31st Annual International Conference of the IEEE EMBS
Minneapolis, Minnesota, USA, September 2-6, 2009

LOOS: An Extensible Platform for the Structural Analysis of Simulations

Tod D. Romo and Alan Grossfield

Abstract— **We have developed LOOS (Lightweight Object-Oriented Structure-analysis library) as an object-oriented library designed to facilitate the rapid development of tools for the structural analysis of simulations. LOOS supports the native file formats of most common simulation packages including AMBER, CHARMM, CNS, Gromacs, NAMD, Tinker, and X-PLOR. Encapsulation and polymorphism are used to simultaneously provide a stable interface to the programmer and make LOOS easily extensible. A rich atom selection language based on the C expression syntax is included as part of the library. LOOS enables students and casual programmer-scientists to rapidly write their own analytical tools in a compact and expressive manner resembling scripting. LOOS is written in C++ and makes extensive use of the Standard Template Library and Boost, and is freely available under the GNU General Public License (version 3) (http://loos.sourceforge.net). LOOS has been tested on Linux and MacOS X, but is written to be portable and should work on most Unix-based platforms.**

I. INTRODUCTION

A common problem facing simulations scientists is the need to analyze their data. They are faced with deciding whether to try to use a very capable but complex and difficult to modify system such as CHARMM[1] or ptraj[2], or to try to write their own tool. The latter choice then begets another: which library to use to parse and organize the data, or, to eschew code reuse and write a tool de novo. While there are high quality libraries available for structural analysis, such as SimTK[3], MMTSB[4], and MMTK[5], they are generally part of a large project and are themselves complex and not always easy to integrate. In the case of SimTK and MMTK, these libraries are also geared more towards creating simulations rather than providing interfaces for rapid development of new analytical tools. Moreover, when used with microsecond timescale simulations, the reliance of interpreted languages such as Python and PERL can become an impediment. These aspects of existing tools and libraries impose a substantial activation energy barrier to the construction of new tools and analytical methods, particularly among students and scientists who do not view themselves as programmers.

The design goals for the Lightweight Object-Oriented Structure analysis library (LOOS) are specifically to make it lightweight yet powerful, easily extensible, and easy to use and difficult to use incorrectly. LOOS eliminates the complex modeling hierarchies and concomitant class structures typical of most libraries, instead focusing on atoms and groups of atoms. Enabling the easy creation of small, focused tools and eschewing the monolithic design of many extant packages makes LOOS-derived tools simultaneously easier to learn and easier to maintain. LOOS is designed primarily for analysis, not for creating simulations, and as such supports reading system and trajectory data from a number of different formats.

Another common problem in building analytical tools is communicating to the tool which atoms are of interest. A variety of solutions are used by these libraries, though most utilize a terse "short-cut" description of atom metadata such as that used in CCP4[6] and MMTSB[4] or as used in the `make_ndx` tool in GROMACS[7]. MMTK instead relies on the Python interpreter for selecting atoms, meaning that the selection criteria must be hard-coded or additional code must be written to interpret a user-specified selection. Alternatively, CHARMM[1] and VMD[8] each provide a very complex and expressive selection language, embedded within the package. This tight integration makes it difficult to access the selection language for adding functionality. LOOS takes an approach similar to VMD in providing a atom selection "expression" language, but LOOS does so by specifying the language grammar and parser using the common free Unix tools Flex and Bison. As a result, the selection language is easy to extend and is exposed to the tool-writer.

LOOS is made even more powerful through object oriented program design and by using the C++ Standard Template Library (STL) and components of the Boost Project (a collection of libraries and header files that provide additional functionality to C++). This design makes it easy, for example, to write a tool that is "agnostic" with respect to what specific file format is used to read in the data (e.g. a Protein Data Bank file as opposed to a CHARMM/NAMD Protein Structure File (PSF).

LOOS was designed from the outset to be as easy to use as possible—to provide the simplicity of a scripting language like Python but with C++ performance. Another key to the design of LOOS is to hide from the end user all of the complexity describe above—encapsulation of data and bundling of high-level operations into member functions within its classes hides much of this complexity from the user.

II. IMPLEMENTATION

The following is a description of the implementation details of the principal classes in LOOS. A number of additional support classes are included that are useful in their

T. Romo and A. Grossfield are with the Department of Biochemistry & Biophysics, University of Rochester Medical School, Rochester, NY, 14642, USA

own right, including a full-feature coordinates class, a class for performing geometric manipulations, as well as matrix and memory management classes. While LOOS is designed primarily for input of data, a specialized class for writing basic CHARMM/NAMD DCD trajectories is included.

