Journal of
Applied
Crystallography
ISSN 0021-8898

Received 1 October 2003
Accepted 10 November 2003

mmLib Python toolkit for manipulating annotated structural models of biological macromolecules

Jay Painter and Ethan A Merritt*

Biomolecular Structure Center, Department of Biochemistry, University of Washington, Seattle, WA 98195-7742 USA. Correspondence e-mail: merritt@u.washington.edu

© 2004 International Union of Crystallography
Printed in Great Britain – all rights reserved

The *Python Macromolecular Library* (*mmLib*) is a software toolkit and library of routines for the analysis and manipulation of macromolecular structural models, implemented in the Python programming language. It is accessed *via* a layered object-oriented application programming interface, and provides a range of useful software components for parsing mmCIF, PDB and MTZ files, a library of atomic elements and monomers, an object-oriented data structure describing biological macromolecules, and an OpenGL molecular viewer. The mmLib data model is designed to provide easy access to the various levels of detail needed to implement high-level application programs for macromolecular crystallography, NMR, modeling and visualization. We describe here the establishment of *mmLib* as a collaborative open-source code base, and the use of mmLib to implement several simple illustrative application programs.

1. Introduction

Structural biologists are both blessed and cursed with a rich assortment of computational tools and visualization programs for manipulating structural models of proteins, nucleic acids and other biological molecules. The core database of structures upon which these programs must operate has for many years been the Protein Data Bank (Bernstein *et al.*, 1977; Berman *et al.*, 2000), and historically the PDB file format (Westbrook & Fitzgerald, 2003) has been by far the most widespread mechanism for storage and transfer of structural models within this otherwise very diverse set of tools and programs. Unfortunately, the PDB file format was not designed to capture the full range of relevant information associated with these structural models. In particular, it suffers severe limitations as a representation of the complex network of links, references, biochemical data and experimental history needed to integrate such a model fully into a relational database. To address these issues, the IUCr sponsored development of the mmCIF file format and an associated Dictionary Definition Language, DDL2. Both the mmCIF standard and the associated dictionary were released in 1997 after a long period of design and community input (Bourne *et al.*, 1997; Fitzgerald *et al.*, 1996). However, to this date very few of the huge number of programs used routinely by structural biologists have been re-written to take advantage of this new and richer representation. Even programs written since 1997 have largely ignored mmCIF. This is partly due to the understandable desire to retain interoperability and data exchange with older programs which recognize only PDB files, but also to the limited set of programming tools available to manipulate mmCIF data files.

The need for a rich structure description language such as mmCIF is directly paralleled by a need for rich structural representation internal to the computer programs used by structural biologists. For example, a general visualization tool used for molecular modeling should provide the user with far more than a bare representation of three-dimensional structure. It should provide mechanisms to pull in relevant sequence information, homologous structures, biochemical, genetic and medical data, and thus provide context for interpretation of the structural model. With both of these needs in mind, we have undertaken the development of a general library of routines for the manipulation of macromolecular structural models.

Work on *mmLib* is complementary to existing efforts elsewhere. A notable example is the EBI/CCP4 Data Harvesting project (Winn, 1999), which extends programs in the *CCP4* suite (Collaborative Computational Project Number 4, 1994) to emit mmCIF records capturing statistical and data quality measures from various stages of crystallographic structure determination and refinement. These records are currently kept in parallel to the PDB representation of the crystallographic model under refinement. The tools in *mmLib* allow programs to carry such information along with the model description itself, as part of a larger range of possible annotations. *mmLib* development is guided by two primary goals. The first goal is to support extensible and program-independent input/output of richly annotated structural models. The initial *mmLib* implementation reported here supports mmCIF and its associated dictionaries, but the implementation approach is general enough to handle other file formats or direct database access through toolkit extensions. The second goal is to provide an extensible set of operations which act on the internal representation of the structural model. Our implementation is both low-level enough to be efficient and powerful enough to support very high level applications, such as interactive visualization and molecular modeling. Furthermore, the *mmLib* design and its use of Python permit such toolkit extensions without modification or recompilation of higher level application programs.

