LOOS: A Tool for Making New Tools for Analyzing Molecular Simulations

Tod D. Romo and Alan Grossfield

University of Rochester Medical School, Rochester, NY, USA

http://loos.sourceforge.net

Abstract

We have developed LOOS (Lightweight Object Oriented Structure-analysis) as a tool for making new tools for analyzing molecular simulations. LOOS is an object-oriented library designed to facilitate the rapid development of new methods for structural analysis. LOOS includes over 120 pre-built tools for common structural analysis tasks. LOOS supports reading the native file formats of most common simulation packages and can write NAMD formats (PDB and DCD). A dynamic atom selection language, based on the C expression syntax, is included as part of the library and is easily accessible to both the programmer and the end user. LOOS is written in C++ and makes extensive use of Boost and the Standard Template Library. Through modern C++ design, LOOS is both simple to use (requiring knowledge of only 4 core classes and a few utility functions) and easily extensible. A Python interface to the core components of LOOS is also available, further facilitating rapid development of analysis tools and broadening the LOOS community by making it accessible to those who would otherwise be deterred by using C++. LOOS also includes a set of libraries and tools for performing elastic network model calculations that are easily extended to accommodate new methods.

Bundled Tools

Over 120 tools total, including 4 packages and 60 core tools

Sample of tools included in LOOS

Core Tools	
aligner	Optimally align trajectory
contact-time	Time-series of atom contacts
density-dist	Electron, mass, or charge density along the z-axis
merge-traj	Merge & subsample trajectories
order_parameters	Order parameters analogous to ^{2}H quadrupolar splitting for lipid chains
rdf	Radial distribution function
rmsds	All-to-all RMSD
svd	Singular Value Decomposition of a trajectory (PCA)
xy_rdf	Radial distribution function 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 & sampling
Density Package	
blobid	Segment a density grid and find non-contiguous blobs of density
grid2xplor	Convert density grid to X-plor electron density map format for visualization
gridmask	Apply a binary mask to a density grid
near_blobs	Find residues that are near a blob for a trajectory
water-hist	Density histogram for atoms in a trajectory
Elastic Network Models	
anm	Anistropic Network Model
enmovie	Visualize ENM motions by generating a trajectory for 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 inter-molecular hydrogen bonds
hcorrelation	Time-correlation of putative hydrogen bonds

Design Goals

- **Lightweight**
  - Tool developers only need 4 core classes: `Coord`, `Atom`, `AtomicGroup`, `Trajectory`
  - Few external dependencies: Boost, scons, atlas/lapack SWIG (optional)
- **Extensible**
  - Polymorphic classes
  - Algorithm encapsulation
  - Design patterns for easy extension
- **Powerful**
  - Rich atom selection language
  - Parser built using standard Unix tools
  - Many useful member functions
  - Shared data via Boost shared pointers
    - Simplifies memory management
    - Copies are lightweight
  - Standard Template Library support
  - Support for basic periodicity
- **Easy to use**
  - Complex tools with minimal code
  - *Python interface to core library*
  - Rapid development of new tools
  - Templates for writing new tools for common cases
  - Tools are self-documenting (embedded detailed help)
  - Consistent command line options across tools
- **Multiplatform Support**
  - Linux
  - MacOS X
  - Windows (cygwin)
- **Multipackage Support**
  - CHARMM/NAMD
  - Amber (including NetCDF)
  - GROMACS/MARTINI
  - Tinker
  - Easy to extend

Selection Language

- Based upon C/C++ expressions
- Built using lex & yacc
- Available via function call for all tools
- Select atoms via atom metadata
- Keywords bound to atom properties
  `-id, name, resname, resid, segid`
- Special keywords
  `-all, none, hydrogen`
- Substring and pattern matching via regular expression operator `=~`
- Number extraction operator `->`
- Complex selections can be stored in a file and used via shell substitution
- Convenience functions in `AtomicGroup` reduce selection complexity: `splitByResidue()`, `splitByMolecule()`,...