A. Atom and AtomicGroup

The `Atom` class is responsible for representing atoms and their associated properties, such as coordinates and charges, and metadata, such as names and residue names. The available properties are based on those in the PDB atom record (e.g. atom name, residue name, coordinates, charge, mass, etc) along with a few other properties such as bond connectivity.

The "workhorse" class in LOOS is the `AtomicGroup`. This class manages a group of `Atom` objects and bundles a number of useful functions together into a single namespace. This class also represents one of the major features of LOOS—the elimination of the complex modeling and class hierarchies typical of structural biology libraries. For example, the classical hierarchy of atom, residue, chain, and molecule may be appropriate for a protein in solvent, but not for a membrane system. Instead of providing specialized classes for each hierarchical component and creating these objects during initialization, LOOS treats these objects as just another `AtomicGroup` and provides mechanisms for creating them as needed based on `Atom`'s metadata. For example, the whole system is generally created as a single `AtomicGroup`. If a program needs to isolate a specific protein residue, the `getResidue()` function can be used to take an atom and return a new `AtomicGroup` with all of the atoms in the same residue. The `splitByMolecule()` function walks the bond connectivity tree and partitions the current group into groups that are connected by bonds. This use of arbitrary groups of atoms is similar to how GROMACS[7] works, however LOOS `AtomicGroup` groups are dynamic (e.g. atoms can be added or removed at run-time, such as pruning a user-supplied list of atoms to only retain the heavy atoms).

Typically, the `Atom` objects stored in an `AtomicGroup` are shared between related groups. Therefore, changes in one group, e.g. coordinate transformations, are reflected in all related groups. This means that copying `AtomicGroup` objects is fast and memory efficient. Additional metadata, such as periodic box information, may also be shared amongst related groups.

One of the most common and fundamental operations that can be performed on an `AtomicGroup` is selecting which atoms to extract for a calculation, such as picking all protein alpha carbon atoms or all solvent oxygens. LOOS provides several mechanisms that make this selection easy. The selection criteria can be embedded in the code using a library of special selection functions provided as part of LOOS. Alternatively, the selection criteria can be specified in a string supplied at run-time that is parsed according to the LOOS selection expression grammar. A single function call is all that is required to parse the expression and construct a new `AtomicGroup` with the selected atoms.

The `AtomicGroup` class also serves as a mechanism for grouping "convenience" functions for analysis that are applicable to all groups of atoms. For example, these functions include computing the center of mass, center of charge, dipole moment, and radius of gyration. More code-intensive techniques, such as principal axes and Kabsch superposition are also implemented in `AtomicGroup`; numerical performance is dramatically improved by using fast linear algebra libraries such as ATLAS (for Linux) and the vecLib framework (bundled with MacOS X).

LOOS supports the native file formats for structures from different packages by subclassing the `AtomicGroup` class. Presently, LOOS supports the native file formats for AMBER[2], CHARMM[1], CNS/X-PLOR[9], and NAMD[10], and Tinker[11], while GROMACS[7] support is planned for the near future. The subclasses are responsible for parsing the file contents as well as handling any extra metadata associated with the format. This polymorphic design means that, from the perspective of a LOOS tool, it does not matter if the system is defined from a PDB, an AMBER parmtop file, or a PSF since they are all `AtomicGroup` objects, so long as enough information is present to complete whatever calculations are intended (e.g. partial charge information is needed to compute dipole moments).

B. Trajectories

The `Trajectory` class is both a template design pattern and a class interface for interacting with simulation trajectories. The `Trajectory` class uses a "template" design pattern to standardize access to the trajectory frames wherein the subclasses actually implement the steps that vary depending on the trajectory format. This makes it very easy to support new formats with very little additional code beyond that required to parse a frame. Just as `AtomicGroup` can be used for format-agnostic tools, any code that follows the `Trajectory` interface will work with any specific trajectory format.

The `Trajectory` class provides basic information about a trajectory, such as number of frames present and time steps used. Once a frame of data has been read, the new coordinates can be mapped onto a corresponding `AtomicGroup` using the `updateGroupCoords()` member function. Since `Atom` objects are typically shared between related `AtomicGroup` objects, using `updateGroupCoords()` has the property that all related groups are also updated at the same time. In addition, the `Trajectory` class and subclasses understand about periodic box information and, if it is present in the trajectory, will update the shared periodic box data in related `AtomicGroup` objects.