2. Implementation

2.1. Overview and current state of the library

We have chosen to implement *mmLib* in the Python programming language because of its flexibility and dynamic extensibility (van Rossum & Drake, 2003). Python is becoming the scripting language of choice in the bioinformatics community, which makes it easy for *mmLib* to provide access to the functionality of toolkits developed elsewhere for sequence analysis and database access (Stajich *et al.*,

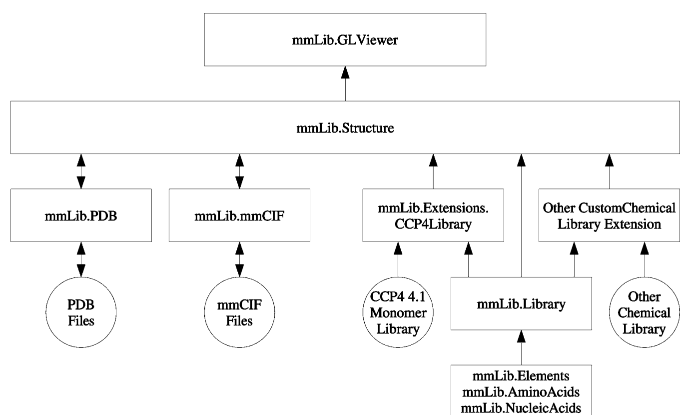


Figure 1
Interaction of *mmLib* modules. The *mmLib.PDB* and *mmLib.mmCIF* implement read/write file parsers for the PDB and mmCIF file formats. *mmLib.Library* implements a basic chemical library, and provides an interface which can be subclassed to create an alternate chemical library, which we have done in the *mmLib.Extensions.CCP4Library* module to retrieve data from the CCP4 monomer library. *mmLib.Structure* implements a hierarchical organization of macromolecular structures, and *mmLib.GLViewer* implements an OpenGL viewer for *mmLib.Structure* objects.

2002), and for molecular simulations (Hinsen, 2000). Python bindings are being developed for the next version of the *CCP4* program suite (Collaborative Computational Project Number 4, 1994). Several crystallographic software packages, including the *Computational Crystallographic Toolbox* and *Phenix*, are being developed as C++/Python hybrid systems (Grosse-Kunstleve *et al.*, 2002; Adams *et al.*, 2002). The widely used *PyMol* (DeLano, 2002) molecular viewer is also written in Python. Our work on *mmLib* has benefited from discussions with the developers of these and other projects.

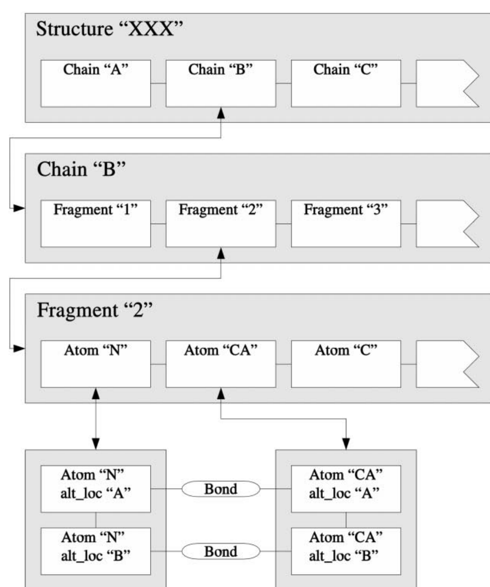
mmLib brings together a range of useful programming modules into one toolkit. Most of the individual capabilities of *mmLib* can also be found in other libraries, but we believe that the integrated design of the *mmLib* components provides a significant advantage for developing extensible application programs that span a wide range of structural research areas. We are currently using *mmLib* for preparation and validation of mmCIF files for PDB deposition of structures determined as part of the structural genomics initiative. We are also using *mmLib* as a programming base for visualization modules to display molecular motions; development of these visualization tools will be reported separately. The initial *mmLib* implementation is less complete in other areas such as molecular modeling, but we hope that application developers in these areas will contribute to the continuing development of *mmLib* through the project's shared code base on SourceForge.

2.2. Python modules

Fig. 1 shows an overview of *mmLib*'s software component layers. It uses the popular Numeric Python (Ascher *et al.*, 2001) and Scientific Python (Hinsen, 2003) libraries to provide vectors, matrixes, matrix algorithms and geometric primitives.

