Complexes Code Architecture

Abstract

This document is intented for people who want to extend complexes or understand the code in more detail. The code is described at a high level outlining the general ideas and design.

Contents

Chapter 1

Complexes++

Complexes++ is a Monte Carlo simulation engine implementing the complexes protein model. The program is written in modern C++14 with a focus on being extensible. The program contains an extensible Monte-Carlo engine, the modified Lennard-Jones potential, **??** and other pair potentials, the Monte-Carlo movements for different domain types, and a cell-list algorithm [1] to speed up the energy evaluations.

1.1 Flexible and Extensible Core Algorithms

The core algorithms of the Monte-Carlo engine have the ability to add new trial moves for domains and to add new Monte-Carlo algorithms for different statistical ensembles or enhanced sampling techniques. This extensibility is achieved thought a combination of run-time polymorphism and use of modern C++14. In Complexes++ , this technique is used to implement domain trial moves, Monte-Carlo algorithms, evaluation of pair potentials, and a cell-list algorithm. In C++ run-time polymorphism is implemented through abstract classes and inheritance. In Complexes++ abstract classes are marked with the preposition "Abstract". The most interesting abstract classes to add new features to Complexes++ are `AbstractDomain`, `AbstractPairKernel`, `AbstractConnection`, `AbstractInteractionGrid` and `AbstractMcAlgo`. See Figure 1.1 for a schematic how these classes interact with each other during a simulation.

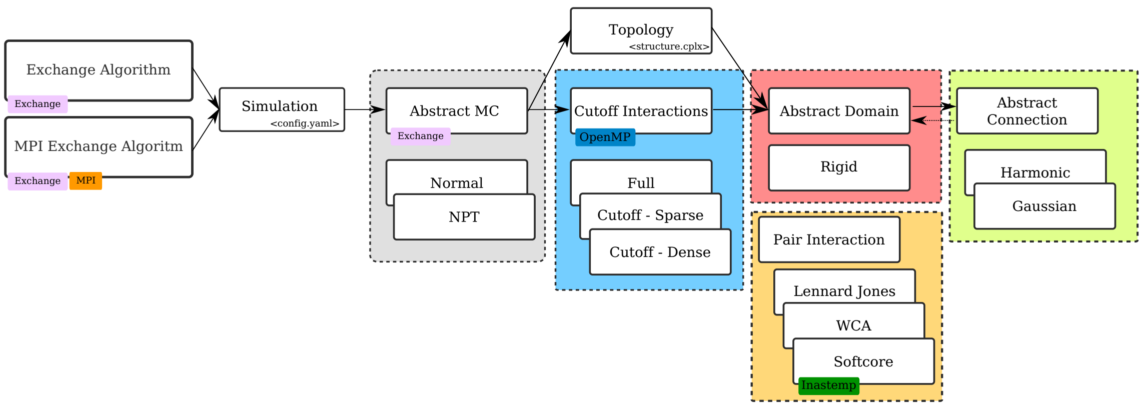

Figure 1.1: Schematic of the classes used in the Complexes++ program and how they interact with each other (arrows). Abstract classes and their explicit implementations are shown as colored boxes. Nested boxes show different implementations of an abstract class. Names are as in the code.

1.2 Monte-Carlo Algorithm

The main logic to run a Monte-Carlo simulation is implement in `AbstractMcAlgo` with a virtual method called `sweep` to implement a sweep. Currently Complexes++ implements sweep functions for the NVT and NΠT ensemble [1]. In addition two different acceptance functions, Metropolis [2] and Glauber [3] have been implemented as well. All Monte-Carlo sweep algorithms work with all acceptance functions.

For a sweep in the NVT ensemble domains are chosen randomly from a uniform distribution. If N is the number of domains that than a sweep will make N trial moves and select a random domain with probability $1/N$ for each trial. In the NΠT ensemble trial moves are generated for the domains and the volume. To account for the additional volume move a sweep consists of $N+1$ trial moves. At each trial, the probability to make a volume move is $1/(N+1)$. If a domain move was chosen the domain to be moved is chosen randomly with a $1/N$ probability. For the volume moves the domains are re-scaled so that the centroid of the domain is scaled by $\sqrt[3]{(V+dV)/V}$. The NΠT algorithm is called NPT in the code and configuration files.

