Lightweight Object Oriented Structure Analysis: Tools for Building Tools to Analyze Molecular Dynamics Simulations

Tod D. Romo, Nicholas Leioatts, and Alan Grossfield*

LOOS (Lightweight Object Oriented Structure-analysis) is a C++ library designed to facilitate making novel tools for analyzing molecular dynamics simulations by abstracting out the repetitive tasks, allowing developers to focus on the scientifically relevant part of the problem. LOOS supports input using the native file formats of most common biomolecular simulation packages, including CHARMM, NAMD, Amber, Tinker, and Gromacs. A dynamic atom selection language based on the C expression syntax is included and is easily accessible to the tool-writer. In addition, LOOS is bundled with over 140 prebuilt tools, including suites of tools for analyzing simulation convergence, three-dimensional histograms, and elastic network models. Through modern C++ design, LOOS is both simple to develop with (requiring knowledge of only four core classes and a few utility functions) and is easily extensible. A python interface to the core classes is also provided, further facilitating tool development. © 2014 Wiley Periodicals, Inc.

DOI: 10.1002/jcc.23753

Introduction

As computers increase in performance and decrease in price, more scientists are using simulations and generating ever more simulation data. The increasing availability of supercomputing resources has only hastened the production of simulation data that now approaches biologically relevant time-scales. With the growth of these data come new questions that can be asked and new ways to analyze the data that are not always addressed by the common analysis software tools, requiring the creation of a new tool. One of the first questions an aspiring tool-writer must ask is how to implement this tool. There are three basic approaches: hook into an existing system at the source code level, extend an existing system through a scripting interface or via plugins, and write a stand-alone tool. An additional concern that is sometimes overlooked is the license for a system. Not all packages permit the tool-writer to distribute their modifications, reducing the benefits to the community at large and reducing the incentive to modifying the system to their needs.

CHARMM,[1] ptraj/cpptraj,[2,3] carma/grcarma,[4,5] and MDAnalysis[6] are popular systems for which source code is available. Using an existing software system has the advantage of a fairly consistent interface for the user and readily available infrastructure for handling file formats and computations. The difficulty in this approach, however, is that the code may be quite large, such as CHARMM, or monolithic, such as carma/grcarma. There are nearly 14,000 lines of C code in carma and grcarma, the graphical front-end to carma, is similarly a monolithic Perl/Tk program. Other packages, such as ptraj, are nearly monolithic. The "actions" module, which contains many analysis routines, consists of nearly 20,000 lines of C code. In these circumstances, it is often difficult to figure out where to place new code and how to integrate it within the supporting environment. Moreover, there is always the danger of side effects from the added code—unforeseen errors that manifest in other parts of the system.

Recently, ptraj was renamed to cpptraj, converted to C++, and refactored so that, among other structural changes, actions reside in separate source code units. Actions use a common programming interface via subclassing and the "command" design pattern, facilitating adding new analyses modules to cpptraj. However, there is still a barrier to adding even a simple analysis module in terms of "glue" code required to hook a new routine into the system. For example, the radial distribution function (RDF) in cpptraj consists of 506 lines of C++ code, whereas the lightweight object oriented structure (LOOS) implementation (rdf) is 434 lines, 50 of which are documentation. Conceptually, actions are decoupled from the I/O and atom selection as these steps are handled by cpptraj itself. There are advantages to this approach, such as relieving the module-writer from having to worry about reading the trajectory and iterating over it. However, this decoupling may be nonintuitive for new programmers and, as we shall see with LOOS, iterating over a trajectory can be accomplished in as little as five lines of code, which is barely a barrier. Moreover, as actions are driven externally (by cpptraj), actions that require multiple passes through the trajectory must cache the requisite data.

Another approach to performing analysis is to integrate it directly into a visualization package, such as Pymol[7] and VMD.[8] The advantage of both approaches is that utilizing a graphical framework, and the possibility of making the analysis interactive, may significantly lower the barrier to use of the tool. Moreover, these tools have very large user communities, which means it is easier for users to find the new tool. The

T. D. Romo, N. Leioatts, A. Grossfield
Department of Biochemistry and Biophysics, University of Rochester Medical Center, Rochester, New York 14642
E-mail: Alan_Grossfield@URMC.Rochester.edu

disadvantage is that using the scripting layer relies on existing functionality from the "host" program, which may or may not have the same flexibility in design as a library would, and it involves an added layer of execution (or interpretation) along with a corresponding increase in execution time. While the plugin approach would seem to solve the execution speed issue, the requirements of the plugin application programming interface (API) and the programming models typically used in interactive graphical applications add several more levels of complexity for even basic tools.

A similar approach is used by ST-Analyzer,[9] which provides an integrated graphical workbench using a web browser as a front-end. While there are clear benefits to providing a graphical interface to analysis tools from the standpoint of the end user, such as ease of use and tracking of results, it often imposes hardships on the programmer that can easily exceed those related to learning a new API, particularly an API with few classes and limited class hierarchies such as LOOS. Adding a new tool to ST-Analyser requires an understanding of the event flow used by the workbench, as well as consideration of both the front and back-ends used, which may require some knowledge of the various technologies used (e.g., AJAX, JQuery, and SQL). The amount of interface code required for typical GUI applications can also be significant. For example, the RDF in ST-Analyzer consists of roughly 1100 lines of Python, JQuery, and HTML code. In contrast, the similar tool in LOOS (xy_rdf) is under 600 lines of C++ code. Although ST-Analyzer provides separate documentation for adding new tools, this document is nearly 20 pages long, which may be quite daunting for less seasoned programmers, or those wanting to quickly test out a new analysis idea. Finally, the implementation of the analysis routine itself will likely still rely on some library for data manipulation and analysis (e.g., MDAnalysis and NumPy) and hence their APIs must also be understood.

Finally, the tool-writer can create a stand-alone tool. While this approach significantly reduces the overhead to creating a new tool, it typically requires a library to handle file formats and basic data structures for storing structural information. MDAnalysis[6] follows the toolkit, or library, paradigm. It provides a set of utility functions and classes that are designed to facilitate the creation of new tools, in addition to providing several useful tools. In many respects, MDAnalysis is a "kindred spirit" to LOOS. However, there are several important differences in both philosophy and implementation. MDAnalysis uses a more "top down" approach to design in that, as much as possible, the library is written in Python. Performance bottlenecks are identified and rewritten in C/C++. LOOS, conversely, uses a "bottom up" approach where most of the library is written in C++ with a Python wrapper on top of it. In practice, the differences in performance, from the Python perspective, are probably small. However, it does mean that should performance or efficient utilization of resources be required, it is possible to only use the C++ layer of LOOS, which is not possible with MDAnalysis. The second, philosophical difference, is that MDAnalysis provides a deeper class hierarchy that mimics common structural elements (e.g., residues, chains, and molecules), while LOOS only uses groups of shared atoms. As we will describe later, this can have important consequences on the idioms used when writing tools, and correspondingly the compactness and complexity of the code implementing the tool.

We believe that one of the paramount goals of any analysis package should be the rapid prototyping of new techniques. It is not always clear, from the outset, what the best method is, nor is it always clear what the implementation should be. During the course of development, it is often necessary to try several different approaches. It is, therefore, critical to minimize the barrier to this programming "noodling." The monolithic and GUI-based approaches impose a substantial barrier to this development methodology, while the toolkit and scripting approaches largely eliminate it, leaving the scientific process of trial and error unimpeded.

Methods

General overview

The design of LOOS is dictated by several fundamental goals. First, it is intended to be lightweight, eschewing the more complex class hierarchies typical of modeling packages, and is, therefore, easy to learn. Tool developers only need to know four core classes (Coord, Atom, AtomicGroup, and Trajectory) and a handful of utility functions. In addition to its simple interface, LOOS has few external dependencies, simplifying installation and maintenance. The primary dependencies are the Boost C++ libraries[10] and LAPACK[11] and the Blas,[12,13] or the ATLAS equivalents,[14] all available through the package managers of most versions of Linux. When compiled using MacOS, LOOS will take advantage of the Accelerate framework thereby gaining both SIMD and multicore parallelization for linear algebra functions.

Another fundamental goal of LOOS is that it is easily extensible. Base classes (e.g., AtomicGroup) are used as much as possible both to fix the API as well as to collect related functions (e.g., most common geometric operations). Finding the common functions that work with an AtomicGroup is simply a matter of consulting the class documentation for AtomicGroup. Relying on base classes for functionality also encourages tools written with LOOS to be agnostic with respect to file format. As a rule, the only time the specialized classes are needed are to address specialized problems.