2.3. Building applications on top of *mmLib*

The modules included in *mmLib* provide a significant portion of the functionality needed for several classes of macromolecular applications. The PDB and mmCIF modules can read and write to and from those file formats. Any application which needs to translate,



(*a*)

```
# Python Example Code
# import mmLib
from mmLib.Structure import *
from mmLib.FileLoader import LoadStructure

# load structure
struct = LoadStructure(fil="XXX.pdb")

# get chain B
chain_B = struct["B"]

# get fragment 2
frag_2 = chain_B["2"]

# get atoms N, CA
atom_N = frag_2["N"]
atom_CA = frag_2["CA"]

# switch to alternate conformation B of the atoms
atom_N = atom_N["B"]
atom_CA = atom_CA["B"]

# get the Bond object connecting the atoms
bond_N_CA = atom_N.get_bond(atom_CA)
```

(*b*)

Figure 2
(*a*) Structures described by the *mmLib.Structure* object hierarchy consist of four basic object classes: Structure, Chain, Fragment (and Fragment subclasses AminoAcidResidue, NucleicAcidResidue), and Atom. Other objects are optionally built into the structural description. Structure objects are the parent objects of the entire structure, and contain one or more Chain objects. Chain objects contain a list of Fragment objects, and also contain information describing the polymer sequence within the chain which may a subset of the Fragments within the Chain. Each Fragment object contains a dictionary of its member Atom objects that can be retrieved by the atom's unique name. If a structure contains multiple models or disordered atoms, the alternate conformations of the logically identical atoms can be retrieved *via* any atom in the set. A default model and alternate location ID, set in the top-level Structure object, is used when iterating or accessing Atom objects from any higher-level structural object. Bond objects link two Atom objects together, creating a graph with Atoms as nodes, and Bonds as edges. Atom objects also contain coordinate, charge, occupancy, and other atom data described in the structure source file description. (*b*) Example Python source code for accessing the structure as presented in (*a*). Structure, Chain, Fragment and Atom classes implement many of the Python list class methods, such as __len__(), __getattr__(), __delitem__(), __contains__(), index(), remove() and sort().

edit or inspect files can do so through *mmLib*'s Python modules for those formats, or use the common Structure representation (Fig. 2) of the data which is independent of file format. Calculation of basic geometric operations such as bond distance, bond angle, torsion angle, planarity, *etc.* is provided directly by the *mmLib* code. More complex stereochemical operations, such as least-squares model fitting and extraction of eigenvalues from crystallographic U^{ij} values, are supported through calls to the linear algebra routines in the Numeric and Scientific Python packages.

2.4. Support for graphics and visualization

Once a structure has been read into a corresponding *mmLib* Structure object, it can be visualized using *mmLib*'s built-in OpenGL molecular viewing component, *mmLib.GLViewer*. The PyOpenGL extension to Python provides a single interface for programming with the OpenGL libraries on any operating system, and *mmLib.GLViewer* uses this interface to display Structure objects in three dimensions on any supported platform. Atom drawing properties, including the color, size, brightness and the drawing style of atoms, can be modified by subclassing the *mmLib.GLViewer* component. A simple viewer illustrating the integration of interaction display into an *mmLib*-based application program is provided with the toolkit.

3. File parsers

3.1. PDB

PDB file parsers exist in many individual application programs, and in several biomolecular and scientific Python packages. Most of these parsers interpret only the subset of PDB records relating to atomic coordinates and connectivity, skipping all other records. This is the case, for example, for the PDB parsing modules in the current versions of Scientific Python (Hinsen, 2003) and the BioPython project (Open Bioinformatics Foundation, 2003). In contrast to this, the *mmLib* PDB parser defines a Python class for each Brookhaven PDB v2.2 record type. The parser converts each PDB record line into its corresponding Python object. The PDB record classes are subclasses of native Python dictionary objects, and the dictionary key names are taken from the PDB record field names defined in the Brookhaven PDB file format document (Westbrook & Fitzgerald, 2003). A customized Python list object contains these PDB record objects, and provides methods for saving and loading the record list to and from a PDB file. Once a PDB file is parsed, record objects can be added, modified or deleted before being written back to disk as a PDB file. This makes *mmLib*'s PDB parser a useful software component that is independent from *mmLib*'s structural object model. It can be used to analyze, edit and clean up existing PDB files. It can also be used, in conjunction with the Python-DBI modules, to create a bridge between PDB files and an SQL database.

