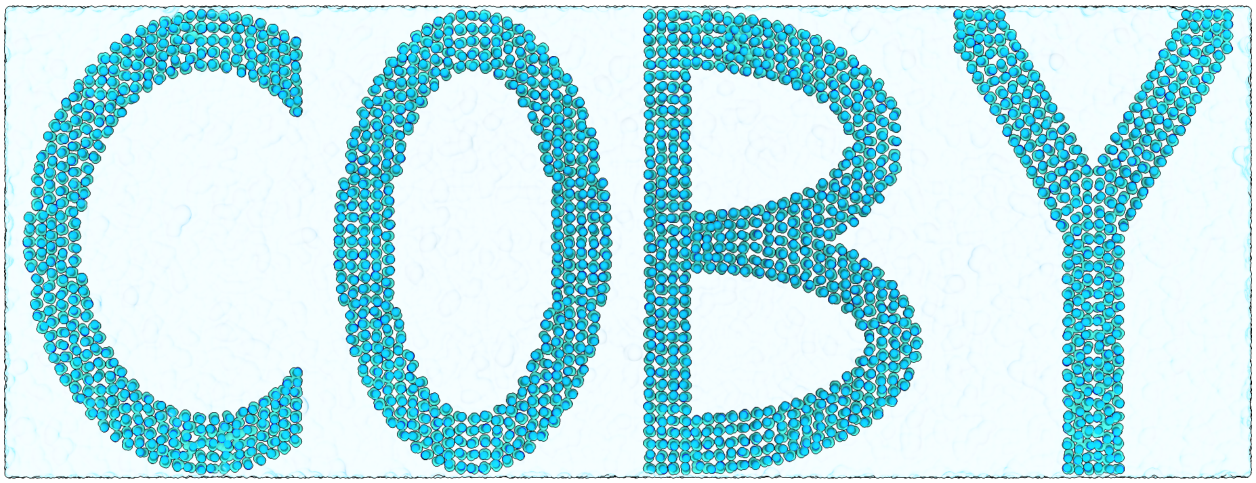
Coarse-Grained System Builder (COBY)

Documentation (for version 0.2.6)

Mikkel D. Andreasen, Lorena Zuzic



Contents

1 Introduction

Coarse-grained System Builder (COBY) is a versatile and easy-to-use Python-based program for building coarse-grained complex membranes for use in molecular dynamics simulations. COBY can handle asymmetric membranes, phase-separated membranes, or multiple bilayers in the same system. Additionally, it performs membrane protein insertion and solvation, and can be used to flood the system with one (or more) molecules of interest.

COBY can be used both as a **package within a Python environment**, or as a **argument-line based software**.

The leading principles guiding the software design were:

- out-of-the box use for simple systems
- high-level of customisability for complex systems
- accuracy in handling requested system properties
- parameter libraries for a large number of Martini lipids
- open-source code

2 Installation

The code and installation instructions are available at: github.com/MikkelDA/COBY. COBY is maintained as a pip package compatible with a Python3.9 environment.

```
conda create --name COBY python==3.9 ipykernel

conda activate COBY

pip install COBY
```

If you wish to use the script as a Jupyter notebook, also run:

```
conda activate COBY

python -m ipykernel install --user --name=COBY
```

3 Cite us

Coming up...

4 Quickstart

Build a simple POPC membrane in water with 0.15 M NaCl in a Python script:

```
import COBY

COBY.COBY(
    box = [10, 10, 10],
    membrane = "lipid:POPC",
    solvation = "default"
)
```

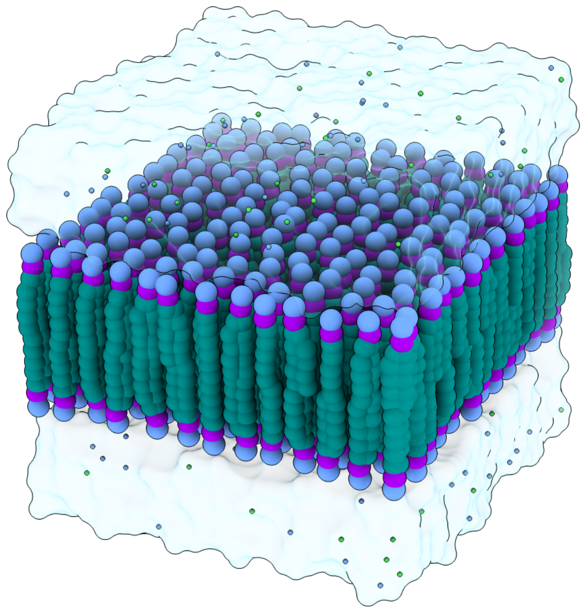


Figure 1: Pure POPC membrane in water with 0.15 M NaCl.