We designed LOOS from the outset for rapid prototyping and to be easy to use; the LOOS functions are highly expressive, so that most codes resemble scripting languages as much as C++. The relatively lean class structure of LOOS facilitates this goal, as does the grouping of related functions and our efforts to hide memory management from the tool developer. To aid the novice tool writer, several tool templates are also included with the distribution. These contain the framework for common analysis tasks, such as iterating through the atoms of a structure, iterating over a trajectory, or applying a transformation to the trajectory. These frameworks handle interfacing with the command-line, instantiating the

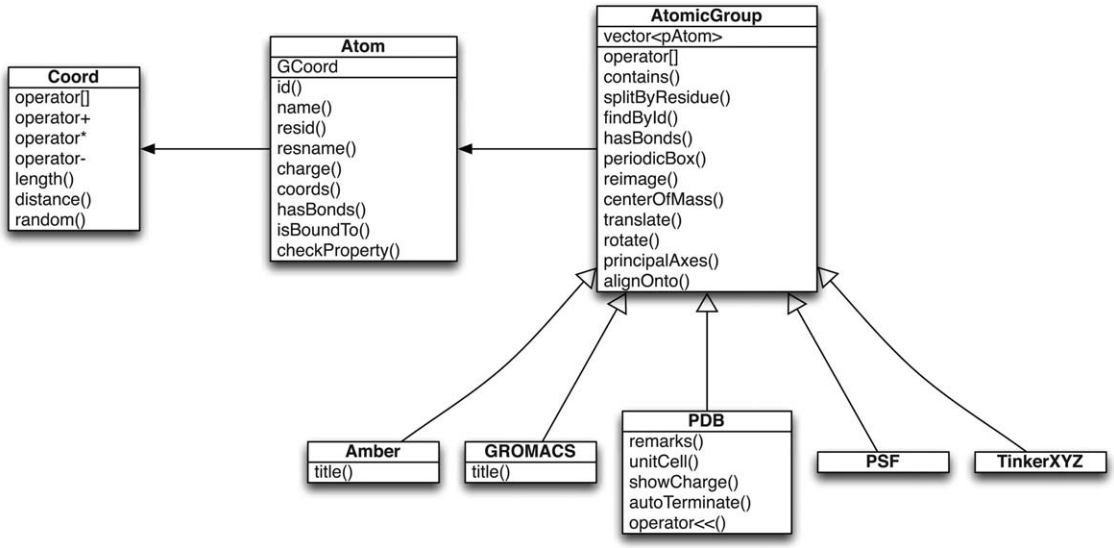

Figure 1. LOOS classes used to model molecular structure and associated file formats. Only a small subset of member functions and operators are shown for illustrative purposes.

appropriate objects, and reading required data. In principal, all that is required is for the tool-writer to insert their analysis code at the appropriate point.

LOOS is also designed to be powerful despite its simplicity. For example, a common problem facing MD analysis tools is how to select what parts of a system are of interest. CHARMM and VMD provide a powerful selection language, but programmatic access to the parser is difficult since the parser is intended to be used interactively. CPPTraj utilizes a masking system with a syntax based originally on MIDAS.[2,15] Gromacs utilizes an index file (in essence, a list of atom numbers) that can be created via different methods.[16] MDAnalysis, in contrast, implements a simple recursive-descent parser in Python for atom selection. The importance of this approach is that the selection can be dynamic (i.e., it can change during the course of the tool's execution). Again, LOOS' approach is similar to that of MDAnalysis, except that the parser is built using the standard UNIX tools lex and yacc. These take a token and grammar specification, respectively, and generate code for the corresponding tokenizer and parser, making it relatively easy to extend the selection language. The selection system is made available to the tool-writer as a simple function call, but the low-level components are also exposed. A unique feature of how selections are implemented in LOOS is that they are actually compiled into a small "program" that performs the selection, and this program can be stored for later use or reuse.

A frequent problem encountered with code libraries (and the C and C++ languages in general) is the need to directly manage dynamically allocated memory. LOOS sidesteps this challenge by working mostly with Boost shared pointers allocated inside the library, effectively removing the responsibility for memory management from the developer. A shared pointer is essentially a reference-counted pointer, that is, a pointer that keeps track of how many things (objects or functions) are using the data it points to. When an object no longer needs the data, the count is decremented and, when the count reaches zero the associated memory is deallocated. Using shared pointers therefore serves as a form of garbage collection, giving LOOS developers the advantages of dynamic memory allocation without the hassles and pitfalls of explicit memory management.

Class structure

There are four fundamental classes in LOOS: three required to represent atomic structure and one that represents molecular dynamics trajectories. The structural classes are shown in Figure 1, along with derived classes used to represent specific file formats. For illustrative purposes, only a subset of member functions are shown.

The first class is Coord, which represents atomic coordinates. This class provides overloaded operators for math involving coordinates (e.g., vector addition, dot products, and cross products), and includes facilities for handling periodic boundary conditions (rectangular boxes). The Coord class uses a template to determine the data type used internally for representing the coordinate. Typically in LOOS, this is a double, although it need not even be a floating point type. For example, the three-dimensional (3D) histogram classes use Coord with an integral coordinate type to represent grid coordinates.

The next fundamental class is Atom, which encapsulate atomic information, such as coordinates (via a Coord), charge, connectivity, and associated metadata (e.g., atom number, residue number, name, etc.). In practice, tools almost never use Atom objects directly. Instead, a Boost shared pointer to the atom, referred to as a pAtom, is used. The job of allocating storage for the Atom is handled internally within LOOS and it is deallocated automatically when no longer needed by the shared pointer "wrapper." There are additional advantages to using shared pointers beyond issues in memory management. Copying of atoms is "lightweight" in that only the pointer

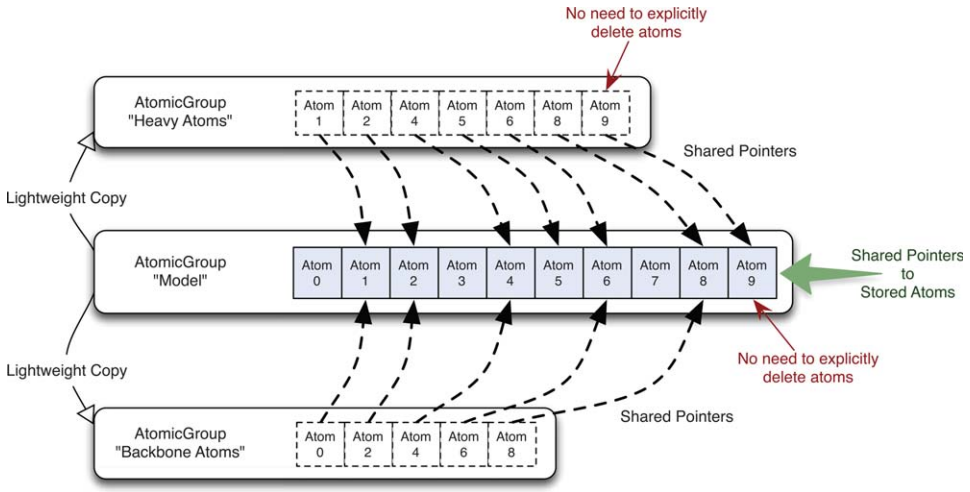

Figure 2. Sharing Atom objects between different AtomicGroup's. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

needs to be copied, not the data pointed to. In addition, sharing pointers to the same atom means that if one function updates the atom, all shared copies of the atom are also updated (e.g., when new coordinates are read from the next frame from a trajectory). This will have important implications in how LOOS emulates traditional structural hierarchies of atoms.

The natural inclination when designing an object-oriented library for simulations analysis is to begin to model real-world components with classes, for example, an atom class, a residue class, a chain class, and so forth. One difficulty with this approach is that the correspondence is not always clear or appropriate. Should a lipid be considered a residue, or a chain? This approach also leads to a large number of classes to model the real-world systems and their hierarchies. Another issue is an implicit tight coupling between the classes and their modeled components. In the older CHARMM27,[17,18] lipids were represented as a multiple "residues" whereas in CHARMM36,[19] lipids consist of a single residue. Such a change requires either a reinterpretation of the associated class, or a redesign of that class.

Yet another difficulty with these hierarchical representations is how to best "drill down" and iterate at the level that the tool-writer is interested in. For example, a loop over all atoms might look like,

```
for molecule in world:
  for lipid in molecule:
    for residue in lipid:
      for atom in residue:
        do_something(atom)
```

Such an organization requires hard-coded nested iterations, which would quickly make the code messy and difficult to read, considering how often this task is performed. Alternatively, the library can provide another method to iterate over all atoms. However, this typically requires internal structures to support the iteration. Even more problematic is at what granularity should iteration be supported: atoms, residues, chains, or all of them?

The approach LOOS takes is to dispense with explicitly modeling a hierarchy. Instead, only a collection of atoms is supported, called an AtomicGroup, which is the third fundamental class in LOOS. When coupled with an extensive system for selecting atoms (usually via atom metadata), virtually any hierarchy can be implicitly modeled. For example, selecting a range of residue ID's returns an AtomicGroup containing all of the corresponding atoms (or strictly speaking, pointers to the atoms selected). If the tool needs to operate only on residues from that range, then this group can be further split into an array of AtomicGroup objects, each containing the atoms corresponding to a single residue.

AtomicGroup can also be considered the "workhorse" class because it collects many common functions that operate on groups of atoms. These include calculating the center of mass, radius of gyration, principal axis vectors, alignment, and basic set operations, as well as aligning one group onto another. One of our design goals was to reduce all common tasks into one-line operations; this not only simplifies new code, but makes existing code more expressive and comprehensible.

As mentioned above, AtomicGroup actually contains not Atom objects, but rather pAtom's (Boost shared pointers to Atom's). When an AtomicGroup is copied, or a set of atoms are selected and returned as a new AtomicGroup, the common Atom objects are shared between the two groups. Changes made to an atom in one group will also appear in the other group. This property is illustrated in Figure 2. The "Model" AtomicGroup contains the pAtom's representing the entire system. The "Heavy Atoms" group was created from the "Model" group using a nonhydrogen selection. Similarly, the "Backbone Atoms" group was created by selecting the backbone atoms by their name metadata. Each AtomicGroup object has its own pAtom objects, but share the underlying Atom objects. Altering the first backbone atom will also alter the corresponding atom in the "Model" and "Heavy Atoms" groups.

