Contents

List of Figures

List of Tables

Appendix

This code is designed to efficiently simulate the wormlike chain polymer model using various coarse-grainings where applicable.

For very stiff polymers, the usual wormlike chain is simulated.

For relatively more flexible polymers, the "stretchable, shearable" chain is used.

For *VERY* stretchable polymers, a purely Gaussian chain is used.

To Run

Simply typing `make` in the top level directory will build the simulator from source. The executable created (`wlcsim.exe`) will use the parameters in the file `input/input` and write its output to the `data` directory. Descriptions of the available parameters can be found at their definitions and where they are read in `src/wlcsim/params.f03`. Example input files are usually more useful, and can be found in the `input` directory. To force a rerun without having to manually delete all the old output files, you can also simply type `make run` at any time.

There are several ways to easily visualize simulation output. There are PyMol scripts in the `vizualization` directory, `python -m wlcsim.plot_wlcsim` from the repo's top level directory will launch a GUI designed to visualize BD simulations, and one can of course simply use the output in the `data` directory, which contains rank two arrays of shape `num_beads*num_polymers-by-3`, with one file per time point. By default, specifying multiple polymers just simulates them in parallel in the same reaction volume, no interactions are assumed.

To scan parameters, the Python script `scan_wlcsim.py` should be used. It takes care of saving the current git commit_hash, all inputs, etc. into a unique directory, and preventing race conditions even on shared filesystems, among other things.

Disclaimer

This codebase is internal to the Spakowitz lab and is not guaranteed to be bug-free at any point. For battle-tested versions of our software, please see the links in the relevant papers.

Contents:

Introducing the WormLike Chain SIMulator This code is designed to efficiently simulate the wormlike chain polymer model using various coarse-grainings where applicable to decrease run time.

For very stiff polymers, the usual wormlike chain is simulated.

For relatively more flexible polymers, the "stretchable, shearable" chain is used.

For *VERY* flexible polymers, a purely Gaussian chain is used.

Quickstart

Simply typing `make` in the top level directory will build the simulator from source. The executable created (`wlcsim.exe`) will use the parameters in the file `input/input` and write its output to the `data` directory. To force a rerun without having to manually delete all the old output files, you can also simply type `make run` at any time.

Several tools for vizualizing the output are included. For simply plotting polymer configurations, see the Python modules described in *Plotting Simulation Output*. Plotting can also be done by hand using the output in the `data` directory, which contains rank two arrays of shape `num_beads*num_polymers-by-3`, with one file per time point. Specifying multiple polymers just simulates them in parallel in the same reaction volume, no interactions are assumed. For more complex analysis, see *Analyzing Simulation Output*.

To scan parameters, the Python script `scan_wlcsim.py` should be used. It takes care of saving the current git commit_hash, all inputs, etc. into a unique directory, preventing race conditions even on shared filesystems. For more details, see *Running Simulations*.

Disclaimer

This codebase is internal to the Spakowitz lab and is not guaranteed to be bug-free at any point. For battle-tested versions of our software, please see the links in the relevant papers.

Running Simulations It is recommended that you always use the scan_wlcsim.py script to run your simulations. This will make it easy to leverage the *wlcsim* functionality in your analysis, and keep track of lots of easy-to-forget things for you, like saving the version of the code used, the parameters used, random seeds, etc.

To use scan_wlcsim.py, simply fill out the input file, input/input, with the default parameters you want to use, then follow the instructions in scan_wlcsim.py to fill out the parameters that you want to scan, how many

repeats of each parameter, and what name you want to give to the simulation run.

Analyzing Simulation Output Assuming you've used the *wlcsim* module for running your simulations, whether they be parameter scans or simple replicas, there are numerous classes and helper functions in the *wlcsim.data* for loading the output, and functions in the module `wlcsim.analysis` for analyzing the data once it's been loaded.

wlcsim Python Module

wlcsim.data

This module "understands" wlcsim.exe's output, and will make it into nice, pretty numpy/pandas arrays for you.

class `wlcsim.data.Scan`(*run_dir*, *limit=None*, **args*, ***kwargs*)

Can tell you what dirs in a run_dir are simulation directories, and what all the input files said, and more.

`rend2end`

Last time point for each simulation as pandas array

class `wlcsim.data.Sim`(*sim_path*, *overwrite_coltimes=False*, *load_method=3*)

Sim provides an interface to the heterogenous data produced by a simulation that saves coltimes, positions, and momentums.

Unless specified during initialization, the coltimes, r, and u are loaded on demand upon first access.

Accessing properties of this class for the first time can return an IOError if a file is moved, or doesn't exist. Afterwards, the values will be cached.

`add_useful_cols_to_coltimes`(*coltimes*)

Only works on coltimes that come from combine_coltimes_csvs.

`polymer_length`

The sum of interbead distances in the polymer at each time

`r`
: Numpy array of positions of each particle at each time point.

`r0`
: Numpy array of positions of each particle pre-first time point.

`rend`
: Numpy array of positions of each particle pre-first time point.

`u`
: Numpy array of positions of each particle at each time point.

`u0`
: Numpy array of positions of each particle at each time point.

`wlcsim.data.add_useful_cols_to_coltimes(`*coltimes*`)`
: Only works on coltimes that come from combine_coltimes_csvs.

`wlcsim.data.combine_coltimes_csvs(`*run_dir*, *conditionals=[]*, *outfile_name='coltimes.csv'*`)`
: takes a directory that scan_wlcsim.py has output to and an optional list of functions that takes a coltimes array and decides whether it may be used, and an output location

`wlcsim.data.combine_initial_dists_csvs(`*run_dir*, **args*, ***kwargs*`)`
: Not yet implemented.

