polychrom

Release 0.1.1

Mirny lab

Aug 21, 2023

CONTENTS

Polychrom is a package for setting up, performing and analyzing polymer simulations of chromosomes. The simulation part is based around VJ Pande's OpenMM library - a GPU-assisted framework for general molecular dynamics simulations. The analysis part is written by the mirnylab.

CHAPTER

ONE

INSTALLATION

Polychrom requires OpenMM, which can be installed through conda: `conda install -c omnia openmm`. See http://docs.openmm.org/latest/userguide/application.html#installing-openmm . In our experience, adding `-c conda-forge` listed in the link above is optional.

CUDA is the fastest GPU-assisted backend to OpenMM. You would need to have the required version of CUDA, or install OpenMM compiled for your version of CUDA.

Other dependencies are simple, and are listed in requirements.txt. All but joblib are installable from either conda/pip, and joblib installs well with pip.

1.1 Installation errors and possible fixes

Error: After installation, importing openmm or running polychrom code leads to the error:

```
version GLIBCXX_3.4.30 not found
```

Fix:

```
conda install -c conda-forge libstdcxx-ng=12
```

CHAPTER

TWO

STRUCTURE

Polychrom is an API, and each simulation has to be set up as a Python script. Simulations are done using a "simulation" module *polychrom.simulation*. Forces that define the simulation are found in *polychrom.forces* and `polychrom.forcekits` modules. Contactmaps from simulated conformations can be generated using *polychrom.contactmaps* module. Loading and saving individual conformations can be done using *polychrom.polymerutils*, while loading/saving whole trajetories is done using *polychrom.hdf5_format*. P(s), R(s), Rg(s) curves and other analyses can be done using *polychrom.polymer_analyses*.

2.1 polychrom.simulation module

2.1.1 Creating a simulation: Simulation class

Both initialization and running the simulation is done by interacting with an instance of *polychrom.simulation.Simulation* class.

Overall parameters

Overall technical parameters of a simulation are generally initialized in the constructor of the Simulation class. *polychrom.simulation.Simulation.__init__()* . This includes

Techcnical parameters not affecting the output of simulations

- Platform (cuda (usually), opencl, or CPU (slow))
- GPU index
- reporter (where to save results): see :py:mod`polychrom.hdf5_reporter`

Parameters affecting the simulation

- number of particles
- integrator (we usually use variable Langevin) + error tolerance of integrator
- collision rate
- Whether to use periodic boundary conditions (PBC)
- timestep (if using non-variable integrator)

Parameters that are changed rarely, but may be useful

- particle mass, temperature and length scale
- kinetic energy at which to raise an error

- OpenMM precision
- Rounding before saving (default is to 0.01)

Starting conformation is loaded using *polychrom.simulation.Simulation.set_data()* method. Many tools for creating starting conformations are in `polychrom.starting_conformations`

Adding forces

Forces define the main aspects of a given simulation. Polymer connectivity, confinement, crosslinks, tethering monomers, etc. are all defined as different forces acting on the particles.

Typicall used forces are listed in *polychrom.forces* module. Forces out of there can be added using *polychrom.simulation.Simulation.add_force()* method.

Forces and their parameters are an essential part of nearly any polymer simulations. Some forces have just a few paramters (e.g. spherical confinement just needs a radius), while other forces may have lots of parameters and can define complex structures. For example, harmonidBondForce with a specially-created bond list was used to create a backbone-plectoneme conformation in Caulobacter simulations (Le et al, Science 2013). Same harmonic bonds that change over time are used to simulate loop extrusion as in (Fudenberg, 2016).

Some forces need to be added together. Those include forces defining polymer connectivity. Those forces are combined into **forcekits**. Forcekits are defined in `polychrom.forcekits` module. The only example of a forcekit for now is defining polymer connectivity using bonds, polymer stiffness, and inter-monomer interaction ("nonbonded force").

Some forces were written for openmm-polymer library and were not fully ported/tested into the polychrom library. Those forces reside in `polychrom.legacy.forces` module. Some of them can be used as is, and some of them would need to be copied to your code and potentially conformed to the new style of defining forces. This includes accepting simulation object as a parameter, and having a `.name` attribute.

Defining your own forces

Each force in *polychrom.forces* is a simple function that wraps creation of an openmm force object. Users can create new forces in the script defining their simulation and add them using add_force method. Good examples of forces are in *polychrom.forces* - all but harmonic bond force use custom forces, and provide explanations of why particular energy function was chosen. Description of the module *polychrom.forces* has some important information about adding new forces.

Running a simulation

To run a simulation, you call `polychrom.simulation.Simulation.doBlock()` method in a loop. Unless specified otherwise, this would save a conformation into a defined reporter. Terminating a simulation is not necessary; however, terminating a reporter using reporter.dump_data() is needed for the hdf5 reporter. This all can be viewed in the example script.

exception polychrom.simulation.**EKExceedsError**

Bases: `Exception`

exception polychrom.simulation.**IntegrationFailError**

Bases: `Exception`

class polychrom.simulation.**Simulation**(***kwargs*)

Bases: `object`

This is a base class for creating a Simulation and interacting with it. All general simulation parameters are defined in the constructor. Forces are defined in *polychrom.forces* module, and are added using *polychrom.simulation.Simulation.add_force()* method.

RG()

Returns

Gyration ratius in units of length (bondlength).

__init__(***kwargs*)

All numbers here are floats. Units specified in a parameter.

Parameters

- **N** (*int*) – number of particles
- **error_tol** (*float, optional*) – Error tolerance parameter for variableLangevin integrator Values of around 0.01 are reasonable for a "nice" simulation (i.e. simulation with soft forces etc). Simulations with strong forces may need 0.001 or less OpenMM manual recommends 0.001, but our forces tend to be "softer" than theirs
- **timestep** (*number*) – timestep in femtoseconds. Mandatory for non-variable integrators. Ignored for variableLangevin integrator. Value of 70-80 are appropriate
- **collision_rate** (*number*) – collision rate in inverse picoseconds. values of 0.01 or 0.05 are often used. Consult with lab members on values.

  In brief, equilibrium simulations likely do not care about the exact dynamics you're using, and therefore can be simulated in a "ballistic" dynamics with col_rate of around 0.001-0.01.

  Dynamical simulations and active simulations may be more sensitive to col_rate, though this is still under discussion/investigation.

