wlcsim Documentation

Release 0.1.8+0.g2b5f727.dirty

Andrew Spakowitz, Bruno Beltran, Sarah Sandholtz, Quinn MacPh

Oct 27, 2020

Contents

CHAPTER 1

What is *wlcsim*?

A project started by the Spakowitz lab for carrying out various polymer physics calculations, especially multi-scale, coarse-grained simulation and theory relating to semiflexible polymers. The library has been applied largely to simulate DNA, and to compare results from polymer field theory to measurements of biological polymer systems, but our universal coarse graining procedure and our field theoretic results should be broadly applicable to any semiflexible polymer system.

For example, combining our coarse graining procedure with field theoretic simulations, we were able to simulate the phase segregation of an entire chromosome due to H3K9 methylation.

There are two largely independent codebases that are each called *wlcsim*. One is a Fortran program that implements our universal coarse graining procedure to allow Monte Carlo and Brownian dynamics simulations of semiflexible polymers with high discretization lengths (i.e. with a very small number of beads). The Monte Carlo routines in this codebase are highly optimized, and integrated with our field theoretic simulations. For details see *wlcsim Fortran Codebase Structure*.

There is an associated Python package, `wlcsim`, which can be used to help process the output of our Fortran code, but also contains much easier to use Monte Carlo and Brownian dynamics routines. For more details on this package, see the *wlcsim Python Module* docs.

The remainder of this section focusus on the FORTRAN code. The features of this codebase are described in *Fortran Simulation Features*.

CHAPTER 2

Setting up a Fortran simulation

To define the system you would like to simulate, set the approparte values `src/defines.inc`. Discriptions of each parameter are found along with their definitions in `src/defines.inc`. In pracatice, the best approach is often to start from examples provided in `input/example_defines/`.

Some parameters that MUST be set are given default values that prevent the code from compiling. This is on purpose so that the code is not accidentally run with something arbitrary for these values (like the length of the chain, the persistence length, etc).

For tips on setting up and running simulations see *Tips*.

CHAPTER 3

Running the Fortran code

Simply typing `make` in the top level directory will build the simulator from source. The executable created (`wlcsim.exe`) will data from the `input/` directoyr 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.

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.

To perform parallel tempering using MPI for multiprocessing using 10 threads first compile using `make` then type `mpirun -np 10 wlcsim.exe`. For more details on parallel tempering see *About Parallel Tempering*.

CHAPTER 4

Output

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.

For more details see *Fortran Simulation Output*.

CHAPTER 5

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:

5.1 wlcsim Python Module

Compute results related to (twistable, stretchable, shearable) wormlike chains.

This package can be largely separated into

1. Modules for processing simulation output from our FORTRAN codebase (`wlcsim.data`, *`wlcsim.input`*) 2. Modules for generating polymer chains at equilibrium (both directly in `wlcsim.chains` and using Monte Carlo `wlcsim.mc`). 3. Modules for simulating diffusing polymers (*`wlcsim.bd`*). 4. Modules for plotting these various chains and their properties (`wlcsim.plot` and `wlcsim.plot_wlcsim`).

5.1.1 wlcsim.bd

wlcsim.bd.rouse

Simulate Rouse polymers.

Notes

There are various ways to parameterize Rouse polymers. In this module, we use the convention that *N* is the number of beads of a Rouse polymer (contrary to e.g. Doi & Edwards, where *N* is the number of Kuhn lengths in the polymer). We instead use *Nhat* for the number of Kuhn lengths in the polymer.

D is the "diffusivity of a Kuhn length", i.e. *kbT/xi*, where *xi* is the dynamic viscosity of the medium in question, as typically found in the Langevin equation for a Rouse polymer

$$\xi \frac{d}{dt}\vec{r}(n,t) = k\frac{d^2}{dn^2}\vec{r}(n,t) + f^{(B)}$$

This means that to simulate a Rouse polymer diffusing in a medium with viscosity *xi*, the diffusion coefficient of each bead should be set to *Dhat = D (N/Nhat)*. Since ξ is in units of "viscosity per Kuhn length" and *(N/Nhat)* is in units of "number of beads over number of Kuhn lengths", this can be thought of as changing units from "viscosity per Kuhn length" to "viscosity per bead".

Some books use *b* for the mean (squared) bond length between beads in the discrete gaussian chain. We instead use *b* to be the real Kuhn length of the polymer, so that the mean squared bond length *bhat**2* is instead given by *L0*b*, where *L0* is L0 = L/(N-1) is the amount of polymer "length" represented by the space between two beads.

The units chosen here make the most sense if you don't actually consider the polymer to be a "real" Rouse polymer, but instead an e.g. semiflexible chain with a real "length" whose long distance statistics are being captured by the simulation.

To compare these results to the `rouse.analytical.rouse` module, use

```
>>> plt.plot(t_save, sim_msd)
>>> plt.plot(t_save, wlcsim.analytical.rouse.rouse_mid_msd(t_save, b, Nhat, D,
...                                                        num_modes=int(N/2)))
>>> plt.plot(t_save, 6*Dhat*t_save)
>>> plt.plot(t_save, 6*(Dhat/N)*t_save)  # or 6*(D/Nhat)*t_save
>>> # constant prefactor determined empirically...
>>> # no, using the value of 6*(Dhat/N)*t_R doesn't work...
>>> plt.plot(t_save, 1.9544100*b*np.sqrt(D)*np.sqrt(t_save))
```

where cutting off the number of modes corresponds to ensuring that the rouse behavior only continues down to the length scale of a single "bead", thus matching the simulation at arbitrarily short time scales. (Notice that we abide by the warning above. Our *Nhat* is *wlcsim.analytical.rouse*'s *N*).

Example

For example, if you want to simulate a megabase of DNA, which has a real linear length of about 1e6 megabase/base * 0.33 nm/base ~ 3e5 nm, and a Kuhn length of 1e2 nm, then

`wlcsim.bd.rouse.`**`jit_confined_srk1`**
: Add an elliptical confinement.

  Energy is like cubed of distance outside of ellipsoid, pointing normally back in. times some factor Aex controlling its strength.

`wlcsim.bd.rouse.`**`jit_confinement_clean`**
: Unfortunately, by "cleaning up" this function, we also make it 2x slower.

  #worthit

`wlcsim.bd.rouse.`**`jit_srk1`**
: Faster version of *wlcsim.bd.rouse.rouse* using jit.

  `N=101,L=100,b=1,D=1` takes about 3.5min to run when `t=np.linspace(0, 1e5, 1e7+1)` adding `t_save` does not slow function down

  Our srk1 scheme is accurate as long as Δt is less than the transition time where the MSD goes from high k behavior (t^1) to Rouse behavior ($t^{1/2}$). This is exactly the time required to diffuse a Kuhn length, so we just need $\Delta t < \frac{\hat{b}^2}{6\hat{D}}$ in 3D. the "crossover" from fast-k to rouse-like behavior takes about one order of magnitude in time, but adding more than that doesn't seem to make the MSD any more accurate, so we suggest setting Δt to one tenth of the bound above.

the number of orders of magnitude of "rouse" scaling the simulation will capture is exactly dictated by the ratio between this time scale and the rouse relaxation time (so like `N**2`?)

recall (doi & edwards, eq 4.25) that the first mode's relaxation time is $\tau_1 = \frac{\xi N^2}{k\pi^2}$. and the pth mode is $\tau_p = \tau_1/p^2$ (this is the exponential falloff rate of the pth mode's correlation function).

`wlcsim.bd.rouse.`**`measured_D_to_rouse`**(*Dapp*, *d*, *N*, *bhat=None*, *regime='rouse'*)

Get the full-polymer diffusion coefficient from the "apparent" D.

In general, a discrete Rouse polymer's MSD will have three regimes. On time scales long enough that the whole polymer diffuses as a large effective particle, the MSD is of the form $6D/\hat{N}t$, where, as is true throughout this module, D is the diffusivity of a Kuhn length, and $\hat{N}$ is the number of Kuhn lengths in the polymer.

This is true down to the full chain's relaxation time (the relaxation time of the 0th Rouse mode, also known as the "Rouse time") $t_R = N^2b^2/D$. For times shorter than this, the MSD will scale as $t^{1/2}$. Imposing continuity of the MSD, this means that the MSD will behave as $\kappa_0 D/\hat{N}(t_R t)^{1/2}$, where κ_0 is a constant that I'm too lazy to compute using $\lim_{t\to 0}$ of the analytical Rouse MSD result, so I just determine it empirically. We rewrite this MSD as $\kappa b D^{-1/2} t^{1/2}$, and find by comparing to the analytical Rouse theory that $\kappa = 1.9544100(4)$.

Eventually (at extremely short times), most "real" polymers will eventually revert to a MSD that scales as t^1. This cross-over time/length scale defines a "number of segments" $\tilde{N}$, where the diffusivity of a segment of length $\tilde{L} = L/(\tilde{N}-1)$ matches the time it takes stress to propagate a distance $\tilde{L}$ along the polymer. Said in other words, for polymer lengths smaller than $\tilde{L}$, the diffusivity of the segment outruns the stress communication time between two neighboring segments.

The diffusivity at these extremely short times will look like $6D(\tilde{N}/\hat{N})t$. In order to simulate this behavior exactly, one can simply use a polymer with $\tilde{N}$ beads, then all three length scales of the MSD behavior will match.

This three-regime MSD behavior can be very easily visualized in log-log space, where it is simply the continuous function defined by the following three lines. From shortest to longest time scales, $\log t + \log 6D(\tilde{N}/\hat{N})$, $(1/2)\log t + \log \kappa b D^{1/2}$, and $\log t + \log 6D\hat{N}$. The lines (in log-log space), have slopes 1, 1/2, and 1, respectively, and their "offsets" (y-intercept terms with the log removed) are typically referred to as D_{app}.

The simulations in this module use the diffusivity of a single Kuhn length as input (i.e. plain old D is expected as input) but typically, measurements of diffusing polymers are done below the Rouse time, and for long enough polymers (like a mammalian chromosome), it may be difficult to impossible to measure unconfined diffusion at time scales beyond the Rouse time. This means you often can't just look at the diffusivity of the whole polymer and multiply by $\hat{N}$ to get the diffusivity of a Kuhn length. And since there's not really a principled way in general to guess $\tilde{N}$, you usually can't use the short-time behavior to get D either, even supposing you manage to measure short enough times to see the cross-over back to t^1 scaling. This means that usually the only way one can extract D is by measuring D_{app}, since we can use the fact that $D_{\text{app}} = \kappa b D^{1/2}$ (the κ value quoted above only works for 3-d motion. I haven't bothered to check if it scales linearly with the number of dimensions or like $\sqrt{d}$ for d-dimensional motion, but this shouldn't be too hard if you are interested).

In short, this function gives you D given D_{app}.

Parameters

- **`D_app`** (*`float`*) – The measured $D_{\text{app}} based on the y-intercept of the MSD in log-log space. This can be computed as$, assuming that *t* is purely in the regime where the MSD scales like $t^{1/2}$.
- **`Nhat`** (*`int`*) – number of beads in the polymer
- **`d`** (*`int`*) – number of dimensions of the diffusion (e.g. 3 in 3-d space)

Returns **D_rouse** – the diffusivity of a Kuhn length of the polymer

Return type float

Notes

What follows is an alternate way of "thinking" about the three regimes of the MSD, from a simulation-centric perspective, that I typed up back when I first wrote this function, but has been obsoleted by the description at the start of this docstring.

In general, a discrete Rouse polymer's MSD will have three regimes. On time scales short enough that chain connectivity is not relevant, the MSD is of the form $6\hat{D}t$.

As soon as chain connectivity begins to affect the dynamics, the form of the MSD will change to $\kappa\hat{b}\hat{D}^{1/2}t^{1/2}$, where we have determined the constant κ empirically (using our exact analytical theory) to be approximately 1.9544100(4). Note that (up to a constant factor), this is just $6\hat{D}(t_R t)^{1/2}$, where t_R is the relaxation time (aka Rouse time) of the polymer, given by $\hat{b}^2\tilde{N}^2/\hat{D}$.

This is because the Rouse behavior only continues up to the relaxation time $t = t_R$, where the MSD transitions into the form $6\frac{\hat{D}}{\tilde{N}}t$. If you ask what "line" with slope 1/2 in log-log space intersects $6\frac{\hat{D}}{\tilde{N}}t$ at $t = t_R$, you get the above form for the sub-Rouse time MSD.

Here, $\tilde{N}$ is distinguished from N only as a formality. For a simulation of a discrete Rouse polymer with a fixed number of beads, $N = \tilde{N}$ is exactly the number of beads. For a "real" polymer, it is simply a numerical parameter that describes how long the Rouse behavior lasts (in the limiit of short times). For a "true" Rouse polymer (the fractal object) N is infinity and there is no time scale on which the MSD transitions back into t^1 behavior. But, for any real polymer, this will "almost always" eventually happen.

wlcsim.bd.rouse.**recommended_dt**(*N*, *L*, *b*, *D*)

Recommended "dt" for use with rouse*jit family of functions.

Currently set to $\frac{1}{10}\frac{b^2}{6D}$.

See the *jit_srk1()* docstring for source of this time scale.

wlcsim.bd.rouse.**with_integrator**(*N*, *L*, *b*, *D*, *t*, *x0=None*, *integrator=<function rk4_thermal_lena>*)

Simulate a Rouse polymer made of N beads free in solution.

Parameters

- **N** (*float*) – Number of beads in the chain.
- **L** (*float*) – Length of chain.
- **b** (*float*) – Kuhn length of the chain (same units as *L*).
- **D** (*float*) – Diffusion coefficient. (Units of `length**2/time`). In order to compute *D* from a single-locus MSD, use *measured_D_to_rouse*.
- **t** (*(Nt,) array_like of float*) – Time points to use for stepping the integrator. None of our current integrators is currently capable of taking time steps larger than the time scale of the highest Rouse mode of the finite chain (use *~.bd.recommended_dt* to compute the optimal *dt*).
- **x0** (*(N, 3) array_like of float, optional*) – The initial locations of each bead. If not specified, defaults to initializing from the free-draining equilibrium, with the first bead at the origin.
- **integrator** (*Callable[[ForceFunc, float, Times, Positions], Positions]*) – Either *~.runge_kutta.rk4_thermal_lena* or *~.runge_kutta.srk1_roberts*

Returns The positions of the *N* beads at each of the *Nt* time points.

Return type (Nt, N, 3) array_like of float

Notes

The polymer beads can be described by the discrete Rouse equations

$$\xi \frac{dx(i,t)}{dt} = -k(x(i,t) - x(i+1,t)) - k(x(i,t) - x(i-1,t)) + R(t)$$

where $\xi = k_B T/D$, $k = 3k_B T/b^2$, b is the Kuhn length of the polymer, D is the self-diffusion coefficient of a bead, and $R(t)/\xi$ is a delta-correlated stationary Gaussian process with mean zero and $\langle R(t)R(t')/\xi^2 \rangle = 2DI\delta(t-t')$.

Notice that in practice, $k/\xi = 3D/b^2$, so we do not need to include mass units (i.e. there's no dependence on $k_B T$).

wlcsim.bd.homolog

BD of two homologously-linked linear Rouse polymers.

See *homolog_points_to_loops_list* for a description of how the homologous network structure is handled.

A specialized

wlcsim.bd.homolog.**_jit_rouse_homologs**
"Inner loop" of rouse_homologs.