Build the same system, but using the argument-line:

```
python -m COBY -box 10 10 10 -membrane lipid:POPC -solvation default
```

5 General feature overview

- It can be imported as a package in a Python environment or run directly from a terminal argument-line.
- Builds one or more **complex, asymmetric membranes**, with a precise handling of lipid type ratios. Membranes are built flat and in the xy plane.
- It can be used to create monolayers, bilayers, or multiple membranes of any given composition.
- It can build **phase-separated bilayer systems** of any given shape (where each segment can feature a different lipid composition).
- It can build **stacked membranes** with specific lipid compositions, inter-membrane distances, and whilst handling the number of inter-membrane solvent-to-lipid ratios.
- It can build **membrane patches or holes** of any arbitrary shape, or the arguments can be combined to create a complex phase-separated system.
- Lipid placements are optimised in order to alleviate building artefacts.
- It handles cubic, hexagonal, rhombic dodecahedron and custom system **box shapes**.
- It handles one or more **membrane proteins** and their placement within the bilayer.
- It performs **system solvation** with correct handling of ion molarities.
- It can perform **partial solvations** within a given volume.
- It can be used to set up **flooding simulations** with one or more molecules of choice.
- Molecule structures can be **imported** and used as solvents/solutes during solvations and floodings and as lipids in membranes.
- Molecule structures can be manually created using the inbuilt **molecule fragment builder**.
- Separate molecule libraries can be **imported**.

5.1 Workflow

Steps are run sequentially, and progress further only if the previous steps have run successfully. In principle, a minimal executable argument requires only box dimensions and a single element of a system: e.g., a lipid (resulting in a homogeneous membrane), a solvent (creating a solvent box), or a structure (placing the structure — e.g., a protein — in a box). In practice, this means that the user has full flexibility in the choice of the building blocks of the desired systems.

5.2 Membrane creation

A notable feature of this software is correct handling of lipid ratios, both within and between the leaflets. Additionally, special attention is paid to replicate the correct area per lipid (APL) within the each leaflet, taking into account the leaflet area occupied by proteins. Multiple methods are available for optimizing the ratio between lipid types within a given leaflet. Membranes are created by inserting an exact number of lipids in a leaflet, after which the distance between the lipids is optimised using a custom-built algorithm that prevents potential bead overlaps.

All membranes are created in the *xy* plane, meaning that the code supports only flat membranes. Individual leaflets in membranes can be created independently of each other, and any number of membranes can be created. The code also supports creation of phase-separated membranes, where each segment can be defined with different lipid compositions.

To enhance performance, large membranes are dynamically split into multiple smaller ones, which significantly improves code speed. This has no effect on the number of lipids placed within a membrane as it is accounted for.

The code includes subarguments that allow for the creation of membrane patches and holes of arbitrary shapes, including circles, squares, rectangles, ellipses, and polygons that can be rotated and scaled in a desired aspect ratio.

Finally, the code features a special argument that can be used for making stacked membrane systems.