  Johannes converged on using 0.1 for loop extrusion simulations, just to be safe.
- **PBCbox** (*(float,float,float) or False; default:False*) – Controls periodic boundary conditions If PBCbox is False, do not use periodic boundary conditions If intending to use PBC, then set PBCbox to (x,y,z) where x,y,z are dimensions of the bounding box for PBC
- **GPU** (*GPU index as a string ("0" for first, "1" for second etc.)*) – Machines with 1 GPU automatically select their GPU.
- **integrator** (*"langevin", "variableLangevin", "verlet", "variableVerlet",*) – "brownian", or tuple containing Integrator from Openmm class reference and string defining integrator type. For user-defined integrators, specify type "brownian" to avoid checking if kinetic energy exceeds max_Ek.
- **mass** (*number or np.array*) – Particle mass (default 100 amu)
- **temperature** (*simtk.units.quantity(units.kelvin), optional*) – Temperature of the simulation. Devault value is 300 K.
- **verbose** (*bool, optional*) – If True, prints a lot of stuff in the command line.
- **length_scale** (*float, optional*) – The geometric scaling factor of the system. By default, length_scale=1.0 and harmonic bonds and repulsive forces have the scale of 1 nm.
- **max_Ek** (*float, optional*) – raise error if kinetic energy in (kT/particle) exceeds this value
- **platform** (*string, optional*) – Platform to use: CUDA (preferred fast GPU platform) OpenCL (maybe slower GPU platofrm, does not need CUDA installed) CPU (medium speed parallelized CPU platform) reference (slow CPU platform for debug)

- **verbose** – Shout out loud about every change.
- **precision** (*str, optional (not recommended to change)*) – single is the default now, because mixed is much slower on 3080 and other new GPUs If you are using double precision, it will be slower by a factor of 10 or so.
- **save_decimals** (*int or False, optional*) – Round to this number of decimals before saving. `False` is no rounding. Default is 2. It gives maximum error of 0.005, which is nearly always harmless but saves up to 40% of storage space (0.6 of the original) Using one decimal is safe most of the time, and reduces storage to 40% of int32. NOTE that using periodic boundary conditions will make storage advantage less.
- **reporters** (*list, optional*) – List of reporters to use in the simulation.

add_force(*force*)

Adds a force or a forcekit to the system.

dist(*i*, *j*)

Calculates distance between particles i and j.

Added for convenience, and not for production code. Not for use in large for-loops.

do_block(*steps=None*, *check_functions=[]*, *get_velocities=False*, *save=True*, *save_extras={}*)

performs one block of simulations, doing steps timesteps, or steps_per_block if not specified.

Parameters

- **steps** (*int or None*) – Number of timesteps to perform.
- **increment** (*bool, optional*) – If true, will not increment *self.block* and *self.steps* counters
- **check_functions** (*list of functions, optional*) – List of functions to call every block. coordinates are passed to a function. If a function returns False, simulation is stopped.
- **get_velocities** (*bool, default=False*) – If True, will return velocities
- **save** (*bool, defualt=True*) – If True, save results of this block
- **save_extras** (*dict*) – A dict of (key, value) with additional info to save

get_data()

Returns an Nx3 array of positions

get_scaled_data()

Returns data, scaled back to PBC box

init_positions()

Sends particle coordinates to OpenMM system. If system has exploded, this is

used in the code to reset coordinates.

init_velocities(*temperature='current'*)

Initializes particles velocities

Parameters

temperature (*temperature to set velocities (default: temerature of the simulation)*) –

`initialize`(***kwargs*)

Initialize, particles, velocities for the first time. Only need to use this function if your system has no forces (free Brownian particles). Otherwise _apply_force() will execute these lines to add particles to the system, initialize their positions/velocities, initialize the context.

`local_energy_minimization`(*tolerance=0.3*, *maxIterations=0*, *random_offset=0.02*)

A wrapper to the build-in OpenMM Local Energy Minimization

See caveat below

Parameters

- **tolerance** (*float*) – It is something like a value of force below which the minimizer is trying to minimize energy to. see openmm documentation for description

  Value of 0.3 seems to be fine for most normal forces.

- **maxIterations** (*int*) – Maximum # of iterations for minimization to do. default: 0 means there is no limit

  This is relevant especially if your simulation does not have a well-defined energy minimum (e.g. you want to simulate a collapse of a chain in some potential). In that case, if you don't limit energy minimization, it will attempt to do a whole simulation for you. In that case, setting a limit to the # of iterations will just stop energy minimization manually when it reaches this # of iterations.

- **random_offset** (*float*) – A random offset to introduce after energy minimization. Should ideally make your forces have realistic values.

  For example, if your stiffest force is polymer bond force with "wiggle_dist" of 0.05, setting this to 0.02 will make separation between monomers realistic, and therefore will make force values realistic.

  See why do we need it in the caveat below.

Notes

If using variable langevin integrator after minimization, a big error may happen in the first timestep. The reason is that enregy minimization makes all the forces basically 0. Variable langevin integrator measures the forces and assumes that they are all small - so it makes the timestep very large, and at the first timestep it overshoots completely and energy goes up a lot.

The workaround for now is to randomize positions after energy minimization

`print_stats`()

Prints detailed statistics of a system. Will be run every 50 steps

`reinitialize`()

Reinitializes the OpenMM context object. This should be called if low-level parameters, such as parameters of forces, have changed

`set_data`(*data*, *center=False*, *random_offset=1e-05*, *report=True*)

Sets particle positions

Parameters

- **data** (*Nx3 array-like*) – Array of positions
- **center** (*bool or "zero", optional*) – Move center of mass to zero before starting the simulation if center == "zero", then center the data such as all positions are positive and start at zero

- **random_offset** (*float or None*) – add random offset to each particle Recommended for integer starting conformations and in general
- **report** (*bool, optional*) – If set to False, will not report this action to reporters.

set_velocities(*v*)

Set initial velocities of particles.

Parameters

v (*(N, 3) array-like*) – initial x, y, z velocities of the N particles

show(*shifts=[0.0, 0.2, 0.4, 0.6, 0.8]*, *scale='auto'*)

shows system in rasmol by drawing spheres draws 4 spheres in between any two points (5 * N spheres total)

2.2 polychrom.polymerutils module

2.2.1 Loading and saving individual conformations

The module *polychrom.polymerutils* provides tools for saving and loading individual conformations. Note that saving and loading trajectories should generally be done using *polychrom.hdf5_format* module. This module provides tools for loading/saving invividual conformations, or for working with projects that have both old-style and new-style trajectories.

For projects using both old-style and new-style trajectories(e.g. in a project that was switched to polychrom, and new files were added), a function *polychrom.polymerutils.fetch_block()* can be helpful as it provides the same interface for fetching a conformation from both old-style and new-style trajectory. Note however that it is not the fastest way to iterate over conformations in the new-style trajectory, and the *polychrom.hdf5_format.list_URIs()* is faster.

A typical workflow with the new-style trajectories should be:

```
URIs = polychrom.hdf5_format.list_URIs(folder)
for URI in URIs:
    data = polychrom.hdf5_format.load_URI(URI)
    xyz = data["pos"]
```

polychrom.polymerutils.**fetch_block**(*folder*, *ind*, *full_output=False*)

A more generic function to fetch block number "ind" from a trajectory in a folder

This function is useful both if you want to load both "old style" trajectories (block1.dat), and "new style" trajectories ("blocks_1-50.h5")

It will be used in files "show"

Parameters

- **folder** (*str, folder with a trajectory*) –
- **ind** (*str or int, number of a block to fetch*) –
- **full_output** (*bool (default=False)*) – If set to true, outputs a dict with positions, eP, eK, time etc. if False, outputs just the conformation (relevant only for new-style URIs, so default is False)

Returns

- *data, Nx3 numpy array*
- *if full_output==True, then dict with data and metadata; XYZ is under key "pos"*

polychrom.polymerutils.**load**(*filename*)

Universal load function for any type of data file It always returns just XYZ positions - use fetch_block or hdf5_format.load_URI for loading the whole metadata

2.2.2 Accepted file types

New-style URIs (HDF5 based storage)

Text files in openmm-polymer format joblib files in openmm-polymer format

param filename
filename to load or a URI

type filename
str

polychrom.polymerutils.**rotation_matrix**(*rotate*)

Calculates rotation matrix based on three rotation angles

polychrom.polymerutils.**save**(*data*, *filename*, *mode='txt'*, *pdbGroups=None*)

Basically unchanged polymerutils.save function from openmm-polymer

It can save into txt or joblib formats used by old openmm-polymer

It is also very useful for saving files to PDB format to make them compatible with nglview, pymol_show and others

2.3 polychrom.hdf5_format module

2.3.1 New-style HDF5 trajectories

The purpose of the HDF5 reporter

There are several reasons for migrating to the new HDF5 storage format:

- Saving each conformation as individual file is producing too many files
- Using pickle-based approaches (joblib) makes format python-specific and not backwards compatible; text is clumsy
- Would be nice to save metadata, such as starting conformation, forces, or initial parameters.
- Compression can be benefitial for rounded conformations: can reduce file size by up to 40%

one file vs many files vs several files

Saving each conformation as an individual file is undesirable because it will produce too many files: filesystem check or backup on 30,000,000 files takes hours/days.