Some typing stuff: needs N, N_tot as int, needs tether_list/loop_list as dtype == int64, and uses loop_list.shape[0] to get num_loops. This is in particular important for empty list cases, where you need to do stuff like

```
>>> x = rouse._jit_rouse_homologs(int(N), int(2*N),
>>>         np.array([]).astype(int),
>>>         np.array([[]]).T.astype(int),
>>>         L, b, D, Aex, R, R, R, t, t_save)
```

wlcsim.bd.homolog.**points_to_loops_list**(*N*, *homolog_points*)
Create loops list for jit_rouse_linked.

Parameters

- **N** (*int*) – number of beads in each polymer if they were unbound
- **homolog_points** (*(L,) array_like*) – list of loci indices $l_i^L, l_i \in [1, N]$ (beads) that are homologously paired.

Returns

- **N_tot** (*int*) – final size of the x0 array for the simulation.
- **loop_list** (*(L,4) np.array*) – data structure used by jit_rouse_linked to correctly compute the spring forces in the network.

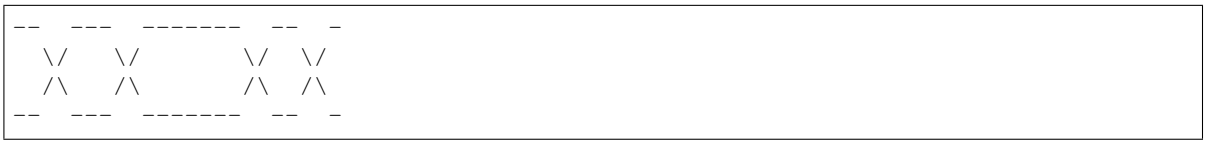
Notes

You can use the output like

```
>>> k1l, k1r, k2l, k2r = loop_list[i]
```

where "k[i][l,r]" refers to the bead demarcating the [l]eft or [r]ight end of a given "homologous loop" on each of the polymers polymer $i \in [1, 2]$.

Namely, the connectivity structure of our polymer "network" is that for each *k1l, k1r, k2l, k2r in loop_list*, the beads *k1l* thru *k1r* are linked sequentially (as are *k2l* thru *k2r*), but the beads *k1[l,r]* are linked to the beads *k2[l,r]*, respectively. This network structure can represent arbitrary "homologously linked" polymers, i.e. those of the form



Notice in the example below that *loop_list[1:,1]* and *loop_list[:-1,0]* are both the same as *homolog_list*.

Example

Here's an example for a typical case. With 101 beads per polymer (so 100 inter-bead segments), and one tenth of the beads bound to each other, we might get the following Poisson-space homolog points:

```
>>> N = 101; FP = 0.1
```

```
>>> homolog_points = np.where(np.random.rand(int(N)) < FP)[0]
```

```
>>> homolog_points
array([ 5, 18, 27, 59, 68, 82, 90, 94])
```

These eight homologous junctions would lead to a simulation array length of *2*N - len(homolog_points) = 202 - 8 = 194*. So *N_tot* will be 194, and the *loop_list* will look like

```
>>> N_tot, loop_list = rouse.homolog_points_to_loops_list(N, homolog_points)
```

```
>>> loop_list
array([[ -1,   5, 101, 105],
       [  5,  18, 106, 117],
       [ 18,  27, 118, 125],
       [ 27,  59, 126, 156],
       [ 59,  68, 157, 164],
       [ 68,  82, 165, 177],
       [ 82,  90, 178, 184],
       [ 90,  94, 185, 187],
       [ 94, 101, 188, 193]])
```

This can be read as follows. The first four beads of each polymer are "free", so first row is saying that beads 0 thru 4 correspond to unlinked beads in polymer 1. bead 5 is the linked bead (has half the diffusivity). since the polymer is of length 101, if there were no connections, the first polymer would extend from bead 0 to bead 100, and the second from bead 101 to bead 201. but since bead "5" is linked to (what would be) bead 106, that bead is omitted from the second polymer, and beads 101-105 are the "free end" of the second polymer.

The second row corresponds to the first actual "loop". The two tethering points are beads 5 and 18 of the first polymer. What would have been beads 106 and 119 (that's beads 5 and 18 of the second polymer) are omitted from the list of simulated beads. So bead 106 is linked to bead 5 and bead 117 is linked to bead 18, and beads 106-117 are linked sequentially.

Similarly, beads 18 and 118 (and 27 and 125) are linked, with beads 118-125 being linked sequentially.

wlcsim.bd.homolog.**rouse**(*N*, *FP*, *L*, *b*, *D*, *Aex*, *rx*, *ry*, *rz*, *t*, *t_save=None*, *tether_list=None*)

BD simulation of two homologous yeast chromosomes in meiosis.

An arbitrary elliptical-shaped confinement can be chosen, and in order to simulate synaptonemal formation in prophase, some fraction *FP* of the "homologous" beads of the polymer can be "rigidly tethered" to each other. The simulation treats these pairs as being one bead (with half the diffusion coefficient). The elastic force can then be communicated between the two polymers via the connecting loci.

Depending on what part of prophase is being simulated, either (or both) of the telomeres or centromeres can be tethered to the confinement by including them in *tether_list*. Currently the confinement force and the tethering force are both cubic w.r.t. the distance from the confining ellipse with the same strength (*Aex*), and have the form described in `f_conf_ellipse()` (and `f_tether_ellipse()`).

Parameters

- **N** (*int*) – Number of beads to use in the discretized Rouse chain
- **FP** (*float*) – *FP*$\in [0,1] is fraction of beads in the chain (out of)$ that will be tethered to each other
- **L** (*float*) – Length of the chain (in desired length units)
- **b** (*float*) – Kuhn length of the chain (in same length units as L)
- **D** (*float*) – The diffusivity of a Kuhn length of the chain. This can often difficult to measure directly, but see the documentation for the function `measured_D_to_rouse()` for instructions on how to compute this value given clean measurements of short-time regular diffusion or Rouse-like MSD behavior
- **Aex** (*float*) – multiplicative prefactor controlling strength of confinment and tethering forces
- **ry, rz** (*rx,*) – radius of confinement in x, y, z direction(s) respectively
- **t** (*(M,) float, array_like*) – The BD integrator will step time from t[0] to t[1] to t[2] to ...
- **t_save** (*(m,) float, array_like*) – *t_save*$\subset$ will be the time steps where the simulation output is saved. Default is *t_save* = *t*.
- **tether_list** (*int, array_like*) – list of bead indices that will be tethered to the confinement surface, for now, both polymers must be tethered at the same beads (but this would not be hard to change).

Returns

- **tether_list** (*List<int>*) – Indices (into compressed *Ntot* output) that were tethered to the confinement surface
- **loop_list** (*(~N*FP, 4) int*) – Output of *points_to_loops_list* used to specify which points are tethered
- **X** (*(len(t_save), Ntot, 3)*) – output of BD simulation at each time in t_save. The number of output beads Ntot <= 2*N excludes the beads on the second polymer whose positions are determined by the fact that they are tethered to the first polymer. Use *split_homologs_X()* to reshape this output into shape *(2, len(t_save), N, 3)*.

wlcsim.bd.homolog.**sim_loc**(*i*, *N*, *loop_list*)

Return actual indices for bead "i" of the second polymer.

Useful for indexing directly into the trucated simulation output array.

Parameters

- **i** (*(M,) int, array_like*) – beads index is requested for
- **N** (*int*) – actual number of beads in each polymer
- **loop_list** (*(~N*FP, 4) int*) – connectivity of the homologous polymers being simulated

Returns **iloc** – index in simulation array (or None if the bead is a paired bead) for each bead in *i*.

Return type (M,) Optional<int>, array_like

wlcsim.bd.homolog.**split_homolog_x**(*x0*, *N*, *loop_list*)
Make an (N_tot,3) array from rouse_homologs into (2,N,3).

wlcsim.bd.homolog.**split_homologs_X**(*X*, *N*, *loop_list*)
Make the (Nt,Ntot,3) output of rouse_homologs to (2,Nt,N,3).

wlcsim.bd.forces

Force fields used in our BD integrators.

Our BD code can be summarized as typically looking like:

```
X = init_function()
for t_i in requested_times:
    X = integrator_step(f_all, X, prev_t, t_i)
    prev_t = t_i
```

where `f_all` takes `X` (size `N_beads` by `N_dimensions`) and `t` and returns the total force on each of the beads at time `t`.

This module holds all the functions that can be used to construct `f_all`. Broadly, they are named based on the source of the force:

- `f_elas_*` are elastic forces, they define the physics of the polymer being simulated. Linear, ring, and different types of network polymers have their own independent implementations.
- `f_conf_*` are confinement forces.
- `f_tether_*` tether individual beads of the polymer to an external structure.
- Others may be added in the future.

Notes

Because they are in the "innermost loop", it is important that these functions are just-in-time compile-able when possible.

wlcsim.bd.forces.**f_conf_ellipse**
Compute soft (cubic) force due to elliptical confinement.

wlcsim.bd.forces.**f_elas_homolog_rouse**
Compute spring forces on two "homologously linked linear rouse polymers.

The two polymers are (homologously) at beads specified by loop_list. While each polymer is of length N beads, the beads that are hooked together move together identically, so you have *x0.shape[0] == 2*N - len(loop_list)*

We assume that, per the spec in *~.homolog_points_to_loops_list*, a polymer of the shape

```
++   ^^^   '''''''   ""   =
  \ /   \ /       \ /  \ /
   .     .         .    .
  / \   / \       / \  / \
--   ---   -------   --   _
```

will be laid out in memory like

```
--.---.-------.--.-++^^^'''''''""=
```

If we call the lower polymer "polymer 1", and all the beads on the top row "polymer 2", then it makes sense to first compute the spring forces between adjacent beads in polymer 1 as if there was no "polymer 2". Then, we can compute all the forces between beads in "polymer 2" and the "." beads. Finally, for each "loop" (represented as symbols of different types above) in polymer 2, we can compute the usual linear forces along the polymer.

wlcsim.bd.forces.**f_elas_linear_rouse**
Compute spring forces on single, linear rouse polymer.

wlcsim.bd.forces.**f_elas_linear_sswlc**
Compute spring forces and torques on each bead of dsswlc.

wlcsim.bd.forces.**f_tether_ellipse**
Compute soft (cubic) force towards an elliptical confinement.

wlcsim.bd.runge_kutta

Some spare stochastic integrators.

Not currently used because if you pass a function as an argument you can't @numba.jit that.

For rouse chains, Bruno (that's me) has tested the srk1 integrator and found it to work extremely well (see notes in wlcsim.bd.rouse on suggested dt).

For WLC/ssWLC, Lena found it useful to use a scheme that was higher deterministic order in time, to resolve the high elastic energies involved. Bruno has not yet tested this explicitly.

You can find Lena's algorithm (hold brownian force constant over an rk4 step) below. It is unlikely to be strongly convergent except if you subsample below the actual desired time resolution.

wlcsim.bd.runge_kutta.**_get_bead_msd**
center bead by default

for 1e4-long time arrays, this takes ~10-30s on my laptop

wlcsim.bd.runge_kutta.**_get_scalar_corr**
fast correlation calculation for testing

wlcsim.bd.runge_kutta.**_get_vector_corr**
fast correlation calculation for testing

wlcsim.bd.runge_kutta.**ou**(*x0*, *t*, *k_over_xi*, *D*, *method=<function rk4_thermal_lena>*)
simulate ornstein uhlenbeck process with theta=k_over_xi and sigma^2/2=D

wlcsim.bd.runge_kutta.**rk4_thermal_bruno**(*f*, *D*, *t*, *x0*)
Test new method: Attempt to keep ω constant.

WARNING: does not converge strongly (autocorrelation function seems higher than should be for OU process...), as is... x'(t) = f(x(t), t) + Xi(t), where Xi is thermal, diffusivity D

x0 is x(t[0]).

$f : R^n x R- > R^n$

wlcsim.bd.runge_kutta.**rk4_thermal_lena** (*f*, *D*, *t*, *x0*)
x'(t) = f(x(t), t) + Xi(t), where Xi is thermal, diffusivity D.

x0 is x(t[0]).

$f : R^n x R- > R^n$

wlcsim.bd.runge_kutta.**srk1_roberts** (*f*, *D*, *t*, *x0*)
From wiki, from A. J. Roberts. Modify the improved Euler scheme to integrate stochastic differential equations. [1], Oct 2012.

If we have an Ito SDE given by

$$d\vec{X} = \vec{a}(t, \vec{X}) + \vec{b}(t, \vec{X})dW$$

then

$$\vec{K}_1 = h\vec{a}(t_k, \vec{X}_k) + (\Delta W_k - S_k\sqrt{h})\vec{b}(t_k, \vec{X}_k)\vec{K}_2 = h\vec{a}(t_{k+1}, \vec{X}_k + \vec{K}_1) + (\Delta W_k - S_k\sqrt{h})\vec{b}(t_{k+1}, \vec{X}_k + \vec{K}_1)\vec{X}_{k+1} = \vec{X}_k + \frac{1}{2}$$

where $\Delta W_k = \sqrt{h}Z_k$ for a normal random $Z_k \sim N(0, 1)$, and $S_k = \pm 1$, with the sign chosen uniformly at random each time.

wlcsim.bd.tsswlc

Simulation twistable, stretchable-shearable wormlike chains.

Notes

The ssWLC is implemented as described in Koslover et. al., Soft Matter, 2013, 9, 7016. This defines the locations ($r^{(i)}(t_j)$) and tangent vectors ($u^{(i)}(t_j)$) of each (ith) bead at each time point (t_j).

The twist energy is (loosely speaking) quadratic in the angle that one must rotate the material normals of one bead to get them to align with the next bead (after the two beads have been rotated once to make the tangent vectors align).

There are other strategies one might take for this, but we will leave them as "#TODO" in the code for now.

wlcsim.bd.tsswlc.**_jit_sswlc_clean** (*N*, *L*, *lp*, *t*, *t_save*, *e_b*, *gam*, *e_par*, *e_perp*, *eta*, *xi_u*, *xi_r*)
WARNING: known to be buggy...

wlcsim.bd.tsswlc.**_jit_sswlc_lena** (*N*, *L*, *lp*, *t*, *t_save*, *e_b*, *gam*, *e_par*, *e_perp*, *eta*, *xi_u*, *xi_r*)
TODO: test

wlcsim.bd.tsswlc.**sswlc** (*N*, *L*, *lp*, *t*, *t_save*)
Simulate a stretchable-shearable WLC.

Parameters

- **N** (*int*) – The number of beads to use.
- **L** (*float*) – The length of the chain.
- **lp** (*float*) – The persistence length of the underlying WLC being approximated. (Must be same units as *L*.
- **t** (*array_like*) – The times for which to simulate the polymer's motion.
- **t_save** (*(M,) array_like*) – The subset of *t* for which we should save the output.

Returns

- **r** (*(M, N, 3) array of float*) – The positions of each bead at each save time.
- **u** (*(M, N, 3) array of float*) – The tangent vectors to the underlying wormlike chain at each bead.

5.1.2 wlcsim.analytical

Modules containing routines to evaluate analytical results for various polymer chains (equilibrium statistics and dynamics).

Rouse polymer, analytical results.

Notes

There are two parameterizations of the "Rouse" polymer that are commonly used, and they use the same variable name for two different things.

In one, N is the number of Kuhn lengths, and in the other, N is the number of beads, each of which can represent an arbitrary number of Kuhn lengths.

`wlcsim.analytical.rouse.`**`end2end_distance`**(*r*, *lp*, *N*, *L*)
For now, always returns values for `r = np.linspace(0, 1, 50001)`.

Parameters

- **r** (*`(N, ) float, array_like`*) – the values at which to evaluate the end-to-end probability distribution. ignored for now (TODO: fix)
- **lp** (*`float`*) – persistence length of polymer
- **N** (*`int`*) – number of beads in polymer
- **L** (*`float`*) – polymer length

Returns

- **x** (*(5001,) float*) – `np.linspace(0, 1, 5001)`
- **g** (*(5001,) float*) – $P(|R| = x|lp, N, L)$

Notes

Uses the gaussian chain whenever applicable, ssWLC tabulated values otherwise. If you request parameters that require a WLC end-to-end distance, the function will ValueError.

