import easy_pickle
import mmtbx.ringer.emringer
# =============================================================================
program_citations = libtbx.phil.parse('''
citation {
  article_id = emringer1
  authors = Barad BA, Echols N, Wang RY, Cheng Y, DiMaio F, Adams PD, Fraser JS
  title = Side-chain-directed model and map validation for 3D Electron Cryomicroscopy.
  journal = Nature Methods
  volume = 10
  pages = 943-46
  year = 2015
  doi_id = "10.1038/nmeth.3541"
  pmid = 26280328
  external = True
}
citation {
  article_id = emringer2
  authors = Lang PT, Ng HL, Fraser JS, Corn JE, Echols N, Sales M, Holton JM, Alber T
  title = Automated electron-density sampling reveals widespread conformational polymorphism in ...
…
}
''')
# =============================================================================
master_phil_str = '''
include scope libtbx.phil.interface.tracking_params
include scope mmtbx.ringer.emringer.master_params
map_label = 2FOFCWT,PH2FOFCWT
  .type = str
  .input_size = 200
  .short_caption = 2Fo-FC map labels
  .help = Labels for 2Fo-Fc map coefficients
show_gui = False
  .type = bool
output_base = None
  .type = str
output_dir = None
  .type = path
  .short_caption = Output directory
quiet = False
  .type = bool
  .short_caption = no graphs
  .help = Don't output files or graphs
'''
# =============================================================================

class Program(ProgramTemplate):
  description = '''
Program for calculating the EMRinger score.

Minimum required inputs:
  Model file
```

Figure 2: Program class for EMRinger (mmtbx/programs/emringer.py)

```
  Map file (or file with map coefficients)

How to run:
  phenix.emringer model.pdb map.ccp4
'''
    datatypes = ['model', 'real_map', 'phil', 'map_coefficients']
    citations = program_citations
    master_phil_str = master_phil_str
    # ---------------------------------------------------------------------------
    def validate(self):
      print('Validating inputs', file=self.logger)
      self.data_manager.has_models(raise_sorry=True)
      if not (self.data_manager.has_real_maps() or
          self.data_manager.has_map_coefficients()):
        raise Sorry("Supply a map file or a file with map coefficients.")
      elif (self.data_manager.has_real_maps() and
          self.data_manager.has_map_coefficients()):
        raise Sorry("Supply either a map file or a file with map coefficients.")
    # ---------------------------------------------------------------------------
    def run(self):
      map_inp = None
      miller_array = None

      print('Using model: %s' % self.data_manager.get_default_model_name(),
        file=self.logger)
      model = self.data_manager.get_model()

      if self.data_manager.has_map_coefficients():
        miller_arrays = self.data_manager.get_miller_arrays()
        miller_array = self.find_label(miller_arrays = miller_arrays)
        print('Using miller array: %s' % miller_array.info().label_string(),
          file=self.logger)
      elif self.data_manager.has_real_maps():
        print('Using map: %s' % self.data_manager.get_default_real_map_name(),
          file=self.logger)
        map_inp = self.data_manager.get_real_map()
        print("CCP4 map statistics:", file=self.logger)
        map_inp.show_summary(out=self.logger, prefix="  ")

      if (self.params.output_base is None) :
        pdb_base = os.path.basename(self.data_manager.get_default_model_name())
        self.params.output_base = os.path.splitext(pdb_base)[0] + "_emringer"

      if not self.params.quiet:
        plots_dir = self.params.output_base + "_plots"
        if (not os.path.isdir(plots_dir)) :
          os.makedirs(plots_dir)

      task_obj = mmtbx.ringer.emringer.emringer(
        model        = model,
        miller_array = miller_array,
        map_inp      = map_inp,
        params       = self.params,
        out          = self.logger)
      task_obj.validate()
      task_obj.run()
      self.results = task_obj.get_results()

      ringer_result = self.results.ringer_result

      if not self.params.quiet:
        # save as pickle
```

Figure 2: Program class for EMRinger (mmtbx/programs/emringer.py) (continued)