1.3 Boundary Conditions

Complexes++ treats boundary effects using periodic boundary conditions [1]. Under these conditions, the centroid of all domains is placed inside of the unit-cell. Complexes++ only implements rectangular unit-cells. Because Complexes++ keeps the centroid of a domain within the box it can happen that some beads are outside of the box.

To determine the distance between two beads Complexes++ uses the minimum image convention (MIC) [1]. Complexes++ uses an efficient implementation to calculate the minimum image distance, Algorithm 1. This algorithm is an extension of a fast algorithm [4], where the while-loop ensures it works for any distance to ensure the algorithm will also work if trial moves displace a domain by more than one periodic image. To ensure that the number of iterations of the while-loop are as small as possible Complexes++ ensures that the centroid of a domain are inside the unit-cell.

ALGORITHM 1: Algorithm to efficiently convert the distance between two beads to the minimum image distance it domain centers are inside of the simulation box.

```
function mic(distance[3], box[3])
    for i=0; i<3; ++i do
        while distance[i] > .5 * box[i] do
            distance[i] -= box[i]
        end
        while distance[i] < .5 * box[i] do
            distance[i] += box[i]
        end
    end
    return distance;
```

1.4 Replica Exchange Algorithms

The theoretical descriptions of the replica exchange algorithms do not specify which replicas are exchanged in an attempt. To increase the probability to accepted an exchange attempt in complexes only neighboring replicas are exchanged [5]. During an exchange, not all neighboring pairs are attempted to exchange, this is because the exchange between replica i and $i+1$ also depends on replica $i-1$. Rather in Complexes++ attempts are only done between even pairs when the attempt number is even and odd pairs otherwise. In an even pair, of replicas i and $i+1$, then i is even and odd for odd pairs. For consistency 0 is counted as an even number and 1 as an odd number.

Replica simulations require that multiple simulations are started. Complexes++ requires that every replica is started in its own folder. Multiple simulations can be started with the `-multidir` flag and a list of the folders containing individual replicas. The `-multidir` option alone only tells complexes to simulate multiple replicas to activate the exchange two flags `-replex` and `-replex-accept` have to be used as well. `-replex` states after how many sweeps an exchange should be attempted. `-replex-accept` states the exchange function to be used.

The replica exchange simulations are also implemented in an extensible manner. Because the different replicas exchange algorithms implemented require different variables to be changed between the replicas, Complexes++ only exchanges the coordinates of the beads and box dimensions during an exchange.

1.5 Random Numbers

Complexes++ uses pseudo random numbers to generate trial moves and accept moves. As a pseudo random number generator (RNG) it is using the mersenne twister [6] implementation in the C++ standard library. Because the RNG cannot be safely used in a multi-threaded program like Complexes++ it uses a different instance of the RNG for each thread. The individual RNG instances are seeded from random numbers from an initial RNG instance during setup. To achieve reproducability Complexes++ requires the user to provide an explicit seed for the initial RNG. This requirement ensures that two runs with the same input are identical on the same computation node.

1.6 Pair Interactions Potentials

The different pair kernels are implemented with a common abstract base class `AbstractPairKernel`. The Lennard-Jones like and Weeks-Chandler-Anderson (WCA) potential have been implemented using Inastemp, a portable single instruction multiple data (SIMD) library, [7].

1.7 Bead and Domain Implementation

No specific bead class exists in Complexes++ , instead, domains contain all information specific to the beads as several lists. This design scheme follows the “struct of arrays” pattern [8] used to optimize memory access. For the beads a domain stores the coordinates in a `m_xyz` field, the charges in `m_charges` and the bead types in `m_beads`. These three fields are the only information needed to evaluate the potential energy with the pair potential functions. Domains are also implemented with run-time polymorphism in an AbstractDomain class that contains information about the beads and coordinates. Derived classes only need to implement trial moves.