`wlcsim.analytical.rouse.`**`end2end_distance_gauss`**(*r*, *b*, *N*, *L*)
in each dimension... ? seems to be off by a factor of 3 from the simulation....

`wlcsim.analytical.rouse.`**`end_to_end_corr`**(*t*, *D*, *N*, *num_modes=10000*)
Doi and Edwards, Eq. 4.35

`wlcsim.analytical.rouse.`**`gaussian_G`**
Green's function of a Gaussian chain at N Kuhn lengths of separation, given a Kuhn length of b

`wlcsim.analytical.rouse.`**`gaussian_Ploop`**
Looping probability for two loci on a Gaussian chain N kuhn lengths apart, when the Kuhn length is b, and the capture radius is a

`wlcsim.analytical.rouse.`**`kp_over_kbt`**
"non-dimensionalized" k_p is all that's needed for most formulas, e.g. MSD.

Type k_p/(k_B T)

wlcsim.analytical.rouse.**linear_mid_msd**
modified from Weber Phys Rev E 2010, Eq. 24.

wlcsim.analytical.rouse.**linear_mscd**(*t*, *D*, *Ndel*, *N*, *b=1*, *num_modes=10000*)
Compute mscd for two points on a linear polymer.

Parameters

- **t** (*(M,) float, array_like*) – Times at which to evaluate the MSCD
- **D** (*float*) – Diffusion coefficient, (in desired output length units). Equal to k_BT/ξ for ξ in units of "per Kuhn length".
- **Ndel** (*float*) – Distance from the last linkage site to the measured site. This ends up being (1/2)*separation between the loci (in Kuhn lengths).
- **N** (*float*) – The full lengh of the linear polymer (in Kuhn lengths).
- **b** (*float*) – The Kuhn length (in desired length units).
- **num_modes** (*int*) – how many Rouse modes to include in the sum

Returns **mscd** – result

Return type (M,) np.array<float>

wlcsim.analytical.rouse.**ring_mscd**(*t*, *D*, *Ndel*, *N*, *b=1*, *num_modes=10000*)
Compute mscd for two points on a ring.

Parameters

- **t** (*(M,) float, array_like*) – Times at which to evaluate the MSCD.
- **D** (*float*) – Diffusion coefficient, (in desired output length units). Equal to k_BT/ξ for ξ in units of "per Kuhn length".
- **Ndel** (*float*) – (1/2)*separation between the loci on loop (in Kuhn lengths)
- **N** (*float*) – full length of the loop (in Kuhn lengths)
- **b** (*float*) – The Kuhn length, in desired output length units.
- **num_modes** (*int*) – How many Rouse modes to include in the sum.

Returns **mscd** – result

Return type (M,) np.array<float>

wlcsim.analytical.rouse.**rouse_large_cvv_g**(*t*, *delta*, *deltaN*, *b*, *D*)
Cvv^delta(t) for infinite polymer.

Lampo, BPJ, 2016 Eq. 16.

wlcsim.analytical.rouse.**rouse_mode**
Eigenbasis for Rouse model.

Indexed by p, depends only on position n/N along the polymer of length N. N=1 by default.

Weber, Phys Rev E, 2010 (Eq 14)

wlcsim.analytical.rouse.**rouse_mode_coef**
Weber Phys Rev E 2010, after Eq. 18.

Type k_p

wlcsim.analytical.rouse.**test_rouse_msd_line_approx**()
Figure out what Dapp is exactly using our analytical result.

if we use msd_approx(t) = 3*bhat*np.sqrt(Dhat*t)/np.sqrt(3)*1.1283791(6) then `|msd(t) - msd_approx(t)|/msd(t) = np.sqrt(2)/N*np.power(t, -1/2).` `msd_approx(t) > msd(t)` in this range, so that means that `msd_approx(t)/(np.sqrt(2)/N*np.power(t, -1/2) + 1) = msd(t).`

if we redefine msd_approx with this correction, the new relative error is about `np.sqrt(2)/N/100*t**(-1/2)`? gotta see if this carries over to other chain parameters though... it does not... this time msd(t) is bigger, so `msd_approx = msd_approx(t)/(1 - np.sqrt(2)/N/100*t**(-1/2)).`

okay actually looks like extra factor is additive?

for N = 1e8+1; L = 25; b = 2; D = 1;, we have msd_approx(t) - msd(t) = 1.01321183e-07

for N = 1e8+1; L = 174; b = 150; D = 166;, we have msd_approx(t) - msd(t) = 5.2889657(3)e-05

for N = 1e8+1; L = 174; b = 15; D = 166;, we have msd_approx(t) - msd(t) = 5.2889657(3)e-06

for N = 1e8+1; L = 17.4; b = 15; D = 166;, we have msd_approx(t) - msd(t) = 5.2889657(3)e-07

for N = 1e8+1; L = 17.4; b = 15; D = 16.6;, we have msd_approx(t) - msd(t) = 5.2889657(3)e-07 (no change)

for N = 1e7+1; L = 17.4; b = 15; D = 16.6;, we have msd_approx(t) - msd(t) = 5.2889657(3)e-06 (no change)

so the answer is like 3*bhat*np.sqrt(Dhat*t)/np.sqrt(3)*1.1283791615 - 0.202642385398*b*L/N

Fractional (Rouse) polymer. A Rouse polymer in a viscoelastic medium.

For the case alpha=1, these formulas should reduce to the analogous ones in *wlcsim.analytical.rouse*.

wlcsim.analytical.fractional.**frac_cv**(*t*, *alpha*, *D*)
Velocity autocorrelation of a fractionally-diffusing particle. Weber, Phys Rev E, 2010 (Eq 32)

wlcsim.analytical.fractional.**frac_discrete_cv**(*t*, *delta*, *alpha*, *D*)
Discrete velocity autocorrelation of a fractionally-diffusing particle. Weber, Phys Rev E, 2010 (Eq 33)

wlcsim.analytical.fractional.**frac_discrete_cv_normalized**(*t*, *delta*, *alpha*)
Normalized discrete velocity autocorrelation of a fractionally-diffusing particle. Should be equivalent to

frac_discrete_cv(t, delta, 1, 1)/frac_discrete_cv(0, delta, 1, 1)

Lampo, BPJ, 2016 (Eq 5)

wlcsim.analytical.fractional.**frac_msd**(*t*, *alpha*, *D*)
MSD of fractionally diffusing free particle.

Weber, Phys Rev E, 2010 (Eq 10)

wlcsim.analytical.fractional.**frac_rouse_cvv**(*t*, *delta*, *n1*, *n2*, *alpha*, *b*, *N*, *D*, *min_modes=500*, *rtol=1e-05*, *atol=1e-08*, *force_convergence=True*)
Velocity cross-correlation of two points on fractional Rouse polymer

rtol/atol specify when to stop adding rouse modes. a particle (t,delta) pair is considered to have converged when np.isclose returns true give rtol/atol and p is even (p odd contributes almost nothing)

Lampo, BPJ, 2016 Eq. 10.

wlcsim.analytical.fractional.**frac_rouse_mid_msd**(*t*, *alpha*, *b*, *N*, *D*, *num_modes=1000*)
Weber Phys Rev E 2010, Eq. 24.

wlcsim.analytical.fractional.**frac_rouse_mode_corr**(*p*, *t*, *alpha*, *b*, *N*, *D*)
Weber Phys Rev E 2010, Eq. 21.

wlcsim.analytical.fractional.**rouse_cv_mid**(*t*, *alpha*, *b*, *N*, *D*, *min_modes=1000*)
Velocity autocorrelation of midpoint of a rouse polymer.

Weber Phys Rev E 2010, Eq. 33.

wlcsim.analytical.fractional.**rouse_cvv_ep**(*t*, *delta*, *p*, *alpha*, *b*, *N*, *D*)
Term in parenthesis in Lampo, BPJ, 2016 Eq. 10.

wlcsim.analytical.fractional.**rouse_nondim_**(*t*, *delta*, *n1*, *n2*, *alpha*, *b*, *N*, *D*)
uses parameters defined by Lampo et al, BPJ, 2016, eq 12

wlcsim.analytical.fractional.**tDeltaN**(*n1*, *n2*, *alpha*, *b*, *D*)
Lampo et al, BPJ, 2016, eq 11

wlcsim.analytical.fractional.**tR**(*alpha*, *b*, *N*, *D*)
Lampo et al, BPJ, 2016, eq 8

wlcsim.analytical.fractional.**un_rouse_nondim**(*tOverDelta*, *deltaOverTDeltaN*, *alpha*, *delN=0.001*)
Takes a requested choice of parameters to compare to Tom's calculated values and generates a full parameter list that satisfies those requirements.

uses approximation defined by Lampo et al, BPJ, 2016, eq 12

In Tom's paper, delN is non-dimensinoalized away into deltaOverTDeltaN, but that only works in the case where the chain is infinitely long. Otherwise, where exactly we choose n1,n2 will affect the velocity correlation because of the effects of the chain ends. While other choices are truly arbitrary (e.g. D, b, N, etc), delN can be used to specify the size of the segment (relative to the whole chain) used to compute the cross correlation.

wlcsim.analytical.fractional.**vc**(*t*, *delta*, *beta*)
velocity correlation of locus on rouse polymer. beta = alpha/2.

wlcsim.analytical.fractional.**vvc_rescaled_theory**(*t*, *delta*, *beta*, *A*, *tDeltaN*)
velocity cross correlation of two points on rouse polymer.

wlcsim.analytical.fractional.**vvc_unscaled_theory**(*t*, *delta*, *beta*, *A*, *tDeltaN*)
velocity cross correlation of two points on rouse polymer.

5.1.3 wlcsim.data

5.1.4 wlcsim.input

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

class wlcsim.input.**InputFormat**
An enumeration.

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_defines_params_file()
Parse file in the format of src/defines.inc. Each line begins with #define WLC_P__[PARAM_NAME] [_a-zA-Z0-9] where WLC_P__[A-Z]* is the parameter name and the piece after the space is the value of the parameter.

TODO:test

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 three 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. 3) "DEFINES" is the format of the src/defines.inc file.

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.

5.1.5 wlcsim.FrankElastic

The calculation of Frank elastic constants occures in the function *get_Frank_values*.

wlcsim.FrankElastic.frank.**get_Frank_values**(*gamma*, *N*, *gammaNmax=None*, *laplace_args={}*)
Returns dictionary of Frank Elastic Values for specified system.

Nondimensionalized Frank elastc constant K'=K*A/(phi_00*L). Maier saupe parameter ether nondimensionalized by L*A*phi_00 or 2*l_p*A*phi_00 for aLAf and a2lpAf respectively. Correlation functions S are also returned.

Parameters

- **gamma** (*real*) – Field strength in kT's per Kuhn length of chain
- **N** (*real*) – How many Kuhn lengths long the polymers are
- **gammaNmax** (*real*) – Cutoff to prevent wasting time.
- **laplace_args** (*dict*) – Arguements to pass to inverse laplace calculation.

Note: when N=0.0, gamma is assume to be gamma*N, that is kT's per chain.

wlcsim.FrankElastic.frank.**get_a**(*gamma*, *N*, *gammaNmax=None*)
Maier-Saupe parameter need to generate field strength gamma.

Parameters

- **gamma** (*real*) – Field strength in kT's per Kuhn length of chain
- **N** (*real*) – How many Kuhn lengths long the polymers are

Note: when N=0.0, gamma is assume to be gamma*N, that is kT's per chain.

wlcsim.FrankElastic.frank.**get_first_pole**(*gamma*, *tol=1e-06*)
Search the real axes for polls of G[0][0,0]. Returns highest real value.

Parameters

- **gamma** (*float*) – Field strength.
- **tol** (*float*) – absolute tolerance of result

wlcsim.FrankElastic.frank.**test_Frank_accuracy**(*N=10*, *gamma=1000*)
Compair result for different laplace_args.

Looks really good at N=10, gamma=25 To within ~0.3% for N=100, gamma=50. Increasing factor is impoartant. At N=100 gamma=100 the factor=400 level gets off by a few percent. At N=10, gamma=100 it is accurate to may descimal points At N=10, gamma=1000 there is a several persent error

Plotting scripts are found in plot_Frank

wlcsim.FrankElastic.plot_frank.**Multi_frank_plot**(*load=None*, *logy=False*)
Plot data from get_Multi_frank_data

Parameters

- **load** (*string*) – File to load
- **logy** (*bool*) – Log y axis

wlcsim.FrankElastic.plot_frank.**find_gamma**(*N*, *aset=None*, *gammaN=True*, *gamma_TOL=1e-06*, *a_TOL=1e-06*)
Frank Elastic data for given N and a

Parameters

- **N** (*float*) – Number of Kuhn length
- **aset** (*itrable*) – aLAf or a2lpAf values
- **gammaN** (*bool*) – If ture use aLAf, else use a2lpAf, amoung other things

wlcsim.FrankElastic.plot_frank.**get_Multi_frank_data**(*log_gamma=False*, *npts=48*, *gammaN=False*, *maxgamma=35*, *saveas='Frank.p'*, *nthreads=25*, *Nset=None*)
Calculate Frank elastic constants for a range of gamma and N.

Parameters

- **log_gamma** (*bool*) – Space gamma logrithmeicly
- **npts** (*int*) – number gamma values
- **N** (*real*) – Kuhn lenths in polymer
- **Where to save results.** (*saveas=*) –
- **gammaN** (*bool*) – Devide gamma by N. I.e. nondimensionalized by polyer length
- **nthreads** (*int*) – Number of threads to use
- **Nset** (*iterable*) – Where to save results

wlcsim.FrankElastic.plot_frank.**get_Nsweep_data**(*saveas='Nsweep.p'*, *MultiProcess=True*, *nthreads=25*, *Ns=None*)

Calculate Frank Elastic Constants for sweep over N

Parameters

- **Where to save results.** (*saveas=*) –
- **nthreads** (*int*) – Number of threads to use
- **MultiProcess** (*bool*) – Use many threads
- **Ns** (*iterable*) – N values

wlcsim.FrankElastic.plot_frank.**get_data_for_Splot**(*name*, *nthreads=25*, *N=0.05*, *npts=45*)

Get data for plot_S

Parameters

- **name** (*string*) – Where to save result.
- **nthreads** (*int*) – Number of computational threads
- **N** (*real*) – Kuhn lenths in polymer
- **npts** (*int*) – number gamma values

wlcsim.FrankElastic.plot_frank.**multi_process_fun**(*inputs*)

pass through for muiltiproccessing

wlcsim.FrankElastic.plot_frank.**plotNsweep**(*load*)

Plot results from get_Nsweep_data

Parameters **load** (*string*) – File to load from.

wlcsim.FrankElastic.plot_frank.**plot_Frank**(*log_gamma=True*, *MultiProcess=True*, *npts=24*, *N=1.0*, *gammaN=False*, *maxgamma=35*, *nthreads=25*)

Calculate Frank Elastic data for range of gamma values.

