VerMoUTH Documentation

Release 0.10.1.dev28

Peter C Kroon, Jonathan Barnoud, Tsjerk A Wassenaar, Siewert-J

Apr 04, 2024

CONTENTS:

CHAPTER

ONE

GENERAL OVERVIEW

VerMoUTH and martinize2 are tools for setting up starting structures for molecular dynamics (MD) simulations starting from atomistic coordinates, with a special focus on polymeric systems (including proteins and DNA). Existing tools that do this are generally limited to strictly linear polymers, while VerMoUTH and martinize2 make *no* assumptions regarding polymer structure. VerMoUTH is a python library that can be used programmatically. Martinize2 is a command line tool build on top of that.

VerMoUTH and martinize2 are also capable of dealing with structures where atom names are not provided, and to some extent with incomplete structures where atoms are missing from the input structure due to e.g. experimental limitations. There is also support for post-translational modifications.

VerMoUTH and martinize2 can be used to generate both atomistic and coarse-grained topologies and are the preferred method of generating topologies for the [Martini3] force field.

1.1 Installation instructions

Vermouth and martinize2 are distributed through pypi and can be installed using pip.

```
pip install vermouth
```

The behavior of the `pip` command can vary depending on the specifics of your python installation. See the documentation on installing a python package to learn more.

1.2 Quickstart

The CLI of martinize2 is very similar to that of [martinize1], and can often be used as a drop-in replacement. For example:

```
martinize2 -f lysozyme.pdb -x cg_protein.pdb -o topol.top
    -ff martini3001 -dssp -elastic
```

This will read an atomistic `lysozyme.pdb` and produce a [Martini3] compatible structure and topology at `cg_protein.pdb` and `topol.top` respectively. It will use the program [DSSP] to determine the proteins secondary structure (which influences the topology), and produce an elastic network. See `martinize2 -h` for more options! Note that if `martinize2` runs into problems where the produced topology might be invalid it will issue warnings. If this is the case it won't write any output files, but also see the `-maxwarn` flag.

1.3 General layout

In VerMoUTH a *force field* is defined as a collection of *Blocks*, *Links* and *Modifications*. Each of these is a graph, where nodes describe atoms (or coarse-grained beads) and edges describe bonds between these. *Blocks* describe idealized residues/monomeric repeat units and their MD parameters and interactions. *Links* are molecular fragments that describe MD parameters and interactions *between* residues/monomeric repeat units. *Modifications* are molecular fragments that describe *deviations* from *Blocks*, such as post-translational modifications and protonation states. *Mappings* describe how molecular fragments can be converted between force fields.

Finally, martinize2 is a pipeline that is built up from *Processors*, which are defined by VerMoUTH. Processors are isolated steps which function on either the complete system, or single molecules.

Martinize2 identifies atoms mostly based on their *connectivity*. We read the bonds present in the input file (as `CONECT` records), and besides that we *guess bonds* based on atom names (within residues) and on distances (between residues, using the same criteria as [VMD]). This means that your input structure must be reasonable.

1.4 Citing

A publication for vermouth and martinize 2 is currently being written. For now, please cite the relevant chapter from the thesis of Peter C Kroon:

Kroon, P.C. (2020). Martinize 2 – VerMoUTH. *Aggregate, automate, assemble* (pp. 16-53). ISBN: 978-94-034-2581-8.

1.5 References

CHAPTER

TWO

MARTINIZE 2 WORKFLOW

2.1 Pipeline

Martinize 2 is the main command line interface entry point for vermouth. It effectively consists of 6 stages:

1) reading input files
2) repairing the input molecule
3) mapping the input molecule to the desired output force field and resolution
4) applying Links to generate inter-residue interactions
5) post-processing, such as building an elastic network
6) writing output files

We'll describe each stage in more detail here. It is good to bear in mind however that in all stages the recognition/identification of atoms/particles is based on their *connectivity* in addition to any atom properties.

Throughout this document, when we refer to an 'edge' we mean a connection between two nodes in a graph. With 'bonds' we mean a chemical connection including the corresponding simulation parameters. Similarly, with 'molecule' we mean a connected graph consisting of atoms and edges. Note that this is not necessarily the same as a protein chain, since these could be connected through e.g. a disulphide bridge.

If martinize2 at some point encounters a situation that might result in an incorrect topology it will issue a warning, and refuse to write output files so that you are forced to examine the situation, but also see the `-maxwarn` CLI option. The options `-v` and `-vv` can be used to print more debug output, while the options `-write-graph`, `-write-repair` and `-write-canon` can be used to write out the system after *Make bonds*, *Repair graph* and *Identify modifications*, respectively. All of these can help you track down what's going wrong where.

2.2 1) Read input files

Martinize2 can currently read input structures from .gro and .pdb files. .pdb files are preferred however, since they contain more information, such as chain identifiers, and `TER` and `CONECT` records.

2.2.1 Reading PDB files

Reading PDB files is done by *PDBInput*. We take into account the following PDB records: `MODEL` and `ENDMDL` to determine which model to parse; `ATOM` and `HETATM`; `TER`, which can be used to separate molecules; `CONECT`, which is used to add edges; and `END`.

We issue a `pdb-alternate` warning if any atoms in the PDB file have an alternate conformation that is not 'A', since those will always be ignored.

Relevant CLI options: `-f`; `-model`; `-ignore`; `-ignh`.

2.2.2 Make bonds

Since atom identification is governed by their connectivity we need to generate bonds in the input structure. Where possible we get them from the input file such as PDB `CONECT` records. Beyond that, edges are added by *MakeBonds*. By default edges will be added based on atom names and distances, but this behaviour can be changed via the CLI option `-bonds-from`.

To add edges based on atom names the *Block* from the input force field is used as reference for every residue in the input structure where possible. This is not possible when a residue contains multiple atoms with the same name, nor when there is no *Block* corresponding to the residue[1]. Note that this will only ever create edges *within* residues.

Edges will be added based on distance when they are close enough together, except for a few exceptions (see below). Atoms will be considered close enough based on their element (taken from either the PDB file directly, or deduced from atom name[2]). The distance threshold is multiplied by `-bonds-fudge` to allow for conformations that are slightly out-of-equilibrium. Edges will not be added from distances in two cases: 1) if edges could be added based on atom names no edges will be added between atoms that are not bonded in the reference *Block*. 2) If the edge would connect 2 residues, and at least one of the atoms involved is a hydrogen atom. Edges added based on distance are logged as debug output.

If your input structure is far from equilibrium and adding edges based on distance is likely to produce erroneous results, make sure to provide `CONECT` records describing at least the edges between residues, and between atoms involved in modifications, such as termini and PTMs.

We issue a `general` warning when it is requested to add edges based on atom names, but this cannot be done for some reason. This commonly happens when your input structure is a homo multimer without `TER` record and identical residue numbers and chain identifiers across the monomers. In this case martinize2 cannot distinguish the atom "N", residue ALA1, chain "A" from the atom "N", residue ALA1, chain "A" in the next monomer. The easiest solution in this case is to place strategic `TER` records in your PDB file.

Relevant CLI options: `-bond-from`; `-bonds-fudge`

2.2.3 Annotate mutations and modifications

As a last step martinize2 allows you to make some changes to your input structure from the CLI, for example to perform point mutations, or to apply PTMs and termini. This is done in part by *AnnotateMutMod*, and completed by *Repair graph*.

The `-mutate` option can be used to change the residue name of one or more residues. For example, you can specify `-mutate PHE42:ALA` to mutate all residues with residue name "PHE" and residue number 42 to "ALA". Or change all "HSE" residues to "HIS": `-mutate HSE:HIS`. Modifications can be specified in a similar way.

[1] Based on residue name.

[2] The method for deriving the element from an atom name is extremely simplistic: the first letter is used. This will go wrong for two-letter elements such as 'Fe', 'Cl', and 'Cu'. In those cases, make sure your PDB file specifies the correct element. See also: *add_element_attr()*

The specifications `nter` and `cter` can be used to quickly refer to all N- and C-terminal residues respectively[3]. In addition, the CLI options `-nter` and `-cter` can be used to change the N- and C-termini. By default martinize2 will try to apply charged protein termini ('N-ter' and 'C-ter'). If this is not what you want, for example because your molecule is not a protein, be sure to provide the appropriate `-nter` and `-cter` options. You can specify the modification `none` to specify that a residue should not have any modifications. Note that if you use this for the termini you may end up with chemically invalid, uncapped, termini.

Relevant CLI options: `-mutate`, `-modify`, `-nter`, `-cter`, `-nt`

2.3 2) Repair the input graph

Depending on the origin of your input structure, there may be atoms missing, or atoms may have non-standard names. In addition, some residues may include modifications such as PTMs.

2.3.1 Repair graph

The first step is to complete the graph so that it contains all atoms described by the reference *Block*, and so that all atoms have the correct names. These blocks are taken from the input force field based on residue names (taking any mutations and modifications into account). `RepairGraph` takes care of all this.

To identify atoms in a residue we consider the *Maximum common induced subgraph* between the residue and its reference since the residue can be both too small (atoms missing in the input) and too large (atoms from PTMs) at the same time. Unfortunately, this is a very expensive operation which scales exponentially with the size of the residue. So if you know beforehand that your structure contains (very) large PTMs, such as lipidations, consider specifying those as separate residues.

The maximum common induced subgraph is found using `ISMAGS`, where nodes are considered equal if their elements are equal. Beforehand, the atoms in the residue will be sorted such that the isomorphism where most atom names correspond with the reference is found. This sorting also speeds up the calculation significantly, so if you're working with a system containing large residues consider correcting some of the atom names.

We issue an `unknown-residue` warning if no *Block* can be retrieved for a given residue name. In this case the entire molecule will be removed from the system.

2.3.2 Identify modifications

Secondly, all modifications are identified. *Repair graph* also tags all atoms it did not recognise, and those are processed by `CanonicalizeModifications`.

Modifications are identified by finding the solution where all tagged atoms are covered by the atoms of exactly one *Modification*, where the modification must be an *induced subgraph* of the molecule. Every modification must contain at least one "anchoring" atom, which is an atom that is also described by a *Block*. Unknown atoms are considered to be equal if their element is equal; anchor atoms are considered equal if their atom name is equal. Because modifications must be *induced subgraphs* of the input structure there can be no missing atoms!

After this step all atoms will have correct atom names, and any residues that include modifications will be labelled. This information is later used during the *resolution transformation*

An `unknown-input` warning will be issued if a modification cannot be identified. In this case the atoms involved will be removed from the system.

[3] N- and C-termini are defined as residues with 1 neighbour and having a higher or lower residue number than the neighbour, respectively. Note that this definition also includes termini for non-proteins, but it does not include zwitterionic amino acids! This also means that if your polymer has a chain break you'll end up with more termini than you would otherwise expect.

2.3.3 Rebuild coordinates for missing atoms

Currently martinize2 is not capable of rebuilding coordinates for missing atoms.

2.4 3) Resolution transformation

The resolution transformation is done by *DoMapping*. This processor will produce your molecules at the target resolution, based on the available mappings. These mappings are read from the `.map` and `.mapping` files available in the library[4]. See also *File formats*. In essence these mappings describe how molecular fragments (nodes and edges) correspond to a block in the target force field. We find all the ways these mappings can fit onto the input molecule, and add the corresponding blocks and modifications to the resulting molecule.

For a molecular fragment to match the input molecule the atom and residue names need to match[5]. This is why we first *repair* the input molecule so that you only need to consider the canonical atom names when adding mappings. Mappings defined by `.mapping` files can also cross residue boundaries (where specified).

Edges and interactions within the blocks will come from the target force field. Edges between the blocks will be generated based on the connectivity of the input molecule, i.e. if atoms A and B are connected in the input molecule, the particles they map to in the output force field will also be connected. Interactions across separate blocks will be added in the next step.

The processor will do some sanity checking on the resulting molecule, and issue an `unmapped-atom` warning if there are atoms in the input molecule for which no mapping can be found. In addition, this warning will also be issued if there are any non-hydrogen atoms that are not mapped to the output molecule. A more serious `inconsistent-data` warning will be issued for the following cases:

- there are multiple modification mappings, which overlap
- there are multiple block mappings, which overlap
- there is an output particle that is constructed from multiple input atoms, and some "residue level" attributes (such as residue name and number) are not consistent between the constructing atoms.
- there is an atom which maps to multiple particles in the output, but these particles are disconnected
- there is an interaction that is being set by multiple mappings

Relevant CLI options: `-ff`, `-map-dir`

2.5 4) Apply Links

Next interactions *between* residues are added by *DoLinks*. We do this based on the concept of *Links*, which are molecular fragments that describe interactions, and which atoms they should apply to. Links are very powerful and flexible tools, and we use them to generate all interactions that depend on the local structure of the polymer. For example, all interactions that depend on the protein sequence or secondary structure are defined by *Links*.

Links can both add, change and remove interactions and nodes. Because of this, the order in which links are applied matters for the final topology. We apply them in the order in which they are defined in the force field files. Therefore it is important to define links in the order of most general to most specific. A link is applied in all the places where it fits onto the molecule produced by *the mapping step*.

[4] When `-ff` (target force field) and `-from` (original force field) are the same the mappings will be generated automatically.

[5] This is only mostly true. All attributes except a few that are not always defined must match. Not all attributes (such as 'mass') are defined in all cases, depending on the source of the mappings. Note that we also take into account that atom names might have changed due to modifications: we use the atom name as it is defined by the *Block*.

For a link to match all its node attributes must match, where the 'order' attribute is a special case. The order attributes are translated to a difference in residue numbers, so that nodes 'BB' and '+BB' must have a difference in residue number of exactly 1[6]. Due to the reliance on residue numbers this can cause complications for non-linear polymers. For those cases order specifications such as '>' (greater than) and '*' (different from)[7] might be useful.

2.6 5) Post processing

There can be any number of post processing steps. For example to add an elastic network, or to generate Go virtual sites. We will not describe their function here in detail. Instead, see for example *ApplyRubberBand* and *VirtualSiteCreator*.

Relevant CLI options: `-elastic`, `-ef`, `-el`, `-eu`, `-ermd`, `-ea`, `-ep`, `-em`, `-eb`, `-eunit`, `-go`, `-go-eps`, `-go-moltype`, `-go-low`, `-go-up`, `-go-res-dist`

2.7 6) Write output

Finally, the topology and conformation are written to files (if no warnings were encountered along the way). Currently martinize2 and VerMoUTH can only write Gromacs itp files. Martinize2 will write a separate itp file for every unique molecule in the system.

Relevant CLI options: `-x`, `-o`, `-sep`, `-merge`

[6] Also '-BB' and 'BB', '+BB' and '++BB', etc.
[7] Remember that links can overlap! The link `BB *BB` will be applied both forwards and backwards!

CHAPTER

THREE

TECHNICAL BACKGROUND

Here we will provide some additional technical background about the chosen data structures and graph algorithms.

3.1 Processor

Processors are relatively simple. They form the fundamental steps of the martinize2 pipeline. Processors are called via their *run_system()* method. The default implementation of this method iterates over the molecules in the system, and runs the *run_molecule()* method on them. This means that implementations of Processors must implement either a `run_system` method, or a `run_molecule` method. If the processor can be run on independent molecules the `run_molecule` method is preferred; `run_system` should be used only for cases where the problem at hand cannot be separated in tasks-per-molecule.

In their `run_molecule` method Processor implementations are free to either modify *molecules* or create new ones. Either way, they must return a *Molecule*. The `run_system` will be called with a *System*, which will be modified in place.

3.2 Data

VerMoUTH knows several data structures, most of which describe atoms (or CG beads) and connections between those. As such, these are modelled as mathematical graphs, where the nodes describe the particles, and edges the bonds between these. In addition, these data structures describe the MD parameters and interactions, such as bonds, atom types, angles, etc.

3.2.1 Molecule

A *Molecule* is a `Graph` where nodes are atoms/beads, and edges are the connections between theses (i.e. bonds[1]) Generally, molecules are a single connected components[2]. Interactions are accessible through the *interactions* attribute. Non-bonded parameters are not fully defined: nodes have an ‘atype’ attribute describing the particle type to be used in an MD simulation, but we don’t store the associated e.g. Lennard-Jones parameters.

Molecules define a few notable convenience methods:

- *merge_molecule()*: Add all atoms and interactions from a molecule to this one. Note that this can also be used to add a *vermouth.molecule.Block* to a molecule! This way you can incrementally build polymers from monomers. This method will always produce a disconnected graph, so be sure to add the appropriate edges afterwards.

[1] But note that not every edge has to correspond to a bond and vice versa.
[2] I.e. there is a path from any node to any other node in the molecule.

- `make_edges_from_interactions()`: To generate edges from bond, angle, dihedral, cmap and constraint interactions. This is the only way interactions and their parameters are interpreted in vermouth.

3.2.2 Block

A `Block` can be seen as a canonical residue containing all atoms and interactions, and where all atom names are correct. A block should be a single connected component, and atom names within a block are assumed to be unique.

Blocks can be defined through Gromacs' `.itp` and `.rtp` file formats.

`Blocks` define a few notable convenience methods:

- `guess_angles()`: Generate all possible angles based on the edges.
- `guess_dihedrals()`: Generate all possible dihedral angles based on the edges.
- `to_molecule()`: Create a new `Molecule` based on this block.

3.2.3 Link

A `Link` is used to describe interactions *between* residues. As such, it consists of nodes and edges describing the molecular fragment it should apply to, as well as the associated changes in MD parameters. For example, a link can describe the addition, change or removal of specific interactions or node attributes. They can also be used to remove nodes. Although it is possible to generate *all* MD parameters and interactions using Links, rather than taking them from constituent blocks, this is not the preferred method. The approach where links only affect the parameters where they depend on the local structure makes it easier to reason about how the final topology is constructed, and the performance is better.

Besides nodes, edges, and interactions, links also describe non-edges, patterns and removed interactions. Non-edges and patterns are used when matching the link to a molecule. Where there is a non-edge in the link there cannot be an edge in the molecule, and the atoms involved do not need to be present in the molecule. Patterns provide a concise way where either one of multiple conditions must be met. For example two neighbouring 'BB' beads, where one must have a helical secondary structure, and the other should be a coil.

Links can be defined through *.ff files*. See also: *Apply Links*.

3.2.4 Modification

A `Modification` describes how a residue deviates from its associated `Block`, such as non-standard protonation states and termini. Modifications differentiate between atoms/particles that should already be described by the block and atoms that are only described by the modification.

A modification can add or remove nodes, change node attributes, and add, change, or remove interactions; much like a *Link*. Note that a modification *must* always add at least one node. Otherwise there will be no unidentified nodes to be picked up by the processor.

Modifications can be defined through *.ff files*. See also: *Identify modifications*.

3.2.5 Force Field

A *force field* is a collection of *Blocks*, *Links* and *Modifications*. Force fields are identified by their `name`, which should be unique. Within a force field blocks and modifications should also have unique names.

Note that this is only a subset of a force field in the MD sense: a VerMoUTH *force field* does not include e.g. non-bonded parameters (only the particle types are included), or functional forms.

3.2.6 Mapping

A *Mapping* describes how molecular fragments can be transformed from one force field to another.

Mappings can be provided through [backward] style `.map` files, or the more powerful (but verbose) *.mapping* format. See also: *Resolution transformation*.

3.3 Graph algorithms

Vermouth describes molecules and molecular fragments as graphs where atoms are nodes and connections between them (e.g. bonds) are edges. This allows us to use the *connectivity* to identify which atom is which, meaning we are no longer dependent on atom names.

3.3.1 Definitions

Graph

A graph $G = (V, E)$ is a collection of nodes (V) connected by edges (E): $e_{ij} = (v_i, v_j) \in E$. In undirected graphs $e_{ij} = e_{ji}$. Unless we specify otherwise all graphs used in vermouth are undirected. The size of a graph is equal to the number of nodes: $|G| = |V|$.

Subgraph

Graph $H = (W, F)$ is a subgraph of graph $G = (V, E)$ if:

$$\begin{aligned}|H| < |G| \\ W \subset V \\ e_{ij} \in F \quad \forall e_{ij} \in E \\ e_{ij} \notin F \quad \forall e_{ij} \notin E\end{aligned}$$

This means that all nodes in H are in G, and that nodes are connected in H if and only if they are connected in G.

Graph isomorphism

A graph isomorphism m between graphs $H = (W, F)$ and $G = (V, E)$ is a bijective mapping $m : V \mapsto W$ such that the following conditions hold:

$$\begin{aligned}|H| = |G| \\ m(v) \simeq v \\ \forall v \in V \\ (m(v_i), m(v_j)) \simeq (v_i, v_j) \\ : (m(v_i), m(v_j)) \in F \ \ \forall (v_i, v_j) \in E\end{aligned}$$

This means that every node in G maps to exactly one node in H such that all connected nodes in G are connected in H. Note that labels/attributes on nodes and edges (such as element or atom name) can affect the equivalence criteria.

Subgraph isomorphism

A subgraph isomorphism is a *Graph isomorphism*, but without the constraint that $|H| = |G|$. Instead, $|H| \leq |G|$ if H is subgraph isomorphic to G.

Induced subgraph isomorphism

As *Subgraph isomorphism* with the added constraint that equivalent nodes not connected in G are not connected in H:

$$(m(v_i), m(v_j)) \notin F \quad \forall (v_i, v_j) \notin E$$

We denote H being induced subgraph isomorphic to G as $H \precsim G$.

It is important to note that a path graph is *not* subgraph isomorphic to the corresponding cycle graph of the same size. For example, n-propane is not subgraph isomorphic to cyclopropane!

Maximum common induced subgraph

The maximum common induced subgraph between G and H is the largest graph J such that $J \precsim G$ and $J \precsim H$. Commonly the answer is given as a general mapping between G and H.

3.3.2 Isomorphism

Vermouth and martinize2 identify atoms by connectivity, generally combined with a constraint on element or atom name. We do this using either a *Maximum common induced subgraph* (during the *Repair graph* step) or a *Induced subgraph isomorphism* (the other steps). In all these cases we effectively find how nodes in the molecule we're working on match with nodes in our reference graphs, such as *blocks*.

During the *Repair graph* step there are two, related, complications: 1) we need a "best" overlay, where as many atom names match as possible; and 2) There can be very many (equivalent) possible overlays/isomorphisms. Let's address the second concern first. As example we'll look at the automorphisms (= self-isomorphism, i.e. how does a graph fit on itself) of propane (`CH3-CH2-CH3`).

There are 2 isomorphisms for the carbons: $C_\alpha - C_\beta - C_\gamma \mapsto C_\alpha - C_\beta - C_\gamma$ and $C_\alpha - C_\beta - C_\gamma \mapsto C_\gamma - C_\beta - C_\alpha$. Similarly, there are 2 isomorphisms for the central methylene group: $H_1 - C_\beta - H_2 \mapsto H_1 - C_\beta - H_2$ and $H_1 - C_\beta - H_2 \mapsto H_2 - C_\beta - H_1$. Each terminal methyl group however, has 6 unique isomorphisms!

$$H_1H_2H_3 \mapsto (H_1H_2H_3, H_1H_3H_2, H_2H_1H_3, H_3H_1H_2, H_2H_3H_1, H_3H_2H_1)$$

This means that in total, propane, a molecule consisting of 11 atoms, has $2(carbons) \times 2(methylene) \times 6(methyl) \times 6(methyl) = 144$ automorphisms! Now imagine how this scales for a lipid. Clearly this spirals out of control very quickly, and it is generally unfeasible to generate all possible isomorphisms[1].

Luckily for us however, we're not interested in finding all these isomorphisms, since we can consider most of these to be equivalent. For our use case it doesn't matter whether H_1 maps to H_1 or H_2 so long as H_1 and H_2 are equivalent. There is one catch however: we need to find the isomorphism where most atom names match. We can achieve this by preferentially using nodes with a lower index[2] when given a choice between symmetry equivalent nodes. The

[1] This problem gets *even* worse when trying to find the *Maximum common induced subgraph*.

[2] In other words, we impose an ordering on the nodes in the graph. We do this by ordering the nodes based on whether there is a node with a corresponding atom name in the reference and subsequently sorting by atom name.

[ISMAGS] algorithm does exactly this: it calculates symmetry unique isomorphisms preferentially using nodes with a smaller index.

Note that this problem only comes up when your graphs are (very) symmetric. In all other steps we constrain the isomorphism such that nodes are only considered equal if their atom names match. Since atom names are generally unique, this means that this problem is sidestepped completely. The only place where we cannot do this is during the *Repair graph* step, since at that point we cannot assume that the atoms names in our molecule are correct.

CHAPTER

FOUR

FILE FORMATS

Vermouth has two main types of data files:

- Force fields describe *blocks*, *links* and *modifications* to generate topologies.
- *Mappings* describe the transformations required for going from one description (force field) to another or vice versa.

The two types of data are contained in data/force_fields and data/mappings, respectively. The force_fields directory has a subdirectory for each force field that is available in vermouth and martinize2. The mappings directory has no mandatory organization; any mapping applies to two force fields, which are specified in the mapping file. For convenience, the mappings may be organized in subfolders. In particular, mappings from and to the canonical description, which is based on the charmm36 force field, are typically placed in a subdirectory with the name of the other force field (e.g., martini30).

4.1 Data structures and file formats

Please note that the file formats described here may undergo changes in the future. Features that are in the code, but are not described here should not be considered stable and will likely be deprecated in the future.

For its description of the force fields, topologies and mappings, Vermouth uses a file format based on the one used by Gromacs, consisting of named sections and subsections referred to as directives, each indicated with tag between square brackets. Directives are divided into top-level and sub-level sections.

4.2 Force field file (.ff)

The top-level directives for the force field and the topologies are macros, variables, citations, moleculetypes, links, and modifications.

4.2.1 Allowed major directives

The format recognizes the following directives:

- **`[ macros ]`**
  - `optional`
  - The macros section has no further subsections and lists substitution patterns to be applied throughout the file being read.
  - Macro values are substituted using the name with a preceding `$`. This is similar to the use of variables in shell scripting and makes it easy to write generalizations and to use and change default values.

    - The following example specifies that the protein default backbone bead type is P2. It can be referred to in the following sections of the file as `$prot_default_bb_type` .. code-block:

      ```
      [ macros ]
      prot_default_bb_type P2
      ```

- **`[ variables ]`**
    - `optional`
    - The variables section has no further subsections and lists a number of variable stored as key value pairs in the *force field object <data: force field>*. This allows retrieving the parameters using `force_field.variables[key] = value`.
    - Variables are used to control force field wide parameters that are tied to a specific force field version.
    - For example, the text below specifies that the bond type of the elastic networkx should be 1 for the force field. .. code-block:

      ```
      [ variables ]
      elastic_network_bond_type 1
      ```

- `[ citations ]`
    - `optional`
    - The citations section lists the citations to be used for the force field. Citations are named and refer to an entry in the bibtex file *citations.bib* in the force field directory.
    - Note that martinize2 automatically adds some citations via processors. Thus it expects them to be present in the citations file.
    - Citations can also be specified for moleculetypes, links, and modifications, in a citation subsection.
    - An example of this is: .. code-block:

      ```
      [ citations ]
      Martini3
      ```

    - If you want to add a citation to a specific molecule, the citation directive can be added as subsection to the moleculetype directive: .. code-block:

      ```
      [ moleculetype ]
      ALA 3
      [ citations ]
      mol_specific_citation
      ```

- **`[ moleculetype ]`**
    - `optional`
    - The moleculetype describes a *block*, i.e., a topological building block, comprising of particles (atoms) with their properties and interactions. This can be a separate molecule or a part of a larger molecule, typically a monomeric unit in a polymer. The moleculetype has a name and the number of bonds to use for exclusions as its first content line. This is followed by one or more subsections. These subsections are listed below.
    - This directive must be followed by a line specifying the residue or molecule name as well as the number of bonded partners excluded when computing the non-bonded interactions.
    - An example of this is: .. code-block:

```
[ moleculetype ]
ALA 1
```

- `[ links ]`
  - `optional`
  - To generate a topology for a polymer or any molecule constructed from joining parts, Vermouth connects moleculetypes using links. A link describes how blocks are to be joined, what changes are effected in the atom lists and which interactions are added, removed, or altered. The changes in the atom and interaction lists are specified using the corresponding subsections as under moleculetype. However, there are also several link exclusive subsections as listed below.
  - There may be any number of lines following the section tag. These lines can list selection statements for filtering atoms in which to search for matching patterns. Each line specifies a property and the corresponding value. The selection statements may include filters based on, e.g., the residue name and the secondary structure type, which are used to determine the structural properties of protein backbone in the Martini force field.
  - An example of this feature is is shown below, where the link only applies to atoms with the resname ALA and the secondary structure assignment coil. .. code-block:

    ```
    [ link ]
    resname "ALA"
    cgsecstruc "C"
    ```

- `[ modification ]`
  - `optional`
  - Modifications can be used to edit molecules or parts thereof (blocks), e.g., for specifying protonation states. Each modification starts with a line with the name. Thereafter may follow subsections as under links. A modification may add, remove, or change atoms, interactions and/or edges, using the corresponding subsections.