```
      easy_pickle.dump("%s.pkl" % self.params.output_base, ringer_result)
      print ('Wrote %s.pkl' % self.params.output_base, file=self.logger)
      # save as CSV
      csv = "\n".join([ r.format_csv() for r in ringer_result])
      open("%s.csv" % self.params.output_base, "w").write(csv)
      print ('Wrote %s.csv' % self.params.output_base, file=self.logger)

    scoring_result = self.results.scoring_result
    scoring_result.show_summary(out = self.logger)

    #rolling_result = self.results.rolling_result

  # It would be good to have central code for this
  # ---------------------------------------------------------------------------
  def find_label(self, miller_arrays):
    best_guess = None
    best_labels = []
    all_labels = []
    miller_array = None
    for array in miller_arrays:
      label = array.info().label_string().replace(" ", "")
      if (self.params.map_label is not None):
        if (label == self.params.map_label.replace(" ", "")):
          miller_array = array
          return miller_array
      elif (self.params.map_label is None):
        if (array.is_complex_array()):
          all_labels.append(label)
          if (label.startswith("2FOFCWT") or label.startswith("2mFoDFc") or
              label.startswith("FWT")) :
            best_guess = array
            best_labels.append(label)
    if (miller_array is None):
      if (len(all_labels) == 0) :
        raise Sorry("No valid (pre-weighted) map coefficients found in file.")
      elif (len(best_labels) == 0) :
        raise Sorry("Couldn't automatically determine appropriate map labels. "+
          "Choices:\n  %s" % "  \n".join(all_labels))
      elif (len(best_labels) > 1) :
        raise Sorry("Multiple appropriate map coefficients found in file. "+
          "Choices:\n  %s" % "\n  ".join(best_labels))
      elif (len(best_labels) == 1):
        miller_array = best_guess
        print("      Guessing   %s   for   input   map   coefficients"%   best_labels[0],
file=self.logger)
        return miller_array

  # ---------------------------------------------------------------------------
  def get_results(self):
    return self.results
```

Figure 2: Program class for EMRinger (mmtbx/programs/emringer.py) (continued)

```
# LIBTBX_SET_DISPATCHER_NAME phenix.emringer
from __future__ import division, print_function

from iotbx.cli_parser import run_program
from mmtbx.programs import emringer

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

Figure 3: Command-line tool for EMRinger (cut from mmtbx/command_line/emringer.py)

is required, the code could look like

```
if self.params.resolution is None:
  raise Sorry("Supply a resolution for the map")
```

The goal of this step is to do a quick check of the user-supplied inputs to determine if it is possible to do the desired work.

4. Run calculation (run) – This step is also required and does the actual calculation. This is program-specific, so it can be as simple or complex as necessary to do the work. For EMRinger, another class is constructed to do the calculation. This brings up the important distinction that the program is separate from core algorithmic functionality. The program, narrowly defined again as a tool exposed to the user, should call core classes/functions to do calculations. The core classes/functions should not call programs. Additionally, log output can be sent to self.logger, which is normally a libtbx.utils.multi_out object.
5. Clean up (clean_up) – This is an optional function for cleaning up any temporary files that may have been created.
6. Get results (`get_results`) – By default, None is returned, but this function can be used to return anything of interest. This function will be important for the new graphical interface because this function will be used for determining the output to be displayed.

The ProgramTemplate defines a standard sequence of steps that programs should use to do something for the user. The steps should be generic enough for any task, but by explicitly defining these steps, we can standardize the behavior of user interfaces.

Furthermore, the class for a program should encapsulate all relevant information about itself. To that end, the `ProgramTemplate` defines several class variables for storing that information. Some basic ones are listed below.

1. description – This is a text description about the program. This will be shown as help text to the user, so some basic information about required files is useful.
2. datatypes – This is a list of data types recognized by the DataManager. For EMRinger, the line from figure 2 is highlighted in pale orange.