Reading the native file formats for different packages is handled by subclasses of AtomicGroup, but the tool-writer normally does not interact with these specialized classes. Instead, a factory function is used to read in a file regardless of the format, returning an AtomicGroup; as a result, LOOS programs are by

Table 1. File formats supported by LOOS for reading in data.

Format	Class Name	Type	File Extensions
Amber Parmtop	Amber	Model	prmtop
Amber NetCDF	AmberNetcdf	Trajectory	mdcrd, crd, nc
Amber Restart	AmberRst	Trajectory (single frame)	inpcrd, rst, rst7
Amber Trajectory	AmberTraj	Trajectory	mdcrd, crd
Concatenated PDB	CCPDB	Trajectory	pdb
CHARMM Coordinates	CHARMM	Model	crd
NAMD DCD	DCD	Trajectory	dcd
GROMACS Model	Gromacs	Model	gro
NAMD PDB	PDB	Model	pdb
Multi-PDB Trajectory	PDBTraj	Trajectory	
NAMD PSF	PSF	Model (no coordinates)	psf
Tinker Arc	TinkerArc	Trajectory	arc
Tinker XYZ	TinkerXYZ	Model	xyz
GROMACS Trajectory	TRR	Trajectory	trr
GROMACS Compressed Trajectory	XTC	Trajectory	xtc

default format-independent. For example, the following code will read in models stored in different package formats,

```
AtomicGroup namd_model = createSystem("model.pdb");
AtomicGroup gro_model = createSystem("model.gro");
AtomicGroup amber_model = createSystem("model.parmtop");
AtomicGroup tinker_model = createSystem("model.xyz");
```

The only exception is when writing out a structure. The only format supported for output is the PDB format, and involves converting an AtomicGroup into a PDB object (using a class function) and printing it out.

The final fundamental class is the Trajectory class, which provides an abstracted interface to all of the trajectory filetypes supported by LOOS (see Table 1). Every supported filetype derives from this class (i.e., DCD implements the CHARMM/NAMD trajectory format and is a child of Trajectory). Since Trajectories need to behave polymorphically, pointers must be used to the appropriate Trajectory-derived object. These are passed as Boost shared pointers called pTraj's (a typedef, similar to pAtom, used to make the code simpler). It is also worth noting that LOOS does not restrict the use of a trajectory type with a specific model, that is, it is possible to mix and match model file types with different trajectory formats, for example, combining a model defined by a NAMD PSF file, and a GROMACS trajectory,

```
AtomicGroup model = createSystem("model.psf");
pTraj traj = createTrajectory("simulation.xtc", model);
```

This can on occasions be convenient, because not all file formats provide identical information; PSF files contain information about connectivity and partial charges, while GRO files do not.

As a common pattern for analysis tools is to work on each frame in a trajectory, a Trajectory object can also be used as an iterator.

```
pTraj traj;
AtomicGroup model;
while (traj->readFrame()) {
  traj->updateGroupCoords(model);
  analyze(model); // Do some analysis task
}
```

To ensure consistent behavior of the iterator, seeking functions are implemented using the "Non-Virtual Interface" idiom. The public seek and read functions maintain the state of the iterator while hidden implementation-specific functions are called to perform the actual seeking. For trajectory formats where the location of a frame is not easily calculable (e.g., GROMACS XTC), LOOS will scan the trajectory at initialization of the Trajectory object and build an index into the file for each frame, permitting rapid random-access to any frame in the trajectory. This is not necessary for formats with a fixed frame size, such as CHARMM/NAMD DCDs.

Associating coordinates within a frame of a trajectory to a given atom can be problematic. LOOS solves this by including extra metadata with each Atom that specifies its location (or index) into the trajectory frame. This index is independent of the atomid property and is set when an AtomicGroup is read in by one of the derived-classes that represent specific file formats. The index is determined by the order the atoms appear in the model file, and are assumed to match the corresponding ordering of data within the trajectory frames.

Selection language

Selecting atoms in LOOS is based on the idea that a selection is really just a filter, picking atoms based on specific properties and placing them into a new AtomicGroup. The selection language LOOS uses is loosely based on the C expression syntax. Atom properties are bound to keywords (summarized in Table 2), such as id for the atom-id and resname for the residue name. In addition, there are special keywords, such as `all` which matches everything, `none` matching nothing, and `hydrogen` matching only hydrogen atoms (based on name or molecular weight). The LOOS operators are summarized in

Table 2. LOOS Keywords.

Keyword	Atom Property	Type
name	Atom name	String
id	Atom id	Number
resname	Residue name	String
resid	Residue number	Number
segid	Segment name	String
segname	Segment name	String
chainid	Chain ID	String
all	Always true	Boolean
none	Never true	Boolean
hydrogen	True if atom is a hydrogen	Boolean

Table 3. LOOS Operators

Operator	Operation	Example
>	Greater than	resid > 10
<	Less than	resid < 10
>=	Greater than or equals	resid >= 10
<=	Less than or equals	resid <= 10
==	Equals	name == "CA"
!=	Not equals	segid! = "SOLV"
=~	Regular expression	name =~"^(CA|C|N|O)$"
&&	Logical and	name == "CA" && segid == "PROT"
||	Logical or	segid == "SOLV" || segid == "BULK"
!	Logical not	!hydrogen
not	Logical not	not hydrogen
->	Number extraction	(segid -> "L(\d+)") < 100

Table 3, which includes the standard C relational operators. There are also two special operators. The first, =~, permits regular expression matching that gives the user a powerful pattern-matching system as well as substring matching. The second, ->, extracts numbers from within strings (e.g., residue names, segment names, etc.). Logical operators (&& and ||) as well as parenthesis are also defined. Operator priority follows the C-convention. As an example, selecting all alpha-carbons in LOOS is written as,

```
name == "CA"
```

Selecting a range of residues based in their resid is simply,

```
resid >= 10 && resid <= 20
```

To select only nonhydrogen atoms from the same block of residues, the previous selection is combined with one to exclude hydrogens,

```
(resid >= 10 && resid <= 20) &&!hydrogen
```

An example of using regular expressions is how to select backbone atoms in LOOS,

```
name =~ "^(C|O|N|CA)$"
```

This selection matches any atom whose name is exactly "C", "O", "N", or "CA". Since regular expressions would normally match a substring, it is necessary to "anchor" the strings we want to match using the ^ and $ operators. In fact, LOOS (via Boost) supports most of the Perl-extensions to regular expressions enabling very sophisticated pattern matching.

Although the LOOS selection language is more verbose than other systems, such as VMD or Gromacs, this verbosity can be an asset in that it makes the selections easier for the user to read and understand. The drawback, however, is that repeatedly entering long expression can be tiring and error-prone. A simple trick of the UNIX shell can eliminate this problem. The selection is stored in a text file and is substituted in-place by the shell. For example, the vibrational subsystem elastic network model (ENM) tool vsa [20] requires a subsystem and an environment selection. Selecting the transmembrane (TM) helices of a G protein-coupled receptor (GPCR) would require seven different residue range selections (i.e., resid >= A && resid <= B). If the subsystem selection was stored in a text file called subsystem, then it could be used in the command line as follows,

```
(`cat subsystem`) && name == 'CA'
```

This combines the TM helices selection with another selection that picks only alpha-carbons. The environment can then be any alpha-carbon not part of the TM helices selection,

```
!(`cat subsystem`) && name =='CA'
```

A unique feature of the selection system used in LOOS is that it is implemented using the "command" design pattern and a minimal virtual machine with the actual "selection" handled by predicate objects. This has two benefits: ease of modification and the storing of selection "programs." Since each operation in the language is encapsulated by an object, it is straightforward to add new operations. Moreover, each selection expression is converted into a set of objects that implement the selection. These can be stored for later use and reused without the overhead of parsing the original selection string. The infrastructure for turning strings into selections as well as the components of the selection system, are exposed to the tool-writer. A selection can be made with a single system call, or it can be built up using the low-level components for efficiency. This is in contrast to environments such as CHARMM, where there is an extensive and powerful selection system, but one that is difficult to access from within FORTRAN.

Extending LOOS

LOOS was designed to make it easy to add support for new file formats. To do so, one simply derives a new class from AtomicGroup and provides an appropriate read function. Adding a new trajectory format is only slightly more complicated. A new class is derived from Trajectory and a handful of functions need to be defined, such as parsing frames, seeking to a frame, returning coordinates, and periodic box information. In both cases, once the code to provide the format-specific functionality is written, very little additional "glue" is needed to integrate with LOOS, such as adding the new object to the factory functions used to read files.

Extending the selection language is also straightforward. Since the selection language is built using the standard UNIX tools lex and yacc, the high-level source for the tokenizer and parser are available and easy to modify. All that remains is to create a new class to handle the corresponding action (derived from the Action class). That said, the parser appears to be feature-complete at this point, in that it contains access to the commonly used metadata currently stored in the Atom datastructure.

Python interface

The C++ language itself can be an impediment to creating new tools for those who are new to programming, due to the

Table 4. Comparing common tasks in C++ and Python with LOOS.