4.2.2 Allowed sub-directives: Moleculetype

- `[ atoms ]`
  - `mandatory`
  - Each line in the atoms section describes one particle, corresponding to a node in the molecular graph. The description comprises the following fields:
    * atom number
    * atom type
    * residue index
    * residue name
    * atom name
    * charge group (optional)
    * charge (optional)
    * mass (optional)
  - An example is shown below: .. code-block:

```
[ atoms ]
;id type resnr residu atom cgnr   charge mass
1   P5   1     GLY    BB     1      0    47
```

- `[ edges ]`
  - `optional`
  - Edges will be added to the molecular graph when required based on the interactions directives, but they can also be added explicitly by listing them under the edges subsections. An edge is specified by the corresponding atom names. Note that these edges do not result in any interactions, but they rather complete the molecular graph.
  - An example is shown below: .. code-block:

```
[ edges ]
BB SC1
```

- `[ interaction_name ]`
  - `optional`
  - There are several options for subsections describing interactions between particles. Of these, bonds, angles, dihedrals, cmap, and constraints will automatically add the corresponding edges to the molecular graph, unless specified explicitly by setting an attribute 'edge' to false in a subsection #meta or following a specific interaction.
  - Each line in an interactions subsection specifies one interaction by listing the atoms involved by name, followed by the interaction parameters. For all interactions, the parameters are read as is and written to the output topology without interpreting and/or checking. Bond/constraint lengths, angles and dihedral angles may be used for generating missing coordinates.
  - A full list of interactions is given below, corresponding to the list of intramolecular interactions available in Gromacs, with a number specifying the number of particles involved in the interaction. Note that improper dihedrals are listed as a separate interaction type, whereas in Gromacs these fall under the dihedrals section.
  - **Known interactions:**
    - bonds(2)
    - angles(3)
    - dihedrals(4)
    - impropers(4)
    - constraints(2)
    - pairs(2)
    - pairs_nb(2)
    - SETTLE(1)
    - virtual_sites2(3)
    - virtual_sites3(4)
    - virtual_sites4(5)
    - position_restraints(1)
    - distance_restraints(2)

        * dihedral_restraints(4)
        * orientation_restraints(2)
        * angle_restraints(4)
        * angle_restraints_z(2)
        * cmap(...)

  – Any of the subsections can be given multiply times, in which case they are additive. Do note that in the output topology specifying the same interaction several times (the same type and particles) will overwrite any previous one, except when they are given different contexts (see below).
  – In order to stack interactions with the same number of atoms but different parameters a special annotation with a version number can be used. This is especially relevant for dihedrals, where multiple ones may be specified. An example is shown below: .. code-block:

```
[ dihedrals ]
BB SC1 SC2 SC3 9  180  5  1 {"version": 1}
BB SC1 SC2 SC3 9  180  1  2 {"version": 2}
BB SC1 SC2 SC3 9    0  2  3 {"version": 3}
```

4.2.3 Allowed sub-directives: Link

- **`[ atoms ]`**
  – `optional`
  – The atoms directive is optional within links. It can be given to for example overwrite link attributes or specify attributes of specific atoms. An attributes statement is a JSON style mapping of key/value pairs, similar to those used in the #meta syntax (see below).
  – Note that here the syntax is different from the moleculetype atoms directive. This directive requires the particle name followed by a dict of attributes e.g `BB {"resname":  "ALA"}`
  – To overwrite atom attributes of existing atoms when a link is applied the user can provide a dict of parameters within using the replace key as follows: .. code-block:

```
[ atoms ]
BB {"replace": {"charge": -1}}
```

- **`[ interaction_name ]`**
  – `optional`
  – A link may list any number of interactions to be added, if a link applies. The syntax is the same as for the *moleculetype* sub- directive. However, when the listed particles are not within the same residue a prefix has to be provided that specifies the order relative to a given residue. The following prefixes are allowed:

```
* +, ++, +++ : first, second, third following residue
* -, --, --- : first, second, third previous residue
* >, >>, >>> : residue with larger resid but unspecified
               difference between the residues
* <, <<, <<< : residue with smaller resid but unspecified
               difference between the residues
* *          : other residue
```

Thus, +CA in amino acids refers to the C-alpha atom in the C-terminal connected neighbor, while >SG in the construction of a disulphide bridge will refer to the SG atom in the partner cysteine.

These prefixes can also be used in the atoms and pattern subsections.

- For example, to specify a bond between the backbone bead of a given amino acid and the next one, we write: .. code-block:

```
[ link ]
[ bonds ]
BB +BB  1 0.47 5000
```

- `[ patterns ]`
  - `optional`
  - If no pattern is given, the link pattern will consists of the particles and their connectivity inferred from the interactions and atoms sub directive.
  - To overwrite this default pattern one can list patterns of atoms to which the link applies. Each line in the subsection describes a pattern. At least one of the patterns must apply for the link to match. A pattern consists of atom identifiers. Each atom identifier consists of a name which may be preceded by a prefix indicating the relative position in terms of residues.
- `[ features]`
  - `optional`
  - The features subsection lists features to apply to the link itself. These can be used to control the application of links during the building of topologies. For example, setting the feature 'scfix' will cause the links to be applied only if the option -scfix is given to martinize2.
- `[ molmeta ]`
  - `optional`
  - The molmeta subsection lists metadata to be added/changed in the molecular graph. These metadata can be used (and modified) by Vermouth's processors and for provenance.
- `[ edges ]`
  - `optional`
  - Within the context of links, the edges specify that an edge should or shouldn't be present, respectively, for the link pattern to match.
  - They should mostly be used in cases, where interactions are applied for which edges cannot be made automatically.
- `[ non-edges ]`
  - `optional`
  - This directive specifies that an edge should be absent in order for the link to apply. Note that the first particle must be present in the link and the second one must the partner with which to not form an edge.
  - This syntax is likely to be deprecated in the near future.

4.2.4 Allowed sub-directives: Modifications

- **[atoms]**
  - `mandatory`
  - The atoms subsection under a modification lists both anchors and atoms to be added to anchors or changed. Entries consist of an atom name followed by an attributes statement. Atoms that are added need to set the "PTM_atom" attribute to True and require a valid "element" attribute. Atoms for which the "PTM_atom" attribute is absent (or False) must already be described by the relevant *block* with the same atomname. The "replace" attribute may be set to a (nested) JSON dict, listing the atom attributes to be changed and the new values corresponding to the modification. Such changes can also be applied to atoms already present in the molecular graph, i.e., the 'non-PTM atoms'.`
- **[interaction_name]**
  - `optional`
  - A modification may list any number of interactions to be added, if a modification applies. The syntax is the same as for the link sub directive.

4.2.5 Special meta data

The ff file format employs some special syntaxes that can be used to affect the order in which interactions are displayed, comment them, or group them.

So called `#meta` statements may be added at any line under an interactions directive. These directives always apply to all entries if the remaining subsection. The metadata is given as a JSON style mapping of key/value pairs. Vermouth currently employs the following possible metadata key/value pairs:

- `{ifdef: value}`, puts interactions within `#ifdef value` statements.
- `{ifndef: value}`, puts interactions within `#ifndef value` statements.
- `{group: value}`, will list all interactions after inserting a comment `; value`
- `{comment: value}`, will put a comment `; value` after each interaction

For example, the meta block below will group all interactions together under a comment 'Side chain bonds' and put these within a #ifdef statement.

```
[ link ]
#meta {"group": "Side chain bonds", "ifdef": "FLEXIBLE"}
```

Metadata can also be added to a single line by adding an attribute statement as the last element.

4.3 Mapping files (.map & .mapping)

A mapping specifies the conversion from one force field description to another. If the transformation is from a higher resolution force field to a lower resolution, e.g., from the canonical description to Martini, the process is typically called 'forward mapping'.

The vermouth library currently utilizes two mapping formats. The `.map` format, which was originally developed for the backward program, is used to describe how two *blocks* correspond to each other. The second format (`.mapping`) is exclusively used in the context of *modifications* and is an extension to the first format.

4.3.1 File structure (.map)

The file is structured into sections, each beginning with a directive enclosed in square brackets (`[]`).

Allowed directives .map

The format recognizes the following directives:

- **`[molecule]`**
  - This directive is immediately followed by a single line containing an alphanumeric string specifying the residue name. This name denotes the residue under consideration. A *block* with this name must be defined in both the `[from]` and `[to]` force fields.
  - `mandatory`
- **`[from]`**
  - The directive is followed by a single line containing an alphanumeric string corresponding to name of the origin (i.e. higher resolution) force field.
  - `mandatory`
- **`[to]`**
  - The directive is followed by a single line containing an alphanumeric string corresponding to the name of the target (i.e. higher resolution) force field.
  - `mandatory`
- **`[martini]`**
  - The directive is followed by any number of lines. Each line must contain space separated bead names.
  - `mandatory`
- **`[atoms]`**
  - This directive introduces a section that can span multiple lines. Each line within this section must adhere to the following format:
    * An integer specifying the atom number.
    * An alphanumeric string corresponding to an atom name in the origin force field.
    * Any number of bead names. These beads must have been previously listed under the `[martini]` directive.
  - All atoms described by the referenced block should be described.
  - `mandatory`
- **`[chiral]`**
  - Contains chirality specifications used for the original backwards program.
  - `ignored`
- **`[trans]`**
  - Contains geometry specifications used for the original backwards program.
  - `ignored`
- **`[out]`**
  - Contains geometry specifications used in the original backwards program.

  - ignored

Example of .map file

```
[ molecule ]
ALA ALA
[ martini ]
BB SC1
[ atoms ]
 1     N    BB
 2    HN    BB
 3    CA    BB
 5    CB    SC1
 9     C    BB
10     O    BB
```

4.3.2 File structure (.mapping)

The file is structured into sections, each beginning with a directive enclosed in square brackets (`[]`).

Allowed directives .mapping

- **`[modification]`**
  - Marks the beginning of a modification block. This directive does not require any following content.
  - `mandatory`
- **`[from]`**
  - Followed by the name of the origin force-field (e.g., amber).
  - `mandatory`
- **`[to]`**
  - Followed by the name of the target force-field (e.g., martini3001).
  - `mandatory`
- **`[from blocks]` and `[to blocks]`**
  - Each followed by the name of the modification in the respective force fields
  - `mandatory`
- **`[from nodes]`**
  - Lists all nodes that should be part of the mapping that are not yet described by `[from block]`
  - `optional`
- **`[from edges]`**
  - Contains all edges that are part of the mapped fragment that are not described by `[from block]`. In particular, all edges concerning nodes in `[from nodes]` must be listed here.
  - `optional`
- **`[mapping]`**

– Contains pairs of atom names and bead names, describing the actual mapping between the high-resolution and coarse-grained representations of the modification.

Example file of .mapping file

Below is an example of a `.mapping` file:

```
[modification]
[from]
amber
[to]
martini3001

[from blocks]
C-ter
[to blocks]
C-ter

[from nodes]
N
HN

[from edges]
HN N
N CA

[mapping]
CA BB
C  BB
O  BB
OXT BB
```

CHAPTER FIVE

TUTORIALS

You can find some examples on how to use martinize 2 in the martinize-examples repository: https://github.com/marrink-lab/martinize-examples

5.1 Adding new residues and links

Occasionally you may need a topology containing a residue that is not yet described by the force fields that ship with vermouth. In this case you will need to create the required data files yourself, and point martinize2 to them with the *-ff-dir* and *-map-dir* flags. The key thing to remember is that you will need to add/edit *three* files. You need to describe your new residue in the input force field (default charmm); the output force field, and the mapping between the two.

For this example we will add the required data files for a phosphorylated serine residue (`OC(=O)C(N)COP(=O)(=O)[O-]`). Note that the parameters presented here are for demonstration purposes only and not fit for actual science or simulations!

All input files for this tutorial can be found on github.

5.1.1 The input force field

The input force field is the force field best describing the structure and atom names in your input PDB file. By default we use the charmm naming scheme. Since the input force field will only be used to *repair* your input structure, only the atom names and edges are relevant.

We'll start by creating a force fields folder we can use to create the tutorial files; and in that folder we need to create a force field named `charmm`:

```
mkdir -p force_fields/charmm
```

Now we need to add the SEP *Block* to our `charmm` folder. Let's put it in the file `force_fields/charmm/sep.ff`:

```
[ moleculetype ]
SEP 3

[ atoms ]
 1 N 1 SEP N    1 0
 2 H 1 SEP HN   2 0
 3 C 1 SEP CA   3 0
 4 H 1 SEP HA   4 0
 5 C 1 SEP CB   5 0
 6 H 1 SEP HB1  6 0
 7 H 1 SEP HB2  7 0
```

(continues on next page)

(continued from previous page)

```
 8 O 1 SEP OG   8 0
 9 C 1 SEP C    9 0
10 O 1 SEP O   10 0
11 P 1 SEP P   11 0
12 O 1 SEP O1  12 0
13 O 1 SEP O2  13 0
14 O 1 SEP O3  14 -1

[ bonds ]
 3  5
 5  8
 1  2
 1  3
 3  9
 3  4
 5  6
 5  7
 9 10
 8 11
11 12
11 13
11 14
```

This file looks a lot like an ITP file, and if you have one of those you can simply drop it in and use it as is. In this case we didn't have an ITP file for SEP yet, so we had to make one. Since we only need atom names and bonds that's all we provide. Note that we added all hydrogens. Finally, if you prefer, you could also provide a RTP file instead of ITP.

5.1.2 The output force field

We also need to add the SEP *Block* to the output force field. Of course we'll use Martini 3 for this. Let's again start by making a martini3001 folder:

```
mkdir -p force_fields/martini3001
```

Now to add the block to `force_fields/martini3001/sep.ff`:

```
;;; PHOSPHOSERINE
[ moleculetype ]
SEP 1

; THESE PARAMETERS ARE FOR DEMONSTRATION PURPOSES ONLY. DO NOT USE.
[ atoms ]
; id type resnr residue atom cgnr charge
 1   P2   1     SEP     BB   1     0
 2   Q5n  1     SEP     SC1  1    -1

[ bonds ]
BB SC1 1 0.33 5000
```

At this point we can run `martinize2 -ff-dir force_fields -list-blocks` to check whether our new SEP blocks are picked up.

5.1.3 The mapping

Finally, we need to add the mapping describing how to get from charmm to martini3001. We need to make a folder:

```
mkdir mappings
```

In that folder, make a file `mappings/sep.charmm36.map`:

```
[ molecule ]
SEP

[ from ]
charmm

[ to ]
martini3001

[ martini ]
BB SC1

[ mapping ]
charmm

[ atoms ]
 1     N  BB
 2    HN  BB
 3    CA  BB
 4    HA  !BB
 5    CB  BB SC1
 6   HB1  !SC1
 7   HB2  !SC1
 8    OG  SC1
 9     C  BB
10     O  BB
11     P  SC1
12    O1  SC1
13    O2  SC1
14    O3  SC1
```

A few things are worth noting here. The HA, HB1, and HB2 atoms are mentioned here, but their mapping weight is 0, due to the exclamation point. In addition, CB will contribute to BB and SC1 with equal weight.

Ok, this great! At this point we can run `martinize2`:

```
martinize2 -ff-dir force_fields -map-dir mappings -f ala-sep-ala.pdb -x AJA.pdb -o topol.
↪top
```

And inspect the resulting `molecule_0.itp` to make sure our final topology is correct:

```
[ moleculetype ]
molecule_0 1

[ atoms ]
1 Q5  1 ALA BB  1    1
```

(continues on next page)

(continued from previous page)

```
2 TC3 1 ALA SC1 2  0.0
3 P2  2 SEP BB  3  0.0
4 Q5n 2 SEP SC1 3 -1.0
5 Q5  3 ALA BB  4   -1
6 TC3 3 ALA SC1 5  0.0

[ bonds ]
3 4 1 0.33 5000

#ifdef FLEXIBLE
; Side chain bonds
1 2 1 0.270 1000000
5 6 1 0.270 1000000
#endif

[ constraints ]
#ifndef FLEXIBLE
; Side chain bonds
1 2 1 0.270
5 6 1 0.270
#endif
```

We can see that we end up with the correct non-bonded parameters for our SEP residue, the C- and N-termini are looking good, and we have the BB-SC1 bond we specified.

There is a problem though, there are no bonds (or constraints) connecting the SEP residue to its neighbouring ALA residues!

5.1.4 The Links

In Vermouth and martinize2 we use *links* to describe interactions between residues. We need to these to the output force field—in this case martini3001.

We can add the following to `force_fields/martini3001/sep.ff`:

```
[ link ]
[ bonds ]
BB {"resname": "SEP"} +BB {"resname": "ALA"} 1 0.35 4000

[ link ]
[ bonds ]
BB {"resname": "SEP"} -BB {"resname": "ALA"} 1 0.35 4000

[ link ]
[ angles ]
-BB {"resname": "ALA"} BB {"resname": "SEP"} +BB {"resname": "ALA"} 10 100 20

[ link ]
[ angles ]
-BB BB {"resname": "SEP"} SC1 2 100 25
```

Links are small molecular fragments. For example, the first one consists of 2 BB beads. The first one has to be part of a SEP residue, and the second has to be part of an ALA residue. In addition, the + means the second BB has to have a resid of exactly one higher than the first BB. In our example, this link will apply a backbone bond between the SEP residue and ALA3.

The second link is almost identical, and applies a backbone bond between ALA1 and SEP. The two angles work in a similar fashion.

This would result in the following topology:

```
[ moleculetype ]
molecule_0 1

[ atoms ]
1 Q5  1 ALA BB  1    1
2 TC3 1 ALA SC1 2  0.0
3 P2  2 SEP BB  3  0.0
4 Q5n 2 SEP SC1 3 -1.0
5 Q5  3 ALA BB  4   -1
6 TC3 3 ALA SC1 5  0.0

[ bonds ]
3 4 1 0.33 5000
3 5 1 0.35 4000
3 1 1 0.35 4000

#ifdef FLEXIBLE
; Side chain bonds
1 2 1 0.270 1000000
5 6 1 0.270 1000000
#endif

[ constraints ]
#ifndef FLEXIBLE
; Side chain bonds
1 2 1 0.270
5 6 1 0.270
#endif

[ angles ]
1 3 5 10 100 20
1 3 4 2 100 25
```

We now have bonds between the backbone beads, as well as the 2 angles we need. In this case, since we don't intend to use this residue for anything other than an ALA-SEP-ALA peptide, we can combine these links:

```
[ link ]
[ atoms ]
-BB {"resname": "ALA"}
BB {"resname": "SEP"}
SC1 {"resname": "SEP"}
+BB {"resname": "ALA"}
[ bonds ]
BB +BB 1 0.35 4000
BB -BB 1 0.35 4000
[ angles ]
-BB BB +BB 10 100 20
```

(continues on next page)

(continued from previous page)

```
-BB BB SC1 2 100 25
```

Which will produce the exact same topology. If you *do* need to add a residue that can be used in any kind of protein please take a look at how the Martini 3 force field is implemented, and deals with e.g. the secondary structure dependence.

Links and Modifications

Something to keep in mind is that Links get applied after Modifications (at the time of writing). This can mean that your Link overwrites, for example, terminal parameters. For this reason, you can filter nodes where Links get applied much like you can limit Links by atom names or secondary structure. In particular, you can add a `"modifications"` attribute to links nodes. This follows the following rules:

1. Links that don't specify modifications simply match.
2. Links that specify empty modifications (`"modifications": []` or `null`) only match atoms that have no modifications.
3. Links that specify a list of modifications (`"modifications": ["C-ter", "ASP-HD2"]`) only match atoms that carry that exact set of modifications.
4. Links that specify a string or Choice of modifications (`"modifications": "C-ter"` or `"C-ter|COOH-ter"`) only match atoms where *all* the atoms modifications match.

5.2 Adding new modifications

In *Adding new residues and links* we added a whole new *Block* in order to describe a phosphoserine residue. This had the (dis)advantage that we had to redefine all the default serine interactions and parameters as well as the inter-residue *links*. There must be a better way!

Fortunately there is. Rather than describing a whole new SEP *Block* and all that entails we can instead describe just the way the phosphorylation *modified* the normal SER residue. This is exactly what *modifications* are for. As before, we need to add the new modification in 3 places: input force field, output force field, and mapping between the two. Note that the parameters presented here are for demonstration purposes only and not fit for actual science or simulations!

All input files for this tutorial can be found on github.

5.2.1 The input force field

During *repair* the regular SER atoms will be repaired, the missing hydrogen (HG) will be added (!), and the phosphate atoms will be annotated as being "nonstandard". During *Identify modifications* the *Processor* will try to identify these tagged atoms by finding a minimal set of modifications that describe all relevant atoms. For a modification to apply here it must be subgraph isomorphic to the input structure.

If we run `martinize2 -f ala-sep-ala.pdb -o topol.top -x AJA.pdb` we get, as expected, the warning that not all modifications could be identified:

```
WARNING - unknown-input - Could not identify the modifications for residues ['SER3'],␣
↪involving atoms ['21-O1', '22-O2', '23-O3', '24-P']
```

So let's define the modification in `force_fields/charmm/modification.ff`:

```
; THESE PARAMETERS ARE FOR DEMONSTRATION PURPOSES ONLY. DO NOT USE.
[ modification ]
SER-phos
[ atoms ]
O1 {"element": "O", "PTM_atom": true}
O2 {"element": "O", "PTM_atom": true}
O3 {"element": "O", "PTM_atom": true}
P  {"element": "P", "PTM_atom": true}
OG {"element": "O"}
HG {"element": "H", "replace": {"atomname": null}}
[ edges ]
OG P
P O1
P O2
P O3
```

As before, the input force field does not define any MD parameters or interactions. This modification contains nodes and edges. The edges are not very interesting, and just define the connections between nodes. Nodes on the other hand define 2 things: 1) the atom name as is should be (first column), and 2) any constraints the node must satisfy during the subgraph isomorphism as a JSON formatted mapping. The constraints should define at least 2 properties: the `element`, and `PTM_atom`. The element property is self explanatory, but the `PTM_atom` needs more explanation.

Modifications contain *2* types of nodes:

1. Nodes that are already described by the parent block (`PTM_atom` is false, this is the default). We call these nodes "anchors".
2. Nodes that are not yet described by the parent block (`PTM_atom` is true).

In addition, it's worth noting that *repair* reconstructed the HG atom (see `-write-repair`) since it's not in the input PDB. We use the "replace" property to describe all node attributes that need to change because of this modification. In this case we indicate that this atom should be removed again, by setting its atomname to "null". You can use the "replace" property to change *any* node property, including *e.g.* atom type and charge.

5.2.2 The output force field

We have to add a similar modification for the output force field in `force_fields/martini3001/modification.ff`:

```
[ modification ]
; THESE PARAMETERS ARE FOR DEMONSTRATION PURPOSES ONLY. DO NOT USE.
SER-PO4
[ atoms ]
BB  {"PTM_atom": false}
SC1 {"PTM_atom": false, "resname": "SER", "replace": {"atype": "Q5n", "charge": -1}}
[ bonds ]
BB SC1 1 0.33 5000
```

Nothing new here compared to the modification for the input force field. Note that here we *do* define the simulation parameters, and we define a bond.

5.2.3 The mapping

Finally, we need to add the mapping describing how to get from charmm to martini3001 in `mappings/SEP.mapping`:

```
; THESE PARAMETERS ARE FOR DEMONSTRATION PURPOSES ONLY. DO NOT USE.
[ modification ]
[ from ]
charmm
[ to ]
martini3001
[ from blocks ]
SER-phos
[ to blocks ]
SER-PO4
[ from nodes ]
N
HN
CA
HA
C
O
CB
HB1
HB2
[ from edges ]
N  HN
N  CA
CA HA
CA C
C  O
CA CB
CB HB1
CB HB2
CB OG
[ mapping ]
N   BB
HN  BB
CA  BB
HA  BB 0
C   BB
O   BB
CB  BB
CB  SC1
HB1 SC1 0
HB2 SC1 0
OG  SC1
P   SC1
O1  SC1
O2  SC1
O3  SC1
```

Firstly, notice that this is a different file format than the backwards format we used before. In this case we have to define between which force fields we're going to define a mapping (`charmm` and `martini3001`), and between which modifications (or blocks) (`SER-phos` and `SER-PO4`). This mapping has to define how to map the phosphate moiety (at least). This moiety will be mapped to the SC1 bead, so we will need to describe the complete mapping for that bead. In addition this mapping affects the mapping of the BB bead (since CB will now also contribute in part to it).

The charmm modification already define some nodes (see above), but not all the nodes required to describe the complete mapping for the BB and SC1 nodes, so these need to be described under *from nodes* and *from edges*. Finally, the actual mapping section should be self explanatory.

Now if we run `martinize2 -f ala-sep-ala.pdb -x AJA.pdb -o topol.top -ff-dir force_fields/ -map-dir mappings/` we see `INFO - general - Applying modification mapping ('SER-phos',)`

Now we need to check the produced itp file:

```
; THESE PARAMETERS ARE FOR DEMONSTRATION PURPOSES ONLY. DO NOT USE.
[ atoms ]
1 Q5  1 ALA BB  1   1
2 TC3 1 ALA SC1 2 0.0
3 P2  2 SER BB  3 0.0
4 Q5n 2 SER SC1 4  -1
5 Q5  3 ALA BB  5  -1
6 TC3 3 ALA SC1 6 0.0

[ bonds ]
3 4 1 0.33 5000

; Backbone bonds
1 3 1 0.350 4000
3 5 1 0.350 4000

#ifdef FLEXIBLE
; Side chain bonds
1 2 1 0.270 1000000
5 6 1 0.270 1000000
#endif

[ constraints ]
#ifndef FLEXIBLE
; Side chain bonds
1 2 1 0.270
5 6 1 0.270
#endif

[ angles ]
; BBB angles
1 3 5 10 127 20

; BBS angles regular martini
1 3 4 2 100 25
3 5 6 2 100 25

; First SBB regular martini
2 1 3 2 100 25
```

What we see here is that the atom type and bond we specified in the modification have been applied, and we can also no longer see the BB-SC1 bond that comes with the normal serine residue (`BB SC1 1 0.287 7500`) is no longer present. In addition, we find the usual backbone/protein interactions.

CHAPTER

SIX

VERMOUTH

6.1 vermouth package

6.1.1 Subpackages

vermouth.dssp package

Submodules

vermouth.dssp.dssp module

Assign protein secondary structures using DSSP.

`class vermouth.dssp.dssp.`**`AnnotateDSSP`**`(`*executable=None*, *savedir=None*`)`

Bases: *`Processor`*

`name = 'AnnotateDSSP'`

`run_molecule``(`*molecule*`)`

`class vermouth.dssp.dssp.`**`AnnotateMartiniSecondaryStructures`**

Bases: *`Processor`*

`name = 'AnnotateMartiniSecondaryStructures'`

`static` **`run_molecule`**`(`*molecule*`)`

`class vermouth.dssp.dssp.`**`AnnotateResidues`**`(`*attribute*, *sequence*, *molecule_selector=<function select_all>*`)`

Bases: *`Processor`*

Set an attribute of the nodes from a sequence with one element per residue.

Read a sequence with one element per residue and assign an attribute of each node based on that sequence, so each node has the value corresponding to its residue. In most cases, the length of the sequence has to match the total number of residues in the system. The sequence must be ordered in the same way as the residues in the system. If all the molecules have the same number of residues, and if the length of the sequence corresponds to the number of residue of one molecule, then the sequence is repeated to all molecules. If the sequence contains only one element, then it is repeated to all the residues ofthe system.

Parameters

- **`attribute`** (*`str`*) – Name of the node attribute to populate.

- **sequence** (*collections.abc.Sequence*) – Per-residue sequence.
- **molecule_selector** (*collections.abc.Callable*) – Function that takes an instance of *vermouth.molecule.Molecule* as argument and returns *True* if the molecule should be considered, else *False*.

name = 'AnnotateResidues'

run_molecule(*molecule*)

Run the processor on a single molecule.

Parameters
molecule (vermouth.molecule.Molecule) –

Return type
vermouth.molecule.Molecule

run_system(*system*)

Run the processor on a system.

Parameters
system (vermouth.system.System) –

Return type
vermouth.system.System

exception vermouth.dssp.dssp.**DSSPError**

Bases: Exception

Exception raised if DSSP fails.

vermouth.dssp.dssp.**annotate_dssp**(*molecule*, *callable=None*, *attribute='secstruct'*)

Adds the DSSP assignation to the atoms of a molecule.

Runs DSSP on the molecule and adds the secondary structure assignation as an attribute of its atoms. The attribute name in which the assignation is stored is controlled with the "attribute" argument.