Select non-hydrogen atoms

```
! hydrogen
```

Select CA atoms

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

Select backbone atoms

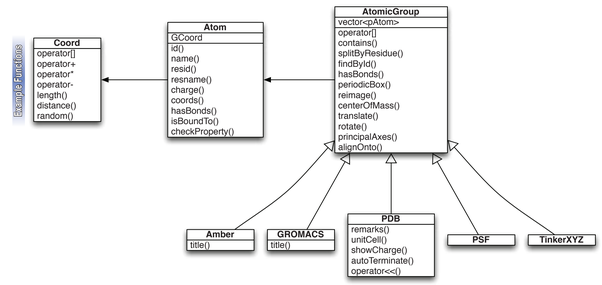
```
name =~ "^(C|O|N|CA)$"
```

Select heavy atoms from a range of residues

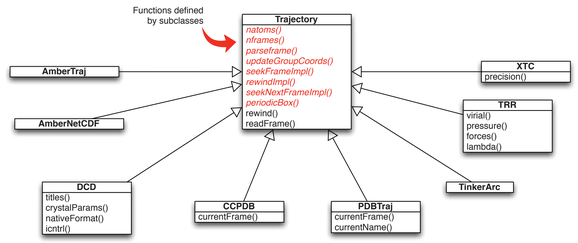
```
(resid >= 10 && resid <= 20) && !hydrogen
```

Class Structures

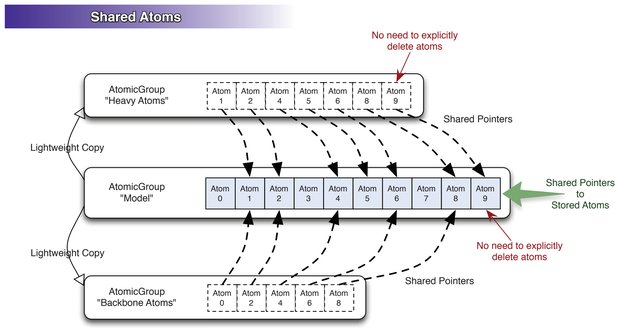
Defining The System (Model)



Trajectories



Shared Atoms



Example Code

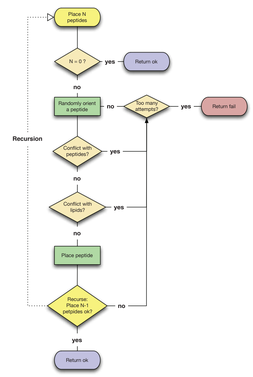
RMSD to Average Structure

```python
from loos import *
import sys

# Read in model and setup trajectory interface
model = createSystem(model_name)
traj = createTrajectory(traj_name, model)

# Select atoms user is interested in
subset = selectAtoms(model, selection)

# Read in entire trajectory
# (but only atoms of interest)
ensemble = AtomicGroupVector()
readTrajectory(ensemble, subset, traj)

# Find optimal alignment
iterativeAlignmentPy(ensemble)
# Compute average structure
average = averageStructure(ensemble)

t = 0
avg_rmsd = 0
# Loop over all stored frames of the trajectory
for structure in ensemble:

    # Align frame onto the optimal average
    structure.alignOnto(average)
    # Compute the RMSD
    rmsd = average.rmsd(structure)
    avg_rmsd = avg_rmsd + rmsd
    print "%d\t%f" % (t, rmsd)
    t = t + 1

print "# Average rmsd = %f" % (avg_rmsd/t)
```

```cpp
#include <loos.hpp>

using namespace loos;
using namespace std;

int main() {
  // Read in model and setup trajectory
  AtomicGroup model = createSystem(model_name);
  pTraj traj = createTrajectory(traj_name, model);

  // Select atoms user is interested in
  AtomicGroup subset = selectAtoms(model, selection);

  // Read in entire trajectory
  // (but only atoms of interest)
  vector<AtomicGroup> ensemble;
  readTrajectory(ensemble, subset, traj);

  // Find optimal alignment
  iterativeAlignment(ensemble);
  // Compute average structure
  AtomicGroup average = averageStructure(ensemble);

  double t = 0.0;
  double avg_rmsd = 0.0;
  // Loop over all stored frame of the trajectory
  for (vector<AtomicGroup>::iterator i = ensemble.begin();
       i != ensemble.end();
       ++i)
    {
      AtomicGroup structure = *i;
      // Align frame onto the optimal average
      structure.alignOnto(average);
      // Compute the RMSD
      double rmsd = average.rmsd(structure);
      avg_rmsd += rmsd;
      cout << t++ << "\t" << rmsd << endl;
    }

  cout << "# Average rmsd = " << avg_rmsd/t << endl;
}
```