Parameters

- **log_gamma** (*bool*) – Space gamma logrithmeicly
- **MultiProcess** (*bool*) – Use multiple threads
- **npts** (*int*) – number gamma values
- **N** (*real*) – Kuhn lenths in polymer
- **gammaN** (*bool*) – Devide gamma by N. I.e. nondimensionalized by polyer length
- **nthreads** (*int*) – Number of threads to use

wlcsim.FrankElastic.plot_frank.**plot_S**(*name*)
Plot S over diffetn gamma values. :param name: File name to load data from. :type name: string

wlcsim.FrankElastic.plot_frank.**seperate_bend_twist_splay**(*load*, *gammaN=False*, *logy=False*)
Plot data from get_Multi_frank_data seperatly

Parameters

- **load** (*string*) – File to load
- **logy** (*bool*) – Log y axis

There are also a number of helper funtions,

wlcsim.FrankElastic.invlaplace.**invLaplace**(*p_values*, *G_values*, *L*, *reduction=0.0*)
Numerical Inverse Laplace from $G(p) \rightarrow G(L)$.

Parameters

- **p_values** (*array_like*) – 1D complex array of p values (i.e. the contour)
- **G_values** (*array_like*) – 1D complex array G(p)
- **L** (*float*) – Polymer length (In Kuhn lengths)
- **reduction** (*float*) – multiply answer by exp(-reduction) for better numerics

wlcsim.FrankElastic.invlaplace.**invLaplace_path**(*path*, *cutts*, *G_values*, *L*, *reduction=0.0*)
Inverse laplace transform based on path.

Parameters

- **path** (*ndarray*) – complex path values
- **cutts** (*dict*) – specified by path
- **G_values** –

wlcsim.FrankElastic.invlaplace.**make_path**(*factor=100*, *scale=1.0*, *lambda_offset=0.1*, *width=2.0*, *depth=2.0*, *nwing=100*, *nside=30*, *nback=20*, *maxp=6000*, *pole_offset=0.0*)
Path for complex integral that draws a half rectangle about the origin.

Parameters

- **factor** (*int*) – Number of points to caltulate (sort of)
- **scale** (*float*) – Scale of path. Recomended: around 10.0/N
- **lambda_offset** (*float*) – distance from complex axis
- **width** (*float*) – width of rectangle
- **depth** (*float*) – depth of rectangle
- **nwing** (*int*) – number of points in wing before `factor`
- **nside** (*int*) – number of points to a side before `factor`
- **back** (*int*) – number of points on the back before `factor`
- **maxp** (*float*) – approximation for infinity
- **pole_offset** (*float*) – add to depth

wlcsim.FrankElastic.stonefence.**Gmll_matrix**
“Matrox of propagators between starting and ending l value.

Parameters

- **Wplus** (*numpy array*) – Result of Wplus_vec for same m, p
- **Wminus** (*numpy array*) – Reult of Wminus_vec for same m, p
- **Am** (*numpy array*) – Result of Am_vec for same m
- **PgammaB** (*numpy array*) – Result of PgammaB_vec for same m, p, gamma
- **gamma** (*float*) – alignment strength, in kT's per Kuhn length
- **m** (*int*) – z component of agular momentum quantum number

Returns An ORDER_L x ORDER_L numpy matrix with propagators that use Maier-Saupe steps to get from l0 to lf.

wlcsim.FrankElastic.stonefence.**PgammaB_vec**
P - gamma beta

Returns vector with index ell

wlcsim.FrankElastic.stonefence.**precalculate_data**(*p, gamma, m_values=[0]*)
Precalculate W_plus, W_minus, W_pm, and G_m_ll

Parameters

- **p** (*complex*) – laplace conjugate of path length
- **gamma** (*real*) – aligning l=2 (Maier-Saupe) field strength
- **m_values** (*list*) – list of integer m values to precalculate for

wlcsim.FrankElastic.yyyreal.**YR**(*m*, *ell*, *phi*, *theta*)
Real spherical harmonic function.

wlcsim.FrankElastic.yyyreal.**YYY**(*m1*, *L*, *M*, *m2*, *ORDER_L=50*)
Same as YYYreal

wlcsim.FrankElastic.yyyreal.**YYY_0mm**(*m*, *ORDER_L*)
Y_{ell1,0} Y_{2,m} Y_{ell2,m}

wlcsim.FrankElastic.yyyreal.**YYY_l1**(*m1*, *m*, *m2*, *ORDER_L*)
Y_{l1,m1} Y_{1,m} Y{l2,m2}

wlcsim.FrankElastic.yyyreal.**YYYreal**(*m1*, *L*, *M*, *m2*, *ORDER_L*)
YYYreal_{l_1,l_2} = int dvec{u} Y_{l_1, m1} Y_{L,M} Y_{l2,m2}

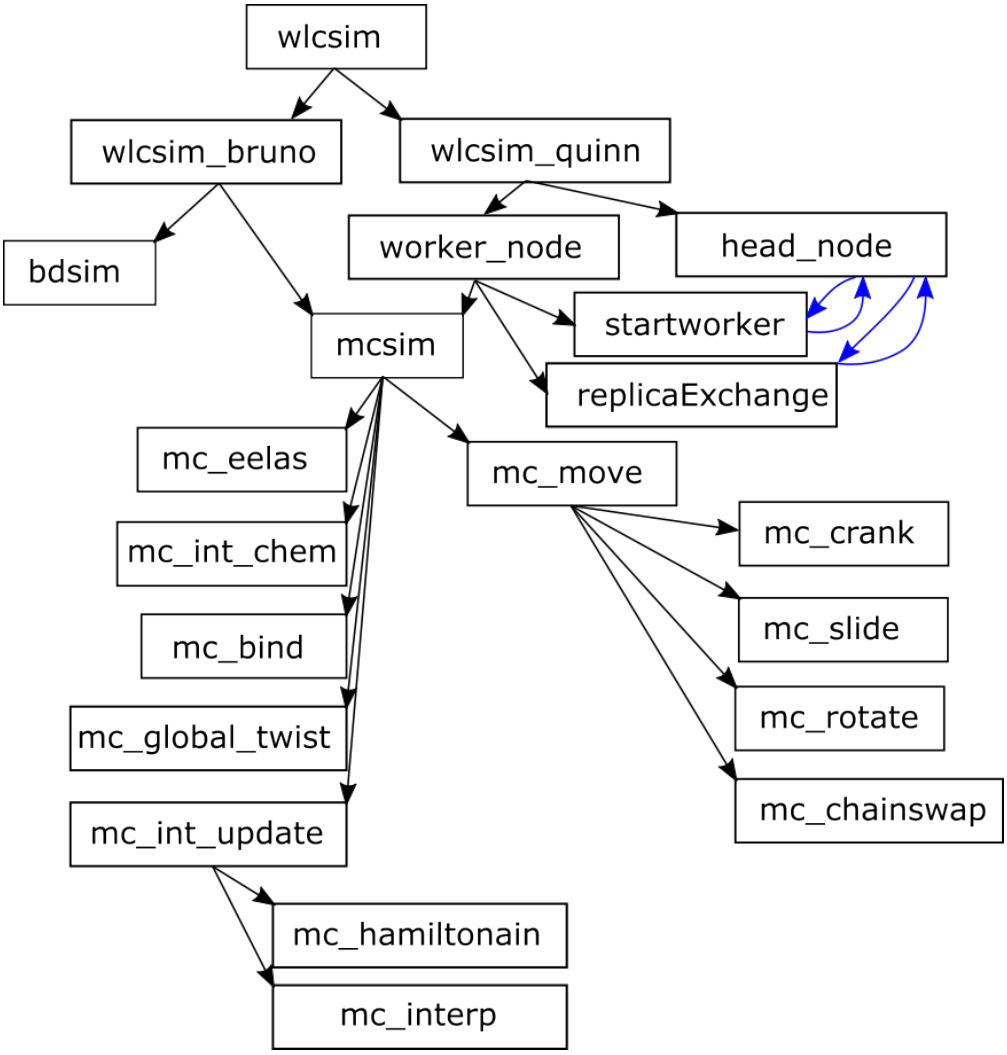
Not able to handle all combiniations of inputs. Can only handle Y_{l_1,0} Y_{2,M} Y_{l_2,M} or Y_{l_1,M} Y_{2,M} Y_{l_2,0} or Y_{l_1,m_1} Y_{1,M} Y{l_2,m_2} or Y_{l_1,m_1} Y_{0,0} Y{l_2,m_2}

wlcsim.FrankElastic.yyyreal.**ind_sphere**(*fun*)
Integrate function on sphere.

Parameters **fun** (*callable*) – fun(theta, phi)

wlcsim.FrankElastic.yyyreal.**intYYYnum**(*l1*, *m1*, *L*, *M*, *l2*, *m2*)
Numerical integral over 3 spherical harmics

5.2 wlcsim Fortran Codebase Structure



program wlcsim
Use a universal discretization scheme to simulate from WLCs through Gaussian chains.

The main entry point of the program is pretty useless as a starting point to understand the codebase, and simply calls one of several "versions" of the program built from our Brownian Dynamics/Monte Carlo API.

Program

program main
Loads in parameters from the input file. Calculates parameters used in simulation from input parameters. For each save point requested, uses either Bruno, Quinn, or Brad's simulators to step forwards (in time or Monte Carlo steps) and writes the output and (optionally) the simulation state needed to restart the simulation.

Use `flap (command_line_interface()), precision (setup_runtime_floats()), params (wlcsim_params(), wlc_repsuffix(), maxfilenamelen(), save_simulation_state(), set_parameters(), initialize_wlcsim_data(), save_parameters(), printdescription(), printwindowstats())`

Call to `stop_if_err()`, `setup_runtime_floats()`, `set_parameters()`, `initialize_wlcsim_data()`, `save_parameters()`, `save_simulation_state()`, *`wlcsim_quinn()`*, *`wlcsim_bruno()`*, `wlcsim_bruno_mc()`, `wlcsim_bruno_looping_events()`

The actual hard work is done in the *wlcsim_** files

Subroutines and functions

subroutine wlcsim_bruno(*save_ind*, *wlc_p*)

values from wlcsim_data

Parameters

- **save_ind** *[integer,in]*
- **wlc_p** *[wlcsim_params,in]*

Use `params (dp()`, `wlcsim_params()`, `pack_as_para()`, `printenergies()`, `printsiminfo())`

Called from *`main`*

Call to `initialize_energies_from_scratch()`, *`mcsim()`*, `verify_energies_from_scratch()`, `bdsim()`, `pack_as_para()`, `stress()`, `stressp()`, `energy_elas()`, `energy_self_chain()`, `printsiminfo()`, `printenergies()`

subroutine wlcsim_quinn(*save_ind*, *wlc_p*)

Quinn's branch of the Fortran codebase. This branch is set up for Monte-Carlo simulations with a particular focus on parallel tempering.

Parameters

- **save_ind** *[integer,in]* :: 1, 2, ...
- **wlc_p** *[wlcsim_params,inout]*

Use `params`, `mpi`

Called from *`main`*

Call to `onlynode()`, *`head_node()`*, *`worker_node()`*, `printenergies()`, `printwindowstats()`, `printlinkingnumber()`

The main entry points for our API are the *mcsim.f03* and *BDsim.f03* routines, which allow you to do Monte Carlo moves and Brownian Dynamics timesteps (respectively) given a set of energies (or forces, respectively) specified by the input in *defines.inc*.

Subroutines and functions

subroutine mcsim(*wlc_p*)

Perform Metropolis Hastings Monte Carlo moves on the system of polymers. Loops over move types (crank shaft, slide,...) and calls each energy function. Accepts move by the Metropolis condition on the change in energy. If accepted update system.

Parameters **wlc_p** *[wlcsim_params,inout]* :: system varibles

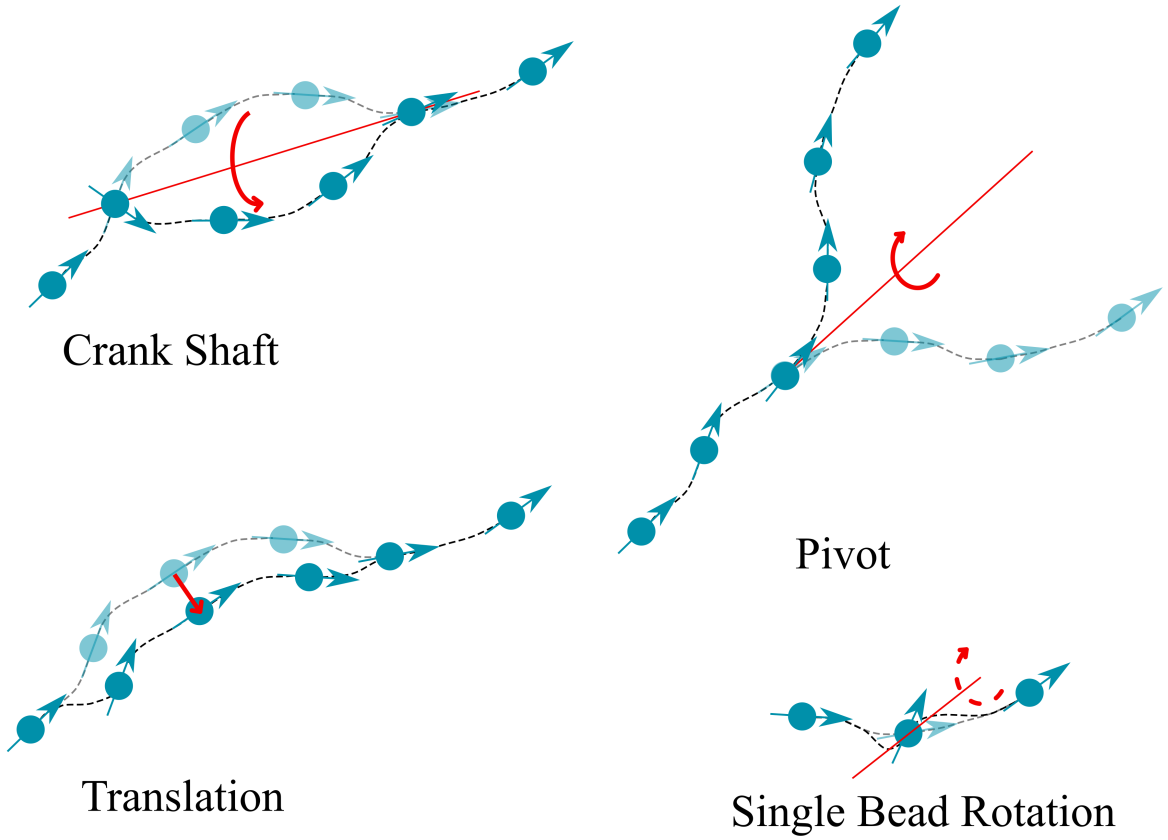
Use `params(wlc_phit()`, `wlc_crossp()`, `wlc_abp()`, `wlc_dphi_l2()`, `wlc_ab()`, `wlc_ncross()`, `wlc_ind_exchange()`, `wlc_indphi()`, `wlc_rand_stat()`, `wlc_cross()`, `wlc_vol()`, `wlc_phi_l2()`, `wlc_nphi()`, `wlc_dphib()`,