C. Atom Selection Language

The atom selection language is a major component of LOOS. When the subset of atoms to consider is only known at run time, the tool must provide the user with a method to arbitrarily change the selection criteria. As has been

previously described, most libraries provide only a limited support for this kind of user interaction. The LOOS selection language addresses this by providing a mechanism for compiling a user-provided selection into a function that can be used to select a subset of atoms from an `AtomicGroup`.

III. RESULTS AND DISCUSSION

The following sections will illustrate the code idioms used to accomplish common tasks with LOOS. Additional examples of common idioms and "best practices" can be found by examining the tools bundled with LOOS.

A. Reading Structures

The first step in most tools is to read in a molecular structure. In LOOS, this is accomplished via "factory" functions for reading in structures and trajectories. These functions determine the type of the file by examining the file name and then instantiate the appropriate object, returning the base `AtomicGroup`. In this fashion, tools can easily be format-agnostic, for example,

```
AtomicGroup system;
system = loos::createSystem(system_filename);
```

used in a tool will read in the given file and return the `AtomicGroup` that corresponds to this system. The caveat is that not all formats may include all of the required information for a calculation (e.g. PSF files do not have coordinates). Furthermore, trajectories can be read in a similar fashion using factory functions.

B. Selecting Atoms

After reading in the structure and setting up the trajectory interface, the next step in a tool is usually to pull out the atoms that will be used in the analysis. This is not a capability that is often addressed in structural analysis libraries. For example, CHARMM has a very complex and powerful selection facility, but it is only available within CHARMM scripts and cannot be leveraged by a standalone tool. In LOOS, a selection language with power comparable to that of CHARMM or VMD is accessible to the tool writer in only one line of code.

The selection language is really an expression language, based on C expression syntax, that supports logical and relational operations and keywords that are automatically assigned properties from `Atom`'s. For example, the selection `name == "CA"` matches any atom whose name is exactly "`CA`". Selections can also be combined with logical operators and grouped with parenthesis, i.e. `name == "CA" && ( resid >= 10 && resid <= 20)` will select all $C\alpha$ atoms from residue 10 through residue 20 inclusive.

The selection language also supports regular expression pattern matching for matching atom and residue names as well as segid's. While regular expressions provide a compact and powerful method for matching complex patterns, their use is not required in LOOS and much of what can be expressed with regular expressions can be emulated with the fundamental operators.

C. Commonly Used Idioms

LOOS was designed from the outset to support many of the common analysis idioms in a more easy and concise manner. For example, after selecting which atoms to use, most tools will need to iterate over these atoms performing some operation. The `AtomicGroup` class is consistent with the Standard Template Library and can therefore be used with idioms common to the STL, e.g. `AtomicGroup::const_iterator`. In addition, simplified iterator-idioms are provided that are often easier to use by novice programmers.

Another common idiom is iterating over the frames of a trajectory. This is facilitated by the `Trajectory::readFrame()` function that acts as an iterator while reading in sequential frames from the trajectory. This leads to the following pattern:

```
traj->readFrame(skip_equil_frames);
while (traj->readFrame())
   {
     traj->updateGroupCoords(system);
     perform_calculation(system);
   }
```

In the example above, the first line primes the trajectory iterator by skipping to the frame index before the one with which to begin the calculation. The next line simply loops, reading in the next frame of the trajectory and stopping when the last frame has already been read. The next line then updates the coordinates for all atoms in `system` (and any related `AtomicGroup` objects) with the coordinates extracted from the trajectory.

D. Bundled Applications

LOOS includes a number of applications that serve as "best practices" examples for developing new tools with LOOS, and are useful in their own right—most were written to support our own research. One tool, `aligner`, aligns the frames of a trajectory to the trajectory average, which is computed iteratively[12]. Several different implementations of tools for calculating radial distribution functions are included. LOOS includes additional tools for calculating average structures and the RMSD between a trajectory and various targets. RMSD maps can show system flexibility and the existence of conformational substates, and LOOS provides a tool to calculate these[13]. Principal component analysis (PCA) is supported via a singular value decomposition (SVD) Tool. LOOS includes a set of tools for calculating an elastic network model (ENM) for a system. A tool for generating "porcupine plots" from both ENM and PCA results is included. LOOS also implements a number of tools aimed specifically at analyzing membrane simulations, including programs to compute planar radial distribution functions (`xy_rdf`), deuterium order parameters, and maps of electron density as well as charge and mass distributions. LOOS also has tools for calculating distances between arbitrary groups of atoms as well as basic rotameric state. Finally, there is a tool for computing a ramachandran plot for proteins as well as a ramachandran-like plot for nucleic acids.