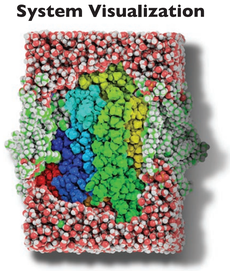
Recursively Insert Peptides

```python
# Recursively place peptides
#   npeptides_remaining = how many to place
#              peptides = AtomicGroup containing placed peptides
#               handler = PeptideHandler object that interfaces into the peptide library
#                lipids = AtomicGroup containing lipids
#               box_min = GCoord for one corner of the system bounding box (NOT the periodic box)
#               box_max = GCoord for the other corner of the system bounding box
#          periodic_box = periodic box for the system
#                  fout = file object for PDB output (info about placed peptides are
#                         written into REMARK records)

def placePeptides(npeptides_remaining, peptides, handler, lipids, box_min,
                  box_max, periodic_box, fout):

    if (npeptides_remaining <= 0):
        return(1)

    box_size = box_max - box_min

    for j in range(max_iterations):

        # Randomly pick a peptide and place it...
        (putative_id, putative) = handler.getRandom()
        (x, a, b, c) = positionPeptide(putative, box_size, box_min, z_loc)

        # Conflict with existing peptides?
        violations = peptides.within(peptide_min_distance, putative, periodic_box)
        if (violations.size() > 0):
            continue

        # Conflict with lipids?
        violations = lipids.within(min_distance, putative, periodic_box)
        if (violations.size() > 0):
            continue

        # Append peptide
        setSegidForPeptide(putative, npeptides_remaining-1)
        peptides.append(putative)

        # Now try to recurse
        ok = placePeptides(npeptides_remaining - 1, peptides, handler, lipids,
                           box_min, box_max, periodic_box, fout)

        # If all subsequent peptides were placed,
        # then accept this placement and return
        if (ok):
            return(1)

        # Failed to place, so must undo
        peptides.remove(putative)
        print "\tCannot place sub-peptide"

    return(0)
```



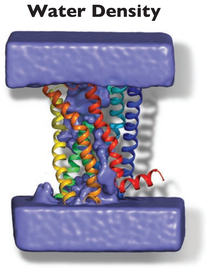
Made with LOOS

System Visualization



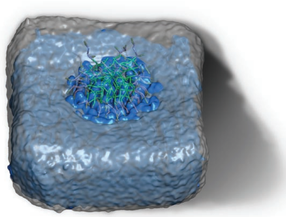
Tools: custom software

Water Density



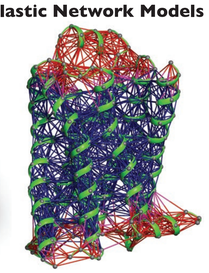
Tools: water-hist

Lipid Density



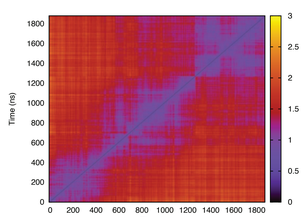
Tools: water-hist

Elastic Network Models



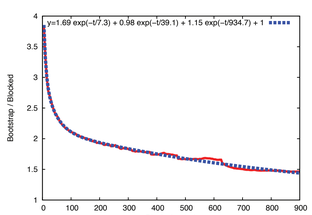
Tools: ANM/VSA (rebond)

Simulation Convergence All-to-All RMSD



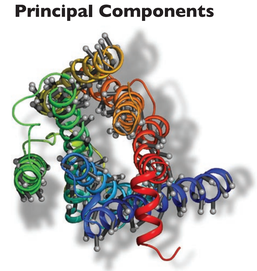
Tools: rmsds

Simulation Convergence Block Covariance Overlap Method



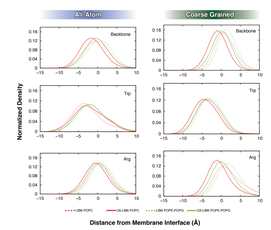
Tools: bootstrap_overlap.pl

Principal Components



Tools: svd & porcupine

Peptide Location in Membrane



Tools: density-dist & custom software