In addition to bead information, a domain also contains a unique id and a type id that are defined at runtime. The type id is used to find the corresponding pair kernel when evaluating the energy with a different domain. Choosing a pair-kernel for domain type pairs and using a struct of arrays pattern allows evaluating the pair-kernel for groups of beads. This pattern can be efficiently implemented using inastemp [7]. Because the evaluation of the pair-kernel is the most expensive calculation of Complexes++ this pattern ensures that Complexes++ has good performance while using run-time polymorphism. The type ids are also used to create domains with the same type of move but different parametrizations for it. For example, the rigid domains have two parameters for translation and rotation. Having the final parametrization set at runtime allows creating two distinct rigid domain types for different proteins. This flexibility is useful for simulations with a mixture of small and large domains. The translation and rotation of larger domains can be chosen smaller to increase the acceptance rate for their moves and small domains can be set to large translation and rotation. Allowing to achieve optimal phase space exploration rate for a simulation.

In Complexes++ only the rigid domain types is implemented. For the Monte-Carlo moves the rigid domains can be translated by an arbitrary vector, each component is chosen randomly from the range $[-a, +a]$, with a the maximal displacement. The rotations are generated by choosing a random rotation axes in the unit cube and a random rotation angle [1]. This method generates a known artifact that rotation axes along the edges are over sampled. Because the all edges are equally sampled it does not affect generated ensembles. In each trial move the probability to make a translation or a rotation move is one half.

1.8 Flexible Linkers and Connection Potentials

As described in the theory flexible linker domains are replaced with effective potentials from a Gaussian chain polymer model. For this Complexes++ implements connection potentials in a `Connection` class. A connection contains the ids of the two domains that are connected and the corresponding bead ids in the domains. For each domain, a list of connections is stored in the class. So if domains i and j are connected both contain the same connection. This duplication of data makes it easier finding the corresponding connections for a domain during the energy calculation. This compromise was chosen as it is expected that the number of connections is significantly smaller the number of beads. Currently implemented connection potentials are a flat potential (zero everywhere), a harmonic potential, and a Gaussian chain potential of mean force (PMF) potential, **??**.

1.9 Cell List Algorithm

The other important part of the performance of complexes is the cell list algorithm [1] used to reduce the number of interactions needed to evaluate. Cell-list algorithms reduce the computation time to calculate the full energy from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$ with N being the number of beads in a simulation. The algorithm stores for every cell continues intervals of beads that are in the corresponding cell. Every bead in a domain is given a unique id defined by the order in which they appear in the input files. The interval will therefore only store the id of the first bead entering the cell and the length of beads in the cell. For domains that are larger than a single cell it can happen that two different intervals are in a cell, see the red domain in Figure 1.2 that has two intervals in the cell (1, 4). In that case, the cell will store two intervals for the same domain. An intervals is stored in the `CoInterval` class, Listing 1. Each cell stores a list of CoInterval instances. As an example of the data stored for a list take the cell (1, 4) containing the red domain in Figure 1.2.

To calculate the completely potential energy between all domains we iterate over all cells in the cell-list, see Algorithm 2. For each cell we then iterate over the list of CoIntervals. During a Monte Carlo move only a single domain will be moved. We therefore do not

```
class CoCell {
  // The list of intervals inside the current
  cell std::vector<CoInterval> m_intervals;
  // ...
};

class CoInterval {
  // The domain related to the current interval
  int m_domainId;
  // The position of the first element of the element-list
  int m_beginingOfInterval;
  // The number of elements in the current interval
  int m_nbElementsInInterval;
  // ...
};
```

Listing 1: Definition of the CoCell and CoInterval class used in the cell list algorithm implemented in Complexes++ . The comment "..." indicates other implementation details.