   This list is used to construct a DataManager object for EMRinger that will only recognize model files (PDB or CIF), real-space maps, PHIL files, and map coefficients (MTZ or CIF). If some other kind of file is supplied, the DataManager will not be able to process it. The CCTBXParser takes this information and displays output about which files are recognized and which files are not used. The goal is to inform the user of any extra files that have no effect in the program.

3. master_phil_str – This is the regular text that defines the PHIL scope for a program. The include keyword is processed, so the program's PHIL scope can be constructed as a collection of other PHIL scopes.
4. citations – This is a PHIL scope object that contains a list of citations following the format defined in libtbx/citations.py. This is useful for citations that do not exist in the central citation database (libtbx/citations.params). For EMRinger, the citations are created by parsing a text string, as shown in figure 2 slightly truncated from the actual file and highlight in pale green.
5. known_article_ids – This is a list of known citations. The known article ids are stored in libtbx/citations.params. This class variable, along with the citations class variable, are used to construct a list of citations for the program.
6. epilog – This is a text string that is shown at the end of help screen on the command-line. For CCTBX, it is defined as shown on the next page.

```
epilog = '''
For additional help, you can contact the developers at cctbx@cci.lbl.gov

'''
```

Since the ProgramTemplate is just a class, other CCTBX-based projects can subclass this class and redefine any defaults. For example, Phenix redefines the epilog and changes the email address to help@phenix-online.org.

When combined with the other new classes, the ProgramTemplate organizes programs into a consistent format with information in predefined categories. This helps ensure that users get a consistent interface. Also, the standard location for programs is <project>/programs. So for the mmtbx subproject, the programs exist in mmtbx/programs. This is separate from the <project>/command_line directory, which will only contain the command-line interface for these programs.

DataManager (iotbx/data_manager/):

The DataManager maps files provided by the user to CCTBX data structures used by developers. The class handles file reading so developers do not have to worry about reading files and trapping IOError for errors in accessing files. Files are categorized by data types, which are basically the type of data in the file (e.g. models, sequences, reciprocal-space data, real space maps, etc.). Currently, the data types recognized by DataManager are listed below.

1. model – model files (PDB or CIF)
2. phil – PHIL files (text)
3. sequence – sequence files (most formats)
4. restraint – restraint files (CIF)
5. ncs_spec – NCS files (text)
6. real_map – real-space maps (CCP4 format)
7. miller_array – general reciprocal-space data (most formats, e.g. MTZ, CIF, …)
8. map_coefficients – subclass of miller_array that has known labels for map coefficients

There are other data types in development, specifically, more subclasses for miller_array to handle intensities, amplitudes and other reciprocal space data.

This class also introduces the idea of a default file, or the first file encountered of each data type. For programs that only need one file of a specific type, developers do not need to specify a PHIL parameter for that file. For more complicated situations where files and parameters need to be mapped to one another, using PHIL parameters to define that mapping is still required. In the EMRinger program, only a model and a map are required, so instead of having to define PHIL parameters for each file, the first model is recognized as the default model and the first map is recognized as the default map. Since the validate step (shown in the above section) checks that only either real-space maps or map coefficients are provided, there is no case where there is a default real-space map and a default map coefficients file. In the event that multiple files of the same type are provided, the EMRinger program displays which files are used in its analysis. Furthermore, the description in the EMRinger program informs the user that one model and one map (real-space or map coefficients) are the inputs.

The DataManager also defines a set of functions for each data type to access the data structures. Some basic functions for the model data type are listed below, but the “model”

part can be replaced with any other data type (e.g. get_real_map instead of get_model).