Task	Language	System		
		LfB	Opsin	GPCR-complex
Iteratively Align Structures	C++	3	58	276
	Python	20	80	394
All-to-all RMSD	C++	91	170	484
	Python	148	233	534
Inter-atomic Distance	C++	0	6	37
	Python	0	6	38
Trajectory size (GB)		0.15	2.25	12.19
System size (atoms)		2,746	46,210	276,122
Number of Frames		4,285	4,058	3,678
Selection Size (atoms)		24	205	1,076

All times reported in seconds as average of five runs. The systems used were LfB in water (LfB), Opsin molecule in membrane with water (Opsin), and a cannabiniod receptor dimer bound to cognate G protein in a membrane with water (GPCR-complex). These timing runs were performed using LOOS v2.1.1 on a modern workstation (Intel i7–3770 CPU @ 3.40GHz, 32 GB memory).

complexity of the language (in actuality, a federation of languages[21]), the difficulty in unwinding compilation errors from the volumes of messages compilers typically emit, and the time involved in the write-compile-debug cycle. Python, in contrast, is readily accessible to new programmers, provides many high-quality higher-level constructs and libraries, and requires no explicit compilation step. Exposing the core library and classes of LOOS to Python greatly expands number of people who can create their own custom tools using LOOS and reduces the development time for most common tasks. The Python interface is useful even for experienced programmers who wish to do many common tasks, such as calculating angles, distances, and various distributions, to more complex tasks such as building and inserting peptides into a membrane system.

The Python interface to LOOS (PyLOOS) is implemented using the Simplified Wrapper and Interface Generator (SWIG).[22] Only the core classes are exposed, such as AtomicGroup, Coord, Atom, Trajectory, and required dependent classes. In addition, the PDB and DCDWriter classes are available for writing the respective file types. Many utility functions are available as well, including the selectAtoms function that invokes the selection language. The Boost shared pointers used by LOOS can be transparently used in Python. It should also be noted, for those with a C++ background, that there is a nuance to Python assignments, in that it is in fact setting a reference or alias, not a copy operation. This aspect of Python and PyLOOS is described in more detail in Supporting Information S1.

Some C++ constructs simply do not translate well into Python, such as template classes and policy classes. With a few notable exceptions, such as templates from Boost and STL, we have avoided providing a Python interface to these features of LOOS. Similarly, low level functions and C++ idioms that might be confusing to novice programmers, particularly when translated into Python, have been omitted. Much of this functionality, however, is available through higher level functions, albeit with some loss in performance (see Table 4 for benchmarks). Any loss however is generally offset by the ease of prototyping code in Python and by its accessibility.

In addition to supporting most of the commonly used C++ idioms in PyLOOS, more Python-esque ones are also supported. For example, it is more natural for Python to iterate through a trajectory using a for-loop rather than the while-loop/updateGroupCoords() that is typical in C++, although we do make the latter available as well. PyLOOS includes two higher-level trajectory classes—PyTraj and PyAlignedTraj—that wrap the lower-level LOOS trajectory. Instances of these classes are iterable and can be sliced, like Python arrays and lists. In addition, the set of frames to be used from the trajectory can be configured at instantiation, such as skipping the first frames (equilibration) and skipping frames at each iteration (striding for data reduction). For example,

```
model = createSystem('foo.pdb')
traj = createTrajectory('foo.dcd', model)
ptraj = PyTraj(traj, model, skip = 50, stride = 2)

for frame in ptraj:
    calculate(frame)
```

Here, the trajectory is wrapped in a PyTraj. The first 50 frames are skipped, and every other frame afterward is skipped.

The PyAlignedTraj is similar, except that the entire trajectory is first optimally aligned using an iterative procedure.[23] The trajectory is not altered and not stored in memory; only the transforms needed to align each frame are stored. As each frame is retrieved, it is transformed before being passed to the user code.

Performance is a valid concern when using Python code for MD analysis, particularly considering the layers of code imposed by the SWIG interface between Python and C++. However, in practice, the performance of PyLOOS is quite reasonable, since much of the computation is performed by the underlying C++ library. When more of the work shifts to being done in Python, such as double-loops over computations, PyLOOS performance can suffer.

The relative performance of PyLOOS is illustrated in Table 4, which compares the performance of common tasks using only the C++ library versus using PyLOOS. Three different system sizes are also compared. As the system size increases, proportionally more time is spent within the core C++ library and PyLOOS performance moves toward that of the C++ implementation. For example, in the all-to-all RMSD benchmark, nearly half of the execution time is spent reading in the trajectory, which is dominated by the C++ code. The bulk of the remaining time is spent calculating the least-squares superposition using a C++ member function from AtomicGroup.

Results and Discussion

Bundled tools

Beyond the development libraries, LOOS is also distributed with roughly 140 prewritten tools; most were initially

Table 5. Examples of tools and packages that are included with LOOS.

Core Tools	
aligner	Optimally align trajectory
contact-time	Time-series of atom contacts
density-dist	Electron, mass, or charge density along z-axis
merge-traj	Merge and subsample trajectories
order_parameters	Order parameters analogous to 2H quadrupolar splitting for lipid chains
rdf	Radial distribution function
rmsds	All-to-all RMSD for one or two trajectories
svd	Singular Value Decomposition of a trajectory (PCA)
xy_rdf	RDF in the *x,y*-plane
Convergence Package	
block_average	Block average of arbitrary time-series data
coscon	Cosine content of a trajectory
decorr_time	Decorrelation time of a trajectory
bcom, boot_bcom	Block covariance overlap method for determining convergence and sampling
Density Package	
blobid	Segment a density grid and find noncontiguous blobs of density
grid2xplor	Convert density grid to X-plor electron density map format for visualization
near_blobs	Find residues in a trajectory that are near a blob of density
water-hist	3D histogram of atoms in a trajectory
Elastic Network Models	
anm	Anisotropic network model
enmovie	Visualize ENM motions by generating a trajectory based on an ENM solution
vsa	Vibrational subsystem analysis
Hydrogen Bonds	
hbonds	Find occupancies of putative hydrogen bonds in a trajectory
hcontacts	Time-series of possible intra and intermolecular hydrogen bonds
hcorrelation	Time-correlation of putative hydrogen bonds

developed for internal use in our lab, and were added to the distribution because of their general utility. These tools implement analysis tasks commonly used in macromolecular molecular dynamics for protein and membrane systems. The tools can be divided into a set of core tools, representing the more common tasks, and a set of four packages that cover hydrogen bonding, assessing the convergence of simulations, constructing and analyzing ENMs, and building 3D histograms for visualization. The packages are a logical grouping of more specialized tools. Frequently, these tools also require libraries that would not fit cleanly into the core of LOOS. A subset of the tools is listed in Table 5.

LOOS tends to follow the "Unix Philosophy" in the design of its tools; that is, tools are short, simple, and modular. This is in contrast to monolithic tools such as ptraj/cpptraj[3] and CHARMM.[1] The advantages of the small, modular approach include ease of maintenance, increased flexibility (using any UNIX shell or a high-level language, such as Perl or Python, to combine tools into analysis pipelines), and ready code reuse. Often, a new analytical method is similar to an existing method (tool), and can be written by copying and modifying the existing code. This is a far easier proposition when the tool is focused and is small, because no glue code is needed to integrate the new functionality into a larger package. That said, there are a few exceptions, there multiple options are combined into a single tool. For example, the merge-traj tool not only efficiently combines trajectory fragments into a single large file, it also downsamples, recenters molecules, and fixes issues with periodic imaging. While including this functionality makes the tool somewhat complex, the alternative would be to require several passes to create a cleanly curated trajectory, which would consume significant time and disk space for large data sets.

A drawback to the UNIX philosophy for tools is that they can be difficult to learn, particularly if multiple authors do not use the same command-line options. Internally, LOOS uses a framework to handle the command-line that is built using the strategy design pattern[24] and utilizes ProgramOptions from BOOST to parse the command-line. Common sets of command-line options and arguments are represented by different classes (e.g., ModelWithCoords) that may be combined as needed by the tool. This approach helps to canonicalize how the command-line is handled by most LOOS tools. It is important to note that this framework is primarily for tools distributed and built with LOOS, and is entirely optional for the end-user/programmer. In our lab, we often find that we initially develop tools with a minimal command line interface, and only apply the full program options interface when preparing the tools for public release.

The other challenge with the "many small tools" approach can be figuring out which is the right tool and how to use it. With LOOS, we take a two-tiered approach to this problem. First, the documentation contains a list of all tools included with LOOS (including the packages) with a short description of each, including a reference to the paper describing the method, where appropriate. Second, nearly all tools support the options --help and --fullhelp. The former simply lists all available command line options, but the latter is far more expansive. The full help provides a detailed description of the tool, including use cases, example command lines, descriptions of logical workflows ("use tool A, then take the output and run it through tool B"), alternative tools ("use tool C instead if you want to do X"), and potential pitfalls ("this tool assumes you have already centered the membrane at $z=0$").

Basic tools

Here, we present several typical use-cases of LOOS covering both the core bundled tools and most of the included packages. We will illustrate basic molecular dynamics simulations analysis with LOOS followed by membrane-specific applications. Next, we will examine how LOOS can be used to create ENMs and how these can be compared with MD simulations. We will also describe using LOOS for assessing sample

convergence in simulations. Finally, we will give examples of using the "density" package for visualizing water and lipid density histograms created from simulations.