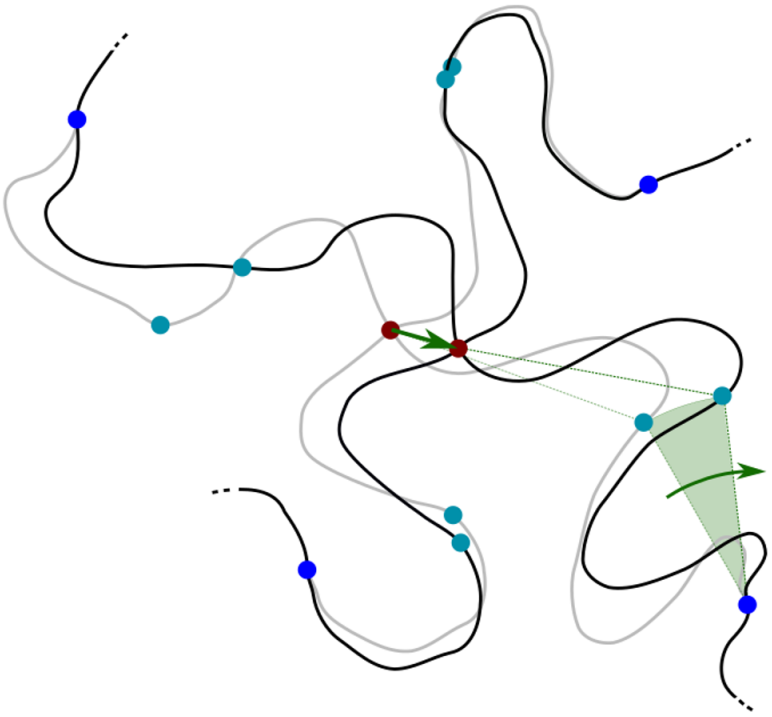
wlc_attempts(), wlc_up(), wlc_crosssize(), wlc_ncrossp(), wlc_r(), wlc_success(), wlc_rp(), wlc_meth(), wlc_dphia(), wlc_phib(), printenergies(), wlcsim_params(), wlc_phia(), int_min(), nan(), wlc_nbend(), wlc_npointsmoved(), pack_as_para(), nmovetypes(), wlc_pointsmoved(), wlc_bendpoints(), wlcsim_params_recenter(), wlc_lk0(), wlc_lk(), wlc_tw(), wlc_wr(), wlc_vp(), wlc_u(), wlc_v(), wlc_r_gjk(), wlc_nucleosomewrap(), wlc_basepairs(), wlc_basepairs_prop()), energies, umbrella (umbrella_energy()), mersenne_twister, binning (addbead(), removebead()), update_ru (update_r()), polydispersity (length_of_chain(), chain_id(), leftmost_from()), linkingnumber (get_del_tw_wr_lk())

Called from *wlcsim_bruno()*, wlcsim_bruno_looping_events(), wlcsim_bruno_mc(), *worker_node()*, onlynode()

Call to list_confinement(), set_all_denergy_to_zero(), mc_move(), mc_cylinder(), mc_sterics(), mc_internucleosome(), check_rp_for_nan(), alexanderp_crank(), length_of_chain(), chain_id(), alexanderp_slide(), alexanderp(), mc_eelas(), mc_global_twist(), mc_bind(), pack_as_para(), de_self_crank(), energy_self_slide(), mc_int_chem(), mc_int_rep(), mc_int_super_rep(), *mc_int_update()*, mc_external_field(), mc_2bead_potential(), mc_explicit_binding(), umbrella_energy(), get_del_tw_wr_lk(), apply_energy_ison(), calc_all_de_from_dx(), sum_all_denergies(), accept_all_energies(), stop_if_err(), update_r(), printenergies(), leftmost_from(), *mc_adapt()*, wlcsim_params_recenter()

mcsim.f03 calls different subroutines that define possible Monte Carlo moves (like *MC_reptationMove.f03*) and then checks for whether the move should succeed by summing the various energies that are turned on (like *mc_wlc.f90*). Diagrams describing a few of the moves are provided here:





MC_wlc

Paramters of moves are optimized over time by *adapt.f90*.

Subroutines and functions

subroutine mc_adapt(*wlc_p*, *mctype*)
values from wlcsim_data

Parameters

- **wlc_p** *[wlcsim_params,in]*
- **mctype** *[integer,in]* :: Type of move

Use `params`

Called from *mcsim()*

5.3 Fortran Simulation Features

5.3.1 Semiflexable polymers

The feature that wlcsim was originally designed around and for which it is named is it's ability to quickly and accurately simulation polymers with stiffness ranging from quite rigid to very flexible.

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.

Of these, the middle strategy is where this code base stands out. The "stretchable, shearable wormlike chain" is described in "Discretizing elastic chains for coarse-grained polymer models" by E. F. Koslover and A. J. Spakowitz (2013). This model allows for quick but still accurate simulation of polymer statistics of semiflexible polymers of a wide range of stiffnesses (i.e. persistence lengths).

The effective potential for the stretchable, searable wormlike chain is given in terms of constants. The bending modulus, the stretch modulus, the shear modulus, the bend-shear coupling, and the ground state segment length. These constants are tabulated for different descritizations in *input/dssWLCparams*. The module *MC_wlc* handles interpolating from this table and calculating the potential between two beads. See MC_wlc.

5.3.2 Field theoretic interactions

Monte-Carlo simulations of solutions and melts of polymers are greatly accelerated by the use of a field-theoretic, binned-density approach for calculating interbead interactions. This approach is described in "Theoretically informed coarse grain simulations of polymeric systems." by Pike et. al. (2009). With a further description of our implementation given in "Field-theoretic simulations of random copolymers with structural rigidity" by Mao et. al. (2017).

Under this approach the local volume fraction of each constituent is calculated for each of a grid of bins. The code for calculating the volume fractions and updating them after a Monte-Carlo move are

Subroutines and functions

subroutine mc_int_intitialize(*wlc_p*)
This subroutine calculates the field energy from scratch based on the positions and types of the beads as described in the hamiltonian. This subroutine also sets the value fractions PhiA and PhiB to 0 and sets x_chi. Volume fractions are linearly interpolated between bins.

Parameters **wlc_p** *[wlcsim_params,in]*

Use `params(dp(),wlcsim_params()),energies(energyof(),maiersaupe_())`

Called from `calculate_energies_from_scratch()`

Call to `y2_calc()`, *interp()*, *hamiltonian()*, *calc_dphi()*

subroutine mc_int_update(*wlc_p*)

This subroutine calculates the change in the field energy after a Monte Carlo move. Calculation only performed for bins containing moved (or changed) beads.

Parameters **wlc_p** *[wlcsim_params,in]*

Use `params(wlcsim_params())`

Called from *mcsim()*

Call to *calc_dphi()*, *hamiltonian()*

subroutine calc_dphi(*wlc_p*, *ib*)

This subroutine calculated the change in volume fraction in various bins assocated with the motion of bead IB. The old positon is taken from wlc_R while the new position is taken from wlc_RP.

Developer Note: This subroutine assumes you have set wlc_bin_in_list=FALSE some time before the start of the move.

Parameters

- **wlc_p** *[wlcsim_params,in]* :: data structure
- **ib** *[integer,in]* :: Index of moved bead

Use `params(dp(),wlcsim_params()),energies(energyof(),maiersaupe_())`

Called from *mc_int_intitialize()*, *mc_int_update()*

Call to *interp()*, `y2_calc()`

The process of calculating the change in volume fractions after a move is often the computational bottle-neck in the simulation. Routines for doing this that are optimized for different move types are documented here other_int_fun.

The density is interpolated linearly between bins as described by Pike. This interpolation is performed in

Subroutines and functions

subroutine interp(*wlc_p*, *rbin*, *ix*, *iy*, *iz*, *wx*, *wy*, *wz*)

This program linearly interpolates a bead at RBin into 8 bins indexed by IX, IY, IZ with weights WX, WY, WZ. Interpolation is based on Pike (2009), "Theoretically informed coarse grain simulation of polymeric systems".

For exampe a if a bead is closer to the center of IX(1) then IX(2) then WX(1) will be proportionally higher then WX(2). The total weight in any of the eight bins is given by WX*WY*WZ.

Parameters

- **wlc_p** *[wlcsim_params,in]* :: system varibles
- **rbin** (3) *[real,inout]* :: position of bead
- **ix** (2) *[integer,out]* :: Range of bins in x direction
- **iy** (2) *[integer,out]* :: Range of bins in y direction
- **iz** (2) *[integer,out]* :: Range of bins in z direction
- **wx** (2) *[real,out]* :: WX(1) = (IX(2)*WLC_P__DBIN-RBin(1))/(IX(2)*WLC_P__DBIN-IX(1)*WLC_P__DBIN)

- **wy** (2) *[real,out]* :: WY(1) = (IY(2)*WLC_P__DBIN-RBin(2))/(IY(2)*WLC_P__DBIN-IY(1)*WLC_P__DBIN)
- **wz** (2) *[real,out]* :: Fractional contributions in z

Use `params(dp(),wlcsim_params())`

Called from *mc_int_intitialize()*, *calc_dphi()*, `mc_int_chem()`, `mc_int_rep()`, `mc_int_super_rep()`, `mc_int_swap()`

To run a simulation with the field theoretic interaction turned on set *WLC_P__FIELD_INT_ON* to true. You will also need to specify the relevant interaction parameters to your problem, for example *WLC_P__CHI*. The details of the interaction depend on problem (i.e. *WLC_P__FIELDINTERACTIONTYPE*). What each of these interactions are (or to add your own) see *src/mc/mc_hamiltonain.f03*.

Subroutines and functions

`subroutine hamiltonian`(*wlc_p*, *initialize*)

This subroutine calculates the field energy (Hamiltonian) based on volume fractions phi_A and phi_B. The expresion used to calculate the energy is different for different systems. The setting WLC_P__FIELDINTERACTIONTYPE determines which expression to use.

If *initialize* is true then the energy for all bins is calculated. Otherwise calculate only for bins specified by wlc_inDPHI.

Parameters

- **wlc_p** *[wlcsim_params,inout]* :: data
- **initialize** *[logical,in]* :: Need to do all beads?

Use `params(dp(),wlcsim_params())`, `energies(energyof(),chi_(),couple_(),kap_(),field_(),maiersaupe_())`

Called from *mc_int_intitialize()*, *mc_int_update()*, `mc_int_chem()`, `mc_int_rep()`, `mc_int_super_rep()`, `mc_int_swap()`

If you edit this file be sure to edit both the initial calculation of volume fraction and change in volume fraction options.

As a final note, for liquid crystal systems the simulation has additional volume fraction fields that are turned on by *WLC_P__CHI_L2_ABLE*. There are five such fields and they correspond to the *m* values for sphericals harmonics of *l=2* between -2 and 2.

5.3.3 Nucleosome geometry

For simulations of chromatin that require the geometry of nucleosomes to determine the entry exit angles of the DNA linkers the module *nucleosome* has the details.

nucleosome

5.3.4 Streamlined Energy Components

The are many different energy contributions. To keep track of these (or to add more) you should look to the *energies* module.

energies

5.4 Fortran Simulation Output

5.4.1 Bead positions and identity

Periodically while running the simulation will take a save point and save the configuration of the polymer(s). These are saved in the *data/* folder in files such as *data/r100* where the number 100 denotes what savepoint it is. When using MPI to parallel temper, the data will be saved in the format *data/r110v6* where 100 refers to the save point and 6 refers to the replica. This data will have three columns for the x, y, and z positions of each bead. If there are multiple polymers they will be printed one after the next with no delineation.

If *WLC_P__SAVEAB* is specified as true than *data/r100v6* will contain a fourth column specifying the bead type. For two tail methylation profiles a value of 0 in this column refers to either tail bound by HP1, a value of 1 or 2 refers to one or the other tail bound by HP1, while a value of 3 refers to both tails being bound by HP1. If *WLC_P__CHANGINGCHEMICALIDENTITY* is true than a fifth column will specify the number of methylated tails (0, 1 or 2).

For example

```
21.702    39.628    14.415  0  0
21.715    39.617    14.892  3  1
21.205    39.449    14.758  3  2
20.718    39.472    14.672  3  2
20.463    39.763    14.803  0  0
20.209    39.667    15.418  1  0
19.888    39.643    14.826  2  0
...
```

5.4.2 Other files

If *WLC_P__SAVEU* is true then the orientation unit vectors for each bead will be stored as in files of a similar format to the positions. For example *data/u100* or *data/u100v6*. If *WLC_P__LOCAL_TWIST* is also true then there will be three additional columns in the *data/u100*.

The file *data/energiesv6* contains the various energies and corresponding coefficients. All energies are in kT's. There is one line in this file per savepoint. The file *data/adaptationsv6* contains the adapted window size, success rate, and so on the various move types. These files are useful for diagnosing problems and determining if the simulation is equilibrated.

A file *data/repHistory* contains various information about replica coupling if it is in use.

5.4.3 Visualization

We provide visualization code based on pymol. To run, navigate to 'visualization/polymer_mole', enter the savepoint and other settings into *run_visualization.py* and then type *python3 run_visualization.py*. This will require both python3 and pymol. The way it works is that the output described above is converted into a pdb file which is then passed to pymol. Various sets of settings that have been used in the past are included in *run_visualization.py*. To use one of thes set *image* to the name of the corrisponing set.

If *closePymol* is set to true in *run_visualization* than the output will be saved to the *data/* directiory. By editing the line *for rep in . . .* you can save many snapshots.

5.5 About Parallel Tempering

This code is set up to allow parallel tempering over multiple variables if desired. The replicas will be along a path in parameter space parameterized by *s* which goes between 0 and *WLC_P__INITIAL_MAX_S*.

To determine which variables will be parallel tempered over set the corresponding switches. For example, to parallel temper over chi set *WLC_P__PT_CHI* to true. By default, the path is set to linearly go between 0 and *WLC_P__CHI* in this case. However, you can define a different path in the function *cof_path_by_energy_type* in the file *src/wlcsim/wlcsim_quinn*. For example, MU starts from -2.5 rather than zero.

5.5.1 How to run with parallel tempering

1. Install MPI fortran if not already installed.
2. In defines file, set *WLC_P__PTON=.TRUE.*, set the variable(s) you want to temper over, and set *WLC_P__CODENAME="quinn"*.
3. In file *Makefile* set 'FC=mpifort'
4. Compile using *make*.
5. Run using *mpirun -np 16*. This will run with 15 replicas and 1 head node. You may need to provide the path to mpirun on your computer, mine is located at *usr/bin/mpirun*.

5.5.2 Adjustable spacing

By default, replicas are spaced evenly between 0 and the maximum value. You can adjust the spacing by writing a more complicated function into cof_path_by_energy_type' in the file *src/wlcsim/wlcsim_quinn*. There is a built in way to adjust the spacing of the replicas to achieve an acceptable exchange rate between sequential replicas. To do this set *WLC_P__INDSTARTREPADAT*, and *WLC_P__INDENDREPADAPT* to the save point range during which you would like the adaptation to occur. For more details see the file *src/mc/adaptCof*.

5.5.3 Structure of code

The code preforms the following sequence of events

1- The *head_node* calculates starting coefficient values and sends them to the workers.

2- The *worker_node* runs *start_worker*.

3- The *worker_node* runs Monte Carlo moves with *mcsim*.

4- The *worker_node* call *replicaExchange* to return values to the *head_node*.

5- The *head_node* attempts a replica exchange.

6- The *head_node* return new coefficient values to the worker via *replicaExchange*

7- Return to step 3 and repeat.

subroutine head_node (*process*)

Head node for multiple thread parallel tempering using MPI. Sends worker threads their coefficients, receives x back, then attempts replica swap move.

Parameters **process** *[integer,in]* :: number of therads

Use `params`, `mersenne_twister`, `mpi`, `energies`

Called from *`wlcsim_quinn()`*

Call to `cof_path_by_energy_type()`, `save_rephistory()`, `adapt_cof()`

subroutine worker_node(*wlc_p*)

Worker node for MPI parallel tempering. Recieves Coefficients from head node, runs many Monte-Carlo steps, then returns x values to head node.

Parameters **wlc_p** *[wlcsim_params,inout]*

Use `params`, `energies` (`energyof()`, `number_of_energy_types()`), `mpi`, `mersenne_twister`

Called from *wlcsim_quinn()*

Call to `stop_if_err()`, *startworker()*, `schedule()`, `calculate_energies_from_scratch()`, `verify_energies_from_scratch()`, *mcsim()*, *replicaexchange()*

Subroutines and functions

subroutine startworker()

Start worker node.