1. add_model(filename, data) – creates an entry for <filename> that is associated to <data>
2. set_default_model(filename) – sets <filename> as the default for that data type
3. get_model(filename) – returns the <data> associated with <filename>. If <filename> is not provided, the default is returned.
4. get_model_names – returns a list of known filenames for that data type
5. get_default_model_name – returns the default <filename> for that data type
6. has_models(expected_n=1, exact_count=False, raise_sorry=False) – returns True or False. The expected_n parameter can be changed to the minimum number of items expected. If exact_count is set to True, the number of items has to equal expected_n for True to be returned. If exact_count is set to True, an error (libtbx.utils.Sorry) is raised instead of returning False. This is useful in the validate step of the ProgramTemplate for checking if the expected data exist.
7. process_model_file(filename) – tries to read <filename> and store the associated <data> from the file

Some data types may also have other functions specific for that data type. To see the full list of available functions for each data type, the source code can be browsed in the iotbx/data_manager directory. There is a file for each data type where the name is the data type name (e.g. model.py for the "model" data type).

For general developers, the DataManager will already exist and populated with data, so the main functions for interacting with the DataManager are get_model, get_model_names, get_default_model_name, and has_models and their equivalents for other data types. These functions provide the basic means for getting the data structures and the filenames used for creating those data structures.

CCTBXParser (iotbx/cli_parser.py):

The CCTBXParser is the standard interface for parsing command-line input into DataManager and PHIL objects for creating a program (subclass of ProgramTemplate) object. This parser is a subclass of the standard Python class, argparse.ArgumentParser, that is used for parsing command-line arguments. There is another standard Python parser, optparse.OptionParser, but that module is deprecated.

The only required argument for CCTBXParser is the program class and the parser will pull the relevant information defined in that class to build a standard command-line interface. For example, when a user runs phenix.emringer on the command-line, the default output is shown in schema 1.

The description class variable in figure 2 is shown by the parser as help text and the epilog class variable from the Phenix subclass of the ProgramTemplate is shown at the end. If Phenix were not available, the standard ProgramTemplate in libtbx will be used and that epilog will be displayed instead.

The CCTBXParser class also defines a set of default command-line flags. They are explained in the default output, but generally, the flags are related to showing and saving the parameters in the program, overwriting existing files (e.g. PHIL parameter files) and showing citations. Any program-specific setting should not be a command-line flag; they should be PHIL parameters. This ensures that settings can be encapsulated in files,

```
[bkpoon@eeyore:~] phenix.emringer
usage: phenix.emringer [-h] [--show-defaults [{0,1,2,3}]]
                       [--attributes-level [{0,1,2,3}]] [--write-data]
                       [--write-modified] [--write-all] [--overwrite]
                       [--citations [{default,cell,iucr}]]
                       [files [files ...]] [phil [phil ...]]

--------------------------------------------------------------------------------
Program for calculating the EMRinger score.

Minimum required inputs:
  Model file
  Map file (or file with map coefficients)

How to run:
  phenix.emringer model.pdb map.ccp4
--------------------------------------------------------------------------------

positional arguments:
  files                 Input file(s) (e.g. model.cif)
  phil                  Parameter(s) (e.g. d_min=2.0)

optional arguments:
  -h, --help            show this help message and exit
  --show-defaults [{0,1,2,3}], --show_defaults [{0,1,2,3}]
                        show default parameters with expert level (default=0)
  --attributes-level [{0,1,2,3}], --attributes_level [{0,1,2,3}]
                        show parameters with attributes (default=0)
  --write-data, --write_data
                        write DataManager PHIL parameters to file
                        (emringer_data.eff)
  --write-modified, --write_modified
                        write modifed PHIL parameters to file
                        (emringer_modified.eff)
  --write-all, --write_all
                        write all (modified + default + data) PHIL parameters
                        to file (emringer_all.eff)
  --overwrite           overwrite files, this overrides the output.overwrite
                        PHIL parameter
  --citations [{default,cell,iucr}]
                        show citation(s) for program in different formats

--------------------------------------------------------------------------------
For additional help, you can contact the developers at help@phenix-online.org
[bkpoon@eeyore:~]
```

Schema 1: Default output of EMRinger.

which will help with reproducibility of results. By making these options standard for all programs based on the ProgramTemplate, users will more easily find the parameters available to them and be able to save them. In fact, the --write-all flag will save all the PHIL parameters and when given to the program, will reproduce a previous run.