Saving all trajectory as a single files is undesirable because 1. backup software will back up a new copy of the file every day as it grows; and 2. if the last write fails, the file will end up in the corrupted state and would need to be recovered.

Solution is: save groups of conformations as individual files. E.g. save conformations 1-50 as one file, conformations 51-100 as a second file etc.

This way, we are not risking to lose anything if the power goes out at the end. This way, we are not screwing with backup solutions. This way, we have partial trajectories that can be analyzed. Although partial trajectories are not realtime, @golobor was proposing a solution to it for debug/development.

Polychrom storage format

We chose the HDF5-based storage that roughly mimics the MDTraj HDF5 format. It does not have MDTraj topology because it seemed a little too complicated. However, full MDTraj compatibility may be added in the future

Separation of simulation and repoter

Polychrom separates two entities: a simulation object and a reporter. When a simulation object is initialized, a reporter (actually, a list of reporters in case you want to use several) is passed to the simulation object. Simulation object would attempt to save several things: __init__ arguments, starting conformation, energy minimization results, serialized forces, and blocks of conformations together with time, Ek, Ep.

Each time a simulation object wants to save something, it calls reporter.report(...) for each of the reporters. It passes a string indicating what is being reported, and a dictionary to save. Reporter will have to interpret this and save the data. Reporter is also keeping appropriate counts. Users can pass a dict with extra variables to *polychrom.simulation.Simulation.do_block()* as `save_extras` paramater. This dict will be saved by the reporter.

Note: Generic Python objects are not supported by HDF5 reporter. Data has to be HDF5-compatible, meaning an array of numbers/strings, or a number/string.

The HDF5 reporter used here saves everything into an HDF5 file. For anything except the conformations, it would immmediately save the data into a single HDF5 file: numpy array compatible structures would be saved as datasets, and regular types (strings, numbers) would be saved as attributes. For conformations, it would wait until a certain number of conformations is received. It will then save them all at once into an HDF5 file under groups /1, /2, /3... /50 for blocks 1,2,3...50 respectively, and save them to *blocks_1-50.h5* file

Multi-stage simulations or loop extrusion

We frequently have simulations in which a simulation object changes. One example would be changing forces or parameters throughout the simulation. Another example would be loop extrusion simulations.

In this design, a reporter object can be reused and passed to a new simulation. This would keep counter of conformations, and also save applied forces etc. again. The reporter would create a file "applied_forces_0.h5" the first time it receives forces, and "applied_forces_1.h5" the second time it receives forces from a simulation. Setting *reporter.blocks_only=True* would stop the reporter from saving anything but blocks, which may be helpful for making loop extrusion conformations. This is currently implemented in the examples

URIs to identify individual conformations

Because we're saving several conformations into one file, we designed an URI format to quickly fetch a conformation by a unique identifyer.

URIs are like that: */path/to/the/trajectory/blocks_1-50.h5::42*

This URI will fetch block #42 from a file blocks_1-50.h5, which contains blocks 1 through 50 including 1 and 50 *polychrom.polymerutils.load()* function is compatible with URIs Also, to make it easy to load both old-style filenames and new-style URIs, there is a function *polychrom.polymerutils.fetch_block()*. fetch_block

will autodetermine the type of a trajectory folder. So it will fetch both */path/to/the/trajectory/block42.dat* and */path/to/the/trajectory/blocks_x-y.h5::42* automatically

class polychrom.hdf5_format.**HDF5Reporter**(*folder*, *max_data_length=50*, *h5py_dset_opts=None*, *overwrite=False*, *blocks_only=False*, *check_exists=True*)

Bases: object

__init__(*folder*, *max_data_length=50*, *h5py_dset_opts=None*, *overwrite=False*, *blocks_only=False*, *check_exists=True*)

Creates a reporter object that saves a trajectory to a folder

Parameters

- **folder** (*str*) – Folder to save data to.
- **max_data_length** (*int, optional (default=50)*) – Will save data in groups of max_data_length blocks
- **overwrite** (*bool, optional (default=False)*) – Overwrite an existing trajectory in a folder.
- **check_exists** (*bool (optional, default=True)*) – Raise an error if previous trajectory exists in the folder
- **blocks_only** (*bool, optional (default=False)*) – Only save blocks, do not save any other information

continue_trajectory(*continue_from=None*, *continue_max_delete=5*)

Continues a simulation in a current folder (i.e. continues from the last block, or the block you specify). By default, takes the last block. Otherwise, takes the continue_from block

You should initialize the class with "check_exists=False" to continue a simulation

NOTE: This funciton does not continue the simulation itself (parameters, bonds, etc.) - it only manages counting the blocks and the saved files.

Returns (block_number, data_dict) - you should start a new simulation with data_dict["pos"]

Parameters

- **continue_from** (*int or None, optional (default=None)*) – Block number to continue a simulation from. Default: last block found
- **continue_max_delete** (*int (default = 5)*) – Maximum number of blocks to delete if continuing a simulation. It is here to avoid accidentally deleting a lot of blocks.

Returns

- *(block_number, data_dict)*
- *block_number is a number of a current block*
- *data_dict is what load_URI would return on the last block of a trajectory.*

dump_data()

report(*name*, *values*)

Semi-internal method to be called when you need to report something

Parameters

- **name** (*str*) – Name of what is being reported ("data", "init_args", anything else)

- **values** (*dict*) – Dict of what to report. Accepted types are np-array-compatible, numbers, strings. No dicts, objects, or what cannot be converted to a np-array of numbers or strings/bytes.

polychrom.hdf5_format.**list_URIs**(*folder*, *empty_error=True*, *read_error=True*, *return_dict=False*)

Makes a list of URIs (path-like records for each block). for a trajectory folder Now we store multiple blocks per file, and URI is a Universal Resource Identifier for a block.

It is be compatible with polymerutils.load, and with contactmap finders, and is generally treated like a filename.

This function checks that the HDF5 file is openable (if read_error==True), but does not check if individual datasets (blocks) exist in a file. If read_error==False, a non-openable file is fully ignored. NOTE: This covers the most typical case of corruption due to a terminated write, because an HDF5 file becomes invalid in that case.