Only proteins can be annotated. Non-protein molecules are returned unmodified, so are empty molecules, and molecules for which no positions are set.

The atom names are assumed to be compatible with DSSP. Atoms with no known position are not passed to DSSP which may lead to an error in DSSP.

Warning: The molecule is annotated **in-place**.

Parameters

- **molecule** (Molecule) – The molecule to annotate. Its atoms must have the attributes required to write a PDB file; other atom attributes, edges, or molecule attributes are not used.
- **callable** (*Callable*) – The function to call to generate DSSP secondary structure assignments. See also: *run_dssp()*, *run_mdtraj()*
- **attribute** (*str*) – The name of the atom attribute in which to store the annotation.

See also:

run_mdtraj, *run_dssp*, *read_dssp2*

`vermouth.dssp.dssp.`**`annotate_residues_from_sequence`**`(`*molecule*, *attribute*, *sequence*`)`

Sets the attribute *attribute* to a value from *sequence* for every node in *molecule*. Nodes in the n'th residue of *molecule* are given the n'th value of *sequence*.

Parameters

- **`molecule`** (*`networkx.Graph`*) – The molecule to annotate. Is modified in-place.
- **`attribute`** (*`collections.abc.Hashable`*) – The attribute to set.
- **`sequence`** (*`collections.abc.Sequence`*) – The values assigned.

Raises

`ValueError` – If the length of *sequence* is different from the number of residues in *molecule*.

`vermouth.dssp.dssp.`**`convert_dssp_annotation_to_martini`**`(`*molecule*, *from_attribute='secstruct'*, *to_attribute='cgsecstruct'*`)`

For every node in *molecule*, translate the *from_attribute* with *`convert_dssp_to_martini()`*, and assign it to the attribute *to_attribute*.

Parameters

- **`molecule`** (*`networkx.Graph`*) – The molecule to process. Is modified in-place.
- **`from_attribute`** (*`collections.abc.Hashable`*) – The attribute to read.
- **`to_attribute`** (*`collections.abc.Hashable`*) – The attribute to set.

Raises

`ValueError` – If not all nodes have a *from_attribute*.

`vermouth.dssp.dssp.`**`convert_dssp_to_martini`**`(`*sequence*`)`

Convert a sequence of secondary structure to martini secondary sequence.

Martini treats some secondary structures with less resolution than dssp. For instance, the different types of helices that dssp discriminates are seen the same by martini. Yet, different parts of the same helix are seen differently in martini.

In the Martini force field, the B and E secondary structures from DSSP are both treated as extended regions. All the DSSP helices are treated the same, but the different part of the helices (beginning, end, core of a short helix, core of a long helix) are treated differently.

After the conversion, the secondary structures are: * :F: Collagenous Fiber * :E: Extended structure (sheet) * :H: Helix structure * :1: Helix start (H-bond donor) * :2: Helix end (H-bond acceptor) * :3: Ambivalent helix type (short helices) * :T: Turn * :S: Bend * :C: Coil

Parameters

`sequence` (*`str`*) – A sequence of secondary structures as read from dssp. One letter per residue.

Returns

A sequence of secondary structures usable for martini. One letter per residue.

Return type

str

`vermouth.dssp.dssp.`**`read_dssp2`**`(`*lines*`)`

Read the secondary structure from a DSSP output.

Only the first column of the "STRUCTURE" block is read. See the documentation of the DSSP format for more details.

The secondary structures that can be read are:

H
: -helix

B
: residue in isolated -bridge

E
: extended strand, participates in ladder

G
: 3-helix (3-10 helix)

I
: 5 helix (-helix)

T
: hydrogen bonded turn

S
: bend

C
: loop or irregular

The "C" code for loops and random coil is translated from the gap used in the DSSP file for an improved readability.

Only the version 2 and 3 of DSSP is supported. If the format is not recognized as comming from that version of DSSP, then a `IOError` is raised.

Parameters
: **`lines`** – An iterable over the lines of the DSSP output. This can be *e.g.* a list of lines, or a file handler. The new line character is ignored.

Returns
: **secstructs** – The secondary structure assigned by DSSP as a list of one-letter secondary structure code.

Return type
: list[str]

Raises
: **`IOError`** – When a line could not be parsed, or if the version of DSSP is not supported.

`vermouth.dssp.dssp.`**`run_dssp`**(*system*, *executable='dssp'*, *savedir=None*, *defer_writing=True*)

Run DSSP on a system and return the assigned secondary structures.

Run DSSP using the path (or name in the research PATH) given by "executable". Return the secondary structure parsed from the output of the program.

In order to call DSSP, a PDB file is produced. Therefore, all the molecules in the system must contain the required attributes for such a file to be generated. Also, the atom names are assumed to be compatible with the 'charmm' force field for DSSP to recognize them. However, the molecules do not require the edges to be defined.

DSSP is assumed to be in version 2 or 3. The secondary structure codes are described in *`read_dssp2()`*.

Parameters

- **`system`** (`System`) –
- **`executable`** (*`str`*) – Where to find the DSSP executable.
- **`savefile`** (*`None or str or pathlib.Path`*) – If set to a path, the output of DSSP is written in this directory.

- **defer_writing** (*bool*) – Whether to use *write()* for writing data

Returns

The assigned secondary structures as a list of one-letter codes. The secondary structure sequences of all the molecules are combined in a single list without delimitation.

Return type

list[str]

Raises

- *DSSPError* – DSSP failed to run.
- IOError – The output of DSSP could not be parsed.

See also:

read_dssp2

Parse a DSSP output.

vermouth.dssp.dssp.**run_mdtraj**(*system*)

Compute DSSP secondary structure assignments for the system by using `mdtraj.compute_dssp`.

During processing, a PDB file is produced. Therefore, all the molecules in the system must contain the required attributes for such a file to be generated. Also, the atom names are assumed to be compatible with the 'charmm' force field for MDTraj to recognize them. However, the molecules do not require the edges to be defined.

Parameters

system (System) – The system to process

Returns

The assigned secondary structures as a list of one-letter codes. The secondary structure sequences of all the molecules are combined in a single list without delimitation.

Return type

list[str]

vermouth.dssp.dssp.**sequence_from_residues**(*molecule*, *attribute*, *default=None*)

Generates a sequence of *attribute*, one per residue in *molecule*.

Parameters

- **molecule** (vermouth.molecule.Molecule) – The molecule to process.
- **attribute** (*collections.abc.Hashable*) – The attribute of interest.
- **default** (*object*) – Yielded if the first node of a residue has no attribute *attribute*.

Yields

object – The value of *attribute* for every residue in *molecule*.

Module contents

vermouth.gmx package

Submodules

vermouth.gmx.gro module

Provides functionality to read and write GRO96 files.

`vermouth.gmx.gro.`**`read_gro`**`(`*file_name*, *exclude=('SOL',)*, *ignh=False*`)`

Parse a gro file to create a molecule.

Parameters

- **`filename`** (*`str`*) – The file to read.
- **`exclude`** (*`collections.abc.Container[str]`*) – Atoms that have one of these residue names will not be included.
- **`ignh`** (*`bool`*) – Whether hydrogen atoms should be ignored.

Returns

The parsed molecules. Will not contain edges.

Return type

vermouth.molecule.Molecule

`vermouth.gmx.gro.`**`write_gro`**`(`*system*, *file_name*, *precision=7*, *title='Martinized!'*, *box=(0, 0, 0)*, *defer_writing=True*`)`

Write *system* to *file_name*, which will be a GRO96 file.

Parameters

- **`system`** (`vermouth.system.System`) – The system to write.
- **`file_name`** (*`str`*) – The file to write to.
- **`precision`** (*`int`*) – The desired precision for coordinates and (optionally) velocities.
- **`title`** (*`str`*) – Title for the gro file.
- **`box`** (*`tuple[float]`*) – Box length and optionally angles.
- **`defer_writing`** (*`bool`*) – Whether to use *`write()`* for writing data

vermouth.gmx.itp module

Handle the ITP file format from Gromacs.

`vermouth.gmx.itp.`**`write_molecule_itp`**`(`*molecule*, *outfile*, *header=()*, *moltype=None*, *post_section_lines=None*, *pre_section_lines=None*`)`

Write a molecule in ITP format.

The molecule must have a *nrexcl* attribute. Each atom in the molecule must have at least the following keys: *atype*, *resid*, *resname*, *atomname*, and *charge_group*. Atoms can also have a *charge* and a *mass* key.

If the *moltype* argument is not provided, then the molecule must have a "moltype" meta attribute.

Parameters

- **molecule** (`Molecule`) – The molecule to write. See above for the minimal information the molecule must contain.
- **outfile** (*`io.TextIOBase`*) – The file in which to write.
- **header** (*`collections.abc.Iterable[str]`*) – List of lines to write as comment at the beginning of the file. The comment character and the new line should not be included as they will be added in the function.
- **moltype** (*`str, optional`*) – The molecule type. If set to *None* (default), the molecule type is read from the "moltype" key of *molecule.meta*.
- **post_section_lines** (*`dict[str, collections.abc.Iterable[str]], optional`*) – List of lines to write at the end of some sections of the file. The argument is passed as a dict with the keys being the name of the sections, and the values being the lists of lines. If the argument is set to *None*, the lines will be read from the "post_section_lines" key of *molecule.meta*.
- **pre_section_lines** (*`dict[str, collections.abc.Iterable[str]], optional`*) – List of lines to write at the beginning of some sections, just after the section header. The argument is formatted in the same way as *post_section_lines*. If the argument is set to *None*, the lines will be read from the "post_section_lines" key of *molecule.meta*.

Raises

`ValueError` – The molecule is missing required information.

vermouth.gmx.itp_read module

Read GROMACS .itp files.

`class vermouth.gmx.itp_read.`**`ITPDirector`**`(`*`force_field`*`)`

Bases: *`SectionLineParser`*

class for reading itp files.

`COMMENT_CHAR = ';'`

```
METH_DICT = {('macros',): (<function SectionLineParser._macros>, {}),
('moleculetype',): (<function ITPDirector._block>, {}), ('moleculetype',
'angle_restraints'): (<function ITPDirector._interactions>, {}), ('moleculetype',
'angle_restraints_z'): (<function ITPDirector._interactions>, {}), ('moleculetype',
'angles'): (<function ITPDirector._interactions>, {}), ('moleculetype', 'atoms'):
(<function ITPDirector._block_atoms>, {}), ('moleculetype', 'bonds'): (<function
ITPDirector._interactions>, {}), ('moleculetype', 'constraints'): (<function
ITPDirector._interactions>, {}), ('moleculetype', 'dihedral_restraints'):
(<function ITPDirector._interactions>, {}), ('moleculetype', 'dihedrals'):
(<function ITPDirector._interactions>, {}), ('moleculetype', 'distance_restraints'):
(<function ITPDirector._interactions>, {}), ('moleculetype', 'exclusions'):
(<function ITPDirector._interactions>, {}), ('moleculetype', 'impropers'):
(<function ITPDirector._interactions>, {}), ('moleculetype',
'orientation_restraints'): (<function ITPDirector._interactions>, {}),
('moleculetype', 'pairs'): (<function ITPDirector._interactions>, {}),
('moleculetype', 'pairs_nb'): (<function ITPDirector._interactions>, {}),
('moleculetype', 'position_restraints'): (<function ITPDirector._interactions>,
{}), ('moleculetype', 'settles'): (<function ITPDirector._interactions>, {}),
('moleculetype', 'virtual_sites1'): (<function ITPDirector._interactions>, {}),
('moleculetype', 'virtual_sites2'): (<function ITPDirector._interactions>, {}),
('moleculetype', 'virtual_sites3'): (<function ITPDirector._interactions>, {}),
('moleculetype', 'virtual_sites4'): (<function ITPDirector._interactions>, {}),
('moleculetype', 'virtual_sitesn'): (<function ITPDirector._interactions>, {})}
```

A dict of all known parser methods, mapping section names to the function to be called and the associated keyword arguments.

```
atom_idxs = {'angle_restraints': [slice(0, 4, None)], 'angle_restraints_z': [0,
1], 'angles': [0, 1, 2], 'bonds': [0, 1], 'constraints': [0, 1],
'dihedral_restraints': [slice(0, 4, None)], 'dihedrals': [0, 1, 2, 3],
'distance_restraints': [0, 1], 'exclusions': [slice(None, None, None)],
'orientation_restraints': [0, 1], 'pairs': [0, 1], 'pairs_nb': [0, 1],
'position_restraints': [0], 'settles': [0], 'virtual_sites1': [0],
'virtual_sites2': [0, 1, 2], 'virtual_sites3': [0, 1, 2, 3], 'virtual_sites4':
[slice(0, 5, None)], 'virtual_sitesn': [0, slice(2, None, None)]}
```

`dispatch`(*line*)

Looks at *line* to see what kind of line it is, and returns either `parse_header()` if *line* is a section header or `vermouth.parser_utils.SectionLineParser.parse_section()` otherwise. Calls `vermouth.parser_utils.SectionLineParser.is_section_header()` to see whether *line* is a section header or not.

Parameters
`line` (*str*) –

Returns
The method that should be used to parse *line*.

Return type
collections.abc.Callable

`finalize`(*lineno=0*)

Called at the end of the file and checks that all pragmas are closed before calling the parent method.

`finalize_section`(*previous_section*, *ended_section*)

Called once a section is finished. It appends the current_links list to the links and update the block dictionary with current_block. Thereby it finishes reading a given section.

Parameters

- **previous_section** (*list[str]*) – The last parsed section.
- **ended_section** (*list[str]*) – The sections that have been ended.

`static is_pragma(`*line*`)`

Parameters
line (*str*) – A line of text.

Returns
`True` iff *line* is a def statement.

Return type
bool

`parse_header(`*line*, *lineno=0*`)`

Parses a section header with line number *lineno*. Sets *vermouth.parser_utils.SectionLineParser.section* when applicable. Does not check whether *line* is a valid section header.

Parameters

- **line** (*str*) –
- **lineno** (*str*) –

Returns
The result of calling *finalize_section()*, which is called if a section ends.

Return type
object

Raises
KeyError – If the section header is unknown.

`parse_pragma(`*line*, *lineno=0*`)`

Parses the beginning and end of define sections with line number *lineno*. Sets attr current_meta when applicable. Does check if ifdefs overlap.

Parameters

- **line** (*str*) –
- **lineno** (*str*) –

Returns
The result of calling *finalize_section()*, which is called if a section ends.

Return type
object

Raises
IOError – If the def sections are missformatted

`vermouth.gmx.itp_read.read_itp(`*lines*, *force_field*`)`

Parses *lines* of itp format and adds the molecule as a block to *force_field*.

Parameters

- **lines** (*list*) – list of lines of an itp file
- **force_field** (*vermouth.forcefield.ForceField*) –

vermouth.gmx.rtp module

Handle the RTP format from Gromacs.

vermouth.gmx.rtp.**read_rtp**(*lines*, *force_field*)

Read blocks and links from a Gromacs RTP file to populate a force field

Parameters

- **lines** (*collections.abc.Iterator*) – An iterator over the lines of a RTP file (e.g. a file handle, or a list of string).
- **force_field** (vermouth.forcefield.ForceField) – The force field to populate in place.

Raises

IOError – Something in the file could not be parsed.

vermouth.gmx.topology module

I/O of topology parameters that are not molecules.

class vermouth.gmx.topology.**Atomtype**(*molecule*, *node*, *sigma*, *epsilon*, *meta*)

Bases: tuple

Create new instance of Atomtype(molecule, node, sigma, epsilon, meta)

epsilon

Alias for field number 3

meta

Alias for field number 4

molecule

Alias for field number 0

node

Alias for field number 1

sigma

Alias for field number 2

class vermouth.gmx.topology.**NonbondParam**(*atoms*, *sigma*, *epsilon*, *meta*)

Bases: tuple

Create new instance of NonbondParam(atoms, sigma, epsilon, meta)

atoms

Alias for field number 0

epsilon

Alias for field number 2

meta

Alias for field number 3

sigma

Alias for field number 1

`vermouth.gmx.topology.`**`sigma_epsilon_to_C6_C12`**(*sigma*, *epsilon*)

Convert the LJ potential from sigma epsilon form to C6 C12 form.

`vermouth.gmx.topology.`**`write_atomtypes`**(*system*, *itp_path*, *C6C12=False*)

Writes the [atomtypes] directive to file. All atomtypes are defined in system.gmx_topology_params. Masses and further information are taken from the molecule directly.

`vermouth.gmx.topology.`**`write_gmx_topology`**(*system*, *top_path*, *itp_paths={'atomtypes': 'extra_atomtypes.itp', 'nonbond_params': 'extra_nbparams.itp'}*, *C6C12=False*, *defines=()*, *header=()*)

Writes a Gromacs .top file for the specified system. Gromacs topology files are defined by directives for example *[atomtypes]*. However, Gromacs supports writing parts of the topology to so called .itp files which can be inculded into a toplevel topology file with the extension .top using #include statements. The topology writer will generate such a toplevel topology file where the different directives are written to seperate .itp files and included into the toplevel file.

Parameters

- **`system`** (`vermouth.system.System`) –
- **`top_path`** (*`pathlib.Path`*) – path for topology file
- **`itp_paths`** (*`dict[str, pathlib.Path]`*) – list of paths for writing the topology parameters like atomtypes with the key being the name of the directive.
- **`C6C12`** (*`bool`*) – write non-bonded interaction parameters using LJ C6C12 form
- **`defines`** (*`tuple(str)`*) – define statments to include in the topology
- **`header`** (*`tuple(str)`*) – any comment lines to include at the beginning

`vermouth.gmx.topology.`**`write_nonbond_params`**(*system*, *itp_path*, *C6C12=False*)

Writes the [nonbond_params] directive to file. All atomtypes are defined in system.gmx_topology_params. Masses and further information are taken from the molecule directly.

Module contents

Provides functionality to read and write Gromacs specific files.

vermouth.pdb package

Submodules

vermouth.pdb.pdb module

Provides functions for reading and writing PDB files.

`class` `vermouth.pdb.pdb.`**`PDBParser`**(*exclude=('SOL',)*, *ignh=False*, *modelidx=1*)

Bases: *`LineParser`*

Parser for PDB files

`active_molecule`

The molecule/model currently being read.

Type

vermouth.molecule.Molecule

`molecules`

All complete molecules read so far.

Type
list[*vermouth.molecule.Molecule*]

`modelidx`

Which model to take.

Type
int

Parameters

- **exclude** (*collections.abc.Container[str]*) – Container of residue names. Any atom that has a residue name that is in *exclude* will be skipped.
- **ignh** (*bool*) – Whether all hydrogen atoms should be skipped
- **modelidx** (*int*) – Which model to take.

`static anisou`(*line*, *lineno=0*)

Does nothing.

`atom`(*line*, *lineno=0*)

Parse an ATOM or HETATM record.

Parameters

- **line** (*str*) – The line to parse. We do not check whether it starts with either “ATOM “ or “HETATM”.
- **lineno** (*int*) – The line number (not used).

`static author`(*line*, *lineno=0*)

Does nothing.

`static caveat`(*line*, *lineno=0*)

Does nothing.

`static cispep`(*line*, *lineno=0*)

Does nothing.

`static compnd`(*line*, *lineno=0*)

Does nothing.

`conect`(*line*, *lineno=0*)

Parse a CONECT record. The line is stored for later processing.

Parameters

- **line** (*str*) – The line to parse. Should start with CONECT, but this is not checked
- **lineno** (*int*) – The line number (not used).

`cryst1`(*line*, *lineno=0*)

Parse the CRYST1 record. Crystal structure information are stored with the parser object and may be extracted later.

`static dbref`(*line*, *lineno=0*)

Does nothing.

static dbref1(*line*, *lineno=0*)

Does nothing.

static dbref2(*line*, *lineno=0*)

Does nothing.

dispatch(*line*)

Returns the appropriate method for parsing *line*. This is determined based on the first 6 characters of *line*.

Parameters
`line` (*str*) –

Returns
The method to call with the line, and the line number.

Return type
collections.abc.Callable[str, int]

do_conect()

Apply connections to molecule based on CONECT records read from PDB file

end(*line=''*, *lineno=0*)

Finish parsing the molecule. *active_molecule* will be appended to *molecules*, and a new *active_molecule* will be made.

endmdl(*line=''*, *lineno=0*)

Finish parsing the molecule. *active_molecule* will be appended to *molecules*, and a new *active_molecule* will be made.

static expdta(*line*, *lineno=0*)

Does nothing.

finalize(*lineno=0*)

Finish parsing the file. Process all CONECT records found, and returns a list of molecules.

Parameters
`lineno` (*int*) – The line number (not used).

Returns
All molecules parsed from this file.

Return type
list[*vermouth.molecule.Molecule*]

static formul(*line*, *lineno=0*)

Does nothing.

static header(*line*, *lineno=0*)

Does nothing.

static helix(*line*, *lineno=0*)

Does nothing.

static het(*line*, *lineno=0*)

Does nothing.

hetatm(*line*, *lineno=0*)

Parse an ATOM or HETATM record.

Parameters

- **line** (*str*) – The line to parse. We do not check whether it starts with either "ATOM " or "HETATM".
- **lineno** (*int*) – The line number (not used).

`static hetnam`(*line*, *lineno=0*)

Does nothing.

`static hetsyn`(*line*, *lineno=0*)

Does nothing.

`static jrnl`(*line*, *lineno=0*)

Does nothing.

`static keywds`(*line*, *lineno=0*)

Does nothing.

`static link`(*line*, *lineno=0*)

Does nothing.

`static master`(*line*, *lineno=0*)

Does nothing.

`static mdltyp`(*line*, *lineno=0*)

Does nothing.

`model`(*line*, *lineno=0*)

Parse a MODEL record. If the model is not the same as `modelidx`, this model will not be parsed.

Parameters

- **line** (*str*) – The line to parse. Should start with "MODEL ", but this is not checked.
- **lineno** (*int*) – The line number (not used).

`static modres`(*line*, *lineno=0*)

Does nothing.

`static mtrix1`(*line*, *lineno=0*)

Does nothing.

`static mtrix2`(*line*, *lineno=0*)

Does nothing.

`static mtrix3`(*line*, *lineno=0*)

Does nothing.

`static nummdl`(*line*, *lineno=0*)

Does nothing.

`static obslte`(*line*, *lineno=0*)

Does nothing.

`static origx1`(*line*, *lineno=0*)

Does nothing.

`static origx2`(*line*, *lineno=0*)

Does nothing.

static **origx3**(*line*, *lineno=0*)
Does nothing.

parse(*file_handle*)

static **remark**(*line*, *lineno=0*)
Does nothing.

static **revdat**(*line*, *lineno=0*)
Does nothing.

static **scale1**(*line*, *lineno=0*)
Does nothing.

static **scale2**(*line*, *lineno=0*)
Does nothing.

static **scale3**(*line*, *lineno=0*)
Does nothing.

static **seqadv**(*line*, *lineno=0*)
Does nothing.

static **seqres**(*line*, *lineno=0*)
Does nothing.

static **sheet**(*line*, *lineno=0*)
Does nothing.

static **site**(*line*, *lineno=0*)
Does nothing.

static **source**(*line*, *lineno=0*)
Does nothing.

static **splt**(*line*, *lineno=0*)
Does nothing.

static **sprsde**(*line*, *lineno=0*)
Does nothing.

static **ssbond**(*line*, *lineno=0*)
Does nothing.

ter(*line=''*, *lineno=0*)
Finish parsing the molecule. `active_molecule` will be appended to `molecules`, and a new `active_molecule` will be made.

static **title**(*line*, *lineno=0*)
Does nothing.

vermouth.pdb.pdb.**get_not_none**(*node*, *attr*, *default*)
Returns `node[attr]`. If it doesn't exists or is `None`, return *default*.

Parameters

- **node** (`collections.abc.Mapping`) –
- **attr** (`collections.abc.Hashable`) –

- **default** – The value to return if `node[attr]` is either `None`, or does not exist.

Returns
The value of `node[attr]` if it exists and is not `None`, else *default*.

Return type
object

`vermouth.pdb.pdb.`**`read_pdb`**(*file_name*, *exclude=('SOL',)*, *ignh=False*, *modelidx=1*)

Parse a PDB file to create a molecule.

Parameters

- **filename** (*str*) – The file to read.
- **exclude** (*collections.abc.Container[str]*) – Atoms that have one of these residue names will not be included.
- **ignh** (*bool*) – Whether hydrogen atoms should be ignored.
- **model** (*int*) – If the PDB file contains multiple models, which one to select.

Returns
The parsed molecules. Will only contain edges if the PDB file has CONECT records. Either way, the molecules might be disconnected. Entries separated by TER, ENDMDL, and END records will result in separate molecules.

Return type
list[*vermouth.molecule.Molecule*]

`vermouth.pdb.pdb.`**`write_pdb`**(*system*, *path*, *conect=True*, *omit_charges=True*, *nan_missing_pos=False*, *defer_writing=True*)

Writes *system* to *path* as a PDB formatted string.

Parameters

- **system** (`vermouth.system.System`) – The system to write.
- **path** (*str*) – The file to write to.
- **conect** (*bool*) – Whether to write CONECT records for the edges.
- **omit_charges** (*bool*) – Whether charges should be omitted. This is usually a good idea since the PDB format can only deal with integer charges.
- **nan_missing_pos** (*bool*) – Whether the writing should fail if an atom does not have a position. When set to *True*, atoms without coordinates will be written with ‘nan’ as coordinates; this will cause the output file to be *invalid* for most uses. for most use.
- **defer_writing** (*bool*) – Whether to use *DeferredFileWriter* for writing data

See also:

:func:write_pdb_string

`vermouth.pdb.pdb.`**`write_pdb_string`**(*system*, *conect=True*, *omit_charges=True*, *nan_missing_pos=False*)

Describes *system* as a PDB formatted string. Will create CONECT records from the edges in the molecules in *system* iff *conect* is True.

Parameters

- **system** (`vermouth.system.System`) – The system to write.
- **conect** (*bool*) – Whether to write CONECT records for the edges.

- **omit_charges** (*bool*) – Whether charges should be omitted. This is usually a good idea since the PDB format can only deal with integer charges.
- **nan_missing_pos** (*bool*) – Wether the writing should fail if an atom does not have a position. When set to *True*, atoms without coordinates will be written with 'nan' as coordinates; this will cause the output file to be *invalid* for most uses.

Returns
The system as PDB formatted string.

Return type
str

Module contents

Provides functionality to read and write PDB files.

vermouth.processors package

Submodules

vermouth.processors.add_molecule_edges module

Processor adding edges between molecules.

`class vermouth.processors.add_molecule_edges.`**`AddMoleculeEdgesAtDistance`**`(`*threshold, templates_from, templates_to, attribute='position', min_edges=0*`)`

Bases: `Processor`

Processor that adds edges within and between molecules.

The processor adds edges between atoms, within or between molecules, when the atoms are part of the selections provided for each end of the edges, and the atoms are closer than a given threshold.

Parameters

- **threshold** (*float*) – Distance threshold in nanometers under which to create an edge.
- **templates_from** (*list[dict]*) – List of node templates to select the atoms at one end of the edges.
- **templates_to** (*list[dict]*) – List of node template to select the atoms at the other end of the edges.
- **attribute** (*str*) – Name of the attribute under which are stores the coordinates.

See also:

`vermouth.molecule.attributes_match`

`run_system``(`*system*`)`
Run the processor on the system.

class vermouth.processors.add_molecule_edges.**MergeNucleicStrands**(*threshold=0.3*, *templates_donnors=({'atomname': <Choice at 7fa699273a70 value=['C2', 'N6']>, 'resname': <Choice at 7fa699273a10 value=['DA', 'DA3', 'DA5']>}, {'atomname': <Choice at 7fa69909c0b0 value=['N1', 'N2']>, 'resname': <Choice at 7fa699273aa0 value=['DG', 'DG3', 'DG5']>}, {'atomname': 'N4', 'resname': <Choice at 7fa69909c0e0 value=['DC', 'DC3', 'DC5']>}, {'atomname': 'N3', 'resname': <Choice at 7fa69909c110 value=['DT', 'DT3', 'DT5']>})*, *templates_acceptors=({'atomname': 'N1', 'resname': <Choice at 7fa69909c140 value=['DA', 'DA3', 'DA5']>}, {'atomname': 'O6', 'resname': <Choice at 7fa69909c170 value=['DG', 'DG3', 'DG5']>}, {'atomname': <Choice at 7fa69909c1d0 value=['N3', 'O2']>, 'resname': <Choice at 7fa69909c1a0 value=['DC', 'DC3', 'DC5']>}, {'atomname': <Choice at 7fa69909c230 value=['O2', 'O4']>, 'resname': <Choice at 7fa69909c200 value=['DT', 'DT3', 'DT5']>})*, *attribute='position'*)

Bases: *AddMoleculeEdgesAtDistance*

Add edges between complementary nucleic acid strands.