Also by default, the parser will look for files and PHIL parameters (positional arguments) since these are common inputs to CCTBX-based programs. Furthermore, the parser will recognize any PHIL files and apply the settings stored in these files. And if there are any filenames (path PHIL type) in these PHIL files, the files will automatically be added to the DataManager. However, this is not recursive; that is, if the PHIL file provided on the command-line contains a path parameter with a PHIL file, this PHIL file will be added to the DataManager, but any parameters in this

PHIL file will not be processed by the parser.

It is important to note that there is an order of precedence in how settings are applied. Basically, command-line PHIL arguments are applied in the order they are parsed, from left to right, and they override any settings stored in PHIL files, processed in order from left to right. For example, if there were a program called cctbx.test_program, and it was called in the following way,

```
cctbx.test_program settings1.eff settings2.eff phil_param_1=10
    phil_param_2="what" phil_param_1=5
```

the command-line arguments for phil_param_1 and phil_param_2 will override any values in settings1.eff or settings2.eff. Values in settings2.eff will override values in settings1.eff. Finally, because phil_param_1 is specified twice, that parameter will be set to 5, which is the last specification. To avoid confusion, the parser shows the processing in steps and also has a summary of the modified PHIL parameters after all processing is complete.

Summary:

After CCTBXParser completes parsing any files and PHIL parameters from the command-line, DataManager and PHIL objects are created, which can be used to construct the program object. Then, the steps outlined in the ProgramTemplate section can be called in sequence to perform the work. Because these steps will basically be the same for all programs based on the ProgramTemplate, these steps are consolidated into the run_program function in iotbx/cli_parser.py. As shown in figure 3, the file in the command_line directory can be about 5 lines that call run_program with the appropriate program class from the programs directory. This means that the general developer can just focus on writing the program class as a child class of the ProgramTemplate class and use run_program to get a working command-line tool with all the features outlined above.

Future Work:

While the command-line version is working right now, there are other features in development to further improve consistency and simplify development of programs. The main one is a GUI equivalent to CCTBXParser. Because the constructor for programs only requires a DataManager and a PHIL object, there can be a GUI widget for managing files and constructing the DataManager object, and a GUI for changing PHIL parameters and constructing the PHIL object. The GUI can then construct the program object and go through the same steps of the ProgramTemplate. Because the process of running programs is standardized, all programs can have a basic GUI, similar to how CCTXParser provides a common interface for the command-line. Of course, the option for a more customized GUI, especially for presenting output is still available. More importantly, this is how the same code path will be executed regardless of how the user runs a program, command-line or GUI, which ensures that the same result is the same for both interfaces.

Similar to how there is now a default for simple input files for each data type, another planned feature is to provide default names for output files that are based on the program name. This avoids another common PHIL