The typical work-flow for analyzing molecular simulations, particularly on-going ones, involves three steps: merge the latest trajectories into the "master" trajectory, align the system, and calculate something of interest. Simulations, through checkpointing, generally consist of a set of trajectories that grow over time. It is convenient to combine these chunks into a single trajectory for further processing and analyses. As discussed above, the merge-traj tool takes an existing "master" trajectory and the set of source chunks that have been extended with the latest simulation results, and can intelligently append the new results, correctly handling periodic boundary conditions, including molecules broken across the periodic boundary. It is often convenient to remove certain motions from the system, for example, aligning the protein in a soluble system, or putting a membrane's center at $z = 0$; doing so at the point of merging simplifies things, because all future analysis tools can safely assume the coordinates are well curated. The aligner tool provides greater functionality, optimally aligning the trajectory using either an iterative procedure [23] or a reference structure. Additional options are available, such as aligning in x and y only (e.g., to preserve the tilt along a membrane normal aligned with z) and restricting translations in z.

Tools for more of the common analysis tasks are already included with LOOS. For example, many tools will output a timeseries for a particular geometric quantity: the distance between sets of atoms can be found with interdist. Basic measures for shape (e.g., bounding box, radius of gyration, and principal axes) for a user-specified set of atoms can be calculated with molshape. The torsion between the centroids of four sets of atoms can be found with torsion. The χ_1 and χ_2 side chain angles can be found using the rotamer tool, and backbone ϕ and ψ angles can be found with ramachandran. In addition, backbone torsions for nucleic acids, as defined by Wadley et al.[25] are supported. Conformational flexibility can be assessed per residue by the root mean square fluctuations (RMSF) using rmsf. The root mean square difference from a reference structure can be determined with the rmsd2ref tool. The RMSD to the average structure can be easily found by first using averager to find the average, and then rmsd2ref to calculate the RMSD. The RDF is another common analytical method. LOOS includes tools for calculating a RDF two different ways. The first, rdf, takes two selections and treats the selections an groups that can be split into separate residues, molecules, segments, on left as a single unit. The second, atomic-rdf, instead treats each selection as a set or individual atoms rather than groups.

If you want to track the process by which to molecules interact (e.g., a peptide binding to a lipid bilayer), one way to do so it to track the contacts formed between individual residues on the peptide and specific components of the lipids.[26–28] The tool fcontacts will create a matrix of time series showing the fraction of contacts made between a "probe" set of atoms and a series of "target" sets, versus all possible contacts with the probe. The tool contact-time can be used to obtain the raw contact count for desired pairs. Finally, similar to the use of intramolecular contacts as a reaction coordinate for protein folding, native_contacts and transition_contacts count the number of contacts broken and formed over a trajectory and are useful for tracking structural transitions.[29]

Membrane systems

Membrane systems often require special consideration, because unlike soluble macromolecules, they intrinsically break the isotropic symmetry of the simulation: the membrane normal (usually z) is unique. For this reason, there are a host of analysis tasks that are specific to membrane systems. LOOS includes a number of tools designed with these considerations in mind. For example, the lateral structure of a membrane system can be analyzed using the xy_rdf tool, which computes the RDF for one set of atoms about another, but where only the x, y distance is used.[27,28] The distribution of either charge, mass, or electron density can be computed along the z axis (i.e., along the membrane normal) using density_dist. This can be useful for detecting changes in the membrane thickness, or locating molecules (e.g., peptides) depth when attached to a membrane.[26]

A standard measure of membrane structure is the order parameter,

$$S_{CD} = -\frac{1}{2} < 3\cos^2\theta_{CD} - 1 > \quad (1)$$

where θ_{CD} is the angle between the carbon–hydrogen bond and the membrane normal. This quantity is easily calculable from MD membrane simulations and can be determined experimentally by measuring deuterium quadrupolar splitting using solid-state NMR. The LOOS tool to perform this, order_params, includes a novel method for estimating the statistical error in the order parameters: it applies block averaging[30] to the instantaneous value of the order parameter for a given carbon, averaged over the full system; this approach accounts for both temporal and spatial correlations in the data which, when neglected, cause the statistical errors to be dramatically underestimated. An example order parameter plot with estimated errors is shown in Figure 3a. The order parameters are calculated from the palmitoyl chains of POPE (1-palmitoyl-2-oleoyl phosphatidylethanolamine) from a 400 ns all-atom simulation of a neat bilayer system consisting of POPE and POPG (1-palmitoyl-2-oleoyl phosphatidylglycerol) in a 3:1 ratio in solvent.[26]

These order parameters, though useful, are problematic for coarse-grained representations, which lack hydrogens (or even most of the carbons). LOOS includes two tools for these cases. The first, mops, implements a molecular order parameter: for each molecule in a selection, the principal axes are determined. The second and third axes are treated as fake C—H bonds, and used in eq. (1). This generates a single value giving the average molecular order parameter for the trajectory. However, when something is bound or inserted into a membrane, what is often interesting is how the membrane order is perturbed. These changes may be difficult to detect with a global order parameter, especially if the concentration of perturbants is small. The second tool, dibmops, addresses this by binning the molecular order parameter by the lateral distance to the nearest

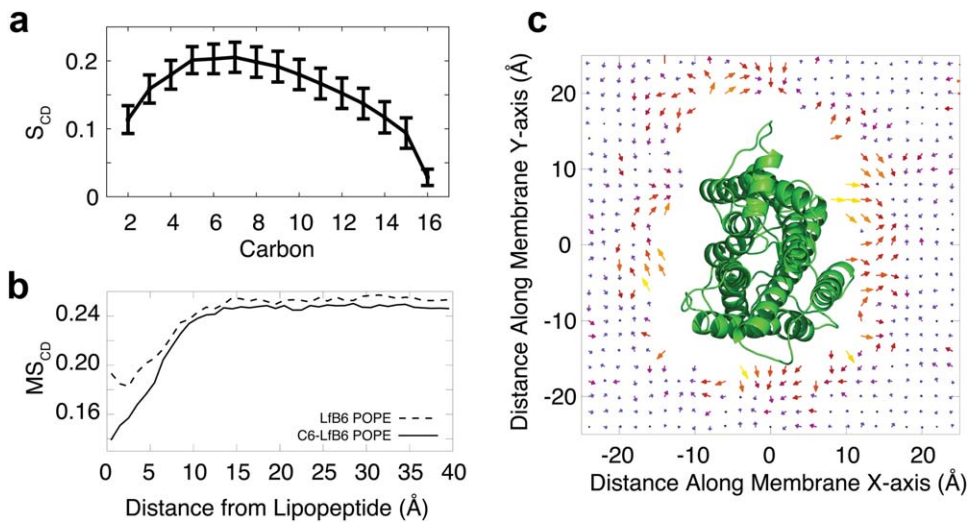

Figure 3. Examples of membrane analysis methods included with LOOS. a) Lipid order parameters with error estimation. b) Distance-based lipid molecular order parameters and c) Lipid orientation map. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

perturbant on the same leaflet. This approach is also useful for all-atom simulations as it can reveal the short-range effects that would have otherwise been lost in the aggregate order parameters. An example of this is shown in Figure 3b, which is taken from an all-atom simulation of a lactoferrin-derived hexapeptide in a bilayer consisting of a 3:1 mixture of POPE:POPG.[26]

Another useful analytical approach for membrane systems is to calculate the two-dimensional distribution of a membrane property about a protein (e.g., the height along the membrane normal, number density, or orientation). This can be represented as a "heat map" of values or a vector field, such as the in-plane "orientation field". The LOOS tool membrane-map implements these calculations. The tool uses a virtual base class to define the interface for calculating membrane properties, with concrete calculators inheriting from it (e.g., densities, molecular order, etc.). This design makes it simple to add new properties to the tool. An example of using the membrane_map is shown in Figure 3c; the data comes from a 1.6 μs all-atom simulation of rhodopsin in a bilayer with explicit water.[31] The vectors show the average orientation of the phosphatidylcholine head groups about the rhodopsin, projected onto the membrane plane (*x*, *y*-plane).

Packages

In addition to the tools bundled in the main distribution, LOOS includes four packages that contain specialized tools and additional libraries useful to the advanced user. Frequently, related tools rely on common code, which is refactored into libraries. However, a fundamental goal of LOOS is to keep the core library and API as simple and compact as possible. Keeping these "package" libraries distinct allows code reuse while maintaining an uncluttered core library. Currently, there are packages for assessing the convergence of MD simulations, constructing and analyzing ENMs, finding hydrogen bonds based on geometrical properties, and the construction of a basic 3D histogram from trajectories. New packages are planned in the future, including Voronoi analysis of membrane systems, calculation of solid state NMR spectra, and construction of membrane-protein systems.