By default, the edges are added in place of the hydrogen bonds between complementary bases.

Parameters

- **threshold** (*float*) – Distance threshold in nanometers under which to create an edge.
- **templates_donnors** (*list[dict]*) – List of templates describing hydrogen donnors.
- **templates_acceptors** (*list[dict]*) – List of templates describing hydrogen acceptors.
- **attribute** (*str*) – Name of the attribute under which are store the node coordinates.

vermouth.processors.annotate_mut_mod module

Provides a processor that annotates a molecule with desired mutations and modifications.

class vermouth.processors.annotate_mut_mod.**AnnotateMutMod**(*modifications=None*, *mutations=None*)

Bases: *Processor*

Annotates residues to have the required 'modification' and 'mutation' attributes on all nodes.

modifications

Type
list[tuple[dict, str]]

mutations

Type
list[tuple[dict, str]]

See also:

annotate_modifications()

run_molecule(*molecule*)

vermouth.processors.annotate_mut_mod.**annotate_modifications**(*molecule*, *modifications*, *mutations*)

Annotate nodes in molecule with the desired modifications and mutations

Parameters

- **molecule** (*networkx.Graph*) –
- **modifications** (*list[tuple[dict, str]]*) – The modifications to apply. The first element is a dictionary contain the attributes a residue has to fulfill. It can contain the elements 'chain', 'resname' and 'resid'. The second element is the modification that should be applied.
- **mutations** (*list[tuple[dict, str]]*) – The mutations to apply. The first element is a dictionary contain the attributes a residue has to fulfill. It can contain the elements 'chain', 'resname' and 'resid'. The second element is the mutation that should be applied.

Raises
NameError – When a modification is not recognized.

vermouth.processors.annotate_mut_mod.**parse_residue_spec**(*resspec*)

Parse a residue specification: [<chain>-][<resname>][[#]<resid>] where resid is /[0-9]+/. If resname ends in a number and a resid is also specified, the # separator is required. Returns a dictionary with keys 'chain', 'resname', and 'resid' for the fields that are specified. Resid will be an int.

Parameters
resspec (*str*) –

Return type
dict

vermouth.processors.annotate_mut_mod.**residue_matches**(*resspec*, *residue_graph*, *res_idx*)

Returns True iff resspec describes residue_graph.nodes[res_idx]. The 'resname's nter and cter match the residues with a degree of 1 and with the lowest and highest residue numbers respectively.

Parameters

- **resspec** (*dict*) – Attributes that must be present in the residue node. 'resname' is treated specially as described above.

- **residue_graph** (*networkx.Graph*) – A graph with one node per residue.
- **res_idx** (*collections.abc.Hashable*) – A node index in residue_graph.

Returns
Whether resspec describes the node res_idx in residue_graph.

Return type
bool

vermouth.processors.apply_posres module

class vermouth.processors.apply_posres.**ApplyPosres**(*selector*, *force_constant*, *functype=1*, *ifdef='POSRES'*)

Bases: *Processor*

run_molecule(*molecule*)

vermouth.processors.apply_posres.**apply_posres**(*molecule*, *selector*, *force_constant*, *functype=1*, *ifdef='POSRES'*)

vermouth.processors.apply_rubber_band module

Provides a processor that adds a rubber band elastic network.

class vermouth.processors.apply_rubber_band.**ApplyRubberBand**(*lower_bound*, *upper_bound*, *decay_factor*, *decay_power*, *base_constant*, *minimum_force*, *res_min_dist=None*, *bond_type=None*, *selector=<function select_backbone>*, *bond_type_variable='elastic_network_bond_type'*, *res_min_dist_variable='elastic_network_res_min_dist'*, *domain_criterion=<function always_true>*)

Bases: *Processor*

Add an elastic network to a system between particles fulfilling the following criteria:

- They must be close enough together in space
- They must be separated far enough in graph space
- They must be either in the same chain/molecule/system
- They must be selected by *selector*
- The resulting elastic bond must be stiff enough

selector
Selection function.

Type
collections.abc.Callable

`lower_bound`

The minimum length for a bond to be added, expressed in nanometers.

Type
float

`upper_bound`

The maximum length for a bond to be added, expressed in nanometers.

Type
float

`decay_factor`

Parameter for the decay function.

Type
float

`decay_power`

Parameter for the decay function.

Type
float

`base_constant`

The base force constant for the bonds in $kJ.mol^{-1}.nm^{-2}$. If 'decay_factor' or 'decay_power' is set to 0, then it will be the used force constant.

Type
float

`minimum_force`

Minimum force constant in $kJ.mol^{-1}.nm^{-2}$ under which bonds are not kept.

Type
float

`bond_type`

Gromacs bond function type to apply to the elastic network bonds.

Type
int or None

`bond_type_variable`

If bond_type is not given, it will be taken from the force field, using this variable name.

Type
str

`domain_criterion`

Function to establish if two atoms are part of the same domain. Elastic bonds are only added within a domain. By default, all the atoms in the molecule are considered part of the same domain. The function expects a graph (e.g. a *`Molecule`*) and two atom node keys as argument and returns `True` if the two atoms are part of the same domain; returns `False` otherwise.

Type
collections.abc.Callable

`res_min_dist`

Minimum separation between two atoms for a bond to be kept. Bonds are kept is the separation is greater or equal to the value given.

Type
int or None

`res_min_dist_variable`

If res_min_dist is not given it will be taken from the force field using this variable name.

Type
str

See also:

apply_rubber_band()

`run_molecule`(*molecule*)

`vermouth.processors.apply_rubber_band.always_true`(**args*, ***kwargs*)

Returns `True` whatever the arguments are.

`vermouth.processors.apply_rubber_band.apply_rubber_band`(*molecule*, *selector*, *lower_bound*, *upper_bound*, *decay_factor*, *decay_power*, *base_constant*, *minimum_force*, *bond_type*, *domain_criterion*, *res_min_dist*)

Adds a rubber band elastic network to a molecule.

The elastic network is applied as bounds between the atoms selected by the function declared with the 'selector' argument. The equilibrium length for the bonds is measured from the coordinates in the molecule, the force constant is computed from the base force constant and an optional decay function.

The decay function for the force constant is defined as:

$$\exp^{-r(d-s)^p}$$

where r is the decay rate given by the 'decay_factor' argument, p is the decay power given by 'decay_power', s is a shift given by 'lower_bound', and d is the distance between the two atoms in the molecule. If the rate or the power are set to 0, then the decay function does not modify the force constant.

The 'selector' argument takes a callback that accepts a atom dictionary and returns `True` if the atom match the conditions to be kept.

Only nodes that are in the same domain can be connected by the elastic network. The 'domain_criterion' argument accepts a callback that determines if two nodes are in the same domain. That callback accepts a graph and two node keys as argument and returns whether or not the nodes are in the same domain as a boolean.

Parameters

- **molecule** (`vermouth.molecule.Molecule`) – The molecule to which apply the elastic network. The molecule is modified in-place.
- **selector** (*collections.abc.Callable*) – Selection function.
- **lower_bound** (*float*) – The minimum length for a bond to be added, expressed in nanometers.
- **upper_bound** (*float*) – The maximum length for a bond to be added, expressed in nanometers.
- **decay_factor** (*float*) – Parameter for the decay function.
- **decay_power** (*float*) – Parameter for the decay function.
- **base_constant** (*float*) – The base force constant for the bonds in $kJ.mol^{-1}.nm^{-2}$. If 'decay_factor' or 'decay_power' is set to 0, then it will be the used force constant.

- **minimum_force** (*float*) – Minimum force constant in $kJ.mol^{-1}.nm^{-2}$ under which bonds are not kept.
- **bond_type** (*int*) – Gromacs bond function type to apply to the elastic network bonds.
- **domain_criterion** (*collections.abc.Callable*) – Function to establish if two atoms are part of the same domain. Elastic bonds are only added within a domain. By default, all the atoms in the molecule are considered part of the same domain. The function expects a graph (e.g. a *Molecule*) and two atom node keys as argument and returns `True` if the two atoms are part of the same domain; returns `False` otherwise.
- **res_min_dist** (*int*) – Minimum separation between two atoms for a bond to be kept. Bonds are kept is the separation is greater or equal to the value given.

`vermouth.processors.apply_rubber_band.`**`are_connected`**(*graph*, *left*, *right*, *separation*)

`True` if the nodes are at most 'separation' nodes away.

Parameters

- **graph** (*networkx.Graph*) – The graph/molecule to work on.
- **left** – One node key from the graph.
- **right** – One node key from the graph.
- **separation** (*int*) – The maximum number of nodes in the shortest path between two nodes of interest for these two nodes to be considered connected. Must be >= 0.

Return type

bool

`vermouth.processors.apply_rubber_band.`**`build_connectivity_matrix`**(*graph*, *separation*, *node_to_idx*, *selected_nodes*)

Build a connectivity matrix based on the separation between nodes in a graph.

The connectivity matrix is a symmetric boolean matrix where cells contain `True` if the corresponding atoms are connected in the graph and separated by less or as much nodes as the given 'separation' argument.

In the following examples, the separation between A and B is 0, 1, and 2. respectively:

` A - B A - X - B A - X - X - B `

Note that building the connectivity matrix with a separation of 0 is the same as building the adjacency matrix.

Parameters

- **graph** (*networkx.Graph*) – The graph/molecule to work on.
- **separation** (*int*) – The maximum number of nodes in the shortest path between two nodes of interest for these two nodes to be considered connected. Must be >= 0.
- **selected_nodes** (*collections.abc.Collection*) – A list of nodes to work on.

Returns

A boolean matrix.

Return type

numpy.ndarray

`vermouth.processors.apply_rubber_band.`**`build_pair_matrix`**(*graph*, *criterion*, *idx_to_node*, *selected_nodes*)

Build a boolean matrix telling if a pair of nodes fulfil a criterion.

Parameters

- **graph** (*networkx.Graph*) – The graph/molecule to work on.
- **criterion** (*collections.abc.Callable*) – A function that determines if a pair of nodes fulfill the criterion. It takes a graph and two node keys as arguments and returns a boolean.
- **selected_nodes** (*collections.abc.Collection*) – A list of nodes to work on.

Returns
A boolean matrix.

Return type
numpy.ndarray

`vermouth.processors.apply_rubber_band.`**`compute_decay`**`(`*distance*, *shift*, *rate*, *power*`)`

Compute the decay function of the force constant as function to the distance.

The decay function for the force constant is defined as:

$$\exp^{-r(d-s)^p}$$

where r is the decay rate given by the 'rate' argument, p is the decay power given by 'power', s is a shift given by 'shift', and d is the distance between the two atoms given in 'distance'. If the rate or the power are set to 0, then the decay function does not modify the force constant.

The 'distance' argument can be a scalar or a numpy array. If it is an array, then the returned value is an array of decay factors with the same shape as the input.

`vermouth.processors.apply_rubber_band.`**`compute_force_constants`**`(`*distance_matrix*, *lower_bound*, *upper_bound*, *decay_factor*, *decay_power*, *base_constant*, *minimum_force*`)`

Compute the force constant of an elastic network bond.

The force constant can be modified with a decay function, and it can be bounded with a minimum threshold, or a distance upper and lower bonds.

If decay_factor = decay_power = 0 all forces applied are = base_constant

Forces applied to distances above upper_bound are removed. Forces below minimum_force are removed.

If decay_factor or decay_power != 0, forces below lower_bound are greater than base_constant, in which case they are set back to = base_constant

`vermouth.processors.apply_rubber_band.`**`make_same_region_criterion`**`(`*regions*`)`

Returns `True` is the nodes are part of the same region.

Nodes are considered part of the same region if their value under the "resid" attribute are within the same residue range. By default the resids of the input file are used (i.e. "_old_resid" attribute).

Parameters

- **graph** (*networkx.Graph*) – A graph the nodes are part of.
- **left** – A node key in 'graph'.
- **right** – A node key in 'graph'.
- **regions** – [(resid_start_1,resid_end_1),(resid_start_2,resid_end_2),...] resid_start and resid_end are included)

Returns
`True` if the nodes are part of the same region.

Return type
bool

`vermouth.processors.apply_rubber_band.`**`same_chain`**(*graph*, *left*, *right*)

Returns `True` is the nodes are part of the same chain.

Nodes are considered part of the same chain if they both have the same value under the "chain" attribute, or if neither of the 2 nodes have that attribute.

Parameters

- **`graph`** (*`networkx.Graph`*) – A graph the nodes are part of.
- **`left`** – A node key in 'graph'.
- **`right`** – A node key in 'graph'.

Returns
`True` if the nodes are part of the same chain.

Return type
bool

`vermouth.processors.apply_rubber_band.`**`self_distance_matrix`**(*coordinates*)

Compute a distance matrix between points in a selection.

Notes

This function does **not** account for periodic boundary conditions.

Parameters
`coordinates` (*`numpy.ndarray`*) – Coordinates of the points in the selection. Each row must correspond to a point and each column to a dimension.

Return type
numpy.ndarray

vermouth.processors.attach_mass module

Provides a processor that assigns a *mass* attribute to every node in a molecule based on it's element.

`class` `vermouth.processors.attach_mass.`**`AttachMass`**(*attribute='mass'*)

Bases: *`Processor`*

`run_molecule`(*molecule*)

`vermouth.processors.attach_mass.`**`attach_mass`**(*molecule*, *attribute='mass'*)

For every atom in *molecule* look up it's element in `ATOM_MASSES`, and assign that value to *attribute*.

Parameters

- **`molecule`** (*`networkx.Graph`*) – The molecule to process. Is modified in-place.
- **`attribute`** (*`collections.abc.Hashable`*) – The attribute the mass is assigned to.

vermouth.processors.average_beads module

Provides a processor that generates positions for nodes based on the weighted average of the positions of the atoms they are constructed from.

`class vermouth.processors.average_beads.`**`DoAverageBead`**`(`*ignore_missing_graphs=False*, *weight=None*`)`

Bases: *Processor*

`run_molecule``(`*molecule*`)`

`vermouth.processors.average_beads.`**`do_average_bead`**`(`*molecule*, *ignore_missing_graphs=False*, *weight=None*`)`

Set the position of the particles to the mean of the underlying atoms.

This requires the atoms to have a 'graph' attributes. By default, a `ValueError` is raised if any atom in the molecule is missing that 'graph' attribute. This behavior can be changed by setting the 'ignore_missing_graphs' argument to *True*, then the average positions are computed, but the atoms without a 'graph' attribute are skipped.

The average is weighted using the 'mapping_weights' atom attribute. If the 'mapping_weights' attribute is set, it has to be a dictionary with the atomname from the underlying graph as keys, and the weights as values. Atoms without a weight set use a default weight of 1.

The average can also be weighted using an arbitrary node attribute by giving the attribute name with the *weight* keyword argument. This can be used to get the center of mass for instance; assuming the mass of the underlying atoms is stored under the "mass" attribute, setting *weight* to "mass" will place the bead at the center of mass. By default, *weight* is set to *None* and the center of geometry is used.

The atoms in the underlying graph must have a position. If they do not, they are ignored from the average.

Parameters

- **`molecule`** (`vermouth.molecule.Molecule`) – The molecule to update. The attribute *position* of the particles is updated on place. The nodes of the molecule must have an attribute *graph* that contains the subgraph of the initial molecule.
- **`ignore_missing_graphs`** (*`bool`*) – If *True*, skip the atoms that do not have a *graph* attribute; else fail if not all the atoms in the molecule have a *graph* attribute.
- **`weight`** (*`collections.abc.Hashable`*) – The name of the attribute used to weight the position of the node. The attribute is read from the underlying atoms.

vermouth.processors.canonicalize_modifications module

Provides a Processor that identifies unexpected atoms such as PTMs and protonations, and canonicalizes their attributes based on modifications known in the forcefield.

`class vermouth.processors.canonicalize_modifications.`**`CanonicalizeModifications`**

Bases: *Processor*

Identifies all modifications in a molecule and corrects their atom names.

See also:

fix_ptm()

`run_molecule``(`*molecule*`)`

`vermouth.processors.canonicalize_modifications.`**`allowed_ptms`**`(`*residue*, *res_ptms*, *known_ptms*`)`

Finds all PTMs in `known_ptms` which might be relevant for `residue`.

Parameters

- **`residue`** (*`networkx.Graph`*) –
- **`res_ptms`** (*`list[tuple[set, set]]`*) – As returned by `find_PTM_atoms`. Currently not used.
- **`known_ptms`** (*`collections.abc.Mapping[str, networkx.Graph]`*) –

Yields

tuple[networkx.Graph, networkx.isomorphism.GraphMatcher] – All graphs in known_ptms which are subgraphs of residue.

`vermouth.processors.canonicalize_modifications.`**`find_ptm_atoms`**`(`*molecule*`)`

Finds all atoms in molecule that have the node attribute `PTM_atom` set to a value that evaluates to `True`. `molecule` will be traversed starting at these atoms until all marked atoms are visited such that they are identified per “branch”, and for every branch the anchor node is known. The anchor node is the node(s) which are not PTM atoms and share an edge with the traversed branch.

Parameters

`molecule` (*`networkx.Graph`*) –

Returns

`[({ptm atom indices}, {anchor indices}), ...]`. Ptm atom indices are connected, and are connected to the rest of molecule via anchor indices.

Return type

list[tuple[set, set]]

`vermouth.processors.canonicalize_modifications.`**`fix_ptm`**`(`*molecule*`)`

Canonizes all PTM atoms in molecule, and labels the relevant residues with which PTMs were recognized. Modifies `molecule` such that atom names of PTM atoms are corrected, and the relevant residues have been labeled with which PTMs were recognized.

Parameters

`molecule` (*`networkx.Graph`*) – Must not have missing atoms, and atom names must be correct. Atoms which could not be recognized must be labeled with the attribute PTM_atom=True.

`vermouth.processors.canonicalize_modifications.`**`identify_ptms`**`(`*residue*, *residue_ptms*, *known_ptms*`)`

Identifies all PTMs in `known_PTMs` necessary to describe all PTM atoms in `residue_ptms`. Will take PTMs such that all PTM atoms in `residue` will be covered by applying PTMs from `known_PTMs` in order. Nodes in `residue` must have correct `atomname` attributes, and may not be missing. In addition, every PTM in must be anchored to a non-PTM atom.

Parameters

- **`residue`** (*`networkx.Graph`*) – The residues involved with these PTMs. Need not be connected.
- **`residue_ptms`** (*`list[tuple[set, set]]`*) – As returned by `find_PTM_atoms`, but only those relevant for `residue`.
- **`known_PTMs`** (*`collections.abc.Sequence[tuple[networkx.Graph, networkx.isomorphism.GraphMatcher]]`*) – The nodes in the graph must have the *PTM_atom* attribute (True or False). It should be True for atoms that are not part of the PTM itself, but describe where it is attached to the molecule. In addition, its nodes must have the *atomname* attribute, which will be used to recognize where the PTM is anchored, or to correct the atom names. Lastly, the nodes may have a *replace* attribute, which is a dictionary

of `{attribute_name: new_value}` pairs. The special case here is if attribute_name is `'atomname'` and new_value is `None`: in this case the node will be removed. Lastly, the graph (not its nodes) needs a 'name' attribute.

Returns
All PTMs from `known_PTMs` needed to describe the PTM atoms in `residue` along with a `dict` of node correspondences. The order of `known_PTMs` is preserved.

Return type
list[tuple[networkx.Graph, dict]]

Raises
`KeyError` – Not all PTM atoms in `residue` can be covered with `known_PTMs`.

`vermouth.processors.canonicalize_modifications.`**`ptm_node_matcher`**`(`*node1*, *node2*`)`

Returns True iff node1 and node2 should be considered equal. This means they are both either marked as PTM_atom, or not. If they both are PTM atoms, the elements need to match, and otherwise, the atom names must match.

vermouth.processors.do_links module

`class vermouth.processors.do_links.`**`DoLinks`**

Bases: *`Processor`*

Apply Links, taken from a molecule's force field, to the molecule.

`run_molecule``(`*molecule*`)`

`vermouth.processors.do_links.`**`match_link`**`(`*molecule*, *link*`)`

`vermouth.processors.do_links.`**`match_order`**`(`*order1*, *resid1*, *order2*, *resid2*`)`

Check if two residues match the order constraints.

The order can be:

an integer
It is then the expected distance in resid with a reference residue.

a series of >
This indicates that the residue must have a larger resid than a reference residue. Multiple atoms with the same number of > are expected to be part of the same residue. The more > are in the serie, the further away the residue is expected to be from the reference, so a residue with >> is expected to have a greater resid than a residue with >.

a series of <
Same as a series of >, but for smaller resid.

a series of *
This indicates a different residue than the reference, but without a specified order. As for the > or the <, atoms with the same number of * are expected to be part of the same residue.

The comparison matrix can be sumerized as follow, with 0 being the reference residue, n being an integer. In the matrix, a ? means that the result depends on the comparison of the actual numbers, a ! means that the comparison should not be considered, and / means that the resids must be different. The rows correspond to the order at the left of the comparison (order1 argument), while the columns correspond to the order at the right of it (order2 argument).

	>	>>	<	<<	n	0	*	**
>	=	<	>	>	!	>	!	!
>>	>	=	>	>	!	>	!	!
<	<	<	=	>	!	<	!	!
<<	<	<	<	=	!	<	!	!
n	!	!	!	!	?	?	!	!
0	<	<	>	>	?	=	/	/
*	!	!	!	!	!	/	=	/
**	!	!	!	!	!	/	/	=

Parameters

- **`order1`** (`int or str`) – The order attribute of the residue on the left of the comparison.
- **`resid1`** (`int`) – The residue id of the residue on the left of the comparison.
- **`order2`** (`int or str`) – The order attribute of the residue on the right of the comparison.
- **`resid2`** (`int`) – The residue id of the residue on the right of the comparison.

Returns
True if the conditions match.

Return type
bool

Raises
`ValueError` – Raised if the order arguments do not follow the expected format.

vermouth.processors.do_mapping module

Provides a processor that can perform a resolution transformation on a molecule.

`class vermouth.processors.do_mapping.`**`DoMapping`**(*mappings*, *to_ff*, *delete_unknown=False*, *attribute_keep=()*, *attribute_must=()*, *attribute_stash=()*)

Bases: `Processor`

Processor for performing a resolution transformation from one force field to another.

This processor will create new Molecules by stitching together Blocks from the target force field, as dictated by the available mappings. Fragments/atoms/residues/modifications for which no mapping is available will not be represented in the resulting molecule.

The resulting molecules will have intra-block edges and interactions as specified in the blocks from the target force field. Inter-block edges will be added based on the connectivity of the original molecule, but no interactions will be added for those.

`mappings`

`{ff_name: {ff_name: {block_name: (mapping, weights, extra)}}}` A collection of mappings, as returned by e.g. `read_mapping_directory()`.

Type
dict[str, dict[str, dict[str, tuple]]]

`to_ff`

The force field to map to.

Type
vermouth.forcefield.ForceField

`delete_unknown`
Not currently used

Type
bool

`attribute_keep`
The attributes that will always be transferred from the input molecule to the produced graph.

Type
tuple[str]

`attribute_must`
The attributes that the nodes in the output graph *must* have. If they're not provided by the mappings/blocks they're taken from the original molecule.

Type
tuple[str]

`attribute_stash`
The attributes that will always be transferred from the input molecule to the produced graph, but prefixed with _old_.Thus they are new attributes and are not conflicting with already defined attributes.

Type
tuple[str]

See also:

do_mapping()

run_molecule(*molecule*)

run_system(*system*)

vermouth.processors.do_mapping.**apply_block_mapping**(*match*, *molecule*, *graph_out*, *mol_to_out*, *out_to_mol*)

Performs a mapping operation for a "block". *match* is a tuple of 3 elements that describes what nodes in *molecule* should correspond to a `vermouth.molecule.Block` that should be added to *graph_out*, and any atoms that should be used a references. Add the required `vermouth.molecule.Block` to *graph_out*, and updates *mol_to_out* and *out_to_mol in-place*.

Parameters

- **match** –
- **molecule** (`networkx.Graph`) – The original molecule
- **graph_out** (`vermouth.molecule.Molecule`) – The newly created graph that describes *molecule* at a different resolution.
- **mol_to_out** (*dict[collections.abc.Hashable, dict[collections.abc.Hashable, float]]*) – A dict mapping nodes in *molecule* to nodes in *graph_out* with the associated weights.
- **out_to_mol** (*dict[collections.abc.Hashable, dict[collections.abc.Hashable, float]]*) – A dict mapping nodes in *graph_out* to nodes in *molecule* with the associated weights.

Returns

- *set* – A set of all overlapping nodes that were already mapped before.
- *set* – A set of none-to-one mappings. I.e. nodes that were created without nodes mapping to them.
- *dict* – A dict of reference atoms, mapping *graph_out* nodes to nodes in *molecule*.

`vermouth.processors.do_mapping.`**`apply_mod_mapping`**(*match*, *molecule*, *graph_out*, *mol_to_out*, *out_to_mol*)

Performs the mapping operation for a modification.

Parameters

- **`match`** –
- **`molecule`** (*`networkx.Graph`*) – The original molecule
- **`graph_out`** (`vermouth.molecule.Molecule`) – The newly created graph that describes *molecule* at a different resolution.
- **`mol_to_out`** (*`dict[collections.abc.Hashable, dict[collections.abc.Hashable, float]]`*) – A dict mapping nodes in *molecule* to nodes in *graph_out* with the associated weights.
- **`out_to_mol`** (*`dict[collections.abc.Hashable, dict[collections.abc.Hashable, float]]`*) – A dict mapping nodes in *graph_out* to nodes in *molecule* with the associated weights.

Returns

- *dict[str, dict[tuple, vermouth.molecule.Link]]* – A dict of all modifications that have been applied by this modification mapping operations. Maps interaction type to involved atoms to the modification responsible.
- *dict* – A dict of reference atoms, mapping *graph_out* nodes to nodes in *molecule*.

`vermouth.processors.do_mapping.`**`attrs_from_node`**(*node*, *attrs*)

Helper function that applies a "replace" operations on the node if required, and then returns a dict of the attributes listed in *attrs*.

Parameters

- **`node`** (*`dict`*) –
- **`attrs`** (*`collections.abc.Container`*) – Attributes that should be in the output.

Return type

dict

`vermouth.processors.do_mapping.`**`build_graph_mapping_collection`**(*from_ff*, *to_ff*, *mappings*)

Function that produces a collection of *`vermouth.map_parser.Mapping`* objects. Hereby deprecated.

Parameters

- **`from_ff`** (`vermouth.forcefield.ForceField`) – Origin force field.
- **`to_ff`** (`vermouth.forcefield.ForceField`) – Destination force field.
- **`mappings`** (*`dict[str, dict[str, `*`vermouth.map_parser.Mapping`*`]]`*) – All known mappings

Returns

A collection of mappings that map from *from_ff* to *to_ff*.

Return type
collections.abc.Iterable

`vermouth.processors.do_mapping.`**`cover`**`(`*to_cover*, *options*`)`

Implements a recursive backtracking algorithm to cover all elements of *to_cover* with the elements from *options* that have the lowest index. In this context "to cover" means that all items in an element of *options* must be in *to_cover*. Elements in *to_cover* can only be covered *once*.

Parameters

- **`to_cover`** (*`collections.abc.MutableSet`*) – The items that should be covered.
- **`options`** (*`collections.abc.Sequence[collections.abc.MutableSet]`*) – The elements that can be used to cover *to_cover*. All items in an element of *options* must be present in *to_cover* to qualify.

Returns
None if no covering can be found, or the list of items from *options* with the lowest indices that exactly covers *to_cover*.

Return type
None or list

`vermouth.processors.do_mapping.`**`do_mapping`**`(`*molecule*, *mappings*, *to_ff*, *attribute_keep=()*, *attribute_must=()*, *attribute_stash=()*`)`

Creates a new *`Molecule`* in force field *to_ff* from *molecule*, based on *mappings*. It does this by doing a subgraph isomorphism of all blocks in *mappings* and *molecule*. Will issue warnings if there's atoms not contributing to the new molecule, or if there's overlapping blocks. Node attributes in the new molecule will come from the blocks constructing it, except for those in *attribute_keep*, which lists the attributes that will be kept from *molecule*.

Parameters

- **`molecule`** (*`Molecule`*) – The molecule to transform.
- **`mappings`** (*`dict[str, dict[str, dict[str, tuple]]]`*) – `{ff_name: {ff_name: {block_name: (mapping, weights, extra)}}}` A collection of mappings, as returned by e.g. *`read_mapping_directory()`*.
- **`to_ff`** (*`ForceField`*) – The force field to transform to.
- **`attribute_keep`** (`Iterable`) – The attributes that will always be transferred from *molecule* to the produced graph.
- **`attribute_must`** (`Iterable`) – The attributes that the nodes in the output graph *must* have. If they're not provided by the mappings/blocks they're taken from *molecule*.
- **`attribute_stash`** (*`tuple[str]`*) – The attributes that will always be transferred from the input molecule to the produced graph, but prefixed with _old_.Thus they are new attributes and are not conflicting with already defined attributes.