Use `params`, `mpi`, `energies` (`number_of_energy_types()`, `energyof()`)

Called from *worker_node()*

subroutine replicaexchange()

This checks in with the mpi head node to For parallel tempering of the form: E = cof*x 1: Tell head node the x value 2: Recive replica assignment from head node 3: Recive assigned cof value

Use `params` (`dp()`, `maxfilenamelen()`, `epsapprox()`), `mpi`, `energies` (`number_of_energy_types()`, `energyof()`)

Called from *worker_node()*

5.6 Tips

This page contains usefull tips that we have found usefull for using this codebase. These may be helpful for your workflow.

5.6.1 Running Remotely

When running wlcsim remotely you may want to use tmux so that you don't get logged off while wlcsim is running. More documentation on this comming shortly.

5.6.2 The defines.inc file

You will likely end up with many different defines.inc files. Generally I keep a few examples around in *input/exampledefines*. One of the best tools for comparing defines files is *vim -d defines.inc path/other_defines.inc*. Look up vim diff online for more shortcuts when using this, they are really helpfull.

Each setting comes with a description. But you will likely run into cases where the description isn't enough. Because the settings are named with uinque strings, it is easy to grep the code base for them. For example:

grep -r "WLC_P__WHATEVERSETTING" src/

5.6.3 Running / storing simulations

Genearally I coppy the *entire* code base around when I want to run it again with different settings. The code is small compaired to the output so this donesn't really take up much extra space. I recompile every time I run with *make clean*. Since compiling *wlcsim* is fast compaired to running you might as well.

5.6.4 Compiling options

When debugging you likely want to run with *FCFLAGS = ${PEDANTICFLAGS}* in *Makefile*. When running long simulations you likely want to switch to *FCFLAGS = ${FASTFLAGS}*. Be sure to *make clean* after changing this setting.

5.6.5 Trouble Shooting

When running Monte Carlo many different runtime errors will lead to the integrated energy differing from the energy when it is calculated again from scratch. You'll want to moniter the error output for this warning. This check is one of the best features of this codebase when it comes to code testing, trust us you want it! Here are some tips for if this should occure:

- Turn off some moves and not others. If the error only occures on some subset that can be informative.
- Are the energies really big? This may have tripped this error. You likely don't acctually want them to be that big.
- Which energy type is not matching? Are there multiple that arn't matching.

If the text in the error message appears to be refering to text that isn't actually there.

- The text may have been replaced by the precomilier to the values givin in *src/defines.inc*. This probably means there's an error with one of your settings.

5.7 Meiotic Homolog Pairing

In collaboration with the Burgess lab at UC Davis, we are interested in simulating chromosomes diffusing in a budding yeast nucleus in order to compare to their single-particle tracking data of homologous (and heterologous) loci diffusing in early meiosis I.

We don't have concrete measurements of the viscoelasticity of these nuclei, we will assume they are water-like (i.e. $\alpha = 1$ in a fractional Langevin model of polymer diffusion). And since we are interested in large-scale motion (frame rate in our real movies is already 30s), we can simply use the Rouse model for the behavior of each chromosome.

5.7.1 Testing Code

The :mod:`wlcsim.bd.rouse` module was originally written by Bruno Beltran to replace Andrew Spakowitz/Steph Weber/Elena Koslover's FORTRAN77 Brownian dynamics codebase for fractional ssWLC simulation. We want to do some tests so make sure that the new code behaves as expected in all limiting cases with known behavior.

Rouse behavior

The simulations should reproduce the Gaussian statistics associated with a Rouse polymer with the given parameters.

This includes an MSD that scales like

$$\begin{cases} 6D(N/\hat{N})t \\ 1.9544100b\sqrt{Dt} \\ 6(D/\hat{N})t \end{cases}$$

and Gaussian inter-bead distributions with mean spacing $L_0 b$.

In order to see this behavior, we turn off the confinement by setting the strength of the confinement force `Aex` to 0, and do not tether any beads to the confinement nor pair any loci together

[134]:
```
import numpy as np
import wlcsim
from wlcsim.bd import rouse, homolog
N = int(1e2+1); L = 17.475; R = 1; b = 0.015; D = 2e1 # ChrV
dt = rouse.recommended_dt(N, L, b, D)
Aex = 0; # no confinement
N_tot, loop_list = homolog.points_to_loops_list(N, []) # no loops
tether_list = np.array([]).astype(int) # no tethered loci
```

The following derived parameters are explained in the docs for :mod:`wlcsim.bd.rouse`.

[135]:
```
Nhat = L/b; L0 = L/(N-1); Dhat = D*N/Nhat; bhat = np.sqrt(L0*b)
```

Saving all `1e6` time steps doesn't take up THAT much memory, but calculating the taMSD (a O(N^2) operation) for that many time points is prohibitive.

We use `bnp.loglinsample` to save sets of equispaced time points (in chunks of length `Nlin`) that cover all time scales of our simulation. `Nlin` is chosen to be small enough that O(`Nlin`^2) is not too big.

You can think of `loglinsample` as returning a bunch of separate arrays

```
1) `np.arange(0, dt*Nlin, dt)`
2) `np.arange(0, dt*Nlin**2, Nlin*dt)`
3) ...
4) `np.linspace(0, Nt*dt, Nlin)`
```

we will compute the MSD using the equally-spaced points in each of these arrays, then recombine that into a single curve below.

[136]:
```
Nt = 1e6; # total number of equi-space time steps
Nlin = 1e4; # see docstring of loglinsample
t = np.arange(0, Nt*dt, dt) # take Nt time steps
from bruno_util import numpy as bnp
t_i, i_i = bnp.loglinsample(Nt, Nlin, 0.6)
t_save = t[t_i]
```

[137]:
```
i_i = [np.round(i).astype(int) for i in i_i] # numba workaround
```

Simulate for long enough to see the cross-over into the :math:`t^{1/2}` regime. (This simulation takes about 4min on my laptop).

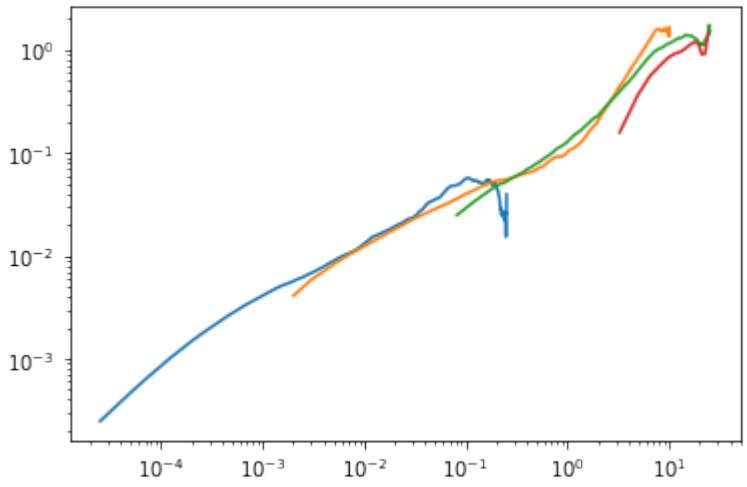
[139]:
```
# run actual simulation
x = homolog._jit_rouse_homologs(N, N_tot, tether_list, loop_list, L, b, D, Aex, R, R,␣
↪R, t, t_save)
```

Post-processing simulation a score or so seconds per `_get_bead_msd` call (time averaged MSD calculation is expensive)

[140]:
```
import wlcsim.bd.runge_kutta
sim_ts = []
sim_msds = []
for i in i_i:
    sim_ts.append(t_save[i])
    # extract msd of 51st (of 101) bead from simulation
    m, c = wlcsim.bd.runge_kutta._get_bead_msd(x[i], 51)
    sim_msds.append(m/c)
```

First plot the MSD curves separately, to demonstrate that the clever time-point choice worked correctly

[141]:
```
for t, msd in zip(sim_ts, sim_msds):
    plt.plot(t[1:], msd[1:])
plt.yscale('log'); plt.xscale('log')
```



now combine them altogether (this is what you would normally do)

[142]:
```
sim_msd = np.zeros_like(t_save)
# for each set of equi-spaced time points we used
for i in i_i:
    # extract msd of 51st (of 101) bead from simulation
    m, c = wlcsim.bd.runge_kutta._get_bead_msd(x[i], 51)
    # recombine into single array, overwriting values computed from
    # more tightly-space points with those from less tightly-spaced
    sim_msd[i[1:]] = m[1:]/c[1:]
```

compare MSD curves to analytical result (by truncating/choosing a finite number of modes matching N/2 beads, we recover the exact analytical theory for a finite rouse chain from that of the infinite chain)

[143]:
```
import matplotlib.pyplot as plt
# exclude t=0 to make log/log msd look good
plt.plot(t_save[1:], 6*D*(N/Nhat)*t_save[1:], label='Short-time behavior')
plt.plot(t_save[1:], 1.9544100*b*np.sqrt(D*t_save[1:]), label='Rouse-like behavior')
plt.plot(t_save[1:], 6*D*t_save[1:]/Nhat, label='Long-time behavior')
plt.plot(t_save[1:], wlcsim.analytical.rouse.rouse_mid_msd(t_save[1:],
            b, Nhat, D, num_modes=int(N/2)), 'k', label='Analytical MSD')
```

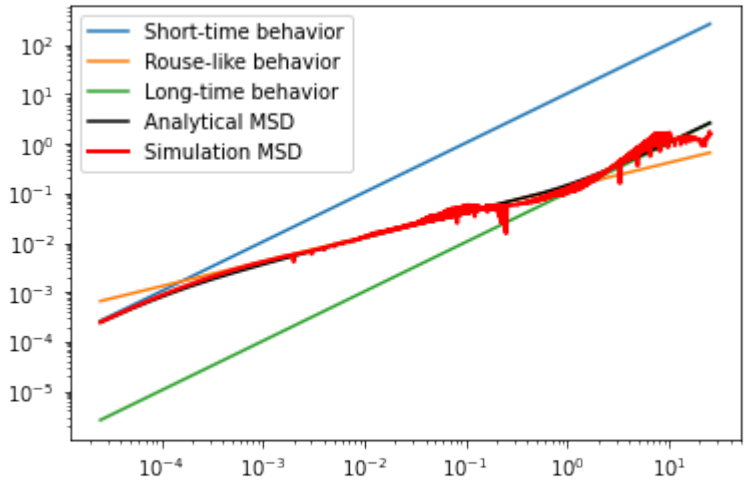
(continues on next page)

(continued from previous page)

```
plt.plot(t_save[1:], sim_msd[1:], 'r', label='Simulation MSD', lw=2)

plt.yscale('log'); plt.xscale('log');
plt.legend()
```

[143]:
```
<matplotlib.legend.Legend at 0x7f3f22d66b50>
```



Ornstein-Uhlenbeck test

Two-bead chains can be well understood as an Ornstein-Uhlenbeck process in the coordinates of one of the beads. The position autocorrelation of the second bed should be an exponential

$$\langle x_s - x_0, x_t - x_0 \rangle = \frac{k_B T}{k} \exp\left(-k|t|/\xi\right)$$

[3]:
```
N = 2; L = 3; R = 1000; b = 1; D = 5 # two O-U processes
dt = rouse.recommended_dt(N, L, b, D)
Aex = 0; # no confinement
N_tot, loop_list = rouse.homolog_points_to_loops_list(N, []) # no loops
tether_list = np.array([]).astype(int) # no tethered loci
```

we run 100 copies of the process, to get better stats (so 200 total 3D O-U processes, since each simulation has two "homologs")

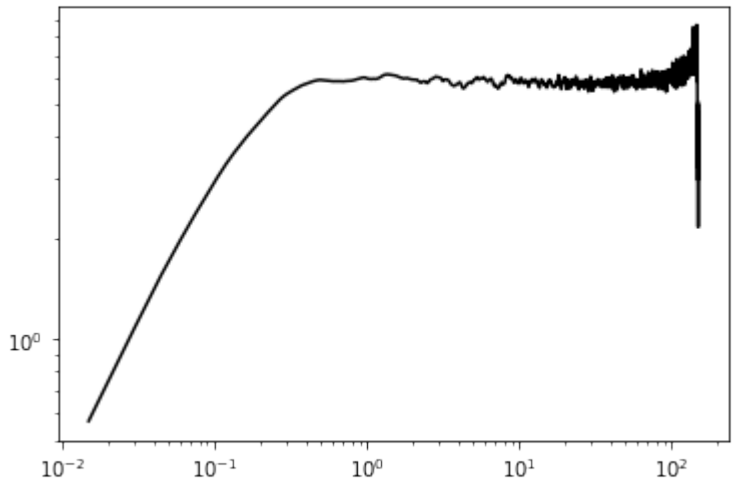
[4]:
```
Nt = 1e4;
t = np.arange(0, Nt*dt, dt) # take Nt time steps
X = []
for i in range(100):
    x = rouse._jit_rouse_homologs(N, N_tot, tether_list, loop_list, L, b, D, Aex, R,␣
↪R, R, t, t)
    X.append(np.stack([x[:,1,:] - x[:,0,:], x[:,3,:] - x[:,2,:]]))
X = np.concatenate(X, axis=0)
```

[7]:
```
X.shape # (num_samples, num_t, d)
```

[7]:
```
(200, 10000, 3)
```

[6]:
```
# abuse get_bead_msd, which expects X.shape == (num_t, num_beads, d)
# to get one of the sample's MSDs by pretending its a "bead"
sim_msd, count = wlcsim.bd.runge_kutta._get_bead_msd(np.transpose(X, [1, 0, 2]), 1)
```

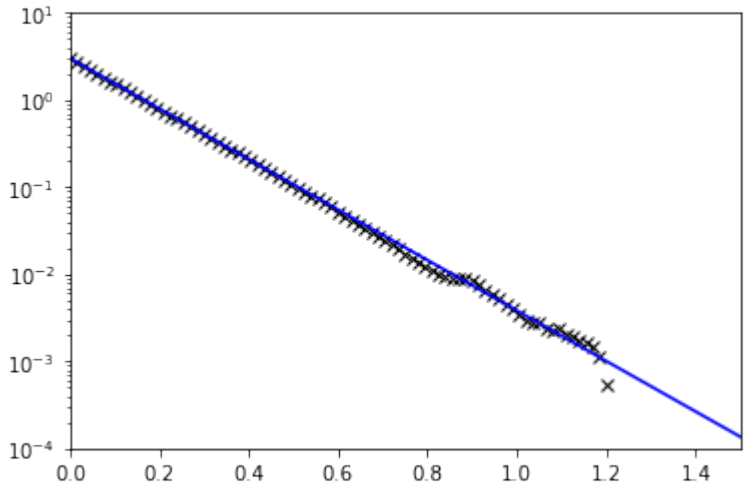
[8]:
```
plt.plot(t[1:], sim_msd[1:]/count[1:], 'k')
plt.yscale('log'); plt.xscale('log')
```



[9]:
```
corr, count = wlcsim.bd.runge_kutta._get_vector_corr(X)
```

[25]:
```
Nhat = L/b; L0 = L/(N-1); Dhat = D*N/Nhat; bhat = np.sqrt(L0*b)
plt.plot(t, corr/count, 'kx')
kbT_over_k = 3*bhat**2/3 # bhat**2/3 per dimension
k_over_xi = 3*(2*Dhat)/bhat**2 # extra (2*Dhat) because it's two beads tethered, not␣
↪one tether to origin
plt.plot(t, kbT_over_k * np.exp(-k_over_xi*t), 'b')
plt.yscale('log')
plt.xlim([0, 1.5])
plt.ylim([1e-4, 1e1])
```

[25]:
```
(0.0001, 10.0)
```



Confinement

As seen above, if the confinement size is made "weak enough", the simulation behaves as it should in the absense of a confinement.