It does not check continuity of blocks (blocks_1-10.h5; blocks_20-30.h5 is valid) But it does error if one block is listed twice (e.g. blocks_1-10.h5; blocks_5-15.h5 is invalid)

TODO: think about the above checks, and check for readable datasets as well

Parameters

- **folder** (*str*) – folder to find conformations in
- **empty_error** (*bool, optional*) – Raise error if the folder does not exist or has no files, default True
- **read_error** (*bool, optional*) – Raise error if one of the HDF5 files cannot be read, default True
- **return_dict** (*bool, optional*) – True: return a dict of {block_number, URI}. False: return a list of URIs. This is a default.

polychrom.hdf5_format.**load_URI**(*dset_path*)

Loads a single block of the simulation using address provided by list_filenames dset_path should be

/path/to/trajectory/folder/blocks_X-Y.h5::Z

where Z is the block number

polychrom.hdf5_format.**load_hdf5_file**(*fname*)

Loads a saved HDF5 files, reading all datasets and attributes. We save arrays as datasets, and regular types as attributes in HDF5

polychrom.hdf5_format.**save_hdf5_file**(*filename*, *data_dict*, *dset_opts=None*, *mode='w'*)

Saves data_dict to filename

2.4 polychrom.polymer_analyses module

2.4.1 Analyses of polymer conformations

This module presents a collection of utils to work with polymer conformations.

2.4.2 Tools for calculating contacts

The main function calculating contacts is: *polychrom.polymer_analyses.calculate_contacts()* Right now it is a simple wrapper around scipy.cKDTree.

Another function *polychrom.polymer_analyses.smart_contacts()* was added recently to help build contact maps with a large contact radius. It randomly sub-samples the monomers; by default selecting N/cutoff monomers. It then calculates contacts from sub-sampled monomers only. It is especially helpful when the same code needs to calculate contacts at large and small contact radii.Because of sub-sampling at large contact radius, it avoids the problem of having way-too-many-contacts at a large contact radius. For ordinary contacts, the number of contacts scales as contact_radius^3; however, with smart_contacts it would only scale linearly with contact radius, which leads to significant speedsups.

2.4.3 Tools to calculate P(s) and R(s)

We provide functions to calculate P(s), Rg^2(s) and R^2(s) for polymers. By default, they use log-spaced bins on the X axis, with about 10 bins per order of magnitude, but aligned such that the last bins ends exactly at (N-1). They output (bin, scaling) for Rg^2 and R^2, and (bin_mid, scaling) for contacts. In either case, the returned values are ready to plot. The difference is that Rg and R^2 are evaluated at a given value of s, while contacts are aggregated for (bins[0].. bins[1]), (bins[1]..bins[2]). Therefore, we have to return bin mids for contacts.

polychrom.polymer_analyses.**R2_scaling**(*data*, *bins=None*, *ring=False*)

Returns end-to-end distance scaling of a given polymer conformation. ..warning:: This method averages end-to-end scaling over all possible

subchains of given length

Parameters

- **data** (*Nx3 array*) –
- **bins** (*the same as in giveCpScaling*) –
- **ring** (*is the polymer a ring?*) –

polychrom.polymer_analyses.**Rg2**(*data*)

Simply calculates gyration radius of a polymer chain.

polychrom.polymer_analyses.**Rg2_matrix**(*data*)

Uses dynamic programming and vectorizing to calculate Rg for each subchain of the polymer. Returns a matrix for which an element [i,j] is Rg of a subchain from i to j including i and j

polychrom.polymer_analyses.**Rg2_scaling**(*data*, *bins=None*, *ring=False*)

Calculates average gyration radius of subchains a function of s

Parameters

- **data** (*Nx3 array*) –
- **bins** (*subchain lengths at which to calculate Rg*) –
- **ring** (*treat polymer as a ring (default: False)*) –

polychrom.polymer_analyses.**calculate_cistrans**(*data*, *chains*, *chain_id=0*, *cutoff=5*, *pbc_box=False*, *box_size=None*)

Analysis of the territoriality of polymer chains from simulations, using the cis/trans ratio. Cis signal is computed for the marked chain ('chain_id') as amount of contacts of the chain with itself Trans signal is the total amount of trans contacts for the marked chain with other chains from 'chains' (and with all the replicas for 'pbc_box'=True)

polychrom.polymer_analyses.**calculate_contacts**(*data*, *cutoff=1.7*)

Calculates contacts between points give the contact radius (cutoff)

Parameters

- **data** (*Nx3 array*) – Coordinates of points
- **cutoff** (*float , optional*) – Cutoff distance (contact radius)

Returns

k by 2 array of contacts. Each row corresponds to a contact.

polychrom.polymer_analyses.**contact_scaling**(*data*, *bins0=None*, *cutoff=1.1*, ***, *ring=False*)

Returns contact probability scaling for a given polymer conformation Contact between monomers X and X+1 is counted as s=1

Parameters

- **data** (*Nx3 array of ints/floats*) – Input polymer conformation
- **bins0** (*list or None*) – Bins to calculate scaling. Bins should probably be log-spaced; log-spaced bins can be quickly calculated using mirnylib.numtuis.logbinsnew. If None, bins will be calculated automatically
- **cutoff** (*float, optional*) – Cutoff to calculate scaling
- **ring** (*bool, optional*) – If True, will calculate contacts for the ring

Returns

- *(mids, contact probabilities) where "mids" contains*
- *geometric means of bin start/end*

polychrom.polymer_analyses.**generate_bins**(*N*, *start=4*, *bins_per_order_magn=10*)

polychrom.polymer_analyses.**getLinkingNumber**(*data1*, *data2*, *simplify=True*, *randomOffset=True*, *verbose=False*)

Ported here from openmmlib as well.

polychrom.polymer_analyses.**kabsch_msd**(*P*, *Q*)

Calculates MSD between two vectors using Kabash alcorithm Borrowed from https://github.com/charnley/rmsd with some changes

rmsd is licenced with a 2-clause BSD licence

ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

polychrom.polymer_analyses.**kabsch_rmsd**(*P*, *Q*)

Calculates MSD between two vectors using Kabash alcorithm Borrowed from https://github.com/charnley/rmsd with some changes

rmsd is licenced with a 2-clause BSD licence

Copyright (c) 2013, Jimmy Charnley Kromann <jimmy@charnley.dk> & Lars Bratholm All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

polychrom.polymer_analyses.**mutualSimplify**(*a*, *b*, *verbose=False*)

Ported here from openmmlib.

Given two polymer rings, it attempts to reduce the number of monomers in each of them while preserving the linking between them. It does so by trying to remove monomers one-by-one. If no other bonds pass through the triangle formed by the 2 old bonds and 1 new bond, it accepts removal of the monomer. It does so until no monomers in either of the rings can be removed.

polychrom.polymer_analyses.**ndarray_groupby_aggregate**(*df*, *ndarray_cols*, *aggregate_cols*, *value_cols=[]*, *sample_cols=[]*, *preset='sum'*, *ndarray_agg=<function <lambda>>*, *value_agg=<function <lambda>>*)

A version of pd.groupby that is aware of numpy arrays as values of columns

- aggregates columns ndarray_cols using ndarray_agg aggregator,
- aggregates value_cols using value_agg aggregator,
- takes the first element in sample_cols,
- aggregates over aggregate_cols

It has presets for sum, mean and nanmean.

polychrom.polymer_analyses.**slope_contact_scaling**(*mids*, *cp*, *sigma=2*)

polychrom.polymer_analyses.**smart_contacts**(*data*, *cutoff=1.7*, *min_cutoff=2.1*, *percent_func=<function <lambda>>*)

Calculates contacts for a polymer, give the contact radius (cutoff) This method takes a random fraction of the monomers that is equal to (1/cutoff).