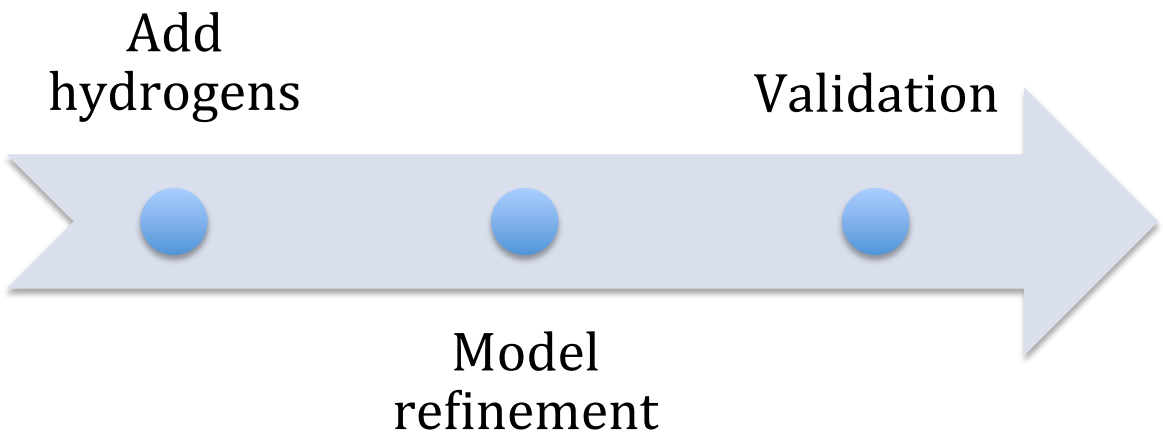


Figure 4: Pipelining example where the blue dot represents the DataManager as it passes through each program. At each step, the default model is updated so that the next step uses the modified model as its input. The "Model refinement" step uses the output of "Add hydrogens," and "Validation" uses the output from "Model refinement." In this case, the data used for refinement and validation is unmodified, so it can just pass through each step normally.

parameter that specifies an output prefix for naming files. Moreover, this should reduce the complexity of developing simple programs that do minor modifications to a file (e.g. converting a model file between PDB and CIF) and just needs to output the result. The DataManager will be responsible for writing output files and this feature will be rolled into the DataManager class. By consolidating the reading and writing of files into the DataManager, the boundary between user files and CCTBX data structures is more clearly defined, which will help with making interfaces more consistent, especially error handling with file input/output.

Longer term, because the DataManager stores information about filenames and data structures for both inputs and outputs, it can be a persistent data repository between programs. That is, developers will be able to link different programs together without having to write scripts or dump data to files between steps because the DataManager has all the relevant information. For example, figure 4 shows a potential pipeline where the first program adds hydrogens to a model file, then the model refinement program is run, and finally validation is performed by the last program on the refined structure. At each step, the default model is updated so that the next step uses the modified model as their default. The "Model refinement" program uses the output from "Add hydrogens," and "Validation" uses the output from "Model refinement." The data used for refinement and validation is unmodified, so it can just pass between steps normally. The data can even be added at the very beginning. The "Add hydrogens" program will just ignore it because the datatypes class variable for that program will not have an appropriate data type and also because the code in the program will not try to do anything with data. Updating the DataManager will probably require the

introduction of a new function to the ProgramTemplate class, most likely called update_data_manager. But because of the nature of classes and inheritance, we can add this function to all programs relatively easily and then customize the implementation for individual programs that need it.

While this is a hypothetical example for pipelining programs, this is currently done in the Phenix GUI for phenix.refine, but in a more *ad hoc* manner and is only available in the GUI. By adopting this new approach, we can standardize the way these pipelines are built and make them available to both the GUI and the command-line. This further improves overall consistency and reduces maintenance overhead by ensuring that the same code path is executed consistently regardless of the interface.

Acknowledgements:

The author wishes to thank members of the Phenix collaboration for helpful discussions about their desired features and requirements for a general program template.

References:

1. Reenskaug, T. "Dynabook System Requirements." 1979 (http://folk.uio.no/trygver/1979/sysreq/SysReq.pdf)
2. Reenskaug, T, Wold, P, and Lehne, OA. "Working with objects - The OOram Software Engineering Method." Manning 1996, ISBN 978-1-884777-10-3, pp. I-XXI, 1-366 (http://heim.ifi.uio.no/trygver/1996/book/WorkingWithObjects.pdf)
3. Barad BA, Echols N, Wang RY, Cheng Y, DiMaio F, Adams PD, Fraser JS. (2015) "Side-chain-directed model and map validation for 3D Electron." Cryomicroscopy. Nature Methods 10:943-46.
4. Lang PT, Ng HL, Fraser JS, Corn JE, Echols N, Sales M, Holton JM, Alber T. (2010) "Automated electron-density sampling reveals widespread conformational polymorphism in proteins." Protein Sci. 7:1420-31.