The MSD should plateau at the radius of the confinement.

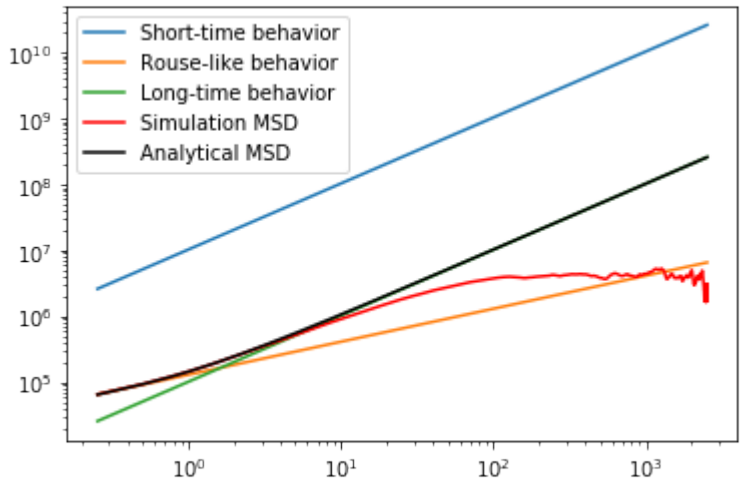
[29]:
```
N = int(1e2+1); L = 17.475; R = 1.000; b = 0.015; D = 2e1 # ChrV
dt = rouse.recommended_dt(N, L, b, D)
Aex = 100; # multiplier on confinement force
N_tot, loop_list = rouse.homolog_points_to_loops_list(N, []) # no loops
tether_list = np.array([]).astype(int) # no tethered loci
```

[35]:
```
Nt = 1e8; Nt_save = 1e4;
t = np.arange(0, Nt*dt, dt) # take Nt time steps
t_save = t[::int(Nt/Nt_save)] # save only a subsample
# run actual simulation
x = rouse._jit_rouse_homologs(N, N_tot, tether_list, loop_list, L, b, D, Aex, R, R, R,
↪ t, t_save)
```

[36]:
```
# extract msd of 51st (of 101) bead from simulation
sim_msd, count = wlcsim.bd.runge_kutta._get_bead_msd(x, 51)
```

[37]:
```
Nhat = L/b; L0 = L/(N-1); Dhat = D*N/Nhat; bhat = np.sqrt(L0*b)
# exclude t=0 to make log/log msd look good
plt.plot(t_save[1:], 6*D*(N/Nhat)*t_save[1:], label='Short-time behavior')
plt.plot(t_save[1:], 1.9544100*b*np.sqrt(D*t_save[1:]), label='Rouse-like behavior')
plt.plot(t_save[1:], 6*D*t_save[1:]/Nhat, label='Long-time behavior')
plt.plot(t_save[1:], sim_msd[1:]/count[1:], 'r', label='Simulation MSD')
plt.plot(t_save[1:], wlcsim.analytical.rouse.rouse_mid_msd(t_save[1:],
            b, Nhat, D, num_modes=int(N/2)), 'k', label='Analytical MSD')
plt.yscale('log'); plt.xscale('log');
plt.legend()
```

[37]:
```
<matplotlib.legend.Legend at 0x7f0ac12c9710>
```



Beads that are "tethered" to the confinement should be preferentially "close" to it compared to the rest of the beads in the simulation.

[51]:
```
N = int(1e2+1); L = 17.475; R = 1.000; b = 0.015; D = 2e1 # ChrV
dt = rouse.recommended_dt(N, L, b, D)
Aex = 100; # multiplier on confinement force
N_tot, loop_list = rouse.homolog_points_to_loops_list(N, []) # no loops
tether_list = np.array([0, 51, 100]).astype(int) # tethered telomeres and centromere
```

[52]:
```
%%debug
Nt = 1e6; Nt_save = 1e4;
t = np.arange(0, Nt*dt, dt) # take Nt time steps
t_save = t[::int(Nt/Nt_save)] # save only a subsample
# run actual simulation
x = rouse._jit_rouse_homologs(N, N_tot, tether_list, loop_list, L, b, D, Aex, R, R, R,
↪ t, t_save)
```

```
NOTE: Enter 'c' at the ipdb>  prompt to continue execution.
> <string>(2)<module>()

ipdb> s
> <string>(3)<module>()

ipdb> s
> <string>(4)<module>()

ipdb>
> <string>(6)<module>()

ipdb>
--Call--
> /home/bbeltr1/.miniconda/lib/python3.7/site-packages/numba/dispatcher.py(521)typeof_
↪pyval()
    519         return "%s(%s)" % (type(self).__name__, self.py_func)
    520
--> 521     def typeof_pyval(self, val):
    522         """
    523         Resolve the Numba type of Python value *val*.

ipdb> n
> /home/bbeltr1/.miniconda/lib/python3.7/site-packages/numba/dispatcher.py(529)typeof_
↪pyval()
    527         # Not going through the resolve_argument_type() indirection
    528         # can save a couple µs.
--> 529         try:
    530             tp = typeof(val, Purpose.argument)
    531         except ValueError:

ipdb>
> /home/bbeltr1/.miniconda/lib/python3.7/site-packages/numba/dispatcher.py(530)typeof_
↪pyval()
    528         # can save a couple µs.
    529         try:
--> 530             tp = typeof(val, Purpose.argument)
    531         except ValueError:
    532             tp = types.pyobject

ipdb>
> /home/bbeltr1/.miniconda/lib/python3.7/site-packages/numba/dispatcher.py(534)typeof_
↪pyval()
```

(continues on next page)

(continued from previous page)

```
    532             tp = types.pyobject
    533         else:
--> 534             if tp is None:
    535                 tp = types.pyobject
    536         return tp

ipdb>
> /home/bbeltr1/.miniconda/lib/python3.7/site-packages/numba/dispatcher.py(536)typeof_
↪pyval()
    534             if tp is None:
    535                 tp = types.pyobject
--> 536         return tp
    537
    538

ipdb>
--Return--
array(float64, 1d, A)
> /home/bbeltr1/.miniconda/lib/python3.7/site-packages/numba/dispatcher.py(536)typeof_
↪pyval()
    534             if tp is None:
    535                 tp = types.pyobject
--> 536         return tp
    537
    538

ipdb>
--Call--
> /home/bbeltr1/.miniconda/lib/python3.7/site-packages/numba/dispatcher.py(325)_
↪compile_for_args()
    323         return self._compiling_counter
    324
--> 325     def _compile_for_args(self, *args, **kws):
    326         """
    327         For internal use.  Compile a specialized version of the function

ipdb> s
> /home/bbeltr1/.miniconda/lib/python3.7/site-packages/numba/dispatcher.py(330)_
↪compile_for_args()
    328         for the given *args* and *kws*, and return the resulting callable.
    329         """
--> 330         assert not kws
    331
    332         def error_rewrite(e, issue_type):

ipdb>
> /home/bbeltr1/.miniconda/lib/python3.7/site-packages/numba/dispatcher.py(332)_
↪compile_for_args()
    330         assert not kws
    331
--> 332         def error_rewrite(e, issue_type):
    333             """
    334             Rewrite and raise Exception `e` with help supplied based on the

ipdb>
> /home/bbeltr1/.miniconda/lib/python3.7/site-packages/numba/dispatcher.py(345)_
↪compile_for_args()
```

(continues on next page)

(continued from previous page)

```
    343                 reraise(type(e), e, None)
    344
--> 345         argtypes = []
    346         for a in args:
    347             if isinstance(a, OmittedArg):

ipdb>
> /home/bbeltr1/.miniconda/lib/python3.7/site-packages/numba/dispatcher.py(346)_
↪compile_for_args()
    344
    345         argtypes = []
--> 346         for a in args:
    347             if isinstance(a, OmittedArg):
    348                 argtypes.append(types.Omitted(a.value))

ipdb>
> /home/bbeltr1/.miniconda/lib/python3.7/site-packages/numba/dispatcher.py(347)_
↪compile_for_args()
    345         argtypes = []
    346         for a in args:
--> 347             if isinstance(a, OmittedArg):
    348                 argtypes.append(types.Omitted(a.value))
    349             else:

ipdb>
> /home/bbeltr1/.miniconda/lib/python3.7/site-packages/numba/dispatcher.py(350)_
↪compile_for_args()
    348                 argtypes.append(types.Omitted(a.value))
    349             else:
--> 350                 argtypes.append(self.typeof_pyval(a))
    351         try:
    352             return self.compile(tuple(argtypes))

ipdb>
--Call--
> /home/bbeltr1/.miniconda/lib/python3.7/site-packages/numba/dispatcher.py(521)typeof_
↪pyval()
    519         return "%s(%s)" % (type(self).__name__, self.py_func)
    520
--> 521     def typeof_pyval(self, val):
    522         """
    523         Resolve the Numba type of Python value *val*.

ipdb>
> /home/bbeltr1/.miniconda/lib/python3.7/site-packages/numba/dispatcher.py(529)typeof_
↪pyval()
    527         # Not going through the resolve_argument_type() indirection
    528         # can save a couple µs.
--> 529         try:
    530             tp = typeof(val, Purpose.argument)
    531         except ValueError:

ipdb>
> /home/bbeltr1/.miniconda/lib/python3.7/site-packages/numba/dispatcher.py(530)typeof_
↪pyval()
    528         # can save a couple µs.
    529         try:
```

(continues on next page)

(continued from previous page)

```
--> 530             tp = typeof(val, Purpose.argument)
    531         except ValueError:
    532             tp = types.pyobject

ipdb>
--Call--
> /home/bbeltr1/.miniconda/lib/python3.7/site-packages/numba/typing/typeof.
↪py(24)typeof()
     22 _TypeofContext = namedtuple("_TypeofContext", ("purpose",))
     23
---> 24 def typeof(val, purpose=Purpose.argument):
     25     """
     26     Get the Numba type of a Python value for the given purpose.

ipdb>
> /home/bbeltr1/.miniconda/lib/python3.7/site-packages/numba/typing/typeof.
↪py(29)typeof()
     27     """
     28     # Note the behaviour for Purpose.argument must match _typeof.c.
---> 29     c = _TypeofContext(purpose)
     30     ty = typeof_impl(val, c)
     31     if ty is None:

ipdb> exit
```

[42]:
```
%matplotlib qt5
```

[43]:
```
hv = rouse.HomologViewer(x, N, loop_list)
```

Connectivity

If we make the only paired bead be one of the end beads, then the polymers should behave as one long linear polymer.

[ ]:
```
N = int(1e2+1); L = 17.475; R = 1.000; b = 0.015; D = 2e1 # ChrV
dt = rouse.recommended_dt(N, L, b, D)
Aex = 0; # no confinement
N_tot, loop_list = rouse.homolog_points_to_loops_list(N, [0]) # only one end tethered
tether_list = np.array([]).astype(int) # no tethered loci
```

[ ]:
```
x = rouse._jit_rouse_homologs(N, N_tot, tether_list, loop_list, L, b, D, Aex, R, R, R,
↪ t, t_save)
```

The above was verified to work visually using

[ ]:
```
%matplotlib qt5
hv = rouse.HomologViewer(x, int(N), loop_list)
```

If we make both end beads be homologously paired, then the two polymers should behave as one larger ring polymer.

[ ]:
```
N = int(1e2+1); L = 17.475; R = 1.000; b = 0.015; D = 2e1 # ChrV
dt = rouse.recommended_dt(N, L, b, D)
Aex = 0; # no confinement
```

(continues on next page)

(continued from previous page)

```
N_tot, loop_list = rouse.homolog_points_to_loops_list(N, [0, 100]) # only one end␣
↪tethered
tether_list = np.array([]).astype(int) # no tethered loci
```

[ ]:
```
x = rouse._jit_rouse_homologs(N, N_tot, tether_list, loop_list, L, b, D, Aex, R, R, R,
↪ t, t_save)
```

The above was verified to work visually using

[ ]:
```
%matplotlib qt5
hv = rouse.HomologViewer(x, int(N), loop_list)
```

5.7.2 Generating Pairing Sites

Our simulation as written is purely Brownian Dynamics, (no Gillespie component). This means that we treat the "existing" homolog paired sites as fixed and ask how the polymer fluctuates assuming no new connections are formed.

Thus, our method for choosing which sites are "paired" in a given simulation will greatly affect our results.

By default, all of our simulations will simply choose pairing sites uniformly according to some density `FP`. That is, each individual bead will have a probability $0 \leq \texttt{FP} \leq 1$ of being paired.

However, we present below an alternative strategy for choosing these sites, that involves explicitly simulating the dynamic process of homolog pairing.

$k_h \lll k_{\textbf{loop}}$

In the limit as the rate of polymer loop formation eclipses the rate of stable homolog junction formation (i.e. if it requires many "kissing" events to form a stable homolog junction), then we can treat the homolog pairing process approximately as a Markov chain where the transition rates from a given loci being unpaired to paired simply are proportional to the looping probabilities for the ring polymer defined by its nearest neighbors (where there already exists a junction) to the left and right.

No Unpairing

In the case $k_h \lll k_{\text{loop}}$ and where we don't allow unpairing events to occur, we can actually analytically compute the final loop size distribution.

[1]:
```
#TODO copy from notes to here
```

[ ]:
```
```

5.7.3 Simulations

Simulate two rouse chains in confinement, various connectivities.

The ura3 gene is on ChrV, spanning bases 116167–116970. ChrV is 576874 bases, and budding yeast have a mean linker length of 15bp (as estimated based on a nucleosome repeat length (NRL) of ~165bp). This corresponds to a WLC with an effective persistence length of ~15nm in theta-solvent conditions (see Beltran & Kannan et al, PRL 2019). This number has elsewhere been estimated to be as low as 5nm experimentally (Hajjoul et al, Genome Research 2013). Since only 15/165~9% of DNA is in linkers (linear distance along this effective WLC) ChrV's ~3496

nucleosomes each cover about one third of a Kuhn length. Meaning the chain can be thought of as being made of 1165 Kuhn lengths of 15nm each (for 17475nm total).

So in μm, L=17.475, b=0.015, and N=101 is used to give 100 segments of equal length. (which would make the ura3 locus bead 20).

The nucleus has a radius of about a micron, and loci reach the confinement after about a minute. Because their Rouse diffusion has basically the form $1.9544100 * b * D^{1/2} t^{1/2}$, then we can calculate that $1^2 \mu m^2 = 1.95441 \times 0.015 \mu m \times D^{1/2} um/s^{1/2} \times 60^{1/2} s^{1/2}$. In other words, $D \approx 19.39 \mu m^2/s$. So R=1.000, D=2e1.

Notice that since the terminal relaxation time of the polymer is $N^2 b^2 / D \ldots$

Assuming that we can treat the maximum attained MSD value as the confinement size, approximately, we can just use teh MSD at time 30 as a proxy for the confinement size, since it's basically flat after that.

If we look at the distribution of these values in the data, we see that the squared displacement has a distribution which is linear

[ ]:
```
msds = msds.msd(burgess.df_flat, mscd=True, include_z=True, traj_group=burgess.cell_
↪cols, groups=burgess.cell_cols)
m = sds.reset_index()
data = m[m['delta'] == 30]['mean'].values
data = data[~np.isnan(data)]
Y, X = np.histogram(data)
scipy.stats.linregress(X[3:], Y[2:])
```

the above gives output of

```
>>> LinregressResult(slope=-648.6107382252995, intercept=1340.988095238095, rvalue=-0.
↪9865990280608326, pvalue=5.956260174887604e-06, stderr=43.79162733110961)
```

if we don't square the values, we see that we get a peak at around 0.8um, which I guess will be the mean confinement radius. this is close enough to the current simulation value of ~1um.