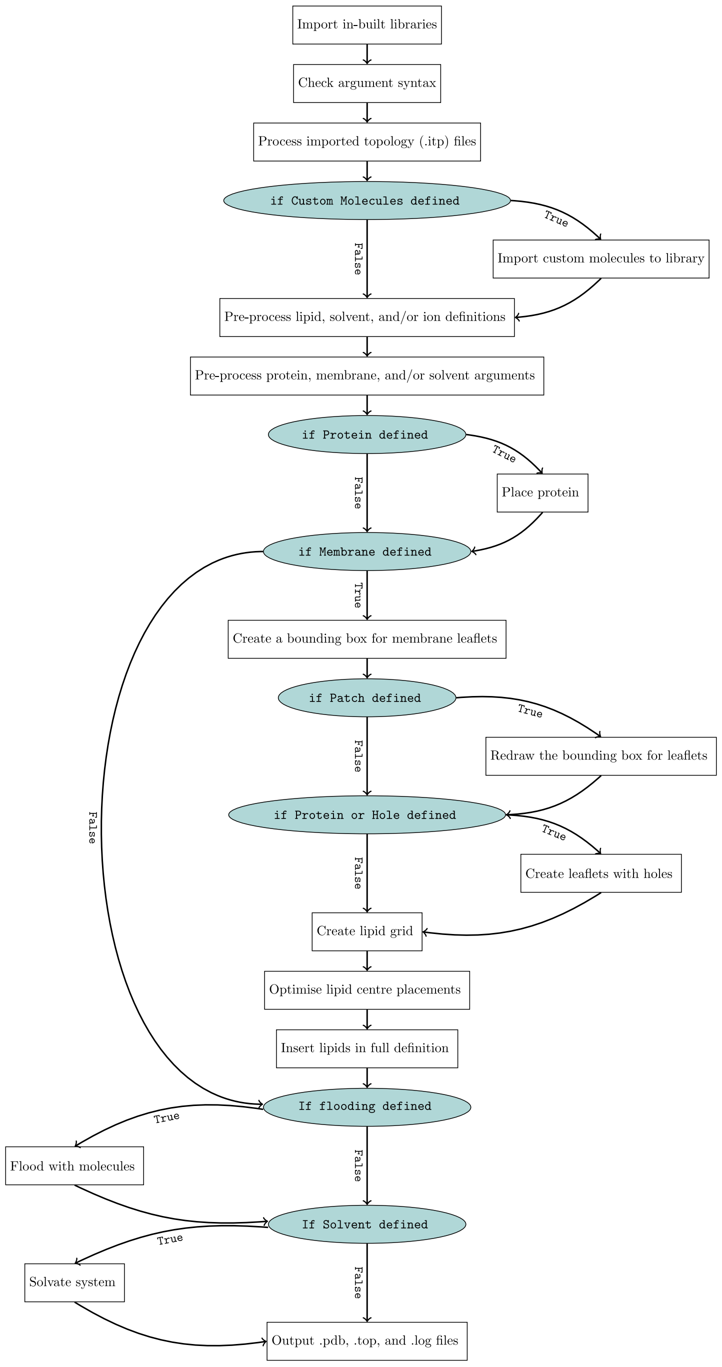

Import in-built libraries
Check argument syntax
Process imported topology (.itp) files
if Custom Molecules defined
True
Import custom molecules to library
False
Pre-process lipid, solvent, and/or ion definitions
Pre-process protein, membrane, and/or solvent arguments
if Protein defined
True
Place protein
False
if Membrane defined
True
Create a bounding box for membrane leaflets
if Patch defined
True
Redraw the bounding box for leaflets
False
if Protein or Hole defined
True
Create leaflets with holes
False
Create lipid grid
Optimise lipid centre placements
Insert lipids in full definition
False
If flooding defined
True
Flood with molecules
False
If Solvent defined
True
Solvate system
False
Output .pdb, .top, and .log files

Figure 2: **COBY workflow.**

5.3 Structure (protein) insertion

Structure files given in a .pdb or .gro format can be inserted into the membrane. In most cases, this is a protein structure, but in principle the code can handle any given structure (if provided with a coordinate file).

Any number of structure files can be inserted, and they can be moved and rotated based on their centre. Multiple methods are available for designating the structure's centre, based on the structure centre of geometry (COG), a coordinate point, the axial mean coordinate, one or multiple residues, or one or multiple beads.

In order to trace the edges of the inserted structure and correctly handle membrane building around it, COBY utilizes a powerful geometry module for Python called `shapely`, which creates an accurate "footprint" of abnormalities present in a membrane (e.g., protein-occupied areas).

5.4 Flooding

If a `flooding` argument is used, the system can be flooded with a specified number of one or more molecules of choice. The program requires the coordinate file of the molecule, number of residues per molecule, total charge, and the number of desired molecules to be included in the system.

5.5 Solvation

First, the free volume is estimated using the number of other particles present in the box.

$$V_{free} = V_{box} - V_{proteins} - V_{lipids} - V_{solutes} \tag{1}$$

If the flooding preceded the solvation step, the volume of the added flooding molecules is also subtracted from the free volume. The free volume dictates the number of required water beads.

$$N_{solvent} = \frac{N_A \cdot V_{free} \cdot c_{solvent}}{K_{solvent}} \tag{2}$$

where N_A is the Avogadro's constant, and $K_{solvent}$ is an atomistic-to-CG mapping specified under solvent parameters (e.g., $K_{RW} = 4$). Molarity is set for solvation argument call, but is by default $c = 55.56$ mol L^{-1}. Next, a number of ions is calculated based on the solvent volume.

$$V_{solvent} = \frac{N_{solvent} \cdot K_{solvent} \cdot M_{solvent}}{N_A \cdot \rho_{solvent}} \tag{3}$$

where $M_{solvent}$ is molecular weight specified under solvent parameters ($M_{water} = 18.016$ g mol^{-1}).

$$N_{ion} = \frac{N_A \cdot V_{solvent} \cdot c_{ion}}{K_{ion}} \tag{4}$$

A solvent placement algorithm ensures that no solvent or ion is placed within the hydrophobic volume of a membrane, and ensures a minimum distance between solvent beads and other particles.

Additionally, the code implements multiple solvation procedures that can be chosen by the user and which apply at specific solvation steps (e.g., in terms of interpretation of mixed solvent ratios, charge neutralisation procedure, or whether the solvent should be added in proportion to the number of lipids).

5.6 Topology processing

Charge information can be read from topology files, if they are supplied.

The script is able to understand and recursively process the `#include Path/To/top.itp` statements. Similarly to `gmx grompp`, it is a requirement that the `#include` statements are listed in the same order as they appear in the `[ molecules ]` section of the topology file.

In order for the program to link coordinates with topologies, proteins and custom solutes must have their name(s) matching their respective `[ moleculetype ]` specified under their designated arguments.

The program will also write `#include` statements found in a topology file to the output .top file, though only those present in the zeroth-layer of recursion (i.e., it does not write `#include` statements found within other `#include` statements, but only those written in the original topology file). This allows the program to write "gromppable" topology files.

6 Arguments

6.1 Syntax

COBY builds the system by using a single-argument call which takes arguments that describe all required elements of the system (e.g., `box`, `membrane`, `solvent`, `randseed`) that take a single value - a float, integer, list, or a string. If a string is passed onto an argument, it can consist of one or several **subarguments** - keywords - followed by a colon and a designated value.

A general syntax for both terminal and script use is shown here, with **argument** and **subargument** being a stand-in for specific calls that can be used.

Python script:

```
COBY.COBY(
    argument = "subargument:value"
)
```

Terminal argument-line:

```
python -m COBY -argument subargument:value
```

The colour scheme will follow the convention (`Python script` vs. terminal argument-line) accordingly.

Multiple calls to the same argument in Python require either creating a list of strings, or using numbered arguments multiple times:

```
COBY.COBY(
    argument = ["subarg1", "subarg2"]
)
```

or

```
COBY.COBY(
    argument1 = "subarg1",
    argument2 = "subarg2"
)
```

In Terminal, the flags must be repeated without numbering:

```
python -m COBY -argument subarg1 -argument subarg2
```

Subsequent subarguments are given as `subargument:value`. The number of values that can be assigned to a subargument varies depending on the specific subargument.

Subarguments given to an argument only affect that specific argument string. For example, building multiple bilayers requires multiple argument calls. Under those circumstances, properties assigned to one do not transfer to another.

```
COBY.COBY(
    box = [10, 10, 20],
    membrane1 = "lipid:POPC apl:0.5 cz:+5",
    membrane2 = "lipid:POPE apl:0.4 cz:-5"
)
```

Similarly, subarguments can repeat within the same argument, in which case the properties link together only until the second instance of the subargument appears. For example, an asymmetric membrane requires lipid composition specification for each individual leaflet. This requires two separate calls to the `leaflet` subargument. Under these circumstances, properties assigned to one leaflet do not transfer to another.

```
COBY.COBY(
    box = [10, 10, 10],
    membrane = " ".join([
        "leaflet:upper lipid:POPC apl:0.5",
        "leaflet:lower lipid:POPC",
    ])
)
```

In this example, `apl:0.5` is applied only to the upper leaflet. Because this value is unspecified in the lower leaflet, it defaults to `0.6`. Note that both subarguments are passed onto the argument as a part of the same string (for clarity, they are written as individual strings and then joined together by `" ".join()` - the resulting string in fact looks like this: `"leaflet:upper lipid:POPC apl:0.5 leaflet:lower lipid:POPC"`.

6.2 Box arguments: box, x, y, z

The `box` or `pbc` argument is used to set the side lengths of the box in x, y, and z direction. The arguments `x` / `y` / `z` can instead be used to set the coordinates individually.

```
box = [10, 10, 10]       pbc = [10, 10, 10]       x = 10, y = 10, z = 10
```

If only one value is specified, it is applied to all three dimensions of the cubic box.

`box = [10]`
`x = 10`
`y = 10`
`z = 10`
	converted to `box = [10, 10, 10]`

6.3 Box types: `box_type`

The `box_type` or `pbc_type` arguments can be used to set the type of box. The available box types are rectangular (default), hexagonal, skewed hexagonal, and a rhombic dodecahedron with a hexagonal xy plane. A rectangular box takes three box side length values. A hexagonal box takes two box side length values. A rhombic dodecahedron box takes a single box side length value.

`box_type = "rectangular"`	default; 3 sidelength values; $\angle\vec{x}\vec{y} = \angle\vec{x}\vec{z} = \angle\vec{y}\vec{z} = 90°$
`box_type = "hexagonal"`	2 sidelength values; $\angle\vec{x}\vec{y} = 60°$; $\angle\vec{x}\vec{z} = \angle\vec{y}\vec{z} = 90°$
`box_type = "skewed_hexagonal"`	1 sidelength value; $\angle\vec{x}\vec{y} = \angle\vec{x}\vec{z} = \angle\vec{y}\vec{z} = 60°$
`box_type = "dodecahedron"`	1 side length value; $\angle\vec{x}\vec{y} = 90°$; $\angle\vec{x}\vec{z} = \angle\vec{y}\vec{z} = 60°$

6.4 Manually designated unit cell: `pdb_unitcell`, `gro_unitcell`

One can also manually designate a unit cell. This can be done by either designating the unit cell parameters formatted as in .pdb files (with `pdb_unitcell`), or as in .gro files (with `gro_unitcell`). Regardless of the input unit cell format, COBY can output both file formats (with correct box designations).

The .pdb unit cell parameters can be given in two different ways, as shown below. In the first line, the $x/y/z$ box lengths are designated, while the angles are assumed to be 90°. In the second example, the $x/y/z$ box lengths and the $\alpha/\beta/\gamma$ angles are designated in degrees. Note that the angles α, β, γ are described with the vectors $\vec{y}$ and $\vec{z}$, $\vec{x}$ and $\vec{z}$, and $\vec{x}$ and $\vec{y}$, respectively (i.e., $\alpha = \angle\vec{y}\vec{z}$, $\beta = \angle\vec{x}\vec{z}$, $\gamma = \angle\vec{x}\vec{y}$).

```
pdb_unitcell = [x, y, z]
```

```
pdb_unitcell = [x, y, z, α, β, γ]
```

The following example is equivalent to a `dodecahedron` box with a sidelength of 15.

```
pdb_unitcell = [15, 15, 15, 60, 60, 90]
```

The .gro unit cell parameters differ from the .pdb format. Instead of specifying the three box side lengths and angles between them, the box in the .gro format is described by three vectors (1-3), each consisting of three dimensions (x, y, and z), thus requiring nine values to describe them. The order of vector components is as follows:

$$v_1(x) \quad v_2(y) \quad v_3(z) \quad v_1(y) \quad v_1(z) \quad v_2(x) \quad v_2(z) \quad v_3(x) \quad v_3(y)$$

In case of rectangular boxes, only the first three components need to be specified (the rest default to zero). For other box types, all nine vector components need to be specified.