`wlcsim.data.nan_fill_coltimes(`*coltimes*`)`
: Takes a raw coltimes pandas array from e.g. write_coltimes_csvs and puts nan's where the -1.0's are, i.e. where the simulation did not go out far enough.

`wlcsim.data.pairwise_distances3`
: Returns np.array of pairwise distances of points in R^3 stored in a 3-by-num_points numpy array.

wlcsim.input

This module "understands" the input format of wlcsim.exe

class `wlcsim.input.InputFormat`

An enumeration.

`_member_type_`

alias of `object`

class `wlcsim.input.ParsedInput`(*input_file=None*, *params=None*)

Knows how to handle various input file types used by wlcsim simulator over the years, and transparently converts into new parameter naming conventions.

input = ParsedInput(file_name) print(input.ordered_param_names) # see params in order defined print(input.ordered_param_values) # to see values input.write(outfile_name) # write clone of input file

`decide_input_format()`

Decide between the two input formats we know of. Not too hard, since one uses Fortran-style comments, which we can look out for, and the other uses bash style comments. Further, the former specifies param names and values on separate lines, while the latter specifies them on the same line.

`parse_lena_params_file()`

Lena-style input files have comment lines starting with a "#". Any other non-blank lines must be of the form "[identifier][whitespace][value]", where an identifier is of the form "[_a-zA-Z][_a-zA-Z0-9]*", and a value can be a boolean, float, int or string. They will always be stored as strings in the params dictionary for downstream parsing as needed.

Identifiers, like fortran variables, are interpreted in a case-insensitive manner by the wlcsim program, and so will be store in all-caps within the ParsedInput to signify this.

`parse_original_params_file()`

Original-style input files have three garbage lines at the top used for commenting then triplets of lines describing one variable each. The first line of the triplet is a counter, for ease of hardcoding input reader, the second line contains the variable name (which is not used by the input reader) in order to make parsing possible outside of the ahrdcoded wlcsim input reader, and the third line

contains the value of the parameter itself. The first two lines of each triplet have a fixed form that we use to extract the record numbers and parameter names, but these forms are not used by wlcsim itself, which ignores these lnies completely. Thus, it is possible that the user specified a particular name for a parameter but that name does not match what wlcsim interpreted it as, since wlcsim simply uses the order of the parameters in this file to determine their identities.

`parse_params_file()`
: Parse and populate ParsedInput's params, ordered_param_names This parser currently understands two file formats: 1) "ORIGINAL" is the input method understood by Andy's hardcoded input reader in the original code used the Spakowitz lab was founded. 2) "LENA" is a spec using slightly more general input reader written by Elena Koslover while Andy's student.

`write(`*output_file_name*`)`
: writes out a valid input file for wlcsim.exe given parameters in the format returned by read_file

`wlcsim.input.correct_param_name(`*name*`)`
: Takes messy param names from different generations of the simulator and converts them to their newest forms so that simulations from across different years can be tabulated together.

`wlcsim.input.correct_param_value(`*name*, *value*`)`
: Some old param names also have new types. This takes messy param names from different generations of the simulator and converts their names and values to their newest forms so that simulations from across different years can be tabulated together.

wlcsim.analyze

Currently, dumping place for all analysis routines.

class `wlcsim.analyze.MapMeanColtimes(`*group_params*`)`
: Function object, allows the same-named method to be passed to

Pool.map

`wlcsim.analyze.get_coltimes_by_pos(`*scan_dir*, *linear_distance*`)`
Get all coltimes that happened at a fixed linear distance as a function of their position on the polymer

`wlcsim.analyze.map_cdf_coltimes(`*sim_dir, dt, group_params=['linear_distance']*`)`
TODO: actually implement

`wlcsim.analyze.striding_map_mean_coltimes(`*sim_dir, idx_gap=0, group_params=['linear_distance']*`)`
Not yet implemented. In the future, this should calculate the mean for a particular simulation of all the beads of each separation possible, without overlapping looped sections. Gah. Don't feel like typing that better.

Plotting Simulation Output Assuming you've used the *wlcsim* module for running your simulations, whether they be parameter scans or simple replicas, there are numerous helper methods in the `wlcsim.plot` for plotting the output, either in matplotlib or with PDB.

Generating Random Sequences Throughout the code, anytime the polymer needs to be labelled with a binary sequence, a markov chain (whose states are the values of the label at each bead) is used to initialize the chain.

If we call the labels $L_1 = A$ and $L_2 = B$ for now, this Markov chain is defined in terms of two parameters, the fraction of A and the non-unit eigenvalue of the transition state matrix, $$M = left(begin{bmatrix} p_{1,1} & p{2,1} \ p_{1,2} & p{2,2} end{bmatrix}right)$$ which is simply $1 - p_{1,2} - p_{2,1}$, where $p_{i,j} coloneq P(X^{(n+1)} == L_j | X^{(n)} == L_i)$. Of course, $p_{1,2} = 1 - p_{1,1}$ and $p_{2,1} = 1 - p_{2,2}$, so there are just two parameters of the transition matrix, $p_1 = p_{1,1}$ and $p_2 = p_{2,2}$.

Straightforward calculation gives that the average run length of A's is $frac{1}{1 - p_1}$, that the fraction of A's is just $frac{1 - p_2}{2 - p_1 - p_2}$, and that the eigenvalues of the transition matrix are 1 and $-1 + p_1 + p_2$.

So to specify a particular Markov chain defined by p_1 and p_2, use the quantites $texttt{lam} = -1 + p_1 + p_2$ and $texttt{fA} = frac{1 - p_2}{2

- p_1 - p_2}$ as simulation inputs.

Params

Indices and tables

- genindex
- modindex
- search

Python Module Index

w