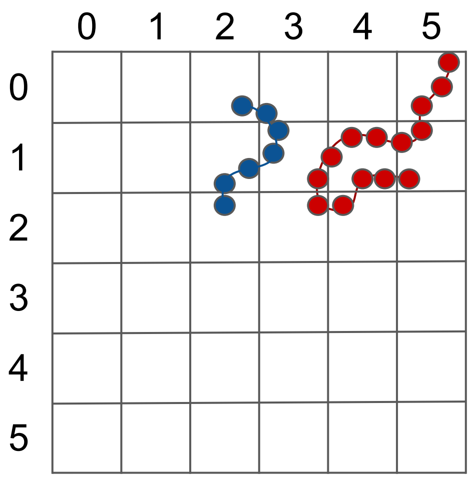

Figure 1.2: Example configuration for two domains (blue and red) in a 2D grid. Numbers on the top and left are used to index cells.

need to recalculate the complete potential energy. Instead we need iterate over the cells occupied by the moved domain and update the energy difference appropriately, replacing the outer most loop in Algorithm 2 with a loop over only the cell currently occupied by the domain. The last step further reduces the computational effort to calculate energies during a simulation.

The interface to the cell list algorithm to calculate pair interactions has also been implemented without references to the underlying algorithm and can be swapped out with different algorithms. In Complexes++ , there are two data-structures available for the cutoff grid. The *dense* data structure is allocated cell objects for all cells in a simulation box and offers efficient computation of neighboring cells. The memory consumption of this data structure scales with $(L_{\text{Box}}/L_{\text{Cell}})^3$, where L_{Box} is the edge-length of the simulation box and L_{Cell} is the edge-length of a cell, see Figure 1.3. This data structure is optimal for dense simulations. The *sparse* data structure uses a hash-map to only allocate cells that

ALGORITHM 2: Cell list algorithm to calculate all pair interactions.

```
function computeEnergy(Cells[K], Domains[M])
    energy = 0
    // Compute energy (particle to particle interactions)
    for cell in Cells do
        for target in cell do
            for source in cell do
                if source.domainId ≠ target.domainId then
                    target_dom = Domains[target.domainId]
                    source_dom = Domains[source.domainId]
                    energy += kernel(target_dom, source_dom, target, source)
                end
            end
            for nb_cell in cell.neighbors() do
                for source in nb_cell do
                    if source.domainId ≠ target.domainId then
                        target_dom = Domains[target.domainId]
                        source_dom = Domains[source.domainId]
                        energy += kernel(target_dom, source_dom, target,
                          source)
                    end
                end
            end
        end
    end
    return energy
```

are occupied by beads. The memory consumption of this data structure scales with the number of beads in a simulation and is independent of the box size. It is suited for sparse simulations. The high-level interface is also agnostic to the underlying cell-list algorithm allowing to chose a completely different algorithm if desired.

1.10 Task based parallelism

Monte Carlo algorithms have two points which can be parallelized, the energy calculation and the trial move generation. The trial move generation in complexes is not computationally expensive and a minuscule amount of time is spend on it. Leaving the energy calculation for the single replica simulations. In replica exchange Monte-Carlo all replicas can be executed in parallel with few synchronization points to exchange coordinates. Ideally, the single replica and replica exchange simulation can use the same parallelisms scheme. It has to be taken into account that replicas can be unbalanced or change during a simulation if a phase transition occurs (from liquid to solid). Another requirement was that it should be possible to always use the maximal number of available central processing unit (CPU) cores independent of the number of replicas. Meaning a single replica can use multiple threads and in the case that more replicas than available threads are simulated the different replicas have a scheduler queue.

In Complexes++ this problem is solved using the task application programming interface (API) in OpenMP [9]. Tasks are generated based on either domains or cell lists (if activate). Meaning tasks are small and plentiful. At the beginning of the energy cal-

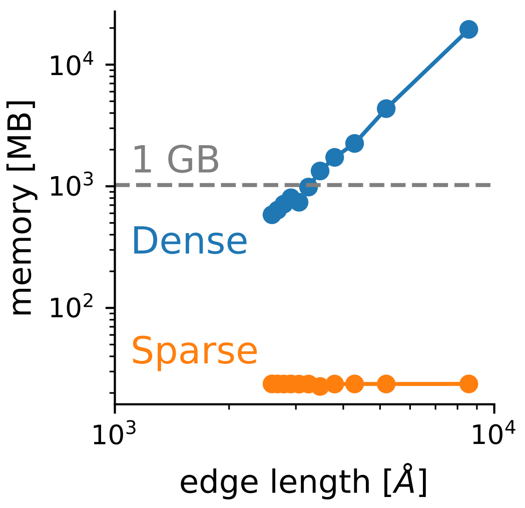

Figure 1.3: Memory consumption of the dense (blue) and sparse (orange) cell-list data-structure with a constant number of beads. The cutoff is set to 12 Å. The gray line marks one gigabyte.