3.2. mmCIF

The *mmLib.mmCIF* module provides a parser for files conforming to mmCIF grammar (Bourne *et al.*, 1997). The mmCIF file format is quite similar in structure to an SQL database; mmCIF files are organized into large labeled data blocks; these data blocks are further broken down into sections and subsections. The mmCIF sections are equivalent to SQL tables, and the subsections become column names. The actual data then become rows within the table. Our mmCIF parser translates the file into a hierarchical object model with the SQL naming conventions. One possible point of confusion comes from mmCIF's distinction between a data section defining a single set of values, and a data section defining an array, each of whose elements contains a set of values. By using the SQL-like object model, we ignore this distinction and pack the single value set into the table as a single row of data, *i.e.* an array with one element. When the parser writes the mmCIF file back to disk, this translation is reversed. *mmLib*'s mmCIF object model is a hierarchy of subclassed Python lists and dictionaries with additional methods for performing some primitive SQL-like selections. Any row, table or data block of the mmCIF data structure can be edited. Tables and rows from multiple mmCIF files can be combined into one mmCIFFile object. The mmCIFFile object can then be written back to disk as a properly formatted CIF file. The mmCIF parser also contains a modified version of the basic parser capable of parsing mmCIF dictionary files. A reference implementation of a visual editor for mmCIF files is provided with the *mmLib* toolkit.

4. Elements and monomers

4.1. Native library of chemical properties

Calculations involving biological macromolecules often require data for atomic element constants, charge, monomer descriptions and chemical bonds. The core *mmLib* source includes a minimal library of such chemical data for individual elements, amino acids and nucleic acids. Values can be added or changed by editing the Python files, or by writing a new subclass of *mmLib.Library* to load monomer descriptions from a different source.

4.2. External libraries

Applications which require an extended chemical library can do so by creating a subclass of *mmLib*'s core library. This is done in the module *mmLib.Extensions.CCP4Library* to provide access to the *CCP4* monomer library. The *CCP4Library* module is a good example of how *mmLib*'s component design can simplify this type of software development. The *CCP4* monomer library is a directory tree of files in mmCIF format, each wrapped by a simple HTML header and footer. By using the core *mmLib.mmCIF* parser, the *mmLib CCP4Library* module implements dynamic import of the *CCP4* monomer library in less than 100 lines of Python.

5. Underlying structural description

A well designed hierarchical data structure for representing biological macromolecules needs to serve at least two purposes: it must contain all the data from the structure description file, and it must provide access methods that simplify otherwise complex programming tasks. Depending on the task at hand, biological macromolecules are hierarchically conceptualized as functional assemblies, molecules, individual polymer chains, folding units (domains), secondary structure elements, residues, side chains, small molecules, and atoms. This organizational hierarchy seems easy enough to model with a parent–child tree, but it turns out to be more difficult than one would expect. Disorder and alternate conformations do not easily fit into the tree hierarchy. Functional units such as an active site may be constituted of residues from distant parts of the linear sequence, and indeed may span multiple molecules. This presents two challenges. One is to decide what levels of detail are required for the core hierarchical description of the macromolecule. The other challenge is to represent the relationships between individual objects in disparate parts of the hierarchy, for example residues from two or more subunits making up a single active site.

5.1. Implementation

After considering a number of possible data structures, we chose a structural representation consisting of a series of nested Python list and dictionary objects. Access methods for Python lists and dictionaries can be overridden by implementation of their well known class interface methods, which we have done to add more complex behavior to the native Python types. For example, Chains are specialized Python lists which store the chemical Fragments and Residues in order according to their sequence number (and insertion code, if applicable), so the Chain's Fragment and Residue items are always correctly ordered. We have also borrowed another useful concept from Microsoft's Component Object Model (COM) programming techniques (Williams & Kindel, 1994). Every object in the structure hierarchy includes functions for retrieving any other object in the structure. This significantly reduces the number of arguments that need to be passed when calculations on a particular target object necessarily depend on references to a larger set of neighboring objects.