This is done to make contact finding faster, and because if cutoff radius is R, and monomer (i,j) are in contact, then monomers (i+a), and (j+b) are likely in contact if **|a|** + **|b|** <~ R (the polymer could not run away by more than R in R steps)

This method will have # of contacts grow approximately linearly with contact radius, not cubically, which should drastically speed up computations of contacts for large (5+) contact radii. This should allow using the same code both for small and large contact radius, without the need to reduce the # of conformations, subsample the data, or both at very large contact radii.

Parameters

- **data** (*Nx3 array*) – Polymer coordinates
- **cutoff** (*float , optional*) – Cutoff distance that defines contact
- **min_cutoff** (*float, optional*) – Apply the "smart" reduction of contacts only when cutoff is less than this value
- **percent_func** (*callable, optional*) – Function that calculates fraction of monomers to use, as a function of cutoff Default is 1/cutoff

Returns

k by 2 array of contacts. Each row corresponds to a contact.

polychrom.polymer_analyses.**streaming_ndarray_agg**(*in_stream*, *ndarray_cols*, *aggregate_cols*, *value_cols=[]*, *sample_cols=[]*, *chunksize=30000*, *add_count_col=False*, *divide_by_count=False*)

Takes in_stream of dataframes

Applies ndarray-aware groupby-sum or groupby-mean: treats ndarray_cols as numpy arrays, value_cols as normal values, for sample_cols takes the first element.

Does groupby over aggregate_cols

if add_count_col is True, adds column "count", if it's a string - adds column with add_count_col name

if divide_by_counts is True, divides result by column "count". If it's a string, divides by divide_by_count column

This function can be used for automatically aggregating P(s), R(s) etc. for a set of conformations that is so large that all P(s) won't fit in RAM, and when averaging needs to be done over so many parameters that for-loops are not an issue. Examples may include simulations in which sweep over many parameters has been performed.

2.5 polychrom.contactmaps module

2.5.1 Building contact maps

This module is the main workhorse of tools to calculate contactmaps, both from polymer simulations and from other simulations (e.g. 1D simulations of loop extrusion). All of the functions here are designed to be parallelized, and lots of efforts were put into making this possible.

The reasons we need parallel contactmap code is the following:

- Calculating contact maps is slow, esp. at large contact radii, and benefits greatly from parallelizing
- Doing regular multiprocesing.map has limitations

- It can only handle heataps up to some size, and transferring gigabyte-sized heatmaps between processes takes minutes
- It can only do as many heatmaps as fits in RAM, which on 20-core 128GB machine is no more than 5GB/heatmap

The structure of this class is as follows.

On the outer level, it provides three methods to average contactmaps:

- *monomerResolutionContactMap()*
- *binnedContactMap()*,
- *monomerResolutionContactMapSubchains()*.

The first two create contact map from an entire file: either monomer-resolution or binned. The last one creates contact maps from sub-chains in a file, starting at a given set of starting points. It is useful when doing contact maps from several copies of a system in one simulation.

The first two methods have a legacy implementation from the old library that is still here to do the tests.

On the middle level, it provides a method "averageContacts". This method accepts a "contact iterator", and can be used to average contacts from both a set of filenames and from a simulation of some kind (e.g. averaging positions of loop extruding factors from a 1D loop extrusion simulation). All of the outer level functions (monomerResolutionContactMap for example) are implemented using this method.

On the lower level, there are internals of the "averageContacts" method and an associated "worker" function. There is generally no need to understand the code of those functions. There exists a reference implementation of both the worker and the *averageContacts()* function, `simpleWorker` and *averageContactsSimple()*. They do all the things that "averageContacts" do, but on only one core. In fact, "averageContacts" defaults to "averageContactsSimple" if requested to run on one core because it is a little bit faster.

`polychrom.contactmaps.`**`averageContacts`**(*contactIterator*, *inValues*, *N*, ***kwargs*)

A main workhorse for averaging contacts on multiple cores into one shared contact map. It mostly does managing the arguments, and initializing the variables. All of the logic of how contacts are actually put in shared memory buckets is in the worker defined above.

Parameters

- **contactIterator** (*iterator*) – an iterator. See descriptions of "filenameContactMap" class below for example and explanations
- **inValues** (*iterable*) – an array of values to pass to contactIterator. Would be an array of arrays of filenames or something like that.
- **N** (*int*) – Size of one side of the resulting contactmap
- **arrayDtype** (*ctypes dtype (default c_int32) for the contact map*) –
- **classInitArgs** (*args to pass to the constructor of contact iterator*) –
- **classInitKwargs** (*dict of keyword args to pass to the constructor*) –
- **contactProcessing** (*function f(contacts), should return processed contacts*) –
- **nproc** (*int, number of processors(default 4)*) –
- **bucketNum** (*int (default = nproc) Number of memory buckets to use*) –
- **contactBlock** (*int (default 500k) Number of contacts to aggregate before writing*) –
- **useFmap** (*True, False, or callable*) – If True, uses mirnylib.systemutils.fmap If False, uses multiprocessing.Pool.map Otherwise, uses provided function, assuming it of a

fork-map type (different initializations are needed for forkmap and multiprocessing-style map)

Sorry, no outside multiprocessing-style maps for now, it's easy to fix Let me know if it is needed.

Code that calcualtes a contactmap from a set of polymer conformation is in the methods below (averageMonomerResolutionContactMap, etc.)

An example code that would run a contactmap from a simulation is below:

..code-block:: python

class simContactMap(object):
"contactmap 'finder' for a simulation" def __init__(self, ind): # accept a parameter (e.g. random number generator seed)

self.model = initModel(ind) # pass parameter to the functon that returns me a model object self.count = 10000000 # how many times to run a step of the model self.model.steps(10000) # initial steps of the model to equilibrate it

def next(self): # actual realization of the self.next method

if self.count == 0: # terminate the simulation if we did self.count iterations
raise StopIteration

self.count -= 1 #decrement the counter self.model.steps(30) # advance model by 30 steps return np.array(self.model.getSMCs()).T # return current LEF positions

mymap = polychrom.contactmaps.averageContacts(simContactMap, range(20), 30000, nproc=20)

polychrom.contactmaps.**averageContactsSimple**(*contactIterator*, *inValues*, *N*, ***kwargs*)

This is a reference one-core implementation

Parameters

- **contactIterator** – an iterator. See descriptions of "filenameContactMap" class below for example and explanations
- **inValues** – an array of values to pass to contactIterator. Would be an array of arrays of filenames etc.
- **N** – Size of the resulting contactmap
- ****kwargs** – arrayDtype: ctypes dtype (default c_int32) for the contact map classInitArgs: args to pass to the constructor of contact iterator as second+ args (first is the file list) classInitKwargs: dict of keyword args to pass to the coonstructor uniqueContacts: whether contact iterator outputs unique contacts (true) or contacts can be duplicate (False) contactProcessing: function f(contacts), should return processed contacts

Returns
contactmap

polychrom.contactmaps.**binnedContactMap**(*filenames*, *chains=None*, *binSize=5*, *cutoff=5*, *n=8*, *contactFinder=<function calculate_contacts>*, *loadFunction=<function load>*, *exceptionsToIgnore=None*, *useFmap=False*)

polychrom.contactmaps.**chunk**(*mylist*, *chunksize*)