Elastic network modeling

LOOS provides the capacity to calculate and analyze (ENMs)[32] through the "ElasticNetworks" package. ENMs are a method for computing macromolecular fluctuations using a harmonic model. This is done by reducing the system to a network of nodes connected by springs. Such modeling can be done at a number of resolutions, but typically single residues (modeled by the Cα or phosphate position) are used.[33,34] One then solves for the normal modes of motion analytically by diagonalizing the Hessian matrix. Normal mode analysis yields a set of eigenvalues and eigenvectors. Each eigenvector describes a direction of motion that is applied to all atoms in the system. The eigenvalues contain the frequency of its paired eigenvector. These simple models have been shown to predict relevant motions in diverse systems such as HIV reverse transcriptase, GroEL, the ribosome, GPCRs, and many others.[26,35–39] The accuracy of predicted large-scale fluctuations can also be as good as hundred-nanosecond scale all-atom explicit solvent simulations.[37,40,41] These models are quite useful for predicting large-scale (slow) fluctuations and testing many hypotheses very quickly while requiring only modest computational resources. Indeed, normal mode analysis on a 200–400 node network (a typical monomeric protein with one node per residue) can be performed on a modern desktop in about a minute.

The "ElasticNetworks" package contains tools to define connectivity and resolution for an ENM, to perform the normal mode analysis, and to analyze the resulting motions. Several popular ENM parametrizations are already implemented, including the standard distance-based cutoff,[42] the harmonic CA (HCA),[43] and the parameter-free[44] approaches. In addition, the isotropic guassian network model[33] (gnm), anisotropic network model[42] (anm), and vibrational subsystem analysis[20] (vsa) methodologies are all implemented. The resulting eigensets are written as ASCII-formatted matrices that are easily imported

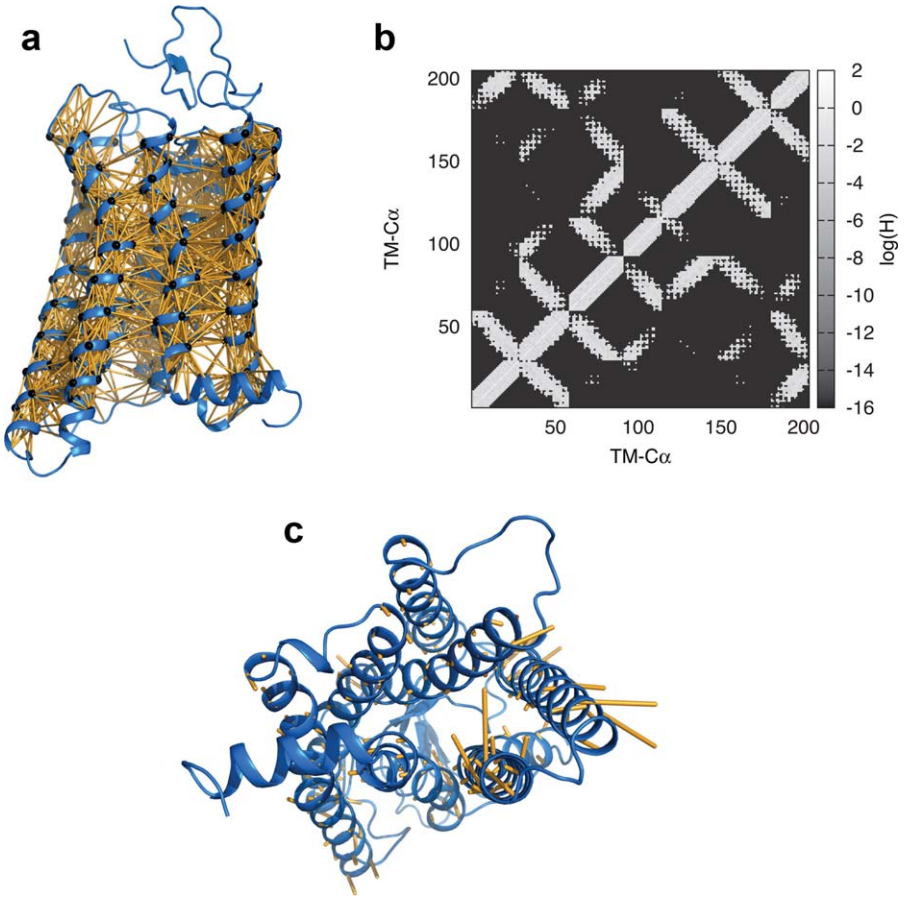

Figure 4. Using ENMs. LOOS provides tools and libraries for normal mode analysis of ENMs. a) Construction of an ENM. The cartoon structure of a protein is shown in blue, with black spheres representing α-carbons. The yellow sticks connecting α-carbons illustrate the springs in a standard distance-cutoff ENM (as defined in Ref. [42]). (b) A representative Hessian matrix of an ENM. Normal model analysis of this matrix yields collective motions (This figure reproduced from Ref. [40]) and c) Reconstruction of a low-frequency motion. The yellow vectors indicate the direction of a given eigenvector (or normal mode). The relative length of these sticks is proportional to each α-carbon's contribution to the mode. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

into Matlab/Octave, Python NumPy, or other LOOS tools for analysis. An object-oriented library is also provided for the rapid development of new parametrizations and new methods. Factory functions are used to control the spring function by the bundled tools, so new spring functions can be used by the existing tools with minimal additional code.

The general workflow for using ENMs in LOOS is shown in Figure 4:

- Select a structure (molecule you are interested in) and spring definition (Fig. 4a)
- Calculate the normal modes via Hessian matrix diagonalization (Fig. 4b). This can be done using any of the three LOOS-bundled tools listed above (gnm, anm, or vsa).
- Analyze the results using tools provided, or written in the language of your choice (Fig. 4c). LOOS libraries may be imported into C++ or Python (see Section "Python interface").

LOOS includes several prebuilt tools for analyzing network model results. For instance, the tool porcupine was used to create the illustration in Figure 4c. Here, the yellow vectors indicate the direction (and relative magnitude) of a particular normal mode. Similarly, the tool enmovie will create a dcd-format "trajectory" where atoms are displaced along a user-specified normal mode; the resulting motions can be easily visualized with standard tools, like VMD. LOOS also includes tools for quantitative ENM analyses, including the prediction of isotropic B-factors (eigenflucc) and comparisons between eigen-spaces (coverlap[45]) coming from both ENMs and principal component analysis results from MD simulations.

Assessing simulation convergence

When discussing MD simulations, the terms "convergence" and "equilibration" are often used imprecisely. What is really of interest to the researcher is the statistical uncertainty in the average of an observable quantity, *f*, that depends on a structural configuration $\vec{x}$. The usual quantity of interest is therefore the standard error, $f(\vec{x})$, computed as

$$\mathrm{SE}(f) = \frac{\sigma_f}{\sqrt{N}} \tag{2}$$

However, this equation only applies to independent and ideal sampling, assuming that σ_f is the standard deviation of $f(\vec{x})$ and that N is the number of samples.[46] In a typical molecular dynamics trajectory, however, there are significant correlations from one frame to the next. This means the number of independent samples is far smaller than the number of snapshots

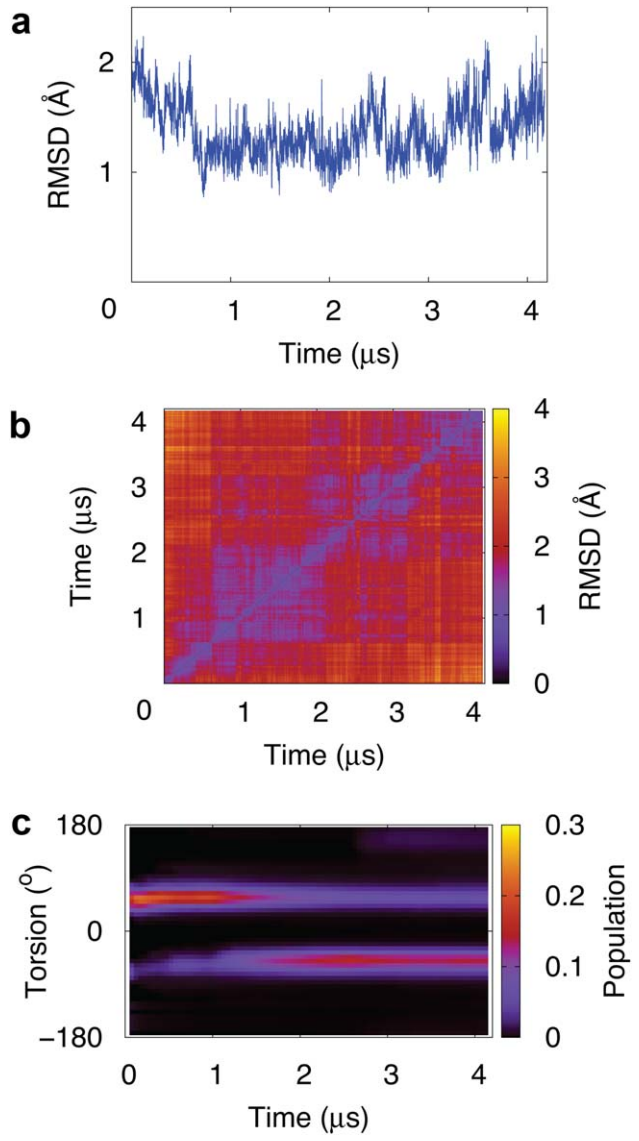

Figure 5. Common methods of assessing quality of sampling using a 4 μs long all-atom rhodopsin simulation. a) RMSD to the average structure. b) All-to-All RMSD. Pairwise comparison of all structures from the dataset. c) Cumulative histogram of ionone ring torsion.

(otherwise, one could reduce statistical error by writing out frames more often). However, knowing just how much smaller *N* is poses a difficult problem.