Returns
A new molecule, created by transforming *molecule* to *to_ff* according to *mappings*.

Return type
`Molecule`

`vermouth.processors.do_mapping.`**`edge_matcher`**`(`*graph1*, *graph2*, *node11*, *node12*, *node21*, *node22*`)`

Checks whether the resids for node11 and node12 in graph1 are the same, and whether that's also true for node21 and node22 in graph2.

Parameters

- **`graph1`** (*`networkx.Graph`*) –

- **graph2** (*networkx.Graph*) –
- **node11** (*collections.abc.Hashable*) – A node key in *graph1*.
- **node12** (*collections.abc.Hashable*) – A node key in *graph1*.
- **node21** (*collections.abc.Hashable*) – A node key in *graph2*.
- **node22** (*collections.abc.Hashable*) – A node key in *graph2*.

Return type
bool

`vermouth.processors.do_mapping.`**`get_mod_mappings`**`(`*mappings*`)`

Returns a dict of all known modification mappings.

Parameters
mappings (*collections.abc.Iterable[*vermouth.map_parser.Mapping*]*) – All known mappings.

Returns
All mappings that describe a modification mapping.

Return type
dict[tuple[str], *vermouth.map_parser.Mapping*]

`vermouth.processors.do_mapping.`**`modification_matches`**`(`*molecule*, *mappings*`)`

Returns a minimal combination of modification mappings and where they should be applied that describes all modifications in *molecule*.

Parameters

- **molecule** (*networkx.Graph*) – The molecule whose modifications should be treated. Modifications are described by the 'modifications' node attribute.
- **mappings** (*collections.abc.Iterable[*vermouth.map_parser.Mapping*]*) – All known mappings.

Returns

A list with the following items:

Dict describing the correspondence of node keys in *molecule* to
node keys in the modification.

The modification.

Dict with all reference atoms, mapping modification nodes to
nodes in *molecule*.

Return type
list[tuple[dict, *vermouth.molecule.Link*, dict]]

`vermouth.processors.do_mapping.`**`node_matcher`**`(`*node1*, *node2*`)`

Checks whether nodes should be considered equal for isomorphism. Takes all attributes in *node2* into account, except for the attributes "atype", "charge", "charge_group", "resid", "replace", and "_old_atomname".

Parameters

- **node1** (*dict*) –
- **node2** (*dict*) –

Return type
bool

`vermouth.processors.do_mapping.`**`node_should_exist`**`(`*modification*, *node_idx*`)`

Returns True if the node with index *node_idx* in *modification* should already exist in the parent molecule.

Parameters

- **`modification`** (*`networkx.Graph`*) –
- **`node_idx`** (*`collections.abc.Hashable`*) – The key of a node in *modification*.

Returns

True iff the node *node_idx* in *modification* should already exist in the parent molecule.

Return type

bool

`vermouth.processors.do_mapping.`**`ptm_resname_match`**`(`*mol_node*, *map_node*`)`

As *`node_matcher()`*, except that empty resname and false PTM_atom attributes from *node2* are removed.

vermouth.processors.gro_reader module

Provides a processor that reads a GRO file.

See also:

`vermouth.gmx.gro`

`class` `vermouth.processors.gro_reader.`**`GROInput`**`(`*filename*, *exclude=()*, *ignh=False*`)`

Bases: *`Processor`*

`run_system``(`*system*`)`

vermouth.processors.locate_charge_dummies module

Provides a processor that generates positions for every charge dummy.

`class` `vermouth.processors.locate_charge_dummies.`**`LocateChargeDummies`**`(`*attribute_tag='charge_dummy'*`)`

Bases: *`Processor`*

`run_molecule``(`*molecule*`)`

`vermouth.processors.locate_charge_dummies.`**`colinear_pair`**`()`

Build two points on a line around the origin at a random orientation.

`vermouth.processors.locate_charge_dummies.`**`fibonacci_sphere`**`(`*n_samples*`)`

Place points near-evenly distributed on a sphere.

Use the Fibonacci sphere algorithm to place 'n_samples' points at the surface of a sphere of radius 1, centered on the origin.

Parameters

`n_samples` (*`int`*) – Number of points to place.

Returns

3D coordinates of the points.

Return type

numpy.ndarray

`vermouth.processors.locate_charge_dummies.`**`find_anchor`**`(`*molecule*, *node_key*, *attribute_tag='charge_dummy'*`)`

Find the non-dummy bead to which a charge dummy is anchored.

Each charge dummy has to be attached to exactly one non-dummy atom. This function returns the node key for that non-dummy atom.

Parameters

- **`molecule`** (*`networkx.Graph`*) – The molecule to work on.
- **`node_key`** – The node key of the charge dummy.
- **`attribute_tag`** (*`str`*) – The name of the atom attribute used to describe charge dummies.

Returns
The node key of the anchor in the molecule graph.

Return type
collections.abc.Hashable

Raises
`ValueError` – Raised if there are no anchor, or more than one anchor, found. Raised also if the charge dummy is not a charge dummy.

`vermouth.processors.locate_charge_dummies.`**`locate_all_dummies`**`(`*molecule*, *attribute_tag='charge_dummy'*`)`

Set the position of all charge dummies of a molecule.

The molecule is modified in-place.

The charge dummies are placed at a distance to the anchor defined in nm by their charge dummy attribute, the name of which is given in the 'attribute_tag' argument.

Parameters

- **`molecule`** (`vermouth.molecule.Molecule`) – The molecule to work on.
- **`attribute_tag`** (*`str`*) – Name of the atom attribute that describe charge dummies.

`vermouth.processors.locate_charge_dummies.`**`locate_dummy`**`(`*molecule*, *anchor_key*, *dummy_keys*, *attribute_tag='charge_dummy'*`)`

Set the position of a group of charge dummies around a non-dummy anchor.

The molecule is modified in-place.

The charge dummies are placed at a distance to the anchor defined in nm by their charge dummy attribute, the name of which is given in the 'attribute_tag' argument.

Parameters

- **`molecule`** (`vermouth.molecule.Molecule`) – The molecule to work on.
- **`anchor_key`** – The key of the non-dummy anchor all the charge dummies are connected to.
- **`dummy_keys`** (*`collections.abc.Iterable`*) – A collection of atom keys for charge dummies to position.
- **`attribute_tag`** (*`str`*) – Name of the atom attribute that describe charge dummies.

vermouth.processors.make_bonds module

Provides a processor that can add edges to a graph based on geometric criteria.

class vermouth.processors.make_bonds.**MakeBonds**(*allow_name=True*, *allow_dist=True*, *fudge=1.2*)

Bases: `Processor`

Processor to add edges to a system and separate it into separate connected molecules.

Two separate criteria are used to decide where to add edges. The system's molecules are separated into residues. Then intra-residue edges are added.

If `allow_names` is True, the corresponding `Block` is looked up in the system's force field. First edges will be added based on the edges in that block. In addition, *non-edges* in the reference block are also stored.

Secondly, if `allow_dist` is True, edges will be added between any atoms that are close enough together. The threshold for "close enough" is determined based on the elements of the atoms in question and their van der Waals radii, multiplied by `fudge`. This way edges will *not* be added between atoms that were marked as 'non-edge' in the previous step, nor between residues if one of the atoms is a hydrogen.

allow_names

Whether edges should be added based on atom names.

Type
bool

allow_dist

Whether edges should be added based on distance.

Type
bool

fudge

A fudge factor used to increase the reference van der Waals radii to allow for conformations that are slightly out of equilibrium.

Type
`Number`

See also:

`make_bonds()`

run_system(*system*)

vermouth.processors.make_bonds.**make_bonds**(*system*, *allow_name=True*, *allow_dist=True*, *fudge=1.2*)

Creates bonds within molecules in the system.

First, edges will be created based on residue and atom names. Second, edges will be created based on a distance criterion. Nodes in system must have *position* and *element* attributes. The possible distance between nodes is determined by values in *VDW_RADII*. Edges within residues will only be guessed between atoms that are not known in the reference Block. The system will be split into connected components, keeping residues (identified by chain, residue name and residue id) within the same molecule. This does mean that the final molecules can be disconnected.

Notes

Edges for residues for which no block can be found will be added based on
the distance criterion. A warning will be issued if this is the case.

Elements that are not in *VDW_RADII* do not make bonds based on distances.

Parameters

- **system** (*System*) – The system in which to add edges.
- **fudge** (Number) – Scale the allowed distance by this factor.

Returns
Molecules in system, in which edges have been added based on atom names and possibly distance. The molecules have been split into connected components keeping residues intact. Molecules can be disconnected within residues.

Return type
List[*Molecule*]

vermouth.processors.merge_all_molecules module

Provides a processor that merges all the molecules from a system.

`class vermouth.processors.merge_all_molecules.`**`MergeAllMolecules`**

Bases: *Processor*

Merge all the molecules from a system.

The molecules are merged into the first molecule of the system. Nothing is done if there are no molecules.

`static` **`run_molecule`**(*molecule*)

`run_system`(*system*)

vermouth.processors.merge_chains module

Merge molecules by chain.

`class vermouth.processors.merge_chains.`**`MergeChains`**(*chains*)

Bases: *Processor*

`name = 'MergeChains'`

`run_system`(*system*)

`vermouth.processors.merge_chains.`**`merge_chains`**(*system*, *chains*)

Merge molecules with the given chains as a single molecule.

Molecules are merged into the resulting molecule if their chain is in the list of chains to merge. The resulting molecule is not connected.

If a molecule comprises multiple chains, then it is merged only if all the chains it comprises are part of the selection.

The meta variable are not conserved in the process.

The input system is modified in-place.

Parameters

- **system** (vermouth.system.System) – The system to modify.
- **chains** (*list[str]*) – A container of chain identifier.

vermouth.processors.name_moltype module

Provides a processor to assign molecule type names to molecules.

A molecule type (moltype) is Gromacs's concept of a molecule. Providing a name for a molecule type is required to write an ITP file for that molecule. We also use the molecule type name to group molecules sharing the same molecule type. Molecule type identity is tested based on *vermouth.molecule.Molecule.share_moltype_with()*.

`class vermouth.processors.name_moltype.`**`NameMolType`**(*deduplicate=True*, *meta_key='moltype'*)

Bases: *Processor*

Assigns molecule type (moltype) names to molecules.

Moltype names are the names given to molecules in an ITP file. This processor assign consecutive names to the molecule. If the *deduplicate* argument is set to *True*, then the processor assigns the same name to all molecules with the same topology.

By default, the moltype name is written under the "moltype" key of the molecule meta attributes. This key can be changed with the *meta_key* argument.

Parameters

- **deduplicate** (*bool*) – If *True*, the same name is given to all the molecules that share the same topology. Else, each molecule is given a different name.
- **meta_key** (*str*) – The name of the key in the molecule *meta* dictionary under which the moltype must be stored.

See also:

vermouth.processors.set_molecule_meta.SetMoleculeMeta
This processor can set key/value pairs in the meta attributes of one molecule, or all molecules in a system. It can be used to set the moltype manually.

vermouth.gmx.itp.write_molecule_itp
Writes the ITP file for a molecule, and use the 'moltype' meta to name the molecule.

`run_system`(*system*)

vermouth.processors.pdb_reader module

Provides a processor that reads a PDB file.

See also:

vermouth.pdb.pdb

`class vermouth.processors.pdb_reader.`**`PDBInput`**(*filename*, *exclude=()*, *ignh=False*, *modelidx=0*)

Bases: *Processor*

Reads PDB files.

filename

The filename to parse.

Type
str

exclude

A collection of residue names that should not be parsed and excluded from the final molecule(s)

Type
collections.abc.Container[str]

ignh

If True, hydrogens will be discarded from the input structure.

Type
bool

modelidx

The model number to parse/use.

Type
int

See also:

read_pdb(), *PDBParser()*

run_system(*system*)

vermouth.processors.processor module

Provides an abstract base class for processors.

class vermouth.processors.processor.**Processor**

Bases: `object`

An abstract base class for processors. Subclasses must implement a *run_molecule* method.

run_molecule(*molecule*)

Process a single molecule. Must be implemented by subclasses.

Parameters
molecule (`vermouth.molecule.Molecule`) – The molecule to process.

Returns
Either the provided molecule, or a brand new one.

Return type
vermouth.molecule.Molecule

run_system(*system*)

Process *system*.

Parameters
system (`vermouth.system.System`) – The system to process. Is modified in-place.

vermouth.processors.quote module

Reads quotes, and produces a random one.

`class vermouth.processors.quote.`**`Quoter`**`(`*quote_file=None*`)`

Bases: *`Processor`*

Processor that can produce random string taken from a file. Useful for e.g. quotes.

Parameters

`quote_file` (*`pathlib.Path or str`*) – The path of the file containing the strings. Must contain at least one line.

`run_system``(`*system*`)`

Logs a random line from the file passed at initialization.

Parameters

`system` – Not used

Return type

None

`vermouth.processors.quote.`**`read_quote_file`**`(`*filehandle*`)`

Iterates over *filehandle*, and yields all strings that are not empty.

Parameters

`filehandle` (*`collections.abc.Iterable[str]`*) – A file opened for reading.

Yields

str – All stripped elements of *filehandle* that are not empty.

vermouth.processors.rename_modified_residues module

Provides a processor that renames residues based on their current residue names and identified modifications, such as PTMs.

`class vermouth.processors.rename_modified_residues.`**`RenameModifiedResidues`**

Bases: *`Processor`*

`run_molecule``(`*molecule*`)`

`vermouth.processors.rename_modified_residues.`**`rename_modified_residues`**`(`*mol*`)`

Renames residue names based on the current residue name, and the found modifications. The new names are found in *force_field.renamed_residues*, which should be a mapping of `{(rename, [modification_name, ...]): new_name}`.

Parameters

`mol` (`Molecule`) – The molecule whose residue names should be changed. Is modified in-place.

vermouth.processors.repair_graph module

Provides a processor that repairs a graph based on a reference.

`class vermouth.processors.repair_graph.`**`RepairGraph`**`(`*delete_unknown=False*, *include_graph=True*`)`

Bases: *`Processor`*

Repairs a molecule such that it contains all atoms with appropriate atom names, as per the blocks in the system's force field, while taking any mutations and modification into account. These should be added as 'mutation' and 'modification' attributes to the atoms of the relevant residues.

`delete_unknown`

If True, removes any molecules that contain residues that are not known to the system's force field.

Type

bool

`include_graph`

If True, every node in the resulting graph will have a 'graph' attribute containing a subgraph constructed using the input atoms.

Type

bool

See also:

`repair_graph()`

`run_molecule``(`*molecule*`)`

`run_system``(`*system*`)`

`vermouth.processors.repair_graph.`**`get_default`**`(`*dictionary*, *attr*, *default*`)`

Functions like `dict.get()`, except that when *attr* is in *dictionary* and *dictionary[attr]* is *None*, it will return *default*.

Parameters

- **`dictionary`** (`dict`) –
- **`attr`** (*`collections.abc.Hashable`*) –
- **`default`** –

Returns

The value of *dictionary[attr]* if *attr* is in *dictionary* and *dictionary[attr]* is not None. *default otherwise.*

Return type

object

`vermouth.processors.repair_graph.`**`make_reference`**`(`*mol*`)`

Takes an molecule graph (e.g. as read from a PDB file), and finds and returns the graph how it should look like, including all matching nodes between the input graph and the references. Requires residue names to be correct.

Notes

The match between hydrogren atoms need not be perfect. See the documentation of `isomorphism`.

Parameters

`mol` (*`networkx.Graph`*) – The graph read from e.g. a PDB file. Required node attributes:

resname
The residue name.

resid
The residue id.

chain
The chain identifier.

element
The element.

atomname
The atomname.

Returns

The constructed reference graph with the following node attributes:

resid
The residue id.

resname
The residue name.

chain
The chain identifier.

found
The residue subgraph from the PDB file.

reference
The residue subgraph used as reference.

match
A dictionary describing how the reference corresponds with the provided graph. Keys are node indices of the reference, values are node indices of the provided graph.

Return type

networkx.Graph

`vermouth.processors.repair_graph.`**`repair_graph`**(*molecule*, *reference_graph*, *include_graph=True*)

Repairs a molecule graph produced based on the information in `reference_graph`. Missing atoms will be added and atom- and residue- names will be canonicalized. Atoms not present in `reference_graph` will have the attribute `PTM_atom` set to `True`.

`molecule` is modified in place. Missing atoms (as per `reference_graph`) are added, atom and residue names are canonicalized, and PTM atoms are marked.

If `include_graph` is `True`, then the subgraph corresponding to each node is included in the node under the “graph” attribute.

Parameters

- **`molecule`** (`molecule.Molecule`) – The graph read from e.g. a PDB file. Required node attributes:

resname
The residue name.

resid
The residue id.

element
The element.

atomname
The atomname.

- **reference_graph** (*networkx.Graph*) – The reference graph as produced by *make_reference()*. Required node attributes:

  resid
  The residue id.

  resname
  The residue name.

  found
  The residue subgraph from the PDB file.

  reference
  The residue subgraph used as reference.

  match
  A dictionary describing how the reference corresponds with the provided graph. Keys are node indices of the reference, values are node indices of the provided graph.

- **include_graph** (*bool*) – Include the subgraph in the nodes.

vermouth.processors.repair_graph.**repair_residue**(*molecule*, *ref_residue*, *include_graph*)

Rebuild missing atoms and canonicalize atomnames

vermouth.processors.set_molecule_meta module

class vermouth.processors.set_molecule_meta.**SetMoleculeMeta**(***meta*)

Bases: *Processor*

run_molecule(*molecule*)

vermouth.processors.sort_molecule_atoms module

Provides a processor that sorts atoms within molecules.

class vermouth.processors.sort_molecule_atoms.**SortMoleculeAtoms**(*sortby_attrs=('chain', 'resid', 'resname', 'insertion_code', 'atomid')*, *target_attr=None*)

Bases: *Processor*

Sort the atoms within a molecule by the attributes listed in the *sortby_attrs*. Optionally, new atom indices are assigned to the node attribute *target_attr*.

Sorting nodes is useful because a lot of software assumes chains and residues are listed contiguously. In particular this gets important when we add atoms — for instance missing atoms identified by *vermouth.processors.repair_graph.RepairGraph*).

Nodes in the molecule are reordered according to the node attributes listed in `sortby_attrs`. The atom keys are left identical, only the order of the nodes is changed. Optionally, the new indices can be assigned to nodes `target_attr` attribute.

`sortby_attrs`

Nodes will be sorted by these node attributes.

Type
collections.abc.Sequence[collections.abc.Hashable]

`target_attr`

If not `None`, new indices will be assigned to this node attribute, starting with 1. It is a good idea to make sure this attribute is also listed in `sortby_attrs` so that the sorting is stable.

Type
collections.abc.Hashable

`run_molecule`(*molecule*)

vermouth.processors.tune_cystein_bridges module

Provides processors that can add and remove cystein bridges.

`class vermouth.processors.tune_cystein_bridges.`**`AddCysteinBridgesThreshold`**(*threshold*, *template=[{'atomname': 'SG', 'resname': 'CYS'}]*, *attribute='position'*)

Bases: `AddMoleculeEdgesAtDistance`

Add edges corresponding to cystein bridges on a distance criterion.

The edge for a cystein bridge is an edge between two atoms that match at least one template from a list of templates if the two ends of the edge are closer than a given distance.

Parameters

- **`threshold`** (*`float`*) – Distance in nanometers under which to consider an edge.
- **`template`** (*`list[dict]`*) – List of node templates.

`class vermouth.processors.tune_cystein_bridges.`**`RemoveCysteinBridgeEdges`**(*template=[{'atomname': 'SG', 'resname': 'CYS'}]*)

Bases: `Processor`

Processor removing edges corresponding to cystein bridges.

The edge for a cystein bridge is an edge between two atoms that match at least one template from a list of templates.

Parameters
`template` (*`list[dict]`*) – List of node templates.

`run_molecule`(*molecule*)

`vermouth.processors.tune_cystein_bridges.`**`remove_cystein_bridge_edges`**(*molecule*, *templates=[{'atomname': 'SG', 'resname': 'CYS'}]*)

Remove all the edges that correspond to cystein bridges from a molecule.

Cystein bridge edges link an atom from a cystein side chain to the same atom on an other cystein. Selecting the correct atom is done with a list of template node dictionaries. A template node dictionary functions in the same way as node matching in links. An atom that can be involved in a cystein bridge must match at least one of the templates of the list. The default template list selects the 'SG' bead of the residue 'CYS': `[{'resname': 'CYS', 'atomname': 'SG'}, ]`.

A template is a dictionary that defines the key:value pairs that must be matched in the atoms. Values can be instances of `LinkPredicate`.

Parameters

- **`molecule`** (*`networkx.Graph`*) – Molecule to modify in-place.
- **`templates`** (*`list[dict]`*) – A list of templates; selected atom must match at least one.

vermouth.processors.water_bias module

`class vermouth.processors.water_bias.`**`ComputeWaterBias`**`(`*auto_bias*, *water_bias*, *idr_regions*`)`

Bases: `Processor`

Processor which computes the water bias for the Martini Go and Martini IDP model.

The water bias strength is defined per secondary structure element in *water_bias* and assigned if *auto_bias* is set to True. Using the *idr_regions* argument the water_bias can be changed for intrinsically disordered regions (IDRs). The IDR bias superseeds the auto bias.

This Processor updates the system.gmx_topology_params attribute.

Subclassing

If the procedure by which to assign the water bias is to be changed this processor is best subclassed and the assign_residue_water_bias method overwritten.

param auto_bias
apply the automatic secondary structure dependent water biasing

type auto_bias
bool

param water_bias
a dict of secondary structure codes and epsilon value for the water bias in kJ/mol

type water_bias
dict[str, float]

param idr_regions
regions defining the IDRs

param prefix
prefix of the Go virtual-site atomtypes

type prefix
str

param system
the system of the molecules is used for storing the nonbonded parameters

type system
vermouth.system.System

`assign_residue_water_bias`(*molecule*, *res_graph*)

Assign the residue water bias for all residues with a secondary structure element or that are defined by the region selector. Region selectors supercede the auto assignment.

Parameters

- `molecule` (*vermouth.molecule.Molecule*) – the molecule
- `res_graph` (*vermouth.molecule.Molecule*) – the residue graph of the molecule

`run_molecule`(*molecule*)

Assign water bias for a single molecule

`run_system`(*system*)

Assign the water bias of the Go model to file. Biasing is always molecule specific i.e. no two different vermouth molecules can have the same bias.

Parameters
`system` (*vermouth.system.System*) –

Module contents

Provides Processors, VerMoUTH's work horses.

vermouth.rcsu package

Submodules

vermouth.rcsu.contact_map module

Read RCSU Go model contact maps.

`vermouth.rcsu.contact_map.read_go_map`(*file_path*)

Read a RCSU contact map from the c code as published in doi:10.5281/zenodo.3817447. The format requires all contacts to have 18 columns and the first column to be a capital R.

Parameters
`file_path` (`pathlib.Path`) – path to the contact map file

Returns
contact as chain id, res id, chain id, res id

Return type
list(tuple)

vermouth.rcsu.go_pipeline module

Wrapper of Processors defining the GoPipline.

`class vermouth.rcsu.go_pipeline.`**`GoProcessorPipline`**(*processor_list*)

Bases: *Processor*

Wrapping all processors for the go model.

`prepare_run`(*system*, *moltype*)

Things to do before running the pipeline.

`run_system`(*system*, ***kwargs*)

vermouth.rcsu.go_structure_bias module

Obtain the structural bias for the Go model.

`class vermouth.rcsu.go_structure_bias.`**`ComputeStructuralGoBias`**(*contact_map*, *cutoff_short*, *cutoff_long*, *go_eps*, *res_dist*, *moltype*, *res_graph=None*)

Bases: *Processor*

Generate the Go model structural bias for a system of molecules. This processor class has two main functions: .contact_selector and .compute_bias. The .run_molecule function simply loops over all molecules in the system and calls the other two functions. The computed structural bias parameters are stored in *system.gmx_topology_params* and can be written out using the *vermouth.gmx.write_topology* function.

Subclassing

In order to customize the Go-model structural bias it is recommended to subclass this function and overwrite the `contact_selector` method and/or the `compute_bias` method. This subclassed Processor then has to be added to the into the martinize2 pipeline in place of the StructuralBiasWriter or as replacement in the GoPipeline.

Initialize the Processor with arguments required to setup the Go model structural bias.

param contact_map
list of contacts defined as by the chain identifier and residue index

type contact_map
list[(str, int, str, int)]

param cutoff_short
distances in nm smaller than this are ignored

type cutoff_short
float

param cutoff_long
distances in nm larger than this are ignored

type cutoff_long
float

param go_eps
epsilon value of the structural bias in kJ/mol

type go_eps
float

param res_dist
if nodes are closer than res_dist along the residue graph they are ignored; this is similar to sequence distance but takes into account disulfide bridges for example

type res_dist
int

param moltype
name of the molecule to treat

type moltype
str

param res_graph
residue graph of the molecule; if None it gets generated automatically

type res_graph
`vermouth.molecule.Molecule`

param system
the system

type system
`vermouth.system.System`

param magic_number
magic number for Go contacts from the old GoVirt script.

type magic_number
float

`compute_go_interaction`(*contacts*)

Compute the epsilon value given a distance between two nodes, figure out the atomtype name and store it in the systems attribute gmx_topology_params.

Parameters
`contacts` (`list[(str, str, float)]`) – list of node-keys and their distance

Returns
dict[frozenset(str, str) – dict of interaction parameters indexed by atomtype

Return type
float]

`contact_selector`(*molecule*)

Select all contacts from the contact map that according to their distance and graph connectivity are elegible to form a Go bond and create exclusions between the backbone beads of those contacts.

Parameters
`molecule` (`vermouth.molecule.Molecule`) –

Returns
list of node keys and distance

Return type
list[(collections.abc.Hashable, collections.abc.Hashable, float)]

`run_molecule`(*molecule*)

run_system(*system*)

Process *system*.

Parameters

system (vermouth.system.System) – The system to process. Is modified in-place.

vermouth.rcsu.go_utils module

Utilities for Go model processors.

vermouth.rcsu.go_utils.**get_go_type_from_attributes**(*molecule*, *prefix*, ***kwargs*)

Find all nodes that satisfy a number of attributes specified as kwargs and have a specific atomtype prefix.

Parameters

- **molecule** (*vermouth.molecule.Molecule*) –
- **prefix** (*str*) – the atom-type prefix of the Go virtual side
- **kwargs** – any number of attributes

Yields

str – the atom-type

Raises

KeyError – If no node can be found that matches attributes and prefix an KeyError is raised.

vermouth.rcsu.go_vs_includes module

class vermouth.rcsu.go_vs_includes.**VirtualSiteCreator**

Bases: *Processor*

Create virtual-sites for the Martini Go model implementation or the specific water biasing options.

See *vermouth.rcsu* for more details.

Every molecule must have a moltype name under the "moltype" key of the molecule meta.

See also:

NameMolType

Assign molecule type names to the molecules in a system.

add_virtual_sites()

add_virtual_sites(*molecule*, *prefix*, *backbone='BB'*, *atomname='CA'*, *charge=0*)

Add the virtual sites for GoMartini in the molecule.

One virtual site is added per backbone bead of the the Martini protein. Each virtual site copies the resid, resname, and chain of the backbone bead. It also copies the *reference* to the position array, so the virtual site position follows if the backbone bead is translated. The virtual sites are added *after* all the other atoms of the molecule, each in its own charge group, with "CA" as atomname, and a charge of 0. The atomname and charge can be set with the *atomname* and *charge* argument, respectively.

The bead type of the virtual sites is names "<prefix>_<resid>". Where *prefix* is provided as an argument of the function, and is expected to be the molecule type name.

Parameters

- **molecule** (vermouth.molecule.Molecule) – The molecule to augment with virtual sites.
- **prefix** (*str*) – The prefix to use for bead type names. Usually the molecule type name.
- **backbone** (*str*) – The atomname of the backbone beads.
- **atomname** (*str*) – The atomname of the virtual sites.
- **charge** (*float or int*) – The charge of the virtual sites.

run_molecule(*molecule*)

run_system(*system*)

Module contents

6.1.2 Submodules

vermouth.citation_parser module

class vermouth.citation_parser.**BibTexDirector**(*force_field*)

Bases: object

Lightweight parser for BibTex files. BibTex files in general have an assortment of entries that describe the corresponding sort of publication to refer to and then a number required and optional fields for the different types of entries. A field for example would be Title giving the title of a publication. The syntax in general looks as follows:

@<entry>{<some custom ID>, field = {<content>},
field = {<content>}}

Alternatively the { } can be replaced by quotation marks.