Parameters

- **mylist** – array

- **chunksize** – int

Returns
list of chunks of an array len chunksize (last chunk is less)

class polychrom.contactmaps.**filenameContactMap**(*filenames*, *cutoff=1.7*, *loadFunction=None*, *exceptionsToIgnore=[]*, *contactFunction=None*)

Bases: object

This is the iterator for the contact map finder

__init__(*filenames*, *cutoff=1.7*, *loadFunction=None*, *exceptionsToIgnore=[]*, *contactFunction=None*)

Init accepts arguments to initialize the iterator. filenames will be one of the items in the inValues list of the "averageContacts" function cutoff and loadFunction should be provided either in classInitArgs or classInitKwargs of averageContacts

When initialized, the iterator should store these args properly and create all necessary constructs

next()

This is the method which gets called by the worker asking for contacts. This method should return new set of contacts each time it is called When there are no more contacts to return (all filenames are gone, or simulation is over), then this method should raise StopIteration

class polychrom.contactmaps.**filenameContactMapRepeat**(*filenames*, *mapStarts*, *mapN*, *cutoff=1.7*, *loadFunction=None*, *exceptionsToIgnore=[]*, *contactFunction=None*)

Bases: object

This is a interator for the repeated contact map finder

__init__(*filenames*, *mapStarts*, *mapN*, *cutoff=1.7*, *loadFunction=None*, *exceptionsToIgnore=[]*, *contactFunction=None*)

Init accepts arguments to initialize the iterator. filenames will be one of the items in the inValues list of the "averageContacts" function cutoff and loadFunction should be provided either in classInitArgs or classInitKwargs of averageContacts

When initialized, the iterator should store these args properly and create all necessary constructs

next()

This is the method which gets called by the worker asking for contacts. This method should return new set of contacts each time it is called When there are no more contacts to return (all filenames are gone, or simulation is over), then this method should raise StopIteration

polychrom.contactmaps.**findN**(*filenames*, *loadFunction*, *exceptions*)

Finds length of data in filenames, handling the fact that files could be not loadable

polychrom.contactmaps.**indexing**(*smaller*, *larger*, *M*)

converts x-y indexes to index in the upper triangular matrix

polychrom.contactmaps.**init**(**args*)

Initializes global arguments for the worker

polychrom.contactmaps.**monomerResolutionContactMap**(*filenames*, *cutoff=5*, *n=8*, *contactFinder=<function calculate_contacts>*, *loadFunction=<function load>*, *exceptionsToIgnore=[]*, *useFmap=False*)

polychrom.contactmaps.**monomerResolutionContactMapSubchains**(*filenames*, *mapStarts*, *mapN*, *cutoff=5*, *n=8*, *method=<function calculate_contacts>*, *loadFunction=<function load>*, *exceptionsToIgnore=[]*, *useFmap=False*)

polychrom.contactmaps.**simple_worker**(*x*, *uniqueContacts*)

A "reference" version of "worker" function below that runs on only one core. Unlike the actual multicore worker, it can write contacts to the matrix directly without sorting. This is useful when your contact finding is faster than sorting a 1D array fo contacts

If uniqueContacts True, assume that contactFinder outputs only unique contacts (like pure contact map) if False, do not assume that (like in binned contact map). Using False is always safe, but True will add a minor speed up, especially for very large contact radius.

polychrom.contactmaps.**tonumpyarray**(*mp_arr*)

Converts mp.array to numpy array

polychrom.contactmaps.**triagToNormal**(*triag*, *M*)

Convert triangular matrix to a regular matrix

polychrom.contactmaps.**worker**(*x*)

This is a parallel implementation of the worker using shared memory buckets This worker is being called by the averageContact method It receives contacts from the contactIterator by calling .next() And puts contacts into the shared memory buckets

All the locks etc. for shared memory objects are handeled here as well

2.6 polychrom.forces module

2.6.1 Detailed description of forces in polychrom

This module defines forces commonly used in polychrom. Most forces are implemented using custom forces in openmm. The force equations were generally derived such that the force and the first derivative both go to zero at the cutoff radius.

Parametrization of bond forces

Most of the bond forces are parametrized using two parametrs: bondLength and bondWiggleDistance. The parameter *bondLength* is length of the bond at rest, while *bondWiggleDistance* is the estension of the bond at which energy reaches 1kT.

Note that the actual standard deviation of the bond length is bondWiggleDistance/sqrt(2) for a harmonic bond force, and is bondWiggleDistance*sqrt(2) for constant force bonds, so if you are switching from harmonic bonds to constant force, you may choose to decrease the wiggleDistance by a factor of 2.

Note on energy equations

Energy equations are passed as strings to one of the OpenMM customXXXForce class (e.g. customNonbondedForce). Note two things. First, sub-equations are separated by semicolon, and are evaluated "bottom up", last equation first. Second, equations seem much more scary than they actually are (see below).

All energy equations have to be continuous, and we strongly believe that the first derivative has to be continuous as well. As a result, all equations were carefully crafted to be smooth functions. This makes things more complicated. For example, a simple "k * abs(x-x0)" becomes "k * (sqrt((x-x0)^2 + a^2) - a)" where a is a small number (defined to be 0.01 for example).

All energy equations have to be calculatable in single precision. Any rounding error will throw you off. For example, you should never have sqrt(A - B) where A and B are expressions, and A >= B. Because by chance, due to rounding, you may and up with A slightly less than B, and you will receive NaN, and the whole simulation will blow up. Similarly, atan(very_large_number), while defined mathematically, could easily become NaN, because very_large_number may be larger than the largest allowable float.

Note that basically all nonbonded forces were written before OpenMM introduced a switching function http://docs.openmm.org/latest/api-python/generated/simtk.openmm.openmm.CustomNonbondedForce.html Therefore, we always manually sticth the value and the first derivative of the force to be 0 at the cutoff distance. For custom user-defined forces, it may be better to use switching function instead. This does not apply to custom external forces, there stitching is still necessary.

Force equations don't have "if" statements, but it is possible to avoid them where they would be normally used. For example, "if a: b= b0 + c" can be replaced with "b = b0 + c * delta(a)". Similarly "f(r) if r < r0; 0 otherwise" is just "f(r) * step(r0 - r)". These examples appear frequently in the forces that we have. One of the finest examples of crafting complex forces with on-the-fly generation of force equation is in *polychrom.forces.heteropolymer_SSW()*. One of the best examples of optimizing complex forces using polynomials is in *polychrom.forces.polynomial_repulsive()*.

polychrom.forces.**angle_force**(*sim_object*, *triplets*, *k=1.5*, *theta_0=<Mock name='mock.pi' id='139910982005648'>*, *name='angle'*, *override_checks=False*)

Adds harmonic angle bonds. k specifies energy in kT at one radian If k is an array, it has to be of the length N. Xth value then specifies stiffness of the angle centered at monomer number X. Values for ends of the chain will be simply ignored.

Parameters

- **k** (*float or list of length N*) – Stiffness of the bond. If list, then determines the stiffness of the i-th triplet Potential is k * alpha^2 * 0.5 * kT
- **theta_0** (*float or list of length N*) – Equilibrium angle of the bond. By default it is np.pi.
- **override_checks** (*bool*) – If True then do not check that no bonds are repeated. False by default.

polychrom.forces.**constant_force_bonds**(*sim_object*, *bonds*, *bondWiggleDistance=0.05*, *bondLength=1.0*, *quadraticPart=0.02*, *name='abs_bonds'*, *override_checks=False*)

Constant force bond force. Energy is roughly linear with estension after r=quadraticPart; before it is quadratic to make sure the force is differentiable.