The *mmLib.StructureBuilder* module automates the complex task of building a Structure object hierarchy from a list of atoms with consistent chain, residue and atom identifiers. The *StructureBuilder* class is designed to be subclassed so that the atom list can be created from a variety of sources.

Some objects in the Structure hierarchy are required, and some are constructed through algorithmic searches and library lookups. The Structure, Chain, Fragment and Atom classes are the minimum object set needed to represent the structure described in a PDB or mmCIF file. Other objects may not be needed in all applications. Construction of Bond objects for standard residues is optional, and is controlled through an argument list passed to the class which builds the structure hierarchy.

Structural information that does not fit within the hierarchical description is stored in a pseudo-mmCIF database which is also part of the Structure object. If the source file of the structure was an mmCIF file, the tables not used in constructing the structural hierarchy are copied and added to the associated database. If the source was instead in PDB file format, the PDB records are first translated into mmCIF using the table of equivalences provided by the RCSB (Berman *et al.*, 2000).

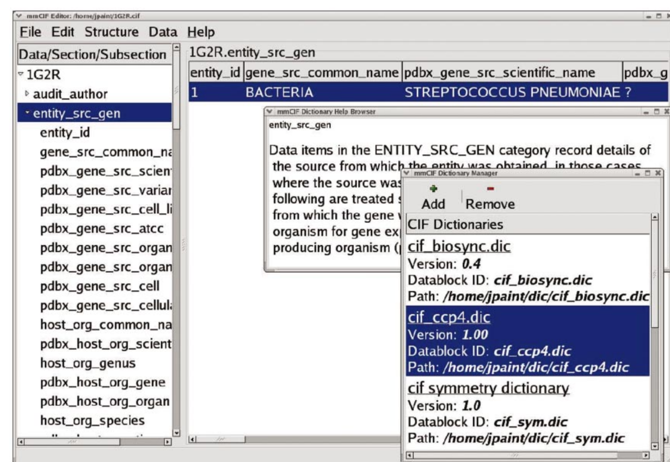


Figure 3
A screenshot of the mmCIF editor included in the *mmLib* toolkit. The editor displays a hierarchical view of the data blocks and dictionary keys on the left, and the corresponding data values on the right. An mmCIF file can be edited without loading the underlying CIF dictionaries, but if the dictionaries themselves are also loaded into the editor then they will be used to provide context-sensitive help on dictionary keys and data entry.

6. Example applications

We have included several sample applications in the *mmLib* distribution. These are primarily intended as coding examples, but they are nevertheless useful in their own right.

6.1. mmCIF input filter

This is a minimal conversion program, essentially two calls to the *mmLib* library, that loads an mmCIF file into a Structure object and then outputs it again as a PDB file. This simple mmCIF to PDB filter is sufficiently robust and lightweight to act as a general input stage for many existing crystallographic programs. It is currently in use as a front-end filter for the Parvati validation server (http://www.bmsc.washington.edu/parvati), allowing the pre-existing analysis programs on the site to handle both mmCIF files and PDB files. There is also a filter program to convert a PDB file to an mmCIF file.

6.2. mmCIF editor

This is a visual editor used to prepare, edit or curate mmCIF files (Fig. 3). It allows easy navigation and viewing of the files through their data blocks, sections and sub-sections. The editor will load mmCIF dictionaries to provide a detailed help description on any field in the file. It also supports field editing in several forms: changing the value of any section/subsection, adding a new row of data to a section, adding a new column to a section, and adding a new section. The editor also allows one to import and merge mmCIF fragments, such as those output by the CCP4 for data harvesting, into an existing mmCIF file.

6.3. Anisotropy

This example program re-implements portions of the Parvati server for calculating the distribution of individual atomic anisotropy in a protein structure (Merritt, 1999). It illustrates the use of simple *mmLib* calls to walk through a Structure object and extract some desired set of properties for statistical analysis. It also illustrates the power of using the Scientific Python library for more complex mathematical operations, in this case the extraction of eigenvalues from the crystallographic U^{ij} matrix for each atom.