One simple method is to calculate the autocorrelation time of the observable itself. However, this analysis can be misleading—an apparently fast relaxation can be coupled to a much slower, but relevant process.[23,47,48] To improve on standard methods, several groups have developed techniques that estimate slow relaxation times based on global protein motions.[45,46,49–54] However, in previous work by Romo et al.[54] it was suggested that there is no single ideal analysis, so that the best practice is to use multiple tests, coupled with intuition.

LOOS provides several tools in the core LOOS for a qualitative assessment, along with a "Convergence" package containing more sophisticated methods for a more quantitative analysis. In general, all of these tools are fast, especially when compared to the time required to run a simulation.

The simplest test is to plot the RMSD between the relevant structures from each step in the simulation with their average structure. The LOOS tool rmsd2ref implements this, using an average structure computed from an optimally aligned ensemble, determined by an iterative alignment method.[23] An example of this is shown in Figure 5a for a 4 μs all-atom simulation of rhodopsin. While a divergent plot is a sure sign of problems with the simulation, a "converged" plot is not indicative of a converged simulation, as the ensemble of structures within a given RMSD radius can be quite large. A better test is the all-to-all RMSD plot (Fig. 5b), where every snapshot is compared to every other[49,55] using the rmsds tool. Conformational substates (ensembles of similar structures) appear as blocks along the diagonal. When substates are revisited, off-diagonal blocks appear. The presence of blocks along the diagonal and few, if any, off-diagonal is a good indication that the relative populations of the different states are not well sampled. A similar qualitative approach is to plot the phase-space projection of the trajectory onto the first few principal component modes.[55,56] The svd tool computes the singular value decomposition (SVD)[56] of the trajectory (equivalent to principal component analysis). The left singular vectors (LSVs) are the eigenvectors, or principal components, of the system while the right singular vectors (RSVs) are the projections of the trajectory along the corresponding LSVs. Plotting the first few RSVs can show the presence of substates and their transitions, or lack thereof. The phase-pdb tool creates a fake structure where each projection point in the phase space formed from three modes, is an atom and all points are connected by bonds. This phase portrait can be viewed in 3D in most molecular visualization programs (e.g., PyMOL, VMD, etc.). Here, substates and their transitions appear as "beads on a string" in the case of poor sampling, and multilobed "furrballs" with better sampling.[57]

Another simple test, particularly when a single quantity is of interest, is to examine how its distribution changes over the simulation. This can be found by calculating a set a histograms for the first *n* samples from the trajectory, where *n* increases from a small value up to the length of the trajectory. Alternatively, the histogram can be constructed from a window slid along the trajectory. The LOOS tool chist implements these methods, giving output suitable for graphing with gnuplot. Figure 5c shows the change in the distribution of the torsion for the ionone ring in the retinal from the same 4 μs all-atom simulation used previously, using the first method in chist. It is distressingly clear that it can take a very long time for the distribution of even a simple quantity to converge when it is coupled to slower modes.

More quantitative methods for assessing sampling and convergence can be found in the "Convergence" package. For example, Hess showed that the time course along principal components for a randomly diffusing system with high dimensionality appeared very cosine-like.[45,49] When a system is poorly sampled, the RSVs for the lowest-frequency, largest-amplitude modes, will resemble cosines, a property he quantified as the cosine content. The LOOS tool (coscon) calculates this quantity (see Ref. [45], eq. 12) using the simulation's principal components. This is done using a block-averaging approach so that the quantity can be tracked as a function of simulation time.

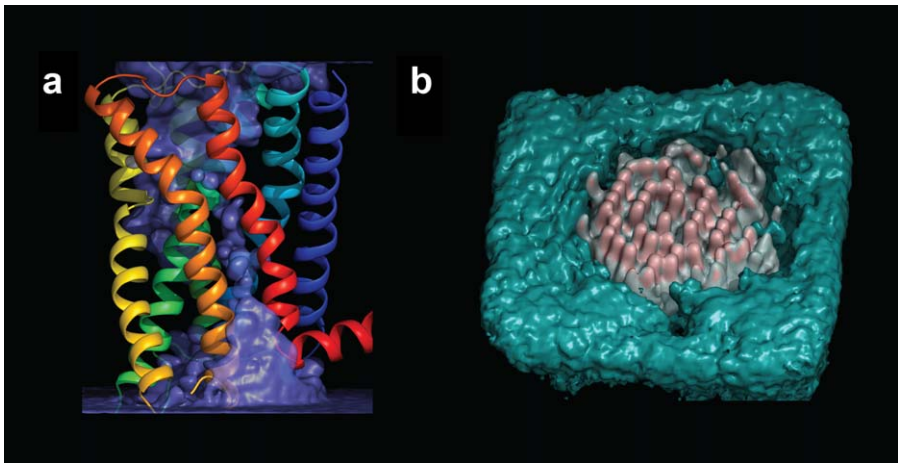

Figure 6. Examples of density calculated using LOOS and rendered with PyMol. a) Water density inside of β_2AR contoured at bulk density[59] b) Membrane lipid density beneath a lipopeptide micelle (not shown). POPE lipids colored white (bulk lipid contour) and red (double bulk), and POPG lipids colored cyan (bulk).[28]

Another approach was developed by Lyman and Zuckerman, using structural histograms to calculate an effective sample size.[52,53] The effsize.pl tool provides a Perl front-end to drive this analysis. This effective sample size estimates the number of statistically independent conformations in a simulation by analyzing the variance in state populations, using randomly defined structural states and increasingly large samples from the trajectory. In addition, the decorrelation time method, also by Lyman and Zuckerman, is implemented in LOOS.[53]

Finally, we've implemented the block covariance overlap method developed by Romo and Grossfield.[54] This analysis combines a block-averaged[30] calculation of the covariance overlap [45] with a bootstrapping approach.[54] The Perl front-end (bootstrap_overlap.pl) will completely run the calculation. Briefly, the algorithm takes principal components of subsets of the trajectory and compares them to PCA results for the full simulations using the covariance overlap. This quantity is normalized by another covariance overlap comparison where bootstrapped trajectories are created by randomly pulling frames from the full-length simulation.

Density

The "Density" package contains a number of tools for calculating 3D histograms from trajectories for visualization. These histograms are written out as X-plor formatted electron density maps[58] suitable for display in most molecular graphics programs (e.g., PyMOL and VMD). Tools for segmenting the density, extracting contiguous regions (blobs), and associating these with the model system are also included. This tool suite was originally designed for visualizing water distributions within membrane proteins, and as such provides options for determining which water molecules are to be considered "internal." One method is to determine the first principal axis for the membrane protein and then accept any water molecule within a given radius of this axis. Simple methods, such as waters within a given radius of any protein atom, are also available. As it is often useful to scale the densities relative to the bulk solvent, the histogramming tool allows the user to specify a horizontal slice through the trajectory for determining the bulk solvent density. An example of visualizing the average water density inside a 1μs long simulation of β_2AR is shown in Figure 6a.[59]

As LOOS is agnostic to chemistry (an atom is like every other atom), there are no restrictions on what can be treated as "water" and "protein" by the density tools. For example, a ligand could be selected as "water." Membrane lipids can also be used. Figure 6b shows an example of visualizing the "crystallization" of one membrane lipid type induced by a bound lipopeptide micelle using a coarse-grained simulation.[28]

Distribution

LOOS is freely distributed as source code under the GNU GPLv3 license[60] through SourceForge (http://loos.sourceforge.net). The ability to build LOOS is tested on most major Linux distributions including Debian, Ubuntu, and Fedora, along with multiple versions of each distribution. In addition, MacOS is supported, as is Windows (via cygwin). In total, 20 different operating systems and releases are tested and supported. Internally, nightly regression tests are run using common tool invocations on multiple datasets. All documentation is generated using the Doxygen[61] tool, and is available with the distribution as well as on-line through Sourceforge.

Conclusions

It is clear that addressing the needs of modern simulation analysis requires more than providing a well-rounded suite of tools. Empowering the researcher with the capacity to quickly and easily create new analytical tools is a necessity, and to that end, a clean, simple, and easy to learn API is critical. LOOS strives to provide just such an API. This design often results in C++ code that resembles a high level scripting language. In addition, including Python bindings has enabled more researchers to be able to quickly develop new tools, either by leveraging the many high-quality scientific libraries available for Python, or by eliminating the write-compile-debug cycle inherent to C++. Indeed, the latter feature allows the researcher to use LOOS to create spur of the moment, disposable scripts. These are particularly useful when developing

new ideas and methods both for testing and for determining what implementation method may work best.

LOOS is not merely a library, however. The distribution includes a number of tools that have general applicability, or tools that we have found useful in our own research. The majority of the tools are geared toward a single task and intended to be combined into a work-flow via a scripting system such as the UNIX shell or Python, although they may also be used interactively through a shell. In contrast to the monolithic programs, such as CHARMM and VMD, the LOOS tools are in many ways, a continuation of the "Unix Philosophy" of small tools that are focused on a single task.

Keywords: molecular dynamics · analysis · software · membranes · convergence

How to cite this article: T. D. Romo, N. Leioatts, A. Grossfield, *J. Comput. Chem.* **2014**, *35*, 2305–2318. DOI: 10.1002/jcc.23753

Additional Supporting Information may be found in the online version of this article.