Force is parametrized using the same approach as bond force: it reaches U=kT at extension = bondWiggleDistance

Note that, just as with bondForce, mean squared extension is actually larger than wiggleDistance by sqrt(2) factor.

Parameters

- **bonds** (*iterable of (int, int)*) – Pairs of particle indices to be connected with a bond.
- **bondWiggleDistance** (*float*) – Displacement at which bond energy equals 1 kT. Can be provided per-particle.
- **bondLength** (*float*) – The length of the bond. Can be provided per-particle.
- **override_checks** (*bool*) – If True then do not check that no bonds are repeated. False by default.

polychrom.forces.**cylindrical_confinement**(*sim_object*, *r*, *bottom=None*, *k=0.1*, *top=9999*, *name='cylindrical_confinement'*)

As it says.

polychrom.forces.**grosberg_angle**(*sim_object*, *triplets*, *k=1.5*, *name='grosberg_angle'*, *override_checks=False*)

Adds stiffness according to the Grosberg paper. (Halverson, Jonathan D., et al. "Molecular dynamics simulation study of nonconcatenated ring polymers in a melt. I. Statics." The Journal of chemical physics 134 (2011): 204904.)

Parameters are synchronized with normal stiffness

If k is an array, it has to be of the length N. Xth value then specifies stiffness of the angle centered at monomer number X. Values for ends of the chain will be simply ignored.

Parameters

- **k** (*float or N-long list of floats*) – Synchronized with regular stiffness. Default value is very flexible, as in Grosberg paper. Default value maximizes entanglement length.
- **override_checks** (*bool*) – If True then do not check that no bonds are repeated. False by default.

polychrom.forces.**grosberg_polymer_bonds**(*sim_object*, *bonds*, *k=30*, *name='grosberg_polymer'*, *override_checks=False*)

Adds FENE bonds according to Halverson-Grosberg paper. (Halverson, Jonathan D., et al. "Molecular dynamics simulation study of

nonconcatenated ring polymers in a melt. I. Statics." The Journal of chemical physics 134 (2011): 204904.)

This method has a repulsive potential build-in, so that Grosberg bonds could be used with truncated potentials. Is of no use unless you really need to simulate Grosberg-type system.

Parameters

- **k** (*float, optional*) – Arbitrary parameter; default value as in Grosberg paper.
- **override_checks** (*bool*) – If True then do not check that no bonds are repeated. False by default.

polychrom.forces.**grosberg_repulsive_force**(*sim_object*, *trunc=None*, *radiusMult=1.0*, *name='grosberg_repulsive'*, *trunc_function='min(trunc1, trunc2)'*)

This is the fastest non-transparent repulsive force. (that preserves topology, doesn't allow chain passing) Done according to the paper: (Halverson, Jonathan D., et al. "Molecular dynamics simulation study of

nonconcatenated ring polymers in a melt. I. Statics." The Journal of chemical physics 134 (2011): 204904.)

Parameters

- **trunc** (*None, float or N-array of floats*) – "transparency" values for each particular particle, which correspond to the truncation values in kT for the grosberg repulsion energy between a pair of such particles. Value of 1.5 yields frequent passing, 3 - average passing, 5 - rare passing.
- **radiusMult** (*float (optional)*) – Multiplier for the size of the force. To make scale the energy larger, set to be more than 1.
- **trunc_function** (*str (optional)*) – a formula to calculate the truncation between a pair of particles with transparencies trunc1 and trunc2 Default is min(trunc1, trunc2)

polychrom.forces.**harmonic_bonds**(*sim_object*, *bonds*, *bondWiggleDistance=0.05*, *bondLength=1.0*, *name='harmonic_bonds'*, *override_checks=False*)

Adds harmonic bonds.

Bonds are parametrized in the following way.

- A length of a bond at rest is *bondLength*
- Bond energy equal to 1kT at bondWiggleDistance

Note that bondWiggleDistance is not the standard deviation of the bond extension: that is actually smaller by a factor of sqrt(2).

Parameters

- **bonds** (*iterable of (int, int)*) – Pairs of particle indices to be connected with a bond.
- **bondWiggleDistance** (*float or iterable of float*) – Distance at which bond energy equals kT. Can be provided per-particle. If 0 then set k=0.
- **bondLength** (*float or iterable of float*) – The length of the bond. Can be provided per-particle.
- **override_checks** (*bool*) – If True then do not check that no bonds are repeated. False by default.

polychrom.forces.**heteropolymer_SSW**(*sim_object*, *interactionMatrix*, *monomerTypes*, *extraHardParticlesIdxs*, *repulsionEnergy=3.0*, *repulsionRadius=1.0*, *attractionEnergy=3.0*, *attractionRadius=1.5*, *selectiveRepulsionEnergy=20.0*, *selectiveAttractionEnergy=1.0*, *keepVanishingInteractions=False*, *name='heteropolymer_SSW'*)

A version of smooth square well potential that enables the simulation of heteropolymers. Every monomer is assigned a number determining its type, then one can specify additional attraction between the types with the interactionMatrix. Repulsion between all monomers is the same, except for extraHardParticles, which, if specified, have higher repulsion energy.

The overall potential is the same as in *polychrom.forces.smooth_square_well()*

Treatment of extraHard particles is the same as in *polychrom.forces.selective_SSW()*

This is an extension of SSW (smooth square well) force in which:

a) You can give monomerTypes (e.g. 0, 1, 2 for A, B, C) and interaction strengths between these types. The corresponding entry in interactionMatrix is multiplied by selectiveAttractionEnergy to give the actual **additional** depth of the potential well.

b) You can select a subset of particles and make them "extra hard". See selective_SSW force for descrition.

2.6.2 Force summary

Potential is the same as smooth square well, with the following parameters for particles i and j:

- Attraction energy (i,j) = attractionEnergy + selectiveAttractionEnergy * interactionMatrix[i,j]
- Repulsion Energy (i,j) = repulsionEnergy + selectiveRepulsionEnergy; if (i) or (j) are extraHard
- Repulsion Energy (i,j) = repulsionEnergy; otherwise

param interactionMatrix
: the **EXTRA** interaction strenghts between the different types. Only upper triangular values are used. See "Force summary" abovec

type interactionMatrix
: np.array

param monomerTypes
: the type of each monomer, starting at 0

type monomerTypes
: list of int or np.array

param extraHardParticlesIdxs
: the list of indices of the "extra hard" particles. The extra hard particles repel all other particles with extra *selectiveRepulsionEnergy*

type extraHardParticlesIdxs
: list of int

param repulsionEnergy
: the heigth of the repulsive part of the potential. E(0) = *repulsionEnergy*

type repulsionEnergy
: float

param repulsionRadius
: the radius of the repulsive part of the potential. E(*repulsionRadius*) = 0, E'(*repulsionRadius*) = 0

type repulsionRadius
: float

param attractionEnergy
: the depth of the attractive part of the potential. E(*repulsionRadius*/2 + *attractionRadius*/2) = *attractionEnergy*

type attractionEnergy
: float

param attractionRadius
: the maximal range of the attractive part of the potential.

type attractionRadius
: float

param selectiveRepulsionEnergy
: the **EXTRA** repulsion energy applied to the "extra hard" particles

type selectiveRepulsionEnergy
: float

param selectiveAttractionEnergy
the **EXTRA** attraction energy (prefactor for the interactionMatrix interactions)

type selectiveAttractionEnergy
float

param keepVanishingInteractions
a flag that determines whether the terms that have zero interaction are still added to the force. This can be useful when changing the force dynamically (i.e. switching interactions on at some point)

type keepVanishingInteractions
bool

polychrom.forces.**polynomial_repulsive**(*sim_object*, *trunc=3.0*, *radiusMult=1.0*, *name='polynomial_repulsive'*)

This is a simple polynomial repulsive potential. It has the value of *trunc* at zero, stays flat until 0.6-0.7 and then drops to zero together with its first derivative at r=1.0.