This parser only parses the version with { } as used by google scholar. In addition we do not check for missing fields or invalid fields. All fields are accepted and no fields are required.

static **extract_fields**(*entry_string*)

Given an entry string without entry type and identified (i.e. ,<field_type> = {<content>}, etc.) split all the contents and field-types using a regular expression.

Parameters
entry_string (*str*) –

Yields
str, str – the field type, the field content

static **find_entries**(*citation_string*)

Look in a string where @ indicates the beginning of a new entry and return the indices.

Parameters
citation_string (*str*) –

Yields
int – position of '@' in citation_string

parse(*lines*)

Given lines from a bibtex file parse them and update the force-field citation instance variable.

parse_entry(*entry_string*)

Given a string describing a single entry, parse it and then update the force_field citations dict with a field dict.

pop_entry_type(*entry_string*)

Given a string describing a single entry strip that entry from the string and return it. Note the string MUST contain the @.

Parameters
entry_string (*str*) –

Returns

- *str* – The entry type
- *str* – The shortened string

static **pop_key**(*entry_string*)

Given a string of a single entry from which the entry_type has already been removed (see pop_entry_type) get the custom ID, strip it and return the entry_string without that ID.

Parameters
entry_string (*str*) –

Returns
the key and the string without key

Return type
str, str

static **prepare_file**(*lines*)

Bibtex is not sensitive to line spacing so we join the line as one string. Comment characters are not allowed.

vermouth.citation_parser.**citation_formatter**(*citation*, *title=False*)

Very basic and minimal formatter for citations. It is adopted from basic ACS style formatting. Fields within [] are optional.

<authors> [journal] <year>; [doi]

Note that the formatter cannot format latex like syntax (e.g. a{"} for ae)

vermouth.citation_parser.**read_bib**(*lines*, *force_field*)

vermouth.edge_tuning module

Set of tools to add and remove edges.

vermouth.edge_tuning.**add_edges_at_distance**(*molecule*, *threshold*, *selection_a*, *selection_b*, *attribute='position'*)

Add edges within a molecule when the distance is below a threshold.

Create edges within a molecule between nodes that have an end part of 'selection_a', the other end part of 'selection_b', and a distance between the ends that is lesser than the given threshold.

All nodes that are part of 'selection_a' or 'selection_b' must have a position stored under the attribute which key is given with the 'attribute' argument. That key is 'position' by default. If at least one node has the position missing, then a KeyError is raised.

Parameters

- **molecule** (*networkx.Graph*) – Molecule to modify in-place.

- **threshold** (*float*) – The distance threshold under which edges will be created. The distance is expressed in nm.
- **selection_a** (*collections.abc.Iterable[collections.abc.Hashable]*) – List of node keys from the molecule.
- **selection_b** (*collections.abc.Iterable[collections.abc.Hashable]*) – List of node keys from the molecule.
- **attribute** (*collections.abc.Hashable*) – Name of the key in the node dictionaries under which the coordinates are stored.

Raises

`KeyError` – At least one node from the selections does not have a position.

`vermouth.edge_tuning.`**`add_edges_threshold`**(*molecules*, *threshold*, *templates_a*, *templates_b*, *attribute='position'*, *min_edges=0*)

Add edges between two selections when under a given threshold.

Edges are added within and between the molecules and connect nodes that match the given template. Molecules that get connected by an edge are merged and the new list of molecules is returned.

Parameters

- **molecules** (*collections.abc.Sequence[*`Molecule`*]*) – A list of molecules.
- **threshold** (*float*) – The distance threshold in nanometers under which an edge is created.
- **templates_a** (*dict*) – A list of templates; a node need to match at least one of them to be selected at one end.
- **templates_b** (*dict*) – A list of templates; a node need to match at least one of them to be selected at the other end.
- **attribute** (*str*) – Name of the key in the node dictionaries under which the coordinates are stored.
- **min_edges** (*int*) – Minimum number of edges between to nodes for an edge to be added.

Returns

A new list of molecules.

Return type

list[*vermouth.molecule.Molecule*]

`vermouth.edge_tuning.`**`add_inter_molecule_edges`**(*molecules*, *edges*)

Create edges between molecules.

The function is given a list of molecules and a list of edges. Each edge is provided as a tuple of two nodes, each node being a tuple of the molecule index in the list of molecule, and the node key in that molecule. An edge therefore looks like `((0, 10), (2, 20))` where `1` and `2` are indices of molecules in *molecules*, `10` is the key of a node from `molecules[0]`, and `20` is the key of a node from `molecules[2]`.

The function **can** create edges within a molecule if the same molecule index is given for both ends of edges.

Molecules that get linked are merged. In a merged molecule, the order of the input molecules is kept. In a list of molecules numbered from 0 to 4, if molecules 1, 2, and 4 are merged, then the result molecules are, in order, 0, 1-2-4, 3.

Parameters

- **molecules** (*collections.abc.Sequence[*`vermouth.molecule.Molecule`*]*) – List of molecules to link.

- **edges** (*collections.abc.Iterable[tuple[int, collections.abc.Hashable]]*) – List of edges in a `(molecule_index, node_key)` format as described above. Edges can have a third element, it is then a dictionary of attributes to be attached to the edge.

Returns
New list of molecules.

Return type
list

`vermouth.edge_tuning.`**`pairs_under_threshold`**(*molecules*, *threshold*, *selection_a*, *selection_b*, *attribute='position'*, *min_edges=0*)

List pairs of nodes from a selection that are closer than a threshold.

Get the distance between nodes from multiple molecules and list the pairs that are closer than the given threshold. The molecules are given as a list of molecules, the selection is a list of nodes each of them a tuple `(index of the molecule in the list, key of the node in the molecule)`. The result of the function is a generator of node pairs followed by the distance between the nodes, each node formated as in the selection.

All nodes from the selection must have a position accessible under the key given as the 'attribute' argument. That key is 'position' by default.

With the *min_edges* argument, one can prevent pairs to be selected if there is a path between two nodes that is shorter than a given number of edges.

Parameters

- **molecules** (*collections.abc.Collection[*`vermouth.molecule.Molecule`*]*) – A list of `vermouth.molecule.Molecule`.
- **threshold** (*float*) – A distance threshold in nm. Pairs are return if the nodes are closer than this threshold.
- **selection_a** (*collections.abc.Iterable[collections.abc.Hashable]*) – List of nodes to consider at one end of the pairs. The format is described above.
- **selection_b** (*collections.abc.Iterable[collections.abc.Hashable]*) – List of nodes to consider at the other end of the pairs. The format is described above.
- **attribute** (*collections.abc.Hashable*) – The dictionary key under which the node positions are stored in the nodes.
- **min_edges** (*int*) – Do not select pairs that are connected by less than that number of edges.

Yields
tuple[collections.abc.Hashable, collections.abc.Hashable, float] – Pairs of node closer than the threshold in the format described above and the distance between the nodes.

Raises
`KeyError` – Raised if a node from the selection does not have a position.

Notes

Symetric node pairs are not deduplicated.

`vermouth.edge_tuning.`**`prune_edges_between_selections`**`(`*molecule*, *selection_a*, *selection_b*`)`

Remove edges which have their ends part of given selections.

An edge is removed if has one end that is part of 'selection_a', and the other end part of 'selection_b'.

Parameters

- **`molecule`** (*`networkx.Graph`*) – Molecule to prune in-place.
- **`selection_a`** (*`collections.abc.Iterable[collections.abc.Hashable]`*) – List of node keys from the molecule.
- **`selection_b`** (*`collections.abc.Iterable[collections.abc.Hashable]`*) – List of node keys from the molecule.

See also:

`prune_edges_with_selectors`

`vermouth.edge_tuning.`**`prune_edges_with_selectors`**`(`*molecule*, *selector_a*, *selector_b=None*`)`

Remove edges with the ends between selections defined by selectors.

An edge is removed if one of its end is part of the selection defined by 'selector_a', and its other end is part of the selection defined by 'selector_b'. A selector is a function that accept a node dictionary as argument and returns `True` if the node is part of the selection.

The 'selection_b' argment is optional. If it is `None`, then 'selector_a' is used for the selection at both ends.

Parameters

- **`molecule`** (*`networkx.Graph`*) – Molecule to prune in-place.
- **`selector_a`** (*`collections.abc.Callable`*) – A selector for one end of the edges.
- **`selector_b`** (*`collections.abc.Callable`*) – A selector for the second end of the edges. If set to `None`, then 'selector_a' is used for both ends.

See also:

`prune_edges_between_selections`

`vermouth.edge_tuning.`**`select_nodes_multi`**`(`*molecules*, *selector*`)`

Find the nodes that correspond to a selector among multiple molecules.

Runs a selector over multiple molecules. The selector must be a function that takes a node dictionary as argument and returns `True` if the node should be selected. The selection is yielded as tuples of a molecule indice from the molecule list input, and a key from the molecule.

Parameters

- **`molecule`** (*`collections.abc.Iterable[Molecule]`*) – A list of molecules.
- **`selector`** (*`collections.abc.Callable`*) – A selector function.

Yields

tuple[int, collections.abc.Hashable] – Molecule/key identifier for the selected nodes.

vermouth.ffinput module

Read .ff files.

The FF file format describes molecule components for a given force field. It is a test format devised for quick prototyping.

The format is built on top of a subset of the ITP format. Describing a block is done in the same way an ITP file describes a molecule.

class vermouth.ffinput.**FFDirector**(*force_field*)

Bases: *SectionLineParser*

COMMENT_CHAR = ';'

```
METH_DICT = {('citations',): (<function FFDirector._pase_ff_citations>, {}),
('link',): (<function FFDirector._link>, {'context_type': 'link'}), ('link',
'!SETTLE'): (<function FFDirector._interactions>, {'context_type': 'link'}),
('link', '!angle_restraints'): (<function FFDirector._interactions>,
{'context_type': 'link'}), ('link', '!angle_restraints_z'): (<function
FFDirector._interactions>, {'context_type': 'link'}), ('link', '!angles'):
(<function FFDirector._interactions>, {'context_type': 'link'}), ('link',
'!bonds'): (<function FFDirector._interactions>, {'context_type': 'link'}),
('link', '!constraints'): (<function FFDirector._interactions>, {'context_type':
'link'}), ('link', '!dihedral_restraints'): (<function FFDirector._interactions>,
{'context_type': 'link'}), ('link', '!dihedrals'): (<function
FFDirector._dih_interactions>, {'context_type': 'link'}), ('link',
'!distance_restraints'): (<function FFDirector._interactions>, {'context_type':
'link'}), ('link', '!exclusions'): (<function FFDirector._interactions>,
{'context_type': 'link'}), ('link', '!impropers'): (<function
FFDirector._interactions>, {'context_type': 'link'}), ('link',
'!orientation_restraints'): (<function FFDirector._interactions>, {'context_type':
'link'}), ('link', '!pairs'): (<function FFDirector._interactions>,
{'context_type': 'link'}), ('link', '!pairs_nb'): (<function
FFDirector._interactions>, {'context_type': 'link'}), ('link',
'!position_restraints'): (<function FFDirector._interactions>, {'context_type':
'link'}), ('link', '!virtual_sites2'): (<function FFDirector._interactions>,
{'context_type': 'link'}), ('link', '!virtual_sites3'): (<function
FFDirector._interactions>, {'context_type': 'link'}), ('link', '!virtual_sites4'):
(<function FFDirector._interactions>, {'context_type': 'link'}), ('link',
'!virtual_sitesn'): (<function FFDirector._interactions>, {'context_type':
'link'}), ('link', 'SETTLE'): (<function FFDirector._interactions>, {'context_type':
'link'}), ('link', 'angle_restraints'): (<function FFDirector._interactions>,
{'context_type': 'link'}), ('link', 'angle_restraints_z'): (<function
FFDirector._interactions>, {'context_type': 'link'}), ('link', 'angles'):
(<function FFDirector._interactions>, {'context_type': 'link'}), ('link', 'atoms'):
(<function FFDirector._link_atoms>, {}), ('link', 'bonds'): (<function
FFDirector._interactions>, {'context_type': 'link'}), ('link', 'citation'):
(<function FFDirector._parse_citation>, {'context_type': 'link'}), ('link',
'constraints'): (<function FFDirector._interactions>, {'context_type': 'link'}),
('link', 'debug'): (<function FFDirector._parse_log_entry>, {'context_type':
'link'}), ('link', 'dihedral_restraints'): (<function FFDirector._interactions>,
{'context_type': 'link'}), ('link', 'dihedrals'): (<function
FFDirector._dih_interactions>, {'context_type': 'link'}), ('link',
'distance_restraints'): (<function FFDirector._interactions>, {'context_type':
'link'}), ('link', 'edges'): (<function FFDirector._edges>, {'context_type':
'link', 'negate': False}), ('link', 'error'): (<function
FFDirector._parse_log_entry>, {'context_type': 'link'}), ('link', 'exclusions'):
(<function FFDirector._interactions>, {'context_type': 'link'}), ('link',
'features'): (<function FFDirector._link_features>, {'context_type': 'link'}),
('link', 'impropers'): (<function FFDirector._interactions>, {'context_type':
'link'}), ('link', 'info'): (<function FFDirector._parse_log_entry>,
{'context_type': 'link'}), ('link', 'molmeta'): (<function FFDirector._link>,
{'context_type': 'molmeta'}), ('link', 'non-edges'): (<function
FFDirector._edges>, {'context_type': 'link', 'negate': True}), ('link',
'orientation_restraints'): (<function FFDirector._interactions>, {'context_type':
'link'}), ('link', 'pairs'): (<function FFDirector._interactions>, {'context_type':
'link'}), ('link', 'pairs_nb'): (<function FFDirector._interactions>,
{'context_type': 'block'}), ('link', 'patterns'): (<function
FFDirector._link_patterns>, {'context_type': 'link'}), ('link',
'position_restraints'): (<function FFDirector._interactions>, {'context_type':
'link'}), ('link', 'virtual_sites2'): (<function FFDirector._interactions>,
{'context_type': 'link'}), ('link', 'virtual_sites3'): (<function
FFDirector._interactions>, {'context_type': 'link'}), ('link', 'virtual_sites4'):
(<function FFDirector._interactions>, {'context_type': 'link'}), ('link',
```

A dict of all known parser methods, mapping section names to the function to be called and the associated keyword arguments.

`finalize_section`(*previous_section*, *ended_section*)

Called once a section is finished. It appends the current_links list to the links and update the block dictionary with current_block. Thereby it finishes the reading a given section.

Parameters

- **previous_section** (*list[str]*) – The last parsed section.
- **ended_section** (*list[str]*) – The sections that have been ended.

`get_context`(*context_type=''*)

`has_context`()

`interactions_natoms = {'SETTLE': 1, 'angle_restraints': 4, 'angle_restraints_z': 2, 'angles': 3, 'bonds': 2, 'constraints': 2, 'dihedral_restraints': 4, 'dihedrals': 4, 'distance_restraints': 2, 'impropers': 4, 'orientation_restraints': 2, 'pairs': 2, 'pairs_nb': 2, 'position_restraints': 1, 'virtual_sites2': 3, 'virtual_sites3': 4, 'virtual_sites4': 5}`

`parse_header`(*line*, *lineno=0*)

Parses a section header with line number *lineno*. Sets `vermouth.parser_utils.SectionLineParser.section` when applicable. Does not check whether *line* is a valid section header.

Parameters

- **line** (*str*) –
- **lineno** (*str*) –

Returns
The result of calling `finalize_section()`, which is called if a section ends.

Return type
object

Raises
KeyError – If the section header is unknown.

`vermouth.ffinput.read_ff`(*lines*, *force_field*)

vermouth.file_writer module

Provides the DeferredFileWriter, which allow writing of files without affecting existing files, until it is clear the written changes are correct.

`class vermouth.file_writer.DeferredFileWriter`(**args*, ***kwargs*)

Bases: `object`

A singleton class/object that is intended to prevent writing output to files that is invalid, due to e.g. warnings further down the pipeline.

If this class is used to open a file for writing, a temporary file is created and returned instead. Once it's clear the output produced is valid the `write()` method can be used to finalize the written changes by moving them to their intended destination. If a file with that name already exists it is backed up according to the Gromacs scheme.

close()

Remove all produced temporary files.

open(*filename*, *mode='r'*, **args*, ***kwargs*)

If mode is either 'w' or 'a', opens and returns a handle to a temporary file. If mode is 'r' opens and returns a handle to the file specified.

Once `write()` is called the changes written to all files opened this way are propagated to their final destination.

Parameters

- **filename** (`os.PathLike`) – The final name of the file to be opened.
- **mode** (`str`) – The mode in which the file is to be opened.
- ***args** (`collections.abc.Iterable`) – Passed to `os.fdopen()`.
- ****kwargs** (`dict`) – Passed to `os.fdopen()`.

Returns

An opened file

Return type

io.IOBase

write()

Finalize writing all open files by moving the created temporary files to their final destinations.

Existing file destinations will be backed up according to the Gromacs scheme.

class vermouth.file_writer.**Singleton**

Bases: `type`

Metaclass for creating singleton objects. Taken from[1].

vermouth.file_writer.**deferred_open**(*filename*, *mode='r'*, **args*, ***kwargs*)

If mode is either 'w' or 'a', opens and returns a handle to a temporary file. If mode is 'r' opens and returns a handle to the file specified.

Once `write()` is called the changes written to all files opened this way are propagated to their final destination.

Parameters

- **filename** (`os.PathLike`) – The final name of the file to be opened.
- **mode** (`str`) – The mode in which the file is to be opened.
- ***args** (`collections.abc.Iterable`) – Passed to `os.fdopen()`.
- ****kwargs** (`dict`) – Passed to `os.fdopen()`.

Returns

An opened file

Return type

io.IOBase

[1] https://stackoverflow.com/questions/50566934/why-is-this-singleton-implementation-not-thread-safe/50567397

vermouth.forcefield module

Provides a class used to describe a forcefield and all associated data.

`class vermouth.forcefield.ForceField`(*directory=None*, *name=None*)

Bases: `object`

Description of a force field.

A force field can be created empty or read from a directory. In any case, a force field must be named. If read from a directory, the base name of the directory is used as force field name, unless the *name* attribute is provided. If the force field is created empty, then *name* must be provided.

Parameters

- **directory** (*str or pathlib.Path, optional*) – A directory to read the force field from.
- **name** (*str, optional*) – The name of the force field.

`blocks`

Type
dict

`links`

Type
list

`modifications`

Type
dict

`renamed_residues`

Type
dict

`name`

Type
str

`variables`

Type
dict

`property features`

List the features declared by the links.

Return type
set

`has_feature`(*feature*)

Test if a feature is declared by the links.

Parameters
feature (*str*) – The name of the feature of interest.

Return type
bool

read_from(*directory*)

Populate or update the force field from a directory.

The provided directory must contain a subdirectory with the same name as the force field.

property reference_graphs

Returns all known blocks.

Return type
dict

vermouth.forcefield.**find_force_fields**(*directory*, *force_fields=None*)

Read all the force fields in the given directory.

A force field is defined as a directory that contains at least one RTP file. The name of the force field is the base name of the directory.

If the force field argument is not `None`, then it must be a dictionary with force field names as keys and instances of *ForceField* as values. The force fields in the dictionary will be updated if force fields with the same names are found in the directory.

Parameters

- **directory** (*pathlib.Path or str*) – The path to the directory containing the force fields.
- **force_fields** (*dict*) – A dictionary of force fields to update.

Returns
A dictionary of force fields read or updated. Keys are force field names as strings, and values are instances of *ForceField*. If a dictionary was provided as the "force_fields" argument, then the returned dictionary is the same instance as the one provided but with updated content.

Return type
dict

vermouth.forcefield.**get_native_force_field**(*name*)

Get a force field from the distributed library knowing its name.

Parameters
name (*str*) – The name of the requested force field.

Return type
ForceField

Raises
`KeyError` – There is no force field with the requested name in the distributed library.

vermouth.forcefield.**iter_force_field_files**(*directory*, *extensions=dict_keys(['.rtp', '.ff', '.bib'])*)

Returns a generator over the path of all the force field files in the directory.

vermouth.geometry module

Geometric operations.

`vermouth.geometry.`**`angle`**(*vector_ba*, *vector_bc*)

Calculate the angle in radians between two vectors.

The function assumes the following situation:

```
  B
 / \
A   C
```

It returns the angle between BA and BC.

`vermouth.geometry.`**`dihedral`**(*coordinates*)

Calculate the dihedral angle in radians.

Parameters
`coordinates` (*`numpy.ndarray`*) – The coordinates of 4 points defining the dihedral angle. Each row corresponds to a point, and each column to a dimension.

Returns
The calculated angle between -pi and +pi.

Return type
float

`vermouth.geometry.`**`dihedral_phase`**(*coordinates*)

Calculate a dihedral angle in radians with a -pi phase correction.

Parameters
`coordinates` (*`numpy.ndarray`*) – The coordinates of 4 points defining the dihedral angle. Each row corresponds to a point, and each column to a dimension.

Returns
The calculated angle between -pi and +pi.

Return type
float

See also:

`dihedral`
Calculate a dihedral angle.

`vermouth.geometry.`**`distance_matrix`**(*coordinates_a*, *coordinates_b*)

Compute a distance matrix between two set of points.

Notes

This function does **not** account for periodic boundary conditions.

Parameters

- **coordinates_a** (*numpy.ndarray*) – Coordinates of the points in the selections. Each row must correspond to a point and each column to a dimension.
- **coordinates_b** (*numpy.ndarray*) – Coordinates of the points in the selections. Each row must correspond to a point and each column to a dimension.

Returns
Rows correspond to the points from *coordinates_a*, columns correspond from *coordinates_b*.

Return type
numpy.ndarray

vermouth.graph_utils module

class vermouth.graph_utils.**MappingGraphMatcher**(**args*, *edge_match=None*, *node_match=None*, ***kwargs*)

Bases: GraphMatcher

semantic_feasibility(*G1_node*, *G2_node*)

Returns True if mapping G1_node to G2_node is semantically feasible. Adapted from networkx.algorithms.isomorphism.vf2userfunc._semantic_feasibility.

vermouth.graph_utils.**add_element_attr**(*molecule*)

Adds an element attribute to every node in *molecule*, based on that node's atomname attribute.

Parameters
molecule (*networkx.Graph*) – The graph of which nodes should get an element attribute.

Raises
ValueError – If no element could be guessed for a node.

vermouth.graph_utils.**categorical_cartesian_product**(*graph1*, *graph2*, *attributes=()*)

vermouth.graph_utils.**categorical_maximum_common_subgraph**(*graph1*, *graph2*, *attributes=()*)

vermouth.graph_utils.**categorical_modular_product**(*graph1*, *graph2*, *attributes=()*)

vermouth.graph_utils.**collect_residues**(*graph*, *attrs=('chain', 'resid', 'resname', 'insertion_code')*)

Creates groups of indices based on the node attributes with keys *attrs*. All nodes in graph will be part of exactly one group.

Parameters

- **graph** (networkx.Graph) – The graph whose node indices should be grouped.
- **attrs** (Sequence) – The attribute keys that should be used to group node indices. The associated values should be hashable.

Returns
The keys are the found node attributes, the values the associated node indices.

Return type
dict[tuple, set]

`vermouth.graph_utils.`**`get_attrs`**(*node*, *attrs*)

Returns multiple values from a dictionary in order.

Parameters

- **`node`** (*`dict`*) – The dict from which items should be taken.
- **`attrs`** (*`collections.abc.Iterable`*) – The keys which values should be taken.

Returns

A tuple containing the value of every key in attrs in the same order, where missing values are *None*.

Return type

tuple

`vermouth.graph_utils.`**`make_residue_graph`**(*graph*, *attrs=('chain', 'resid', 'resname', 'insertion_code')*)

Create a new graph based on *graph*, where nodes with identical attribute values for the attribute names in *attrs* will be contracted into a single, coarser node. With the default arguments it will create a graph with one node per residue. Resulting (coarse) nodes will have the same attributes as the constructing nodes, but only those that have identical values. In addition, they'll have attributes 'graph', 'nnodes', 'nedges' and 'density'.

Parameters

- **`graph`** (*`networkx.Graph`*) – The graph to condense.
- **`attrs`** (*`collections.abc.Iterable[collections.abc.Hashable]`*) – The node attributes that determine node equivalence.

Returns

The resulting coarser graph, where equivalent nodes are contracted to a single node.

Return type

networkx.Graph

`vermouth.graph_utils.`**`partition_graph`**(*graph*, *partitions*)

Create a new graph based on *graph*, where nodes are aggregated based on *partitions*, similar to the networkx *quotient_graph*, except that it only accepts pre-made partitions, and edges are not given a 'weight' attribute. Much fast than the quotient_graph, since it creates edges based on existing edges rather than trying all possible combinations.

Parameters

- **`graph`** (*`networkx.Graph`*) – The graph to partition
- **`partitions`** (*`collections.abc.Iterable[collections.abc.Iterable[collections.abc.Hashable]]`*) – E.g. a list of lists of node indices, describing the partitions. Will be sorted by lowest index.

Returns

The coarser graph.

Return type

networkx.Graph

`vermouth.graph_utils.`**`rate_match`**(*residue*, *bead*, *match*)

A helper function which rates how well `match` describes the isomorphism between `residue` and `bead` based on the number of matching atomnames.

Parameters

- **`residue`** (*`networkx.Graph`*) – A graph. Required node attributes:

atomname
The name of an atom.

- **bead** (*networkx.Graph*) – A subgraph of `residue` where the isomorphism is described by `match`. Required node attributes:

  atomname
  The name of an atom.

Returns
The number of entries in match where the atomname in `residue` matches the atomname in `bead`.

Return type
int

vermouth.ismags module

ISMAGS Algorithm

Provides a Python implementation of the ISMAGS algorithm.[1]

It is capable of finding (subgraph) isomorphisms between two graphs, taking the symmetry of the subgraph into account. In most cases the VF2 algorithm is faster (at least on small graphs) than this implementation, but in some cases there is an exponential number of isomorphisms that are symmetrically equivalent. In that case, the ISMAGS algorithm will provide only one solution per symmetry group.

In addition, this implementation also provides an interface to find the largest common induced subgraph[2] between any two graphs, again taking symmetry into account. Given *graph* and *subgraph* the algorithm will remove nodes from the *subgraph* until *subgraph* is isomorphic to a subgraph of *graph*. Since only the symmetry of *subgraph* is taken into account it is worth thinking about how you provide your graphs:

```
>>> graph1 = nx.path_graph(4)
>>> graph2 = nx.star_graph(3)
>>> ismags = isomorphism.ISMAGS(graph1, graph2)
>>> ismags.is_isomorphic()
False
>>> list(ismags.largest_common_subgraph())
[{1: 0, 0: 1, 2: 2}, {2: 0, 1: 1, 3: 2}]
>>> ismags2 = isomorphism.ISMAGS(graph2, graph1)
>>> list(ismags2.largest_common_subgraph())
[{1: 0, 0: 1, 2: 2},
 {1: 0, 0: 1, 3: 2},
 {2: 0, 0: 1, 1: 2},
 {2: 0, 0: 1, 3: 2},
 {3: 0, 0: 1, 1: 2},
 {3: 0, 0: 1, 2: 2}]
```

However, when not taking symmetry into account, it doesn't matter:

```
>>> list(ismags.largest_common_subgraph(symmetry=False))
[{1: 0, 0: 1, 2: 3},
 {1: 0, 2: 1, 0: 3},
```

(continues on next page)

[1] M. Houbraken, S. Demeyer, T. Michoel, P. Audenaert, D. Colle, M. Pickavet, "The Index-Based Subgraph Matching Algorithm with General Symmetries (ISMAGS): Exploiting Symmetry for Faster Subgraph Enumeration", PLoS One 9(5): e97896, 2014. https://doi.org/10.1371/journal.pone.0097896

[2] https://en.wikipedia.org/wiki/Maximum_common_induced_subgraph

(continued from previous page)

```
 {2: 0, 1: 1, 3: 3},
 {2: 0, 3: 1, 1: 3},
 {1: 0, 0: 2, 2: 3},
 {1: 0, 2: 2, 0: 3},
 {2: 0, 1: 2, 3: 3},
 {2: 0, 3: 2, 1: 3},
 {1: 0, 0: 1, 2: 2},
 {1: 0, 2: 1, 0: 2},
 {2: 0, 1: 1, 3: 2},
 {2: 0, 3: 1, 1: 2}]
>>> list(ismags2.largest_common_subgraph(symmetry=False))
[{1: 0, 0: 1, 2: 3},
 {1: 0, 2: 1, 0: 3},
 {2: 0, 1: 1, 3: 3},
 {2: 0, 3: 1, 1: 3},
 {1: 0, 0: 2, 2: 3},
 {1: 0, 2: 2, 0: 3},
 {2: 0, 1: 2, 3: 3},
 {2: 0, 3: 2, 1: 3},
 {1: 0, 0: 1, 2: 2},
 {1: 0, 2: 1, 0: 2},
 {2: 0, 1: 1, 3: 2},
 {2: 0, 3: 1, 1: 2}]
```

Notes

- The current implementation works for undirected graphs only. The algorithm in general should work for directed graphs as well though.
- Node keys for both provided graphs need to be fully orderable as well as hashable.
- Node and edge equality is assumed to be transitive: if A is equal to B, and B is equal to C, then A is equal to C.