culation, the number of tasks is distributed equally to the available threads. If a thread finishes early it can then steal tasks from other threads. Allowing threads to be always busy. It balances when different replicas have different work loads. Nothing special is done to minimize the numa effect. Therefore, it is maybe better to one process per memory node (explicit sync) and threads inside.

The replica simulations can run in parallel. To make better use of available resources that allow multi-node jobs also a message passing interface (MPI) implementation has been developed. There is no work stealing between replicas in this configuration.

1.11 CPLX File Format

The input for Complexes++ simulation is stored in a CPLX file similar to the tpr files in GROMACS. The CPLX file contains all information about the domains and pair kernels to simulate, with the exception of the parameters for the Monte-Carlo algorithm and the used forcefield parameters. The CPLX file format is based on YAML [10] and divided into four sections *box, definitions*, *topologies*, and *forcefield*.

The *box* section contains the dimensions of the rectangular simulation box as a YAML list. The length in each dimension is given in Å.

The *definitions* section defines the type of domains that are used in the simulation and the pair kernels used to calculate the energy between domains. The *definitions* section is separated into two sub sections to define the final domain-types and their interactions between them, called *domains* and *pair-interactions* respectively. The *domains* section contains a dictionary with the entry names being the domain names and the definition for the domain. A definition contains the type of move the domain can make, currently only rigid is implemented, and the parameters for the move. Allowing to define several rigid bodies in a simulation that have different rotation and translation parameters. The definitions can be used for systems with a mixture of large and small rigid bodies to model the corresponding mobility. The *pair-interaction* section defines which pair interaction potential to choose for each combination of domain types defined in *domains*. In the definition, one can define more domain types than what will be used in a simulation.

Allowing to create standard definitions (like the Kim-Hummer (KH) forcefield) for different applications. See Listing 2 for an example definition using two domain types **A** and **EM**. Here the **EM** type is set to not move at all and interacts with **A** domains using the WCA potential. This definition could be used to fit any domain of type **A** into a domain defined by **EM**.

```
definitions:
  domains:
    A:
      defaults: {rotation: 2, translation: 1}
      move: rigid
   EM:
      defaults: {rotation: 0, translation: 0}
   move: rigid
pair-interaction:
  - domain-type-pair: [A, EM] function: WCA
  - domain-type-pair: [A, A] function: LJH
  - domain-type-pair: [EM, EM] function: None
```

Listing 2: Complexes++ domain definitions for a simulation with two domain types **A** and **EM**. Both domain types move as rigid bodies but they have different pair interaction potentials with each other.

The *topologies* section defines the actual domains that are used in the simulation. It consists of a YAML list of topology entries. Each topology entry contains domain definitions and connections between domains of the same topology. A domain contains the following field: beads, chain-ids, charges, coordinates, nbeads, type, mc-moves, meta-data, name. Here the type field specifies the domain type, which has to be defined in the definitions section. Domains have to be numbered consecutively and uniquely across all topologies.

The *forcefield* section contains definitions for the available bead types as well as the energy and diameter pair parameters.

Chapter 2

pycomplexes

```
box: [100, 100, 100]
topology:
  protein-name:
    coordinate-file: structure.pdb
    move: true
    domains:
      A:
        type: rigid
        selection: 'namd CA and segid A'
      link:
        type: gaussian
        selection: 'name CA and segid L'
        start_connection: [A, 'segid A and resid 10']
        end_connection: [B, 'segid B and resid 10']
      B:
        type: rigid
        selection: 'name CA and segid B'
```

Listing 3: TOP file for a simulation for two rigid domains connected by a Gaussian domain. All selections are written in the atom selection language used by MDAnalysis.