See the gist below with an example of the potential. https://gist.github.com/mimakaev/0327bf6ffe7057ee0e0625092ec8e318

Parameters
trunc (*float*) – the energy value around r=0

polychrom.forces.**pull_force**(*sim_object*, *particles*, *force_vecs*, *name='Pull'*)

adds force pulling on each particle particles: list of particle indices force_vecs: list of forces [[f0x,f0y,f0z],[f1x,f1y,f1z], ...] if there are fewer forces than particles forces are padded with forces[-1]

polychrom.forces.**selective_SSW**(*sim_object*, *stickyParticlesIdxs*, *extraHardParticlesIdxs*, *repulsionEnergy=3.0*, *repulsionRadius=1.0*, *attractionEnergy=3.0*, *attractionRadius=1.5*, *selectiveRepulsionEnergy=20.0*, *selectiveAttractionEnergy=1.0*, *name='selective_SSW'*)

This is a simple and fast polynomial force that looks like a smoothed version of the square-well potential. The energy equals *repulsionEnergy* around r=0, stays flat until 0.6-0.7, then drops to zero together with its first derivative at r=1.0. After that it drop down to *attractionEnergy* and gets back to zero at r=`attractionRadius`.

The energy function is based on polynomials of 12th power. Both the function and its first derivative is continuous everywhere within its domain and they both get to zero at the boundary.

This is a tunable version of SSW: a) You can specify the set of "sticky" particles. The sticky particles are attracted only to other sticky particles. b) You can **smultaneously** select a subset of particles and make them "extra hard".

This force was used two-ways. First was to make a small subset of particles very sticky. In that case, it is advantageous to make the sticky particles and their neighbours "extra hard" and thus prevent the system from collapsing.

Another useage is to induce phase separation by making all B monomers sticky. In that case, extraHard particles may not be needed at all, because the system would not collapse on iteslf.

Parameters

- **stickyParticlesIdxs** (*list of int*) – the list of indices of the "sticky" particles. The sticky particles are attracted to each other with extra *selectiveAttractionEnergy*
- **extraHardParticlesIdxs** (*list of int*) – the list of indices of the "extra hard" particles. The extra hard particles repel all other particles with extra *selectiveRepulsionEnergy*
- **repulsionEnergy** (*float*) – the heigth of the repulsive part of the potential. E(0) = *repulsionEnergy*

- **repulsionRadius** (*float*) – the radius of the repulsive part of the potential. E(*repulsionRadius*) = 0, E'(*repulsionRadius*) = 0
- **attractionEnergy** (*float*) – the depth of the attractive part of the potential. E(*repulsionRadius*/2 + *attractionRadius*/2) = *attractionEnergy*
- **attractionRadius** (*float*) – the maximal range of the attractive part of the potential.
- **selectiveRepulsionEnergy** (*float*) – the **EXTRA** repulsion energy applied to the **extra hard** particles
- **selectiveAttractionEnergy** (*float*) – the **EXTRA** attraction energy applied to the **sticky** particles

polychrom.forces.**smooth_square_well**(*sim_object*, *repulsionEnergy=3.0*, *repulsionRadius=1.0*, *attractionEnergy=0.5*, *attractionRadius=2.0*, *name='smooth_square_well'*)

This is a simple and fast polynomial force that looks like a smoothed version of the square-well potential. The energy equals *repulsionEnergy* around r=0, stays flat until 0.6-0.7, then drops to zero together with its first derivative at r=1.0. After that it drop down to *attractionEnergy* and gets back to zero at r=`attractionRadius`.

The energy function is based on polynomials of 12th power. Both the function and its first derivative is continuous everywhere within its domain and they both get to zero at the boundary.

Parameters

- **repulsionEnergy** (*float*) – the heigth of the repulsive part of the potential. E(0) = *repulsionEnergy*
- **repulsionRadius** (*float*) – the radius of the repulsive part of the potential. E(*repulsionRadius*) = 0, E'(*repulsionRadius*) = 0
- **attractionEnergy** (*float*) – the depth of the attractive part of the potential. E(*repulsionRadius*/2 + *attractionRadius*/2) = *attractionEnergy*
- **attractionRadius** (*float*) – the radius of the attractive part of the potential. E(*attractionRadius*) = 0, E'(*attractionRadius*) = 0

polychrom.forces.**spherical_confinement**(*sim_object*, *r='density'*, *k=5.0*, *density=0.3*, *center=[0, 0, 0]*, *invert=False*, *particles=None*, *name='spherical_confinement'*)

Constrain particles to be within a sphere. With no parameters creates sphere with density .3

Parameters

- **r** (*float or "density", optional*) – Radius of confining sphere. If "density" requires density, or assumes density = .3
- **k** (*float, optional*) – Steepness of the confining potential, in kT/nm
- **density** (*float, optional, <1*) – Density for autodetection of confining radius. Density is calculated in particles per nm^3, i.e. at density 1 each sphere has a 1x1x1 cube.
- **center** (*[float, float, float]*) – The coordinates of the center of the sphere.
- **invert** (*bool*) – If True, particles are not confinded, but *excluded* from the sphere.
- **particles** (*list of int*) – The list of particles affected by the force. If None, apply the force to all particles.

polychrom.forces.**spherical_well**(*sim_object*, *particles*, *r*, *center=[0, 0, 0]*, *width=1*, *depth=1*, *name='spherical_well'*)

A spherical potential well, suited for example to simulate attraction to a lamina.

Parameters

- **particles** (*list of int or np.array*) – indices of particles that are attracted
- **r** (*float*) – Radius of the nucleus
- **center** (*vector, optional*) – center position of the sphere. This parameter is useful when confining chromosomes to their territory.
- **width** (*float, optional*) – Width of attractive well, nm.
- **depth** (*float, optional*) – Depth of attractive potential in kT NOTE: switched sign from openmm-polymer, because it was confusing. Now this parameter is really the depth of the well, i.e. positive = attractive, negative = repulsive

polychrom.forces.**tether_particles**(*sim_object*, *particles*, ***, *pbc=False*, *k=30*, *positions='current'*, *name='Tethers'*)

tethers particles in the 'particles' array. Increase k to tether them stronger, but watch the system!

Parameters

- **particles** (*list of ints*) – List of particles to be tethered (fixed in space). Negative values are allowed.
- **pbc** (*Bool, optional*) – If True, periodicdistance function is applied
- **k** (*int, optional*) – The steepness of the tethering potential. Values >30 will require decreasing potential, but will make tethering rock solid. Can be provided as a vector [kx, ky, kz].

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

P

R

S

T

W