6.4. Molecular viewer

Although the *mmLib.GLViewer* module was not intended to be a complete viewer by itself, we have taken care to design it so it can easily be integrated into larger applications with graphical user interfaces (GUIs). GUI applications are programmed using a number of GUI toolkits on various platforms, and each of these toolkits has its own API for creating a drawing window for OpenGL. Although *mmLib.GLViewer* can create its own windows, it can also draw into an OpenGL-capable window created by a GUI toolkit. The example molecular viewer, *mmView*, included with *mmLib* uses this technique to render a structure within the GTK GUI toolkit, a popular toolkit for building Linux applications. This viewer brings together most of the components of *mmLib* in one sample application.

7. Availability

mmLib is being developed as a collaborative open-source project. The code base and documentation are currently hosted on SourceForge, http://pymmlib.sourceforge.net/. The source code is currently released under the Artistic License, but it is our intention to be as flexible as possible on licensing issues. The components of *mmLib* were written so they could easily be used in other Python macromolecular projects. We will be happy to donate any part of *mmLib* to other projects.

This work was supported by NIH awards GM64655 and GM62617.

References

Adams, P. D., Grosse-Kunstleve, R. W., Hung, L. W., Ioerger, T. R., McCoy, A. T., Moriarty, N. W., Read, R. J., Sacchettini, J. C., Sauter, N. K. & Terwilliger, T. C. (2002). *Acta Cryst.* D**58**, 1948–1954.

Ascher, D., Dubois, P., Hinsen, K., Hugunin, J. & Oliphant, T. (2001). *Numerical Python*, http://pfdubois.com/numpy/.

Berman, H. M., Westbrook, J., Feng, Z., Gilliland, G., Bhat, T. N., Weissig, H., Shindyalov, I. N. & Bourne, P. E. (2000). *Nucleic Acids Res.* **28**, 235–242.

Bernstein, F., Koetzle, T., Williams, G., Meyer, E., Brice, M., Rodgers, J., Kennard, O., Shimanouchi, T. & Tasumi, M. (1977). *J. Mol. Biol.* **112**, 535–542.

Bourne, P. E., Berman, H. M., McMahon, B., Watenpaugh, K. D., Westbrook, J. & Fitzgerald, P. M. D. (1997). *Methods Enzymol.* **277**, 271–590.

Collaborative Computational Project, Number 4 (1994). *Acta Cryst.* D**50**, 760–763.

DeLano, W. (2002). *The Pymol Molecular Graphics System*, http://www.pymol.org.

Fitzgerald, P. M. D., Berman, H. M., Bourne, P. E., McMahon, B., Watenpaugh, K. D. & Westbrook, J. (1996). *Acta Cryst.* A**52** (Suppl.). C-575

Grosse-Kunstleve, R. W., Sauter, N., Moriarty, N. & Adams, P. (2002). *J. Appl. Cryst.* **35**, 126–136.

Hinsen, K. (2000). *J. Comput. Chem.* **21**, 79–85.

Hinsen, K. (2003). *Scientific Python*, http://starship.python.net/hinsen/ScientificPython.

Merritt, E. A. (1999). *Acta Cryst.* D**55**, 1109–1117.

Open Bioinformatics Foundation (2003). *Biopython* 1.22. http://biopython.org

Rossum, G. van & Drake, F. L. Jr (2003). *An Introduction to Python*. New York: Network Theory.

Stajich, J. E., Block, D., Boulez, K., Brenner, S. E., Chervitz, S. A., Dagdigian, C., Fuellen, G., Gilbert, J. G. R., Korf, I., Lapp, H., Lehvaslaiho, H., Matsalla, C., Mungall, C. J., Osborne, B. I., Pocock, M. R., Schattner, P., Senger, M., Stein, L. D., Stupka, E., Wilkinson, M. D. & Birney, E. (2002). *Genome Res.* **12**, 1611–1618.

Westbrook, J. & Fitzgerald, P. (2003). *Structural Bioinformatics*, edited by P. E. Bourne & H. Weissig, pp. 161–179. New York: John Wiley.

Williams, S. & Kindel, C. (1994). *Dr Dobb's J. Interoperable Obj. Spec. Rep.* **19:16**, 14–22.

Winn, M. (1999). *CCP4 Newslett. Protein Crystallogr.* **37**.