Although manipulation of trajectories and models is not a major goal of LOOS, some basic manipulation tools are provided. Chief among these is `subsetter`, which can be used to concatenate trajectories together while extracting an arbitrary subset of frames or atoms. This tool can also be used to recenter the trajectory and adjust or add periodic box information.

CONCLUSIONS

Through designing and using LOOS, we have found that the complex hierarchy of structures automatically imposed by most structure analysis libraries are often unnecessary for analysis. Dispensing with the hierarchy from the start makes our library easier to understand and use. Compositing `AtomicGroup` objects allows the user to easily emulate any hierarchy that is desired. Through the careful design of the LOOS library and the leveraging of object oriented design ideas and the Boost and STL libraries, we have created a library can be used almost like a higher level language. This makes LOOS easy for students and casual programmer-scientists to pick up and use. This also makes LOOS easy to extend and maintain. While not designing specifically for speed, we find that the performance of LOOS is, in general, quite good. In addition, although the goal of LOOS is to provide a framework for facilitating the creation of new tools, LOOS includes many tools that are quite useful in their own right. In the future, we intend to provide Python bindings to make LOOS even easier to use while maintaining the core performance via the C++ side of the interface.

ACKNOWLEDGEMENTS

We thank David Mathews and Julie Hwang for reviewing the manuscript. We also thank David Mathews for input on software design and AMBER support, as well as Matthew Seetin, John Serafino, and Keith van Nostrand for providing examples of AMBER files and for feedback on early versions of the code. We also thank Nathan Baker and his lab for providing samples of Gromacs files.

REFERENCES

[1] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus, "CHARMM: A program for macromolecular energy, minimization, and dynamics calculations," *J Comp Chem.*, vol. 4, pp. 187–217, 1983.

[2] D. A. Case, T. E. Cheatham, T. Darden, H. Gohlke, R. Luo, K. M. Merz, A. Onufriev, C. Simmerling, B. Wang, and R. J. Woods, "The AMBER biomolecular simulation programs." *J Comput Chem*, vol. 26, no. 16, pp. 1668–1688, Dec 2005.

[3] M. A. Sherman, J. L. Middleton, J. P. Schmidt, D. S. Paik, S. S. Blemker, A. W. Habib, F. C. Anderson, S. L. Delp, and R. B. Altman, "The SimTK framework for physics-based simulation of biological structures: preliminary design." in *Proceedings of the Workshop on Component Models and Frameworks in High Performance Computing*, June 2005.

[4] M. Feig, J. Karanicolas, and C. L. Brooks III, "MMTSB tool set: enhanced sampling and multiscale modeling methods for applications in structural biology," *Journal of Molecular Graphics*, vol. 22, no. 22, pp. 377–395, 2004.

[5] K. Hinsen, "The molecular modeling toolkit: A new approach to molecular simulations," *Journal of Computational Chemistry*, vol. 21, no. 2, pp. 79–85, 2000.

[6] C. C. Project, "The ccp4 suite: programs for protein crystallography." *Acta Crystallogr D Biol Crystallogr*, vol. 50, no. Pt 5, pp. 760–763, Sep 1994.

[7] H. J. C. Berendsen, D. van der Spoel, and R. van Drunen, "Gromacs: A message-passing parallel molecular dynamics implementation," *Comp Phys Comm*, vol. 91, pp. 43–56, 1995.

[8] W. Humphrey, A. Dalke, and K. Schulten, "VMD – Visual Molecular Dynamics," *Journal of Molecular Graphics*, vol. 14, pp. 33–38, 1996.

[9] A. T. Brünger, *Xplor-3.1*, Yale University.

[10] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kal, and K. Schulten, "Scalable molecular dynamics with NAMD." *J Comput Chem*, vol. 26, no. 16, pp. 1781–1802, Dec 2005. [Online]. Available: http://dx.doi.org/10.1002/jcc.20289

[11] J. Ponder, "Tinker 4.2." [Online]. Available: http://dasher.wustl.edu/tinker

[12] A. Grossfield, S. Feller, and M. Pitman, "Convergence of molecular dynamics simulations of membrane proteins," *Proteins*, vol. 67, no. 1, pp. 31–40, 2007.

[13] A. Garcia, "Large-amplitude nonlinear motions in proteins," *Physical Review Letters*, vol. 68, no. 17, pp. 2696–2699, 1992.