References

`class vermouth.ismags.ISMAGS(`*graph*, *subgraph*, *node_match=None*, *edge_match=None*, *cache=None*`)`

Bases: `object`

Implements the ISMAGS subgraph matching algorith.[Page 98, 1] ISMAGS stands for "Index-based Subgraph Matching Algorithm with General Symmetries". As the name implies, it is symmetry aware and will only generate non-symmetric isomorphisms.

Notes

The implementation imposes additional conditions compared to the VF2 algorithm on the graphs provided and the comparison functions (`node_equality` and `edge_equality`):

- Node keys in both graphs must be orderable as well as hashable.
- Equality must be transitive: if A is equal to B, and B is equal to C, then A must be equal to C.

`graph`

Type
networkx.Graph

`subgraph`

Type
networkx.Graph

`node_equality`

The function called to see if two nodes should be considered equal. It's signature looks like this: `f(graph1: networkx.Graph, node1, graph2: networkx.Graph, node2) -> bool`. *node1* is a node in *graph1*, and *node2* a node in *graph2*. Constructed from the argument *node_match*.

Type
collections.abc.Callable

`edge_equality`

The function called to see if two edges should be considered equal. It's signature looks like this: `f(graph1: networkx.Graph, edge1, graph2: networkx.Graph, edge2) -> bool`. *edge1* is an edge in *graph1*, and *edge2* an edge in *graph2*. Constructed from the argument *edge_match*.

Type
collections.abc.Callable

Parameters

- **graph** (*networkx.Graph*) –
- **subgraph** (*networkx.Graph*) –
- **node_match** (*collections.abc.Callable or None*) – Function used to determine whether two nodes are equivalent. Its signature should look like `f(n1: dict, n2: dict) -> bool`, with *n1* and *n2* node property dicts. See also `categorical_node_match()` and friends. If *None*, all nodes are considered equal.
- **edge_match** (*collections.abc.Callable or None*) – Function used to determine whether two edges are equivalent. Its signature should look like `f(e1: dict, e2: dict) -> bool`, with *e1* and *e2* edge property dicts. See also `categorical_edge_match()` and friends. If *None*, all edges are considered equal.
- **cache** (*collections.abc.Mapping*) – A cache used for caching graph symmetries.

`analyze_symmetry`(*graph*, *node_partitions*, *edge_colors*)

Find a minimal set of permutations and corresponding co-sets that describe the symmetry of `subgraph`.

Returns

- *set[frozenset]* – The found permutations. This is a set of frozenset of pairs of node keys which can be exchanged without changing `subgraph`.

- *dict[collections.abc.Hashable, set[collections.abc.Hashable]]* – The found co-sets. The co-sets is a dictionary of {node key: set of node keys}. Every key-value pair describes which *values* can be interchanged without changing nodes less than *key*.

find_isomorphisms(*symmetry=True*)

Find all subgraph isomorphisms between `subgraph` <= `graph`.

Parameters
symmetry (*bool*) – Whether symmetry should be taken into account. If False, found isomorphisms may be symmetrically equivalent.

Yields
dict – The found isomorphism mappings of {graph_node: subgraph_node}.

is_isomorphic(*symmetry=False*)

Returns True if `graph` is isomorphic to `subgraph` and False otherwise.

Return type
bool

isomorphisms_iter(*symmetry=True*)

Does the same as `find_isomorphisms()` if `graph` and `subgraph` have the same number of nodes.

largest_common_subgraph(*symmetry=True*)

Find the largest common induced subgraphs between `subgraph` and `graph`.

Parameters
symmetry (*bool*) – Whether symmetry should be taken into account. If False, found largest common subgraphs may be symmetrically equivalent.

Yields
dict – The found isomorphism mappings of {graph_node: subgraph_node}.

subgraph_is_isomorphic(*symmetry=False*)

Returns True if a subgraph of `graph` is isomorphic to `subgraph` and False otherwise.

Return type
bool

subgraph_isomorphisms_iter(*symmetry=True*)

Alternative name for `find_isomorphisms()`.

vermouth.ismags.**intersect**(*collection_of_sets*)

Given an collection of sets, returns the intersection of those sets.

Parameters
collection_of_sets (*collections.abc.Collection[set]*) – A collection of sets.

Returns
An intersection of all sets in *collection_of_sets*. Will have the same type as the item initially taken from *collection_of_sets*.

Return type
set

vermouth.ismags.**make_partitions**(*items, test*)

Partitions items into sets based on the outcome of `test(item1, item2)`. Pairs of items for which *test* returns *True* end up in the same set.

Parameters

- **items** (*collections.abc.Iterable[collections.abc.Hashable]*) – Items to partition
- **test** (*collections.abc.Callable[collections.abc.Hashable, collections.abc.Hashable]*) – A function that will be called with 2 arguments, taken from items. Should return *True* if those 2 items need to end up in the same partition, and *False* otherwise.

Returns

A list of sets, with each set containing part of the items in *items*, such that `all(test(*pair) for pair in itertools.combinations(set, 2)) == True`

Return type

list[set]

Notes

The function *test* is assumed to be transitive: if `test(a, b)` and `test(b, c)` return `True`, then `test(a, c)` must also be `True`.

`vermouth.ismags.`**`partition_to_color`**(*partitions*)

Creates a dictionary with for every item in partition for every partition in partitions the index of partition in partitions.

Parameters

partitions (*collections.abc.Sequence[collections.abc.Iterable]*) – As returned by *make_partitions()*.

Return type

dict[collections.abc.Hashable, int]

vermouth.log_helpers module

Provide some helper classes to allow new style brace formatting for logging and processing the *type* keyword.

class `vermouth.log_helpers.`**`BipolarFormatter`**(*low_formatter*, *high_formatter*, *cutoff*, *logger=None*)

Bases: `object`

A logging formatter that formats using either *low_formatter* or *high_formatter* depending on the *logger*'s effective loglevel.

Parameters

- **low_formatter** (*logging.Formatter*) – The formatter used if *cutoff* <= *logger.getEffectiveLevel()*.
- **high_formatter** (*logging.Formatter*) – The formatter used if *cutoff* > *logger.getEffectiveLevel()*.
- **cutoff** (*int*) – The cutoff used to decide whether the low or high formatter is used.
- **logger** (*logging.Logger*) – The logger whose effective loglevel is used. Defaults to `logging.getLogger()`.

class `vermouth.log_helpers.`**`CountingHandler`**(**args*, *type_attribute='type'*, *default_type='general'*, ***kwargs*)

Bases: `NullHandler`

A logging handler that counts the number of times a specific type of message is logged per loglevel.

Parameters

- **type_attribute** (*str*) – The name of the attribute carrying the type.
- **default_type** (*str*) – The type of message if none is provided.

handle(*record*)

Handle a log record by counting it.

number_of_counts_by(*level=None*, *type=None*)

Return the number of logging calls counted, filtered by level and type.

Parameters

- **level** – Only count log events of this level.
- **type** – Only count log events of this type.

Returns

The number of events counted.

Return type

int

class vermouth.log_helpers.**Message**(*fmt*, *args*, *kwargs*)

Bases: `object`

Class that defers string formatting until it's `__str__` method is called.

class vermouth.log_helpers.**PassingLoggerAdapter**(*logger*, *extra=None*)

Bases: `LoggerAdapter`

Helper class that is actually capable of chaining multiple LoggerAdapters.

addHandler(**args*, ***kwargs*)

log(*level*, *msg*, **args*, ***kwargs*)

property manager

```
Logger.manager = <logging.Manager object>
```

process(*msg*, *kwargs*)

class vermouth.log_helpers.**StyleAdapter**(*logger*, *extra=None*)

Bases: *PassingLoggerAdapter*

Logging adapter that encapsulate messages in *Message*, allowing `{}` style formatting.

log(*level*, *msg*, **args*, ***kwargs*)

class vermouth.log_helpers.**TypeAdapter**(*logger*, *extra=None*, *default_type='general'*)

Bases: *PassingLoggerAdapter*

Logging adapter that takes the *type* keyword argument passed to logging calls and passes adds it to the *extra* attribute.

Parameters

- **logger** (*logging.Logger or logging.LoggerAdapter*) – As described in `logging.LoggerAdapter`.
- **extra** (*dict*) – As described in `logging.LoggerAdapter`.
- **default_type** (*str*) – The type of the messages if none is given.

process(*msg*, *kwargs*)

vermouth.log_helpers.**get_logger**(*name*)

Convenience method that wraps a *TypeAdapter* around `logging.getLogger(name)`

Parameters

name (*str*) – The name of the logger to get. Passed to `logging.getLogger()`. Should probably be `__name__`.

vermouth.log_helpers.**ignore_warnings_and_count**(*counter*, *specifications*, *level=30*)

Count the warnings after deducting the ones to ignore.

Warnings to ignore are specified as tuple `(<warning-type>, <count>)`. The count is `None` if all warnings of that type should be ignored, and the warning type is `None` to indicate that the count is about all not specified types.

In case the same type is specified more than once, only the higher count is used.

vermouth.map_input module

Read force field to force field mappings.

vermouth.map_input.**combine_mappings**(*known_mappings*, *partial_mapping*)

Update a collection of mappings.

Add the mappings from the 'partial_mapping' argument into the 'known_mappings' collection. Both arguments are collections of mappings similar to the output of the *read_mapping_directory()* function. They are dictionary with 3 levels of keys: the name of the initial force field, the name of the target force field, and the name of the block. The values in the third level dictionary are tuples of (mapping, weights, extra).

If a force field appears in 'partial_mapping' that is not in 'known_mappings', then it is added. For existing pairs of initial and target force fields, the blocks are updated and the version in 'partial_mapping' is kept in priority.

Parameters

- **known_mappings** (*dict*) – Collection of mapping to update **in-place**.
- **partial_mapping** (*dict*) – Collection of mappings to update from.

vermouth.map_input.**generate_all_self_mappings**(*force_fields*)

Generate self mappings for a list of force fields.

Parameters

force_fields (*collections.abc.Iterable*) – List of instances of *ForceField*.

Returns

A collection of mappings formatted as the output of the *read_mapping_directory()* function.

Return type

dict

vermouth.map_input.**generate_self_mappings**(*blocks*)

Generate self mappings from a collection of blocks.

A self mapping is a mapping that maps a force field to itself. Applying such mapping is applying a neutral transformation.

Parameters

blocks (*dict[str, networkx.Graph]*) – A dictionary of blocks with block names as keys and the blocks themselves as values. The blocks must be instances of `networkx.Graph` with each node having an 'atomname' attribute.

Returns
mappings – A dictionary of mappings where the keys are the names of the blocks, and the values are tuples like (mapping, weights, extra).

Return type
dict[str, tuple]

Raises
`KeyError` – Raised if a node does not have am 'atomname' attribute.

See also:

`read_mapping_file`
Read a mapping from a file.

`generate_all_self_mappings`
Generate self mappings for a list of force fields.

`vermouth.map_input.`**`make_mapping_object`**(*from_block*, *to_block*, *mapping*, *weights*, *extra*, *name_to_index*)

Convenience method for creating modern *`vermouth.map_parser.Mapping`* objects from old style mapping information.

Parameters

- **`from_blocks`** (*`collections.abc.Iterable`*[`vermouth.molecule.Block`]) –
- **`to_blocks`** (*`collections.abc.Iterable`*[`vermouth.molecule.Block`]) –
- **`mapping`** (*`dict[tuple[int, str], list[tuple[int, str]]]`*) – Old style mapping describing what (resid, atomname) maps to what (resid, atomname)
- **`weights`** (*`dict[tuple[int, str], dict[tuple[int, str], float]]`*) – Old style weights, mapping (resid, atomname), (resid, atomname) to a weight.
- **`extra`** (*`tuple`*) –
- **`name_to_index`** (*`dict[str, dict[str, dict[str, collections.abc.Hashable]]]`*) – Dict force field names, block names, atomnames to node indices.

Returns
The created mapping.

Return type
vermouth.map_parser.Mapping

`vermouth.map_input.`**`read_backmapping_file`**(*lines*, *force_fields*)

Partial reader for modified Backward mapping files.

Read mappings from a Backward mapping file. Not all fields are supported, only the "molecule" and the "atoms" fields are read. If not explicitly specified, the origin force field for a molecule is assumed to be "universal", and the destination force field is assumed to be "martini22".

The mapping collection is a 3 level dictionary where the first key is the name of the initial force field, the second key is the name of the destination force field, and the third key is the name of the molecule.

Parameters

- **`lines`** (*`collections.abc.Iterable[str]`*) – Collection of lines to read.

- **force_fields** (*dict[str, vermouth.forcefield.ForceField]*) – Dict of known force fields.

Return type
dict

`vermouth.map_input.`**`read_mapping_directory`**`(`*directory*, *force_fields*`)`

Read all the mapping files in a directory.

The resulting mapping collection is a 3-level dict where the keys are: * the name of the origin force field * the name of the destination force field * the name of the residue

The values after these 3 levels is a mapping dict where the keys are the atom names in the origin force field and the values are lists of names in the destination force field.

Parameters

- **directory** (*str*) – The path to the directory to search. Files with a '.backmap' extension will be read. There is no recursive search.
- **force_fields** (*dict[str, ForceField]*) – Dict of known forcefields

Returns
A collection of mappings.

Return type
dict

`vermouth.map_input.`**`read_mapping_file`**`(`*lines*, *force_fields*`)`

vermouth.map_parser module

Contains the Mapping object and the associated parser.

class `vermouth.map_parser.`**`Mapping`**`(`*block_from*, *block_to*, *mapping*, *references*, *ff_from=None*, *ff_to=None*, *extra=()*, *normalize_weights=False*, *type='block'*, *names=()*`)`

Bases: `object`

A mapping object that describes a mapping from one resolution to another.

block_from

The graph which this *Mapping* object can transform.

Type
networkx.Graph

block_to

The *vermouth.molecule.Block* we can transform to.

Type
vermouth.molecule.Block

references

A mapping of node keys in *block_to* to node keys in *block_from* that describes which node in blocks_from should be taken as a reference when determining node attributes for nodes in block_to.

Type
collections.abc.Mapping

ff_from

The forcefield of *block_from*.

Type

vermouth.forcefield.ForceField

ff_to

The forcefield of *block_to*.

Type

vermouth.forcefield.ForceField

names

The names of the mapped blocks.

Type

tuple[str]

mapping

The actual mapping that describes for every node key in *block_from* to what node key in *block_to* it contributes to with what weight. `{node_from: {node_to: weight, ...}, ...}`.

Type

dict[collections.abc.Hashable, dict[collections.abc.Hashable, float]]

Note: Only nodes described in *mapping* will be used.

Parameters

- **block_from** (*networkx.Graph*) – As per *block_from*.
- **block_to** (vermouth.molecule.Block) – As per *block_to*.
- **mapping** (*dict[collections.abc.Hashable, dict[collections.abc.Hashable, float]]*) – As per *mapping*.
- **references** (*collections.abc.Mapping*) – As per *references*.
- **ff_from** (vermouth.forcefield.ForceField) – As per *ff_from*.
- **ff_to** (vermouth.forcefield.ForceField) – As per *ff_to*.
- **extra** (*tuple*) – Extra information to be attached to *block_to*.
- **normalize_weights** (*bool*) – Whether the weights should be normalized such that the sum of the weights of nodes mapping to something is 1.
- **names** (*tuple*) – As per *names*.

map(*graph*, *node_match=None*, *edge_match=None*)

Performs the partial mapping described by this object on *graph*. It first find the induced subgraph isomorphisms between *graph* and *block_from*, after which it will process the found isomorphisms according to *mapping*.

None of the yielded dictionaries will refer to node keys of *block_from*. Instead, those will be translated to node keys of *graph* based on the found isomorphisms.

Note: Only nodes described in *mapping* will be used in the isomorphism.

Parameters

- `graph` (*networkx.Graph*) – The graph on which this partial mapping should be applied.
- `node_match` (*collections.abc.Callable or None*) – A function that should take two dictionaries with node attributes, and return *True* if those nodes should be considered equal, and *False* otherwise. If None, all nodes will be considered equal.
- `edge_match` (*collections.abc.Callable or None*) – A function that should take six arguments: two graphs, and four node keys. The first two node keys will be in the first graph and share an edge; and the last two node keys will be in the second graph and share an edge. Should return *True* if a pair of edges should be considered equal, and *False* otherwise. If None, all edges will be considered equal.

Yields

- *dict[collections.abc.Hashable, dict[collections.abc.Hashable, float]]* – the correspondence between nodes in *graph* and nodes in `block_to`, with the associated weights.
- *vermouth.molecule.Block* – `block_to`.
- *dict* – `references` on which `mapping` has been applied.

property reverse_mapping

The reverse of `mapping`. `{node_to: {node_from: weight, ...}, ...}`

class vermouth.map_parser.MappingBuilder

Bases: `object`

An object that is in charge of building the arguments needed to create a `Mapping` object. It's attributes describe the information accumulated so far.

mapping

Type
collections.defaultdict

blocks_from

Type
None or *vermouth.molecule.Block*

blocks_to

Type
None or *vermouth.molecule.Block*

ff_from

Type
None or *vermouth.forcefield.ForceField*

ff_to

Type
None or *vermouth.forcefield.ForceField*

names

Type
list

references

Type
dict

add_block_from(*block*)

Add a block to *blocks_from*. In addition, apply any 'replace' operation described by nodes on themselves:

```
{'atomname': 'C', 'charge': 0, 'replace': {'charge': -1}}
```

becomes:

```
{'atomname': 'C', 'charge': -1}
```

Parameters
block (vermouth.molecule.Block) – The block to add.

add_block_to(*block*)

Add a block to *blocks_to*.

Parameters
block (vermouth.molecule.Block) – The block to add.

add_edge_from(*attrs1*, *attrs2*, *edge_attrs*)

Add a single edge to *blocks_from* between two nodes in *blocks_from* described by *attrs1* and *attrs2*. The nodes described should not be the same.

Parameters

- **attrs1** (*dict[str]*) – The attributes that uniquely describe a node in *blocks_from*
- **attrs2** (*dict[str]*) – The attributes that uniquely describe a node in *blocks_from*
- **edge_attrs** (*dict[str]*) – The attributes that should be assigned to the new edge.

add_edge_to(*attrs1*, *attrs2*, *edge_attrs*)

Add a single edge to *blocks_to* between two nodes in *blocks_to* described by *attrs1* and *attrs2*. The nodes described should not be the same.

Parameters

- **attrs1** (*dict[str]*) – The attributes that uniquely describe a node in *blocks_to*
- **attrs2** (*dict[str]*) – The attributes that uniquely describe a node in *blocks_to*
- **edge_attrs** (*dict[str]*) – The attributes that should be assigned to the new edge.

add_mapping(*attrs_from*, *attrs_to*, *weight*)

Add part of a mapping to *mapping*. *attrs_from* uniquely describes a node in *blocks_from* and *attrs_to* a node in *blocks_to*. Adds a mapping between those nodes with the given *weight*.

Parameters

- **attrs_from** (*dict[str]*) – The attributes that uniquely describe a node in *blocks_from*
- **attrs_to** (*dict[str]*) – The attributes that uniquely describe a node in *blocks_to*
- **weight** (*float*) – The weight associated with this partial mapping.

add_name(*name*)

Add a name to the mapping.

Parameters
name (*str*) – The name to add

add_node_from(*attrs*)

Add a single node to *blocks_from*.

Parameters
attrs (*dict[str]*) – The attributes the new node should have.

add_node_to(*attrs*)

Add a single node to *blocks_to*.

Parameters
attrs (*dict[str]*) – The attributes the new node should have.

add_reference(*attrs_to*, *attrs_from*)

Add a reference to *references*.

Parameters

- **attrs_to** (*dict[str]*) – The attributes that uniquely describe a node in *blocks_to*
- **attrs_from** (*dict[str]*) – The attributes that uniquely describe a node in *blocks_from*

from_ff(*ff_name*)

Sets *ff_from*

Parameters
ff_name –

get_mapping(*type*)

Instantiate a *Mapping* object with the information accumulated so far, and return it.

Returns
The mapping object made from the accumulated information.

Return type
Mapping

reset()

Reset the object to a clean initial state.

to_ff(*ff_name*)

Sets *ff_to*

Parameters
ff_name –

class vermouth.map_parser.**MappingDirector**(*force_fields*, *builder=None*)

Bases: *SectionLineParser*

A director in charge of parsing the new mapping format. It constructs a new *Mapping* object by calling methods of it's builder (default *MappingBuilder*) with the correct arguments.

Parameters

- **force_fields** (*dict[str, ForceField]*) – Dict of known force fields.
- **builder** (MappingBuilder) –

`builder`

The builder used to build the *Mapping* object. By default *MappingBuilder*.

`identifiers`

All known identifiers at this point. The key is the actual identifier, prefixed with either "to_" or "from_", and the values are the associated node attributes.

Type
dict[str, dict[str]]

`section`

The name of the section currently being processed.

Type
str

`from_ff`

The name of the forcefield from which this mapping describes a transfomation.

Type
str

`to_ff`

The name of the forcefield to which this mapping describes a transfomation.

Type
str

`macros`

A dictionary of known macros.

Type
dict[str, str]

`COMMENT_CHAR = ';'`

The character that starts a comment.

METH_DICT = {('block', 'from'): (<function MappingDirector._ff>, {'direction': 'from'}), ('block', 'from blocks'): (<function MappingDirector._blocks>, {'direction': 'from', 'map_type': 'block'}), ('block', 'from edges'): (<function MappingDirector._edges>, {'direction': 'from'}), ('block', 'from nodes'): (<function MappingDirector._nodes>, {'direction': 'from'}), ('block', 'mapping'): (<function MappingDirector._mapping>, {}), ('block', 'reference atoms'): (<function MappingDirector._reference_atoms>, {}), ('block', 'to'): (<function MappingDirector._ff>, {'direction': 'to'}), ('block', 'to blocks'): (<function MappingDirector._blocks>, {'direction': 'to', 'map_type': 'block'}), ('block', 'to edges'): (<function MappingDirector._edges>, {'direction': 'to'}), ('block', 'to nodes'): (<function MappingDirector._nodes>, {'direction': 'to'}), ('macros',): (<function SectionLineParser._macros>, {}), ('modification', 'from'): (<function MappingDirector._ff>, {'direction': 'from'}), ('modification', 'from blocks'): (<function MappingDirector._blocks>, {'direction': 'from', 'map_type': 'modification'}), ('modification', 'from edges'): (<function MappingDirector._edges>, {'direction': 'from'}), ('modification', 'from nodes'): (<function MappingDirector._nodes>, {'direction': 'from'}), ('modification', 'mapping'): (<function MappingDirector._mapping>, {}), ('modification', 'reference atoms'): (<function MappingDirector._reference_atoms>, {}), ('modification', 'to'): (<function MappingDirector._ff>, {'direction': 'to'}), ('modification', 'to blocks'): (<function MappingDirector._blocks>, {'direction': 'to', 'map_type': 'modification'}), ('modification', 'to edges'): (<function MappingDirector._edges>, {'direction': 'to'}), ('modification', 'to nodes'): (<function MappingDirector._nodes>, {'direction': 'to'}), ('molecule',): (<function MappingDirector._molecule>, {})}

A dict of all known parser methods, mapping section names to the function to be called and the associated keyword arguments.

NO_FETCH_BLOCK = '!'

The character that specifies no block should be fetched automatically.

RESIDUE_ATOM_SEP = ':'

The character that separates a residue identifier from an atomname.

RESNAME_NUM_SEP = '#'

The character that separates a resname from a resnumber in shorthand block formats.

SECTION_ENDS = ['block', 'modification']

finalize_section(*previous_section*, *ended_section*)

Wraps up parsing of a single mapping.

Parameters

- **previous_section** (*collections.abc.Sequence[str]*) – The previously parsed section.
- **ended_section** (*collections.abc.Iterable[str]*) – The just finished sections.

Returns

The accumulated mapping if the mapping is complete, None otherwise.

Return type

Mapping or None

`vermouth.map_parser.`**parse_mapping_file**(*filepath*, *force_fields*)

Parses a mapping file.

Parameters

- **filepath** (*str*) – The path of the file to parse.
- **force_fields** (*dict[str, ForceField]*) – Dict of known forcefields

Returns
A list of all mappings described in the file.

Return type
list[*Mapping*]

vermouth.molecule module

class vermouth.molecule.**Block**(*incoming_graph_data=None*, ***attr*)

Bases: *Molecule*

Residue topology template

Two blocks are equal if the underlying molecules are equal, and if the block names are equal.

Parameters

- **incoming_graph_data** – Data to initialize graph. If None (default) an empty graph is created.
- **attr** – Attributes to add to graph as key=value pairs.

name

The name of the residue. Set to *None* if undefined.

Type
str or None

add_atom(*atom*)

Add an atom. *atom* must contain an ‘atomname’. This value will be this atom’s index.

Parameters
atom (*collections.abc.Mapping*) – The attributes of the atom to add. Must contain ‘atomname’

Raises
ValueError – If *atom* does not contain ‘atomname’

property **atoms**

” The atoms in the residue. Each atom is a dict with *a minima* a key ‘name’ for the name of the atom, and a key ‘atype’ for the atom type. An atom can also have a key ‘charge’, ‘charge_group’, ‘comment’, or any arbitrary key.

Return type
collections.abc.Iterator[dict]

guess_angles()

Generates all possible triplets of node indices that correspond to angles.

Yields
tuple[collections.abc.Hashable, collections.abc.Hashable, collections.abc.Hashable] – All possible angles.

guess_dihedrals(*angles=None*)

Generates all possible quadruplets of node indices that correspond to torsion angles.

Parameters

angles (*collections.abc.Iterable*) – All possible angles from which to start looking for torsion angles. Generated from *guess_angles()* if not provided.

Yields

tuple[collections.abc.Hashable, collections.abc.Hashable, collections.abc.Hashable, collections.abc.Hashable] – All possible torsion angles.

has_dihedral_around(*center*)

Returns True if the block has a dihedral centered around the given bond.

Parameters

center (*tuple*) – The name of the two central atoms of the dihedral angle. The method is sensitive to the order.

Return type

bool

has_improper_around(*center*)

Returns True if the block has an improper centered around the given bond.

Parameters

center (*tuple*) – The name of the two central atoms of the improper torsion. The method is sensitive to the order.

Return type

bool

node_dict_factory

alias of `OrderedDict`

to_molecule(*atom_offset=0*, *offset_resid=0*, *offset_charge_group=0*, *force_field=None*, *default_attributes=None*)

Converts this block to a *Molecule*.

Parameters

- **atom_offset** (*int*) – The number at which to start numbering the node indices.
- **offset_resid** (*int*) – The offset for the *resid* attributes.
- **offset_charge_group** (*int*) – The offset for the *charge_group* attributes.
- **force_field** (*None or* `vermouth.forcefield.ForceField`) –
- **default_attributes** (*collections.abc.Mapping[str]*) – Attributes to set to for nodes that are missing them.

Returns

This block as a molecule.

Return type

Molecule

class vermouth.molecule.**Choice**(*value*)

Bases: *LinkPredicate*

Test if an attribute is defined and in a predefined list.

Parameters
`value` (*list*) – The list of value in which to look for the attribute.

`match`(*node*, *key*)

Apply the comparison.

`class vermouth.molecule.DeleteInteraction`(*atoms*, *atom_attrs*, *parameters*, *meta*)

Bases: `tuple`

Create new instance of DeleteInteraction(atoms, atom_attrs, parameters, meta)

`atom_attrs`

Alias for field number 1

`atoms`

Alias for field number 0

`meta`

Alias for field number 3

`parameters`

Alias for field number 2

`class vermouth.molecule.Interaction`(*atoms*, *parameters*, *meta*)

Bases: `tuple`

Create new instance of Interaction(atoms, parameters, meta)

`atoms`

Alias for field number 0

`meta`

Alias for field number 2

`parameters`

Alias for field number 1

`class vermouth.molecule.Link`(*incoming_graph_data=None*, ***attr*)

Bases: *Block*

Template link between two residues.