[1] B. Brooks, R. Bruccoleri, B. Olafson, D. States, S. Swaminathan, M. Karplus, *J. Comput. Chem.* **1983**, *4*, 187.
[2] D. A. Case, T. E. Cheatham, T. Darden, H. Gohlke, R. Luo, K. M. Merz, A. Onufriev, C. Simmerling, B. Wang, R. J. Woods, *J. Comput. Chem.* **2005**, *26*, 1668.
[3] D. R. Roe, T. E. Cheatham, *J. Chem. Theory Comput.* **2013**, *9*, 3084.
[4] N. M. Glykos, *J. Comput. Chem.* **2006**, *27*, 1765.
[5] P. I. Koukos, N. M. Glykos, *J. Comput. Chem.* **2013**, *34*, 2310.
[6] N. Michaud-Agrawal, E. J. Denning, T. B. Woolf, O. Beckstein, *J. Comput. Chem.* **2011**, *32*, 2319.
[7] W. L. DeLano, The Pymol Molecular Graphics System, Available at: http://www.pymol.org., Pymol v1.7.3.0 (svn r4093).
[8] W. Humphrey, A. Dalke, K. Schulten, *J. Mol. Graph.* **1996**, *14*, 33.
[9] J. C. Jeong, S. Jo, E. L. Wu, Y. Qi, V. Monje-Galvan, M. S. Yeom, L. Gorenstein, F. Chen, J. B. Klauda, W. Im, *J. Comput. Chem.* **2014**, *35*, 957.
[10] BOOST C++ Libraries, Available at: http://www.boost.org. Accessed on April 26, **2014**.
[11] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, D. Sorensen, In Supercomputing '90: Proceedings of the 1990 Conference on Supercomputing; IEEE Computer Society Press: Los Alamitos, CA, USA, **1990**; pp. 2–11.
[12] J. J. Dongarra, J. J. Dongarra, J. D. Croz, J. D. Croz, S. Hammarling, S. Hammarling, R. J. Hanson, R. J. Hanlson, *ACM Trans. Math. Softw.* **1988**, *14*, 1.
[13] J. J. Dongarra, J. D. Croz, S. Hammarling, I. Duff, *ACM Trans. Math. Softw.* **1990**, *16*, 1.
[14] R. C. Whaley, J. Dongarra, In SuperComputing 1998: High Performance Networking and Computing, Orlando, FL, **1998**, (in CD-ROM Proceedings).
[15] T. E. Ferrin, C.-C. Huang, L. E. Jarvis, R. Langridge, *J. Mol. Graph.* **1988**, *6*, 13.
[16] D. van der Spoel, E. Lindahl, B. Hess, G. Groenhof, A. E. Mark, H. J. C. Berendsen, *J. Comput. Chem.* **2005**, *26*, 1701.
[17] S. E. Feller, K. Gawrisch, A. D. Mackerell, *J. Am. Chem. Soc.* **2002**, *124*, 318.
[18] J. B. Klauda, B. R. Brooks, A. D. Mackerell, R. M. Venable, R. W. Pastor, *J. Phys. Chem. B* **2005**, *109*, 5300.
[19] J. B. Klauda, R. M. Venable, J. A. Freites, J. W. O'Connor, D. J. Tobias, I. Vorobyov, C. Mondragon-Ramirez, R. W. Pastor, A. D. J. MacKerell, *J. Phys. Chem. B* **2010**, *114*, 7830.
[20] H. L. Woodcock, W. Zheng, A. Ghysels, Y. Shao, J. Kong, B. R. Brooks, *J. Chem. Phys.* **2008**, *129*, 214109.
[21] S. Meyers, Effective C++: 55 Specific Ways to Improve Your Programs and Designs, 3rd ed.; Addison-Wesley Professional: Upper Saddle River, NJ, **2005**.
[22] SWIG-3.0. Available at: http://www.swig.org/Doc3.0/index.html. Accessed on April 21, **2014**.
[23] A. Grossfield, S. E. Feller, M. C. Pitman, *Proteins: Struct. Funct. Bioinf.* **2007**, *67*, 31.
[24] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1st ed.; Addison-Wesley Professional: Boston, MA, **1994**.
[25] L. M. Wadley, K. S. Keating, C. M. Duarte, A. M. Pyle, *J. Mol. Biol.* **2007**, *372*, 942.
[26] T. D. Romo, L. A. Bradney, D. V. Greathouse, A. Grossfield, *Biochim. Biophys. Acta* **2011**, *1808*, 2019.
[27] J. N. Horn, T. D. Romo, A. Grossfield, *Biochemistry* **2013**, *52*, 5604.
[28] J. N. Horn, J. D. Sengillo, D. Lin, T. D. Romo, A. Grossfield, *Biochim. Biophys. Acta* **2012**, *1818*, 212.
[29] N. Leioatts, P. Suresh, T. D. Romo, A. Grossfield, *Proteins: Struct. Funct. Bioinf.* **2014**, *82*, 2538.
[30] H. Flyvbjerg, H. Petersen, *J. Chem. Phys.* **1989**, *91*, 461.
[31] A. Grossfield, M. C. Pitman, S. E. Feller, O. Soubias, K. Gawrisch, *J. Mol. Biol.* **2008**, *381*, 478.
[32] M. Tirion, *Phys. Rev. Lett.* **1996**, *77*, 1905.
[33] I. Bahar, A. R. Atilgan, B. Erman, *Fold. Des.* **1997**, *2*, 173.
[34] I. Bahar, R. L. Jernigan, *J. Mol. Biol.* **1998**, *281*, 871.
[35] W. Zheng, B. R. Brooks, D. Thirumalai, *Biophys. J.* **2007**, *93*, 2289.
[36] A. Bakan, I. Bahar, *Proc. Natl. Acad. Sci. USA* **2009**, *106*, 14349.
[37] N. Leioatts, T. D. Romo, A. Grossfield, *J. Chem. Theory Comput.* **2012**, *8*, 2424.
[38] J. M. Seckler, N. Leioatts, H. Miao, A. Grossfield, *Proteins: Struct. Funct. Bioinf.* **2013**, *81*, 1792.
[39] F. Tama, M. Valle, J. Frank, C. L. Brooks, *Proc. Natl Acad. Sci. USA* **2003**, *100*, 9319.
[40] T. D. Romo, A. Grossfield, *Proteins: Struct. Funct. Bioinf.* **2011a**, *79*, 23.
[41] L. Liu, A. M. Gronenborn, and I. Bahar, *Proteins: Struct. Funct. Bioinf.* **2011**, *80*, 616.
[42] A. R. Atilgan, S. R. Durell, R. L. Jernigan, M. C. Demirel, O. Keskin, I. Bahar, *Biophys. J.* **2001**, *80*, 505.
[43] K. Hinsen, A. Petrescu, S. Dellerue, *Chem. Phys.* **2000**, *261*, 25.
[44] L.-W. Yang, G. Song, R. L. Jernigan, *Proc. Natl. Acad. Sci. USA* **2009**, *106*, 12347.
[45] B. Hess, *Phys. Rev. E* **2002**, *65*, 031910.
[46] A. Grossfield, D. M. Zuckerman, *Annu. Rep. Comput. Chem.* **2009**, *5*, 23.
[47] C. Neale, W. F. D. Bennett, D. P. Tieleman, R. Pomès, *J. Chem. Theory Comput.* **2011**, *7*, 4175.
[48] T. D. Romo, A. Grossfield, *Biophys. J.* **2014**, *106*, 1553.
[49] B. Hess, *Phys. Rev. E Stat. Phys. Plasmas Fluids Relat. Interdiscip. Topics* **2000**, *62*, 8438.
[50] L. J. Smith, X. Daura, W. F. van Gunsteren, *Proteins: Struct. Funct. Bioinf.* **2002**, *48*, 487.
[51] J. D. Faraldo-Gómez, L. R. Forrest, M. Baaden, P. J. Bond, C. Domene, G. Patargias, J. Cuthbertson, M. S. P. Sansom, *Proteins: Struct. Funct. Bioinf.* **2004**, *57*, 783.
[52] E. Lyman, D. M. Zuckerman, *Biophys. J.* **2006**, *91*, 164.
[53] E. Lyman, D. M. Zuckerman, *J. Phys. Chem. B* **2007**, *111*, 12876.
[54] T. D. Romo, A. Grossfield, *J. Chem. Theory Comput.* **2011b**, *7*, 2464.
[55] A. E. García, *Phys. Rev. Lett.* **1992**, *68*, 2696.
[56] T. D. Romo, J. B. Clarage, D. C. Sorensen, G. N. Phillips, *Proteins: Struct. Funct. Bioinf.* **1995**, *22*, 311.
[57] J. B. Clarage, T. D. Romo, B. K. Andrews, B. M. Pettitt, G. N. Phillips, *Proc. Natl. Acad. Sci. USA* **1995**, *92*, 3288.
[58] A. T. Brünger, X-PLOR Version 3.1: A System for X-ray Crystallography and NMR; Yale University Press: New Haven, CT, **1992**.
[59] T. D. Romo, A. Grossfield, M. C. Pitman, *Biophys. J.* **2010**, *98*, 76.
[60] Gnu General Public License. Available at: http://www.gnu.org/licenses/gpl.html, GPL v3 (6-29-2007).
[61] D. van Heesch, Doxygen, Available at: http://www.stack.nl/dimitri/doxygen/, 1.8.3.1.

Received: 13 August 2014
Accepted: 7 September 2014
Published online on 18 October 2014