pycomplexes is a Python library and command line interface (CLI) program that includes several tools to help setup simulations for Complexes++ and analyze them later. The CPLX format accepted by Complexes++ is very complex and allows the user a lot of freedom in setting up the simulation. A lot of simulation setups do not need to leverage the full flexibility of complexes and therefore pycomplexes includes a tool called `convert` that takes as input a simplified format, called a TOP file, for describing simulations and generating CPLX files from it. As part of the conversion, the script will automatically choose charges and interaction energies based on amino acid type. As interaction energies, the user can choose either the KH model or the unaltered Miyazawa and Jernigan (MJ) model. The charges are set according to the KH model in both cases. The interaction potential is chosen based on domain type, so far allowed are rigid and gaussian. The rigid domain type uses the modified lennard jones potential, **??**. For the gaussian domains positions of the C_α beads will be stored in the CPLX as rigid domains that do not move and a connection potential, **??**, will be used between the two rigid domains connected by the gaussian domain. To define a domain in the top file the type and a selection of beads have to be specified. For the gaussian domain, in addition, the beads of the rigid domains connected by the gaussian domain have to be specified as well. The convert script could ignore known beads for a gaussian domain in the structure but keeping the information in the CPLX file allows to generate explicit linker positions in the post processing, i.e. with the `addlinker` tool.

The structures and selections are read and parsed using MDAnalysis[11, 12]. In the molecular dynamics community there exists a large variety of structure file format and they are often not well defined or popular programs write and accept ill formatted files, MDAnalysis helps to read a large variety of different formats with an easy to understand atom selection language.

In addition to the `convert` command pycomplexes also include a variety of other commands to help the user setup and visualize simulations. As of the time of writing the other implemented commands are

- `equilibration` Update coordinates stored in a CPLX from a trajectory.
- `forcefield` Convert a forcefield using scaling parameters λ and e_0, see **??**.
- `visualize` Create VMD scripts from simulations.
- `demux` generate input files for the GROMACS tool `trjcat` to generate time-continuous trajectories from replica exchange simulations.
- `addlinker` generate explicit linker conformations for a simulation with Gaussian linkers.

Bibliography

[1] Daan Frenkel and Berend Smit. *Understanding molecular simulation: from algorithms to applications*, volume 1. Academic press, 2001.

[2] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of State Calculations by Fast Computing Machines. *J. Chem. Phys.*, 21(6):1087–1092, 1953. ISSN 00219606. doi: doi: 10.1063/1.1699114.

[3] Roy J. Glauber. Time-Dependent Statistics of the Ising Model. *J. Math. Phys.*, 4(2): 294, 1963. ISSN 00222488. doi: 10.1063/1.1703954.

[4] Ulrich K. Deiters. Efficient coding of the minimum image convention. *Zeitschrift fur Phys. Chemie*, 227(2-3):345–352, 2013. ISSN 09429352. doi: 10.1524/zpch.2013.0311.

[5] Giovanni Bussi. Hamiltonian replica exchange in GROMACS: a flexible implementation. *Mol. Phys.*, 112(3-4):379–384, 2014. ISSN 0026-8976. doi: 10.1080/00268976.2013.824126.

[6] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8:3–30, 1998.

[7] Berenger Bramas. Inastemp : A Novel Intrinsics-as-Template Library for Portable SIMD-Vectorization. 2017:1–22, 2017. ISSN 10589244. doi: 10.1155/2017/5482468.

[8] https://en.wikipedia.org/wiki/AOS_and_SOA.

[9] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 3.0, 2013. URL `https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf`.

[10] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. http://yaml.org/spec/1.2/spec.html, 2018.

[11] Richard J Gowers, Max Linke, Jonathan Barnoud, Tyler J E Reddy, Manuel N Melo, Sean L Seyler, Jan Domański, David L Dotson, Sébastien Buchoux, Ian M Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. In *Proc. 15th Python Sci. Conf.*, pages 98–105, 2016.

[12] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. Software News and Updates MDAnalysis : A Toolkit for the Analysis of Molecular Dynamics Simulations. *J. Comput. Chem.*, 32(10):2319–2327, 2011. doi: 10.1002/jcc.