Rectangular boxes:

gro_unitcell = [$v_1(x)$, $v_2(y)$, $v_3(z)$]

Non-rectangular boxes:

gro_unitcell = [$v_1(x)$, $v_2(y)$, $v_3(z)$, $v_1(y)$, $v_1(z)$, $v_2(x)$, $v_2(z)$, $v_3(x)$, $v_3(y)$]

The following example is equivalent to a `dodecahedron` box with a sidelength of 15.

```
gro_unitcell = [15, 15, 10.6066, 0, 0, 0, 0, 7.5, 7.5]
```

It should be noted that Gromacs requires that $v_1(y) = v_1(z) = v_2(z) = 0$. If a given unit cell values violate this requirement, then the program will exit with an error.

`pdb_unitcell` and `gro_unitcell` arguments cannot be used within the same COBY call, though, as mentioned earlier, COBY can still create a .pdb file while using `gro_unitcell` and a .gro file while using `pdb_unitcell`.

6.5 System name: `sn`

The `sn` argument can be used to set the name of the output system in the .pdb file (TITLE row), .gro file (first line) and topology file (under `[ system ]`).

```
sn = "Tutorial System Name"
```

6.6 Output file specification: `out_sys`, `out_top`, `out_log`, `out_all`

By default, the script produces two types of output files: a coordinate file (which can be .pdb, .gro or .cif), and a truncated topology file, which contains `[ system ]` and `[ molecules ]` sections. If the output names are unspecified, the files are saved under output.gro, output.pdb, output.cif, and topol.top. One can also save a log file, which will then contain a detailed script output.

```
out_pdb = path/to/file,      # saves only .pdb files
out_gro = path/to/file,      # saves only .gro files
out_cif = path/to/file,      # saves only .cif files
out_sys = path/to/file,      # saves both .pdb, .gro and .cif files
out_top = path/to/topology.top,
out_log = path/to/logfile.log
```

If no extension is specified under `out_sys`, the script saves both .pdb and .gro files. Another way to specify only .pdb of .gro files is to use `out_pdb`, `out_gro` and `out_cif`, respectively.

Alternatively, if one wants identical file names for all output files (.pdb, .gro, .cif, .top and .log files), then the argument `out_all` can be used.

```
out_all = path/to/files
```

6.7 Verboseness of terminal printing: `verbose`

The verboseness of the text printed to the terminal can be modified using the `verbose` argument. The smaller the number, the fewer details are printed to the terminal (ranges between 1 and 3). Note that the `verbose` argument has no impact on the text written to the .log file. The information written to the log file will always be the most verbose possible.

```
verbose = 1        # default
```

6.8 Backup of overwritten files: `backup`

argument `backup` can be used to specify with `True/False` if the overwritten files should be backed up.

```
backup = "True"       # default
```

The backup format follows the same format as Gromacs, namely: `#filename.ext.[0-9]+#`.

6.9 Random seed: `randseed`

The `randseed` argument can be used to set the random seed:

```
randseed = [int]
```

6.10 Topology input: `itp_input`

The `itp_input` argument can be used to give the program topology data which will be used to determine charges. The main topology file that contains `#include` statements can be added using the subargument `file`. The output topology file will also contain all `#include` statements found within that file (but not those found within `#include` statements). Additional topology files can be added by using the subargument `include`, which will also add that specific `#include` statement to the output topology file (`#include` statements found within the file will still be read, but not included in the written topology file).

```
itp_input = "file:top_for_COBY.itp"
itp_input = "include:SUCR.itp"
itp_input = "file:top_for_COBY.itp include:SUCR.itp"
itp_input = ["file:top_for_COBY.itp", "include:SUCR.itp"]
```

One can prevent the `#include` statements from being written to the topology file by using the subargument `write_includes:False`.

```
itp_input = "file:top_for_COBY.itp write_includes:False"
```

COBY reads and stores variables defined within the given topology files, but sometimes one may wish to use information written within `#ifdef` and `#ifndef` statements whose associated variables are designated within mdp files. One can use the `define:[str]` subargument to declare variable names (though no variable values), such that the itp reader will process `#ifdef` and `#ifndef` appropriately. Note that defines are case-sensitive. Multiple ”define” subarguments can be given.

```
itp_input = ["file:top_for_COBY.itp", "define:FLEXIBLE"]
```

Lines can be added to topology files indicating arguments for COBYs molecule fragment builder functionality. You can read about this in the molecule fragment builder section 6.13.1.

6.11 Setting parameter libraries: params

sys_params argument can be used to specify a global parameter library that will be used for all lipids, proteins, and solvents, unless specified otherwise.

The lipid_params, solv_params, and prot_params arguments define libraries of parameters for lipids, solvents, and proteins, respectively. This feature is useful if one wants to use multiple libraries (e.g., corresponding to different development versions), in which molecule types might share the same names, but are defined with different parameters. Note that the name of the parameter library needs to include at least one alphabetical character.

Some examples include:

```
lipid_params = "default" # default
lipid_params = "dev18"
lipid_params = "PhosV13"
```

Parameters specified for groups (lipids, solvents, lipids) trump the global parameters specified under sys_params.

```
sys_params = "default",
lipid_params = "PhosV13"
```

In this example, lipid parameters are taken from the PhosV13 library, while all the others are taken from the default library.

Even more granular, parameters can be set for the whole group, or for each individual lipid/solvent type (addressed in subsubsection 6.15.1) by adding params:LIBRARY to the specific subargument.

```
sys_params = "default",
lipid_params = "PhosV13",
membrane = "lipid:POPC:5 lipid:POPE:3 lipid:CHOL:1:params:dev18",
solv = "solv:W"
```

Here, cholesterol parameters will be linked to the dev18 library, other lipids (POPC and POPE) to the PhosV13 library, and solvent parameters to the default library.

COBY will, by default, search for charge information within topologies, under the condition that they have been provided. If a membrane, solvation or flooding argument, or a specific lipid, solv, solute, pos_ion or neg_ion subargument should have its charges determined by the in-built library, then the subargument charge:VALUE can be added.

Valid options for `charge` (sub)argument are:

`charge:topology` or `charge:top`	(default) charges are read from the topology file
`charge:library` or `charge:lib`	charges are read from the in-built library

In the example below, the membrane argument has been instructed to find charges from the library using the `charge:library` subcommand, and CHOL has been set to acquire charges from the topology by adding `charge:topology` to the `lipid` subargument. This means that POPC and POPE have their charges read from the library, and CHOL from the topology file.

```
membrane = " ".join([
    "lipid:POPC:5 lipid:POPE:2",
    "lipid:CHOL:1:charge:topology",
    "charge:library",
])
```

The syntax is the same for the `solvation` and `flooding` arguments, as well as the `solv`, `solute`, `pos_ion` and `neg_ion` subarguments.

6.12 Importing molecules: `molecule_import`

Molecules can be imported using the argument `molecule_import` and can be used as solvent, solute, negative ions, positive ions or lipids in their respective arguments. The structure file (.pdb/.gro/.cif) must be specified with the `file` subargument.

`file:[str]`	imported molecule structure (pdb, gro or cif) file

`molecule_import` operates in two different ways, dependent on the existence of the topology file for the imported molecule. Note that all topologies are handled in the `itp_input` argument.

Version 1: If molecule topology exists:

```
molecule_import = " ".join([
    "file:[structure.pdb]",
    "moleculetype:[topology moleculetype]",
    "refname:[reference name]" # optional,
])
```

In this instance, molecule charges are read from the topology file using the moleculetype name given. The subargument refname is optional but can be used to give a separate name to the molecule which is used to reference the molecule in other arguments.

Version 2: If molecule topology does not exist:

```
molecule_import = " ".join([
    "file:[structure.pdb]",
    "name:[molname] charge:[float/int]",
])
```

In this instance, the molecules name and charges are specified as subarguments.

In order to be able to reference the molecule in other arguments, the name must be specified. Depending on the existence of the molecule topology file, this can be done with either moleculetype (or refname) or name subarguments.

moleculetype:[str]	Uses the string both as a reference name and as the [moleculetype] for charge determination.
refname:[str]	Optional. Uses the string as a reference name instead of the one supplied to moleculetype:[str]. The name given to moleculetype:[str] is still used for charge determination. Not usable alongside name:[str].
name:[str]	Uses the string as a reference name. Charges must be specified (see below).

If one uses the name subargument, then one must set the charges manually (else they will be assumed to be zero for all beads) using the subargument charge. Note that the bead/residue number uses Python-indexing, meaning that the first bead/residue is number 0.

No `charge` argument	Sets the charge of all beads to zero.
`charge:[float/int]`	Spreads the given charge evenly across all beads in the molecule. Only a single `charge` subargument of this type can be used.
`charge:[float/int]:res:[int]:bead:[int]`	Specifies the charge for a specific bead in a specific residue. Any number of `charge` subarguments of this type can be used.
`charge:[float/int]:bead:[int]`	Specifies the charge for a specific bead. Any number of `charge` subarguments of this type can be used. Only usable for single-residue molecules.

Below are shown examples of the two different ways of importing molecules (in this case the molecule sucrose).

Version 1: If molecule topology exists

The subargument `moleculetype` is used to designate the `[ moleculetype ]` of the molecule. The charge information is gathered from the supplied topology files by matching with the specified name. `moleculetype` is also used as a molecule name in other arguments (e.g., `flooding` or `membrane`).

```
molecule_import = " ".join([
    "file:sucrose.pdb",
    "moleculetype:SUCR",
])
```

Version 2: If molecule topology does not exist

The subarguments `name` and `charge` are used to assign a reference name that is used in other arguments. The charge is set to zero.

```
molecule_import = " ".join([
    "file:sucrose.pdb",
    "name:SUCR",
    "charge:0",
])
```

6.12.1 Solvent import

In order to effectively use solvents with the `solvation` argument, it is necessary to specify the `density`, `molar_mass` and (potentially) the AA-to-CG `mapping` ratio. Below is shown and example where martini 3 regular water is imported with the relevant values.

```
molecule_import = " ".join([
    "file:M3_reg_water.pdb",
    "name:W",
    "charge:0",
    "denisity:099669", # [g/cm]
    "molar_mass:18.01528", # [g/mol]
    "mapping:4", # [number of AA molecules to 1 CG molecule]
])

molecule_import = " ".join([
    "file:wide_lipid.pdb",
    "name:W",
    "charge:0",
    "scale:x:0.75,
    "scale:y:0.75,
])
```

6.12.2 Lipid import

Import of lipids to be used within the `membrane` follows the syntax above, but with the addition of consideration of lipid orientation. The "correct" orientation of a lipid is considered to be a vertical alignment, with the headgroup pointing to the positive z-direction, and the tail pointing to the negative z-direction. If the lipid in the structure file is pre-oriented in this way, then no additional subarguments are needed.

However, if the lipid is not pre-oriented, then we need to designate the upwards-pointing beads and downwards-pointing beads in the `upbead` and `downbead` subarguments, respectively. The syntax for the two subarguments is shown below. Note that the bead/residue number uses Python-indexing, meaning that the first bead/residue is number 0. Multiples of each `upbead` / `downbead` subarguments may be used, in which case the mean position will be used.

If one of the two subcommands is used, then the other is mandatory is well (`upbead` and `downbead` come as a pair).

No `(up/down)bead` subargument	Assumes that the molecule is vertically aligned.
`(up/down)bead:[int]:res:[int]`	Specifies a bead in a residue for lipid alignment. Any number of subarguments of this type can be used.
`(up/down)bead:[int]`	Specifies a bead for lipid alignment. Any number of subarguments of this type may be used. Only usable for single-residue molecules.

Below are shown two examples of importing a POPC molecule. In the first example, the lipid is already vertically aligned; in the second example, the lipid is aligned using the `upbead` and `downbead` subcommands. The first example also shows how to manually designate charges for specific beads. Note that `params:IMPORTED` has been added to both examples because POPC is already present in the in-built lipid parameter library, and thus needs to be placed in its own parameter library. The examples also demonstrate how to use the lipids in `membrane` arguments.

Version 1: The subarguments `name` is used to assign a reference name to be used in other arguments. The subargument `charge` is used to set the charge of the NC3 bead and the PO4 bead. `charge:library` has been added to the `lipid` subargument because the charges have been manually set, and the program should therefore not attempt to look for them in the topology.

```
molecule_import = " ".join([
    "file:POPC.pdb",
    "name:POPC",
    "charge:1:res:0:bead:0",  # Positively charged NC3 bead
    "charge:-1:res:0:bead:1", # Negatively charged PO4 bead
    "params:IMPORTED",
])
membrane = "lipid:POPC:params:IMPORTED:charge:library"
```

Version 2: The subargument `moleculetype` is used to designate the `[ moleculetype ]` of the lipid. The charge information is gathered from supplied topology files, and the same name is used in other arguments to reference the lipids.

```
molecule_import = " ".join([
    "file:POPC_rotated.pdb",
    "moleculetype:POPC",
    "upbead:0:res:0",
    "downbead:7:res:0",
    "downbead:11:res:0",
    "params:IMPORTED",
])
```

```
membrane = "lipid:POPC:params:IMPORTED"
```

6.12.3 Parameter libraries and library types during import

The imported molecule can also have a designated parameter library, which can be useful in cases where there is a molecule with the same name already existing in the in-built library. The parameter library is designated with the `params` subargument. The default parameter library is called `default`.

```
params:[str]
```

Additionally, there are specific library types that correspond to specific system elements.

`library_types:solvent` or `library_types:solute`	The solvent / solute library (placed within the same library).
`library_types:pos_ions`	The positive ion library.
`library_types:neg_ions`	The negative ion library.
`library_types:ions`	Counts as both `pos_ions` and `neg_ions`.
`library_types:lipid`	The lipid library.

Importing the molecule into one of the specific library types can be done using the `library_types` subargument. The same molecule can be added to multiple library types as shown below where the imported POPC is added to both the lipid and solute/solvent libraries.

```
molecule_import = " ".join([
    "file:POPC.pdb",
    "moleculetype:POPC",
    "params:IMPORTED",
    "library_types:lipid:solute",
])
membrane = "lipid:POPC:params:IMPORTED"
flooding = "solute:POPC:params:IMPORTED"
```

6.12.4 Scaling imported structure coordinates

One can scale the size of imported molecules using the `scale` subargument along all axes simultaneously (by using `scale:all`) or along individual axes (by using `scale:x`, `scale:y` or `scale:z`). This can be useful when importing lipids in order to "tighten" their structures for the lipid insertion algorithm. Be careful when using `scale` as scaling coordinates down too much can cause problems during minimizations.

In the example below an imaginary 'wide lipid' is imported and all x- and y-coordinates are scaled down to 75% of their original values (by multiplying the x- and y-values by 0.25).

```
molecule_import = " ".join([
    "file:wide_lipid.pdb",
    "moleculetype:WIDE_LIPID",
    "params:IMPORTED",
    "scale:x:0.25",
    "scale:y:0.25",
])
```

6.13 Molecule Fragment Builder: `molecule_builder`

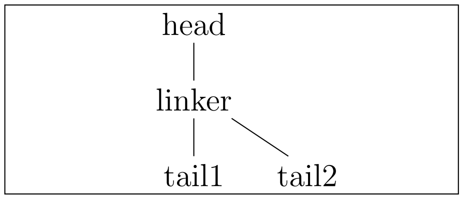
New molecules can be constructed by linking fragments using the `molecule_builder` argument. Common lipid molecule fragments are accessible from an in-built library. Alternatively, custom fragments can be imported (as described in Section 6.12). The available fragment libraries are shown in the following table. They include the Martini 3 launch parameters for phospholipids as well as the lipid task force (LTF) parameters.

`phospholipid`	Covers phosphoglycerides based on the release parameters for Martini 3.
`phospholipid_LTF`	Covers phospholipids (glycerol, ether and plasmalogen linkers) based on the LTF parameters.
`hydrocarbon_LTF`	Covers hydrocarbon chains based on the LTF parameters.
`fattyacid_LTF`	Covers fatty acids based on the LTF parameters.
`monoglyceride_LTF`	Covers glycerides with a single tail based on the LTF parameters.
`diglyceride_LTF`	Covers non-phosphorylated glycerides with two tails based on the LTF parameters.
`triglyceride_LTF`	Covers glycerides with three tails based on the LTF parameters.
`phospholipid_LTF`	Covers phosphoglycerides based on the LTF parameters, ether-glycerides and plasmalogens.
`cardiolipin_LTF`	Covers cardiolipins based on the LTF parameters.
`BMP_LTF`	Covers 2,2' and 3,3' bis(monoacylglycero)phosphates (BMPs) based on the LTF parameters.
`sphingolipid_LTF`	Covers sphingolipids based on the LTF parameters.

Each molecule type contains one or multiple parts. The parts are either prebuilt fragments — defined within parameter libraries that are either in-built or imported, or built fragment, which are dynamically constructed using a building function. In case of lipids, the prebuilt fragments include headgroups (e.g., PC or PE) and linkers (e.g., glycerol GL) while tails are dynamically constructed fragments. By default, linkers are assigned based on the specified lipid type — for instance, phospholipid type will by default have a glycerol linker, although it can be manually changed (e.g., to ET in case of etherphospholipids). The following table includes a list of all currently available parts and associated fragments.

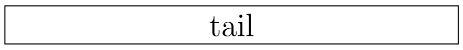
`phospholipid`

`head`	Fragments: PC, PE, PG, PA, PS
`linker`	Fragments: GL - glycerol (default)
`tail1`	Constructed from code containing: C, c, D
`tail2`	Constructed from code containing: C, c, D



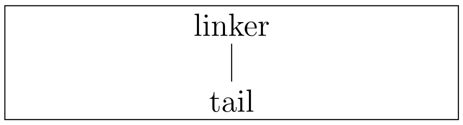
`hydrocarbon_LTF`

`tail`	Constructed from code containing: C, c, D, T, t, F



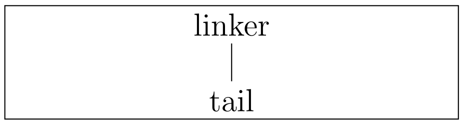
`fattyacid_LTF`

`linker`	Fragments: COO (default)
`tail`	Constructed from code containing: C, c, D, T, t, F



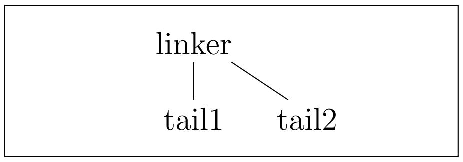
`monoglyceride_LTF`

`linker`	Fragments: GL (default)
`tail`	Constructed from code containing: C, c, D, T, t, F



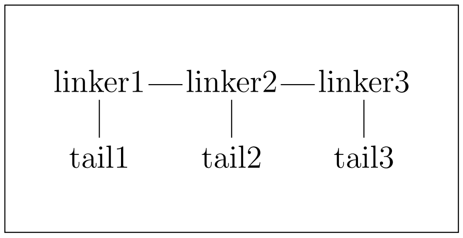
`diglyceride_LTF`

`linker`	Fragments: GL (default)
`tail1`	Constructed from code containing: C, c, D, T, t, F
`tail2`	Constructed from code containing: C, c, D, T, t, F