If we use t=60, 90, or 120s instead, we still get values around 1um, like 1.02um to 1.12um. . .

5.8 Example plots for Frank Elastic Constants

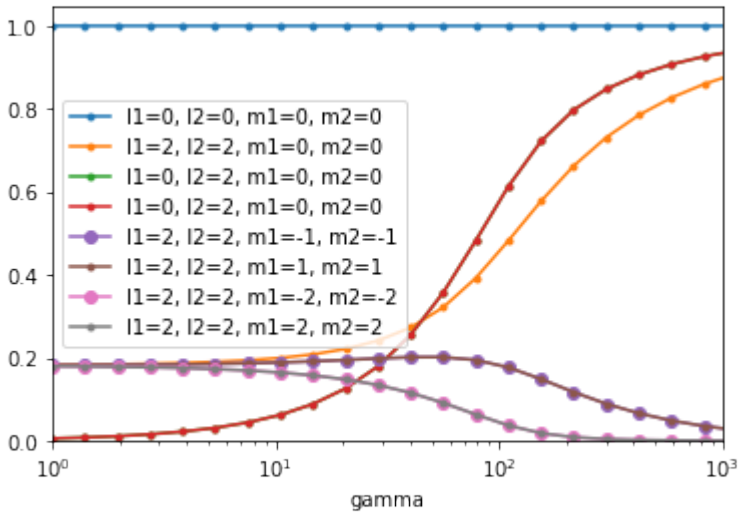
[1]:
```
from FrankElastic.plot_frank import *
```

[2]:
```
%matplotlib inline
import os.path as path
```

5.8.1 Plot correlations

[3]:
```
if not path.exists("FrankElastic/Sdata.p"):
    get_data_for_Splot("FrankElastic/Sdata.p", nthreads=25, N=0.05, npts=25)
```

[4]:
```
plot_S("FrankElastic/Sdata.p")
```

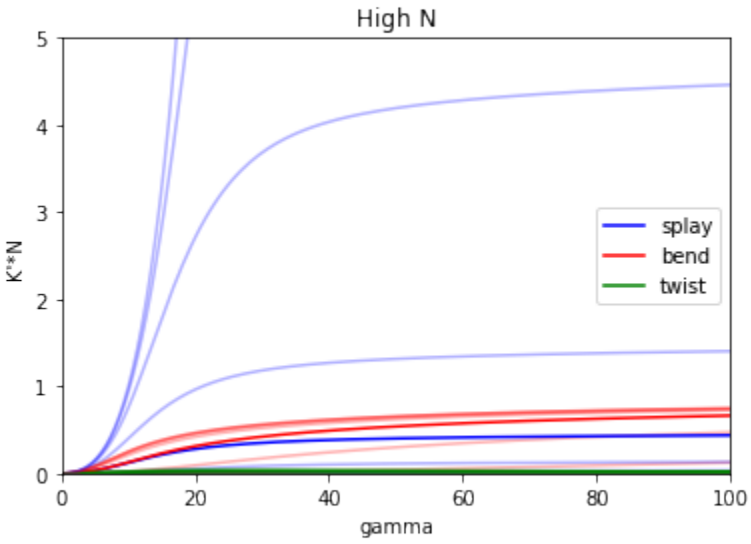


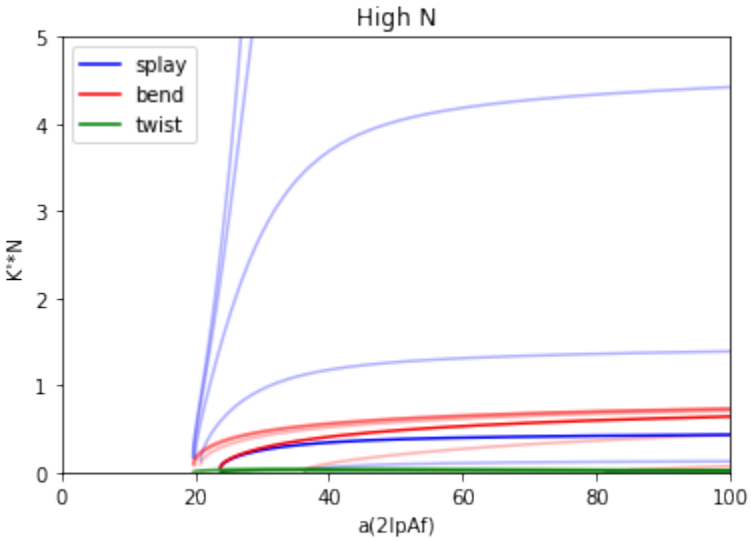
5.8.2 Plots of Frank elastic constants

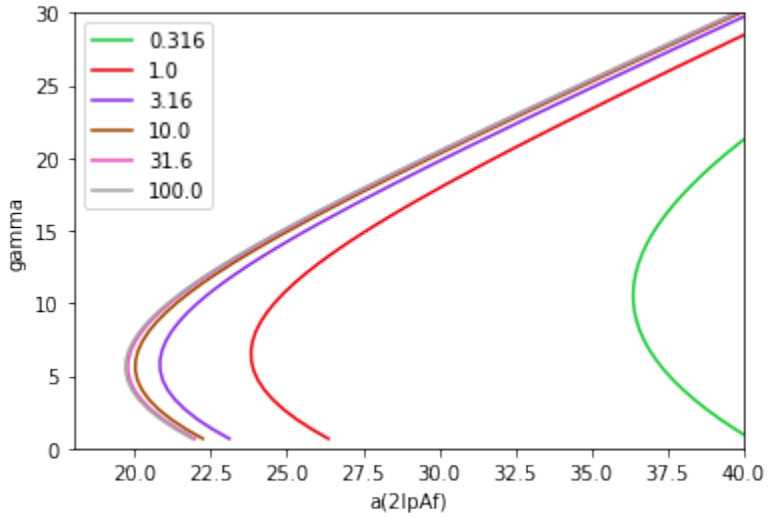
[5]:
```
if not path.exists("FrankElastic/highN.p"):
    get_Multi_frank_data(saveas="highN.p", gammaN=False, nthreads=25, npts=150,␣
↪maxgamma=100)
```

[6]:
```
if not path.exists("FrankElastic/lowN.p"):
    get_Multi_frank_data(saveas="lowN.p", gammaN=True, nthreads=25, npts=48,␣
↪maxgamma=35)
```

[7]:
```
Multi_frank_plot(load="FrankElastic/highN.p")
```

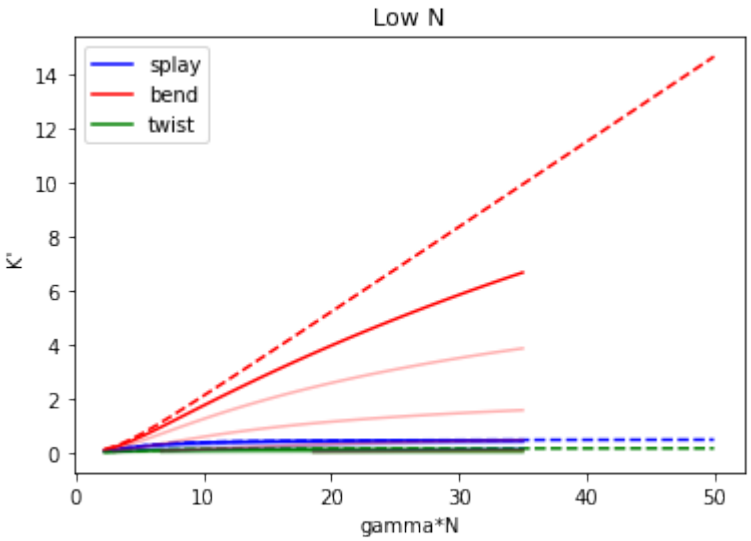


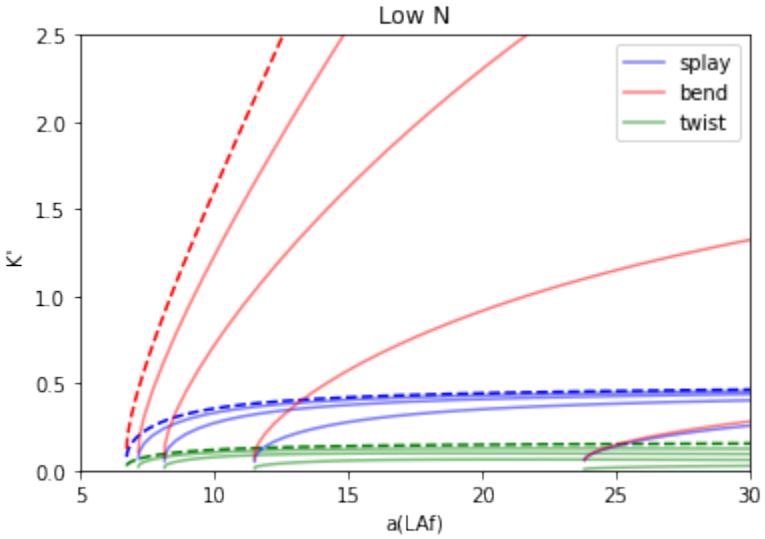


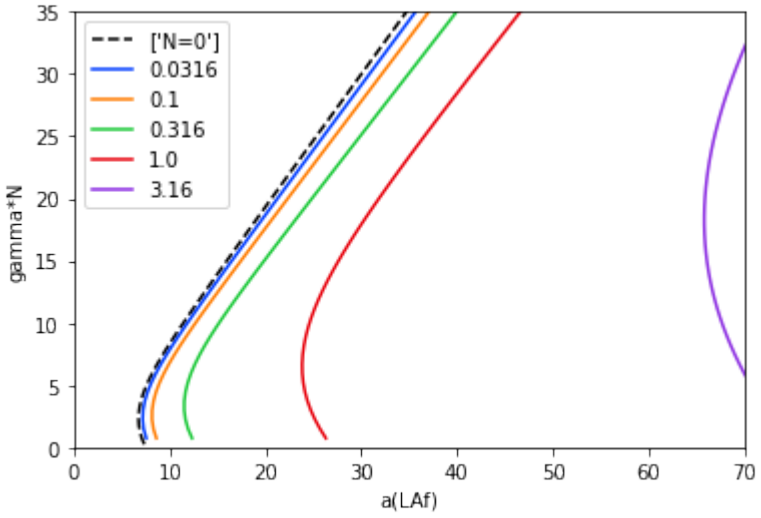


[8]:
```
Multi_frank_plot(load="FrankElastic/lowN.p")
```

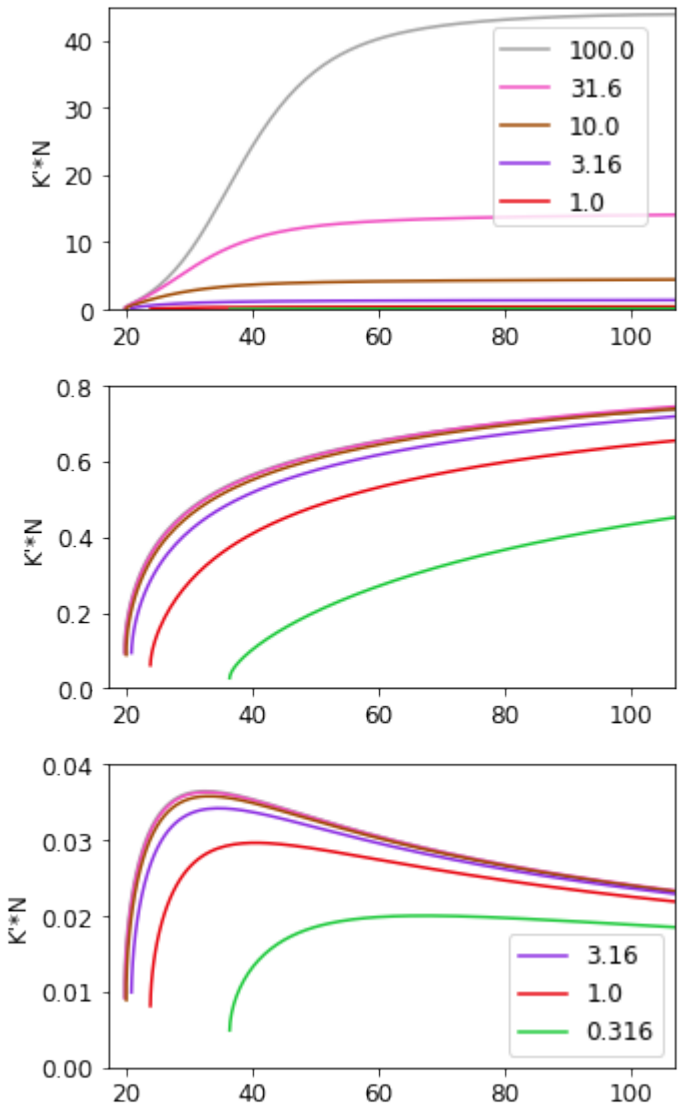
```
/home/users/qmac/combinedCodeBase/wlcsim/wlcsim/FrankElastic/rigidrod.py:119:␣
↪RuntimeWarning: invalid value encountered in double_scalars
  a_prime[ii] = 15*gammas[ii]*eM[0][0,0]/(4*np.sqrt(np.pi)*eM[0][0,2])
```


Low N
splay
bend
twist
K'
gamma*N
0
2
4
6
8
10
12
14
0
10
20
30
40
50


Low N
splay
bend
twist
K'
a(LAf)
0.0
0.5
1.0
1.5
2.0
2.5
5
10
15
20
25
30

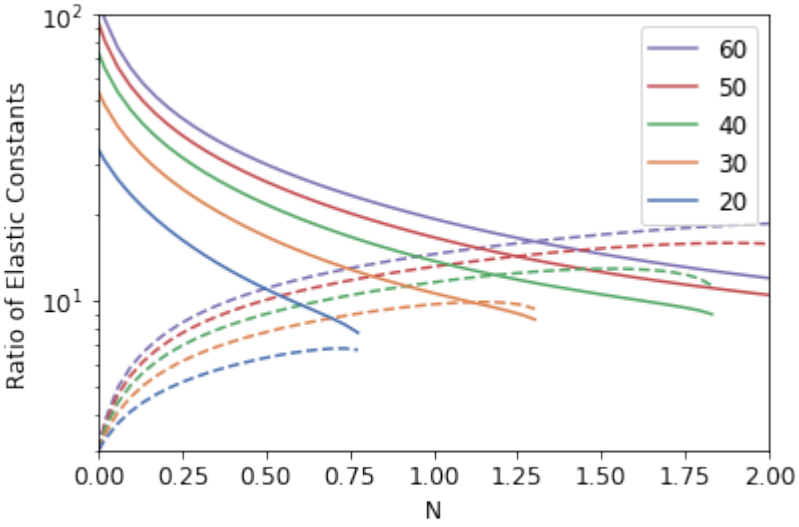

['N=0']
0.0316
0.1
0.316
1.0
3.16
gamma*N
a(LAf)
0
5
10
15
20
25
30
35
0
10
20
30
40
50
60
70

[9]:
```
seperate_bend_twist_splay("FrankElastic/highN.p")
```



[10]:
```
if not path.exists("FrankElastic/Nsweep.p"):
    get_Nsweep_data(saveas="FrankElastic/Nsweep.p", nthreads=25)
```

[11]:
```
plotNsweep("FrankElastic/Nsweep.p")
```


10^2
10^1
Ratio of Elastic Constants
60
50
40
30
20
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
N

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

W

Index

Symbols

C

D

E

F

G

H

U

V

W

Y