Two links are equal if:

- the underlying molecules are equal
- the names are equal
- the negative edges ("non-edges") are equal regardless of order
- the interactions to remove are the same and in the same order
- the meta variables are equal
- the pattern definitions are equal and in the same order
- the features are equals regardless of order

A link does not match if any of the non-edges match the target; their order therefore is not important. Same goes for features that just need to be present or not. The order does matter however for interactions to remove as removing the interactions in a different order may lead to a different set of remaining interactions.

Parameters

- **`incoming_graph_data`** – Data to initialize graph. If *None* (default) an empty graph is created.
- **`attr`** – Attributes to add to graph as key=value pairs.

`node_dict_factory`

alias of `OrderedDict`

`same_non_edges`(*other*)

Returns *True* if all the non-edges of an *other* link are equal to those of this link. Returns *False* otherwise.

`class vermouth.molecule.`**`LinkParameterEffector`**(*keys*, *format_spec=None*)

Bases: `object`

Rule to calculate an interaction parameter in a link.

This class allows to store dynamic parameters in link interactions. The value of the parameter can be computed from the graph using the node keys given when creating the instance.

An instance of this class is first initialized with a list of node keys from the link in which it is defined. The instance is latter called like a function, and takes as arguments a molecule and a match dictionary linking the link nodes with the molecule ones. The format of the dictionary is expected to be `{link key:  molecule key}`.

An instance can also have a format defined. If defined, that format will be applied to the value computed by the *`_apply()`* method causing the output to be a string. The format is given as a 'format_spec' from the python format string syntax. This format spec corresponds to what follows the column the column in string templates. For instance, formating a floating number to have 2 decimal places will be obtained by setting format to *.2f*. If no format is defined, then the calculated value is not modified.

This is a base class; it needs to be subclassed. A subclass must define an *`_apply()`* method that takes a molecule and a list of node keys from that molecule as arguments. This method is not called directly by the user, instead it is called by the *`__call__()`* method when the user calls the instance as a function. A subclass can also set the *`n_keys_asked`* class attribute to the number of required keys. If the attribute is set, then the number of keys provided when initializing a new instance will be validated against that number; else, the user can pass an arbitrary number of keys without validation.

`__call__`(*molecule*, *match*)

Parameters

- **`molecule`** (`Molecule`) – The molecule from which to calculate the parameter value.
- **`match`** (*`dict`*) – The correspondence between the nodes from the link (keys), and the nodes from the molecule (values).

Returns

The calculated parameter value, formatted if required.

Return type

float

`_apply`(*molecule*, *keys*)

Calculate the parameter value from the molecule.

Notes

This method **must** be defined in a subclass.

Parameters

- **molecule** (*Molecule*) – The molecule from which to compute the parameter value.
- **keys** (*list*) – A list of keys to use from the molecule.

Returns
The value for the parameter.

Return type
float

Parameters

- **keys** (*list*) – A list of node keys from the link. If the *n_keys_asked* class argument is set, the number of keys must correspond to the value of the attribute.
- **format_spec** (*str*) – Format specification.

Raises
ValueError – Raised if the *n_keys_asked* class attribute is set and the number of keys does not correspond.

n_keys_asked = None
Class attribute describing the number of keys required.

class vermouth.molecule.LinkPredicate(*value*)
Bases: `object`

Comparison criteria for node and molecule attributes in links.

When comparing an attribute from a link to a corresponding attribute from a molecule or a molecule node, the default behavior is to use the equality as criterion for the correspondence. Some correspondence, however must be broader for the link to be usable. Such alternative criteria are defined as link predicates.

If an attribute in a link is set to an instance of a predicate, then the correspondence is defined as the boolean result of the `match` method.

This is the base class for such predicate. It must be subclassed, and subclasses must define a *match()* method that takes a dictionary and a potential key from that dictionary as arguments.

Parameters
value – The per-instance value that serve as reference. How this value is treated depends on the subclass.

match(*node*, *key*)
Do the comparison with the reference value.

Notes

This function **must** be defined by the subclasses. This docstring describe the *expected* format of the method.

Parameters

- `node` (*dict*) – A dictionary of attributes in which to look up. This can be a node dictionary of a molecule `meta` attribute.
- `key` – A potential key from the `node` dictionary.

Return type
bool

`class vermouth.molecule.Modification`(*incoming_graph_data=None*, ***attr*)

Bases: *Link*

A modification which describes deviations from a *Block*.

`class vermouth.molecule.Molecule`(**args*, ***kwargs*)

Bases: `Graph`

Represents a molecule as per a specific force field. Consists of atoms (nodes), bonds (edges) and interactions such as angle potentials.

Two molecules are equal if:

- the exclusion distance (nrexcl) are equal
- the force fields are equal (but may be different instances)
- the nodes are equal and in the same order
- the edges are equal (but order is not accounted for)
- the interactions are the same and in the same order within an interaction type

When comparing molecules, the order of the nodes is considered as it determines in what order atoms will be written in the output. Same goes for the interactions within an interaction type. The order of edges is not guaranteed anywhere in the code, and they are not writen in the output.

`meta`

Type
dict

`nrexcl`

Type
int

`interactions`

All the known interactions. Each item of the dictionary is a type of interaction, with the key being the name of the kind of interaction using Gromacs itp/rtp conventions ('bonds', 'angles', . . .) and the value being a list of all the interactions of that type in the residue. An interaction is a dict with a key 'atoms' under which is stored the list of the atoms involved (referred by their name), a key 'parameters' under which is stored an arbitrary list of non-atom parameters as written in a RTP file, and arbitrary keys to store custom metadata. A given interaction can have a comment under the key 'comment'.

Type
dict[str, list[*Interaction*]]

`citations`

The citation keys associated with this molecule.

Type
set[str]

`add_interaction`(*type_*, *atoms*, *parameters*, *meta=None*)

Add an interaction of the specified type with the specified parameters involving the specified atoms.

Parameters

- **type** (*str*) – The type of interaction, such as 'bonds' or 'angles'.
- **atoms** (*collections.abc.Sequence*) – The atoms that are involved in this interaction. Must be in this molecule
- **parameters** (*collections.abc.Iterable*) – The parameters for this interaction.
- **meta** (*collections.abc.Mapping*) – Metadata for this interaction, such as comments to be written to the output.

Raises
`KeyError` – If one of the atoms is not in this molecule.

`add_node`(**args*, ***kwargs*)

`add_or_replace_interaction`(*type_*, *atoms*, *parameters*, *meta=None*, *citations=None*)

Adds a new interaction if it doesn't exists yet, and replaces it otherwise. Interactions are deemed the same if they're the same type, and they involve the same atoms, and their `meta['version']` is the same.

Parameters

- **type** (*str*) – The type of interaction, such as 'bonds' or 'angles'.
- **atoms** (*collections.abc.Sequence*) – The atoms that are involved in this interaction. Must be in this molecule
- **parameters** (*collections.abc.Iterable*) – The parameters for this interaction.
- **meta** (*collections.abc.Mapping*) – Metadata for this interaction, such as comments to be written to the output.
- **citations** (*set*) – set of citations that apply when this link is addded to molecule

See also:

add_interaction()

`property atoms`

All atoms in this molecule. Alias for *nodes*.

`copy`()

Creates a copy of the molecule.

Return type
Molecule

`edges_between`(*n_bunch1*, *n_bunch2*, *data=False*)

Returns all edges in this molecule between nodes in *n_bunch1* and *n_bunch2*.

Parameters

- **n_bunch1** (`Iterable`) – The first bunch of node indices.
- **n_bunch2** (`Iterable`) – The second bunch of node indices.

Returns
A list of tuples of edges in this molecule. The first element of the tuple will be in *n_bunch1*, the second element in *n_bunch2*.

Return type
`list`

`find_atoms`(***attrs*)

Yields all indices of atoms that match *attrs*

Parameters
`**attrs` (*`collections.abc.Mapping`*) – The attributes and their desired values.

Yields
collections.abc.Hashable – All atom indices that match the specified *attrs*

`property force_field`

The force field the molecule is described for.

The force field is assumed to be consistent for all the molecules of a system. While it is possible to reassign attribute *Molecule._force_field*, it is recommended to assign the force field at the system level as reassigning *`force_field`* will propagate the change to all the molecules in that system.

`get_interaction`(*type_*)

Returns all interactions of *type_*

Parameters
`type` (*`collections.abc.Hashable`*) – The type which interactions should be found.

Returns
The interactions of the specified type.

Return type
list[*Interaction*]

`iter_residues`()

Returns a generator over the nodes of this molecules residues.

Return type
collections.abc.Generator

`make_edges_from_interaction_type`(*type_*)

Create edges from the interactions of a given type.

The interactions must be described so that two consecutive atoms in an interaction should be linked by an edge. This is the case for bonds, angles, proper dihedral angles, and cmap torsions. It is not always true for improper torsions.

Cmap are described as two consecutive proper dihedral angles. The atoms for the interaction are the 4 atoms of the first dihedral angle followed by the next atom forming the second dihedral angle with the 3 previous ones. Each pair of consecutive atoms generate an edge.

Warning: If there is no interaction of the required type, it will be silently ignored.

Parameters
`type` (*`str`*) – The name of the interaction type the edges should be built from.

make_edges_from_interactions()

Create edges from the interactions we know how to convert to edges.

The known interactions are bonds, angles, proper dihedral angles, cmap torsions and constraints.

merge_molecule(*molecule*)

Add the atoms and the interactions of a molecule at the end of this one.

Atom and residue index of the new atoms are offset to follow the last atom of this molecule.

Parameters
molecule (Molecule) – The molecule to merge at the end.

Returns
A dict mapping the node indices of the added *molecule* to their new indices in this molecule.

Return type
dict

node_dict_factory

alias of OrderedDict

remove_interaction(*type_*, *atoms*, *version=0*)

Removes the specified interaction.

Parameters

- **type** (*str*) – The type of interaction, such as 'bonds' or 'angles'.
- **atoms** (*collections.abc.Sequence*) – The atoms that are involved in this interaction.
- **version** (*int*) – Sometimes there can be multiple distinct interactions between the same group of atoms. This is reflected with their *version* meta attribute.

Raises
KeyError – If the specified interaction could not be found

remove_matching_interaction(*type_*, *template_interaction*)

Removes any interactions that match the template.

Parameters

- **type** (*collections.abc.Hashable*) – The type of interaction to look for.
- **template_interaction** (Interaction) –

See also:

interaction_match()

remove_node(*node*)

Overriding the remove_node method of networkx as we have to delete the interaction from the interactions list separately which is not a part of the graph and hence does not get deleted.

remove_nodes_from(*nodes*)

Overriding the remove_nodes_from method of networkx as we have to delete the interaction from the interactions list separately which is not a part of the graph and hence does not get deleted.

same_edges(*other*)

Compare the edges between this molecule and an other.

Edges are unordered and undirected, but they can have attributes.

Parameters
`other` (*networkx.Graph*) – The other molecule to compare the edges with.

Return type
bool

`same_interactions`(*other*)

Returns *True* if the interactions are the same.

To be equal, two interactions must share the same node key reference, the same interaction parameters, and the same meta attributes. Empty interaction categories are ignored.

Parameters
`other` (`Molecule`) –

Return type
bool

`same_nodes`(*other*, *ignore_attr=()*)

Returns *True* if the nodes are the same and in the same order.

The equality criteria used for the attribute values are those of *`vermouth.utils.are_different()`*.

Parameters

- `other` (`Molecule`) –
- `ignore_attr` (*collections.abc.Container*) – Attribute keys that will not be considered in the comparison.

Return type
bool

`share_moltype_with`(*other*)

Checks whether *other* has the same shape as this molecule.

Parameters
`other` (`Molecule`) –

Returns
True iff other has the same shape as this molecule.

Return type
bool

`static sort_interactions`(*all_interactions*)

Returns keys in interactions sorted by (number_of_atoms, name). Keys with no interactions are skipped.

`property sorted_nodes`

`subgraph`(*nodes*)

Creates a subgraph from the molecule.

Return type
Molecule

`class vermouth.molecule.NotDefinedOrNot`(*value*)

Bases: *`LinkPredicate`*

Test if an attribute is not the reference value.

This test passes if the attribute is not defined, if it is set to `None`, or if its value is different from the reference.

Notes

If the reference is set to `None`, then the test does not pass if the attribute is explicitly set to `None`. It still passes if the attribute is not defined.

Parameters
value – The value the attribute is tested not to be.

match(*node*, *key*)

Apply the comparison.

class vermouth.molecule.**ParamAngle**(*keys*, *format_spec=None*)

Bases: *LinkParameterEffector*

Calculate the angle in degrees between three consecutive nodes.

Parameters

- **keys** (*list*) – A list of node keys from the link. If the *n_keys_asked* class argument is set, the number of keys must correspond to the value of the attribute.
- **format_spec** (*str*) – Format specification.

Raises
ValueError – Raised if the *n_keys_asked* class attribute is set and the number of keys does not correspond.

n_keys_asked = 3

Class attribute describing the number of keys required.

class vermouth.molecule.**ParamDihedral**(*keys*, *format_spec=None*)

Bases: *LinkParameterEffector*

Calculate the dihedral angle in degrees defined by four nodes.

Parameters

- **keys** (*list*) – A list of node keys from the link. If the *n_keys_asked* class argument is set, the number of keys must correspond to the value of the attribute.
- **format_spec** (*str*) – Format specification.

Raises
ValueError – Raised if the *n_keys_asked* class attribute is set and the number of keys does not correspond.

n_keys_asked = 4

Class attribute describing the number of keys required.

class vermouth.molecule.**ParamDihedralPhase**(*keys*, *format_spec=None*)

Bases: *LinkParameterEffector*

Calculate the dihedral angle in degrees defined by four nodes shifted by -180 degrees.

Parameters

- **keys** (*list*) – A list of node keys from the link. If the *n_keys_asked* class argument is set, the number of keys must correspond to the value of the attribute.
- **format_spec** (*str*) – Format specification.

Raises
ValueError – Raised if the *n_keys_asked* class attribute is set and the number of keys does not correspond.

n_keys_asked = 4

Class attribute describing the number of keys required.

class vermouth.molecule.**ParamDistance**(*keys*, *format_spec=None*)

Bases: *LinkParameterEffector*

Calculate the distance between a pair of nodes.

Parameters

- **keys** (*list*) – A list of node keys from the link. If the *n_keys_asked* class argument is set, the number of keys must correspond to the value of the attribute.
- **format_spec** (*str*) – Format specification.

Raises

ValueError – Raised if the *n_keys_asked* class attribute is set and the number of keys does not correspond.

n_keys_asked = 2

Class attribute describing the number of keys required.

vermouth.molecule.**attributes_match**(*attributes*, *template_attributes*, *ignore_keys=()*)

Compare a dict of attributes from a molecule with one from a link.

Returns `True` if the attributes from the link match the ones from the molecule; returns `False` otherwise. The attributes from a link match with those of a molecule if all the individual attribute from the link match the corresponding ones in the molecule. In the simplest case, these attribute match if their values are equal. If the value of the link attribute is an instance of *LinkPredicate*, then the attributes match if the `match` method of the predicate returns `True`.

Parameters

- **attributes** (*dict*) – Attributes from the molecule.
- **template_attributes** (*dict*) – Attributes from the link.
- **ignore_keys** (*list*) – List of keys to ignore from 'template_attributes'.

Return type

bool

vermouth.molecule.**interaction_match**(*molecule*, *interaction*, *template_interaction*)

Compare an interaction with a template interaction or interaction to delete.

An instance of *Interaction* matches a template instance of the same class or of *DeleteInteraction* if, at the minimum, it involves the same atoms in the same order. If the template defines parameters, then they have to match as well. In the case of of a *DeleteInteraction*, atoms may have attributes as well, then they have to match with the attributes of the corresponding atoms in the molecule.

Parameters

- **molecule** (*networkx.Graph*) – The molecule that contains the interaction.
- **interaction** (Interaction) – The interaction in the molecule.
- **template_interaction** (Interaction *or* DeleteInteraction) – The template to match with the interaction.

Return type

bool

See also:

attributes_match

vermouth.parser_utils module

Helper functions for parsers

class vermouth.parser_utils.**LineParser**

Bases: `object`

Class that describes a parser object that parses a file line by line. Subclasses will probably want to override the methods *dispatch()*, *parse_line()*, and/or *finalize()*:

- *dispatch()* is called for every line and should return the function that should be used to parse that line.
- *parse_line()* is called by the default implementation of *dispatch()* for every line.
- *finalize()* is called at the end of the file.

COMMENT_CHAR = '#'

dispatch(*line*)

Finds the correct method to parse *line*. Always returns *parse_line()*.

finalize(*lineno=0*)

Wraps up. Is called at the end of the file.

parse(*file_handle*)

Reads lines from *file_handle*, and calls *dispatch()* to find which method to call to do the actual parsing. Yields the result of that call, if it's not *None*. At the end, calls *finalize()*, and yields its results, iff it's not None.

Parameters

file_handle (*collections.abc.Iterable[str]*) – The data to parse. Should produce lines of data.

Yields

object – The results of dispatching to parsing methods, and of *finalize()*.

parse_line(*line*, *lineno*)

Does nothing and should be overridden by subclasses.

class vermouth.parser_utils.**SectionLineParser**(**args*, ***kwargs*)

Bases: *LineParser*

Baseclass for all parsers that have to parse file formats that are based on sections. Parses the *macros* section. Subclasses will probably want to override *finalize()* and/or *finalize_section()*.

finalize_section() is called with the previous section whenever a section ends.

section

The current section.

Type

list[str]

macros

A set of subsitution rules as parsed from a *macros* section.

Type
dict[str, str]

`METH_DICT = {('macros',): (<function SectionLineParser._macros>, {})}`

A dict of all known parser methods, mapping section names to the function to be called and the associated keyword arguments.

`dispatch`(*line*)

Looks at *line* to see what kind of line it is, and returns either `parse_header()` if *line* is a section header or `parse_section()` otherwise. Calls `is_section_header()` to see whether *line* is a section header or not.

Parameters
line (*str*) –

Returns
The method that should be used to parse *line*.

Return type
collections.abc.Callable

`finalize`(*lineno=0*)

Called after the last line has been parsed to wrap up. Resets the instance and calls `finalize_section()`.

Parameters
lineno (*int*) – The line number.

`finalize_section`(*previous_section*, *ended_section*)

Called once a section is finished. Currently does nothing.

Parameters

- **previous_section** (*list[str]*) – The last parsed section.
- **ended_section** (*list[str]*) – The sections that have been ended.

`static is_section_header`(*line*)

Parameters
line (*str*) – A line of text.

Returns
`True` iff *line* is a section header.

Return type
bool

Raises
IOError – The line starts like a section header but looks misformatted.

`parse_header`(*line*, *lineno=0*)

Parses a section header with line number *lineno*. Sets `section` when applicable. Does not check whether *line* is a valid section header.

Parameters

- **line** (*str*) –
- **lineno** (*str*) –

Returns
The result of calling `finalize_section()`, which is called if a section ends.

Return type
object

Raises
`KeyError` – If the section header is unknown.

`parse_section`(*line*, *lineno*)

Parse *line* with line number *lineno* by looking up the section in `METH_DICT` and calling that method.

Parameters

- **`line`** (*`str`*) –
- **`lineno`** (*`int`*) –

Returns
The result returned by calling the registered method.

Return type
object

`class vermouth.parser_utils.SectionParser`(*name*, *bases*, *attrs*, ***kwargs*)

Bases: `type`

Metaclass (!) that populates the *METH_DICT* attribute of new classes. The contents of *METH_DICT* are set by reading the *_section_names* attribute of all its attributes. You can conveniently set *_section_names* attributes using the `section_parser()` decorator.

`static section_parser`(**names*, ***kwargs*)

Parameters

- **`names`** (*`tuple[collections.abc.Hashable]`*) – The section names that should be associated with the decorated function.
- **`kwargs`** (*`dict[str]`*) – The keyword arguments with which the decorated function should be called.

`vermouth.parser_utils.`**`split_comments`**(*line*, *comment_char=';'*)

Splits *line* at the first occurence of *comment_char*.

Parameters

- **`line`** (*`str`*) –
- **`comment_char`** (*`str`*) –

Returns
line before and after *comment_char*, respectively. If *line* does not contain *comment_char*, the second element will be an empty string.

Return type
tuple[str, str]

vermouth.selectors module

Provides helper function for selecting part of a system, e.g. all proteins, or protein backbones.

`vermouth.selectors.`**`filter_minimal`**`(`*molecule*, *selector*`)`

Yield the atom keys that match the selector.

The selector must be a function that accepts an atom as a argument. The atom is passed as a node attribute dictionary. The selector must return `True` for atoms to keep in the selection.

The function can be used to build a subgraph that only contains the selection:

```
selection = molecule.subgraph(
     filter_minimal(molecule, selector_function)
)
```

Parameters

- **`molecule`** (`Molecule`) –
- **`selector`** (*`collections.abc.Callable`*) –

Yields

collections.abc.Hashable – Keys of the atoms that match the selection.

`vermouth.selectors.`**`is_protein`**`(`*molecule*`)`

Return True if all the residues in the molecule are protein residues.

The function tests if the residue name of all the atoms in the input molecule are in `PROTEIN_RESIDUES`.

Parameters

`molecule` (`Molecule`) – The molecule to test.

Return type

bool

`vermouth.selectors.`**`proto_multi_templates`**`(`*node*, *templates*, *ignore_keys=()*`)`

Return *True* is the node matched one of the templates.

Parameters

- **`node`** (*`dict`*) – The atom/node to consider.
- **`templates`** (*`collections.abc.Iterable[dict]`*) – A list of node templates to compare to the node.
- **`ignore_keys`** (*`collections.abc.Collection`*) – List of keys to ignore from the templates.

Return type

bool

See also:

`vermouth.molecule.attributes_match`

`vermouth.selectors.`**`proto_select_attribute_in`**`(`*node*, *attribute*, *values*`)`

Return True if the given attribute of the node is in a list of values.

To be used as a selector, the function must be wrapped in a way that it can be called without the need to explicitly specify the 'attribute' and 'values' arguments. This can be done using `functools.partial()`:

```
>>> # select an atom if its name is in a given list
>>> to_keep = ['BB', 'SC1']
>>> select_name_in = functools.partial(
...     proto_select_attribute_in,
...     attribute='atomname',
...     values=to_keep
... )
>>> select_name_in(node)
```

Parameters

- **node** (*dict*) – The atom/node to consider.
- **attribute** (*str*) – The key to look at in the node.
- **values** (*list*) – The values the node attribute can take for the node to be selected.

Return type
bool

`vermouth.selectors.`**`select_all`**`(_)`

Returns True for all particles.

`vermouth.selectors.`**`select_backbone`**`(`*node*`)`

Returns True if *node* is in a protein backbone.

`vermouth.selectors.`**`selector_has_position`**`(`*atom*`)`

Return True if the atom have a position.

An atom is considered as not having a position if: * the "position" key is not defined; * the value of "position" is `None`; * the coordinates are not finite numbers.

Parameters
atom (*dict*) –

Return type
bool

vermouth.system module

Provides a class to describe a system.

class `vermouth.system.`**`System`**`(`*force_field=None*`)`

Bases: `object`

A system of molecules.

molecules

The molecules in the system.

Type
list[*Molecule*]

add_molecule`(`*molecule*`)`

Add a molecule to the system.

Parameters
molecule (*Molecule*) –

copy()

Creates a copy of this system and it's molecules.

Returns
A deep copy of this system.

Return type
System

property force_field

The forcefield used to describe the molecules in this system.

property num_particles

The total number of particles in all the molecules in this system.

vermouth.truncating_formatter module

Provides a string formatter that can not only pad strings to a specified length if they're too short, but also truncate them if they're too long.

class vermouth.truncating_formatter.FormatSpec(*fill*, *align*, *sign*, *alt*, *zero_padding*, *width*, *comma*, *decimal*, *precision*, *type*)

Bases: `tuple`

Create new instance of FormatSpec(fill, align, sign, alt, zero_padding, width, comma, decimal, precision, type)

align

Alias for field number 1

alt

Alias for field number 3

comma

Alias for field number 6

decimal

Alias for field number 7

fill

Alias for field number 0

precision

Alias for field number 8

sign

Alias for field number 2

type

Alias for field number 9

width

Alias for field number 5

zero_padding

Alias for field number 4

class vermouth.truncating_formatter.**TruncFormatter**

Bases: `Formatter`

Adds the 't' option to the format specification mini-language at the end of the format string. If provided, the produced formatted string will be truncated to the specified length.

format_field(*value*, *format_spec*)

Implements the 't' option to truncate strings that are too long to the required width.

Parameters

- **value** – The object to format.
- **format_spec** (*str*) – The format_spec describing how *value* should be formatted
- **Returns** –
- **str** – *value* formatted as per *format_spec*

format_spec_re = re.compile('(([\\s\\S])?([<>=\\^]))?([\\+\\-])?(#)?(0)?(\\d*)?(,)?((\\.)(\\d*))?([sbcdoxXneEfFgGn%])?')

vermouth.utils module

Provides several generic utility functions

vermouth.utils.**are_all_equal**(*iterable*)

Returns `True` if and only if all elements in *iterable* are equal; and `False` otherwise.

Parameters

iterable (*collections.abc.Iterable*) – The container whose elements will be checked.

Returns

`True` iff all elements in *iterable* compare equal, `False` otherwise.

Return type

bool

vermouth.utils.**are_different**(*left*, *right*)

Return True if two values are different from one another.

Values are considered different if they do not share the same type. In case of numerical value, the comparison is done with `numpy.isclose()` to account for rounding. In the context of this test, *nan* compares equal to itself, which is not the default behavior.

The order of mappings (dicts) is assumed to be irrelevant, so two dictionaries are not different if the only difference is the order of the keys.

vermouth.utils.**first_alpha**(*search_string*)

Returns the first ASCII letter.

Parameters

string (*str*) – The string in which to look for the first ASCII letter.

Return type

str

Raises

ValueError – No ASCII letter was found in 'search_string'.

vermouth.utils.**format_atom_string**(*node*, *atomid=''*, *chain=''*, *resname=''*, *resid=''*, *atomname=''*)

vermouth.utils.**maxes**(*iterable*, *key=<function <lambda>>*)

Analogous to `max`, but returns a list of all maxima.

```
>>> all(key(elem) == max(iterable, key=key) for elem in iterable)
True
```

Parameters

- **iterable** (*collections.abc.Iterable*) – The iterable for which to find all maxima.
- **key** (*collections.abc.Callable*) – This callable will be called on each element of `iterable` to evaluate it to a value. Return values must support > and ==.

Returns

A list of all maximal values.

Return type

list

6.1.3 Module contents

VerMoUTH: The Very Modular Universal Transformation Helper

Provides functionality for creating MD topologies from coordinate files. Powers the CLI tool martinize2.

CHAPTER SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Martini3] P.C.T. Souza, R. Alessandri, J. Barnoud, S. Thallmair, I. Faustino, F. Grünewald, et al., Martini 3: a general purpose force field for coarse-grained molecular dynamics, Nat. Methods. 18 (2021) 382–388. https://doi.org/10.1038/s41592-021-01098-3

[martinize1] de Jong, D. H., Singh, G., Bennett, W. F. D., Arnarez, C., Wassenaar, T. a, Schäfer, L. v., Periole, X., Tieleman, D. P., & Marrink, S. J. (2013). Improved Parameters for the Martini Coarse-Grained Protein Force Field. Journal of Chemical Theory and Computation, 9(1), 687–697. https://doi.org/10.1021/ct300646g

[VMD] W. Humphrey, A. Dalke and K. Schulten, "VMD - Visual Molecular Dynamics", J. Molec. Graphics, 1996, vol. 14, pp. 33-38. http://www.ks.uiuc.edu/Research/vmd/

[DSSP]

- W.G. Touw, C. Baakman, J. Black, T.A.H. te Beek, E. Krieger, R.P. Joosten, et al., A series of PDB-related databanks for everyday needs, Nucleic Acids Res. 43 (2015) D364–D368. https://doi.org/10.1093/nar/gku1028
- W. Kabsch, C. Sander, Dictionary of protein secondary structure: pattern recognition of hydrogen-bonded and geometrical features., Biopolymers. 22 (1983) 2577–637. https://doi.org/10.1002/bip.360221211

[backward] T.A. Wassenaar, K. Pluhackova, R.A. Böckmann, S.J. Marrink, D.P. Tieleman, Going Backward: A Flexible Geometric Approach to Reverse Transformation from Coarse Grained to Atomistic Models, J. Chem. Theory Comput. 10 (2014) 676–690. doi:10.1021/ct400617g.

[ISMAGS] M. Houbraken, S. Demeyer, T. Michoel, P. Audenaert, D. Colle, M. Pickavet, The Index-Based Subgraph Matching Algorithm with General Symmetries (ISMAGS): Exploiting Symmetry for Faster Subgraph Enumeration, PLoS One. 9 (2014) e97896. doi:10.1371/journal.pone.0097896.

PYTHON MODULE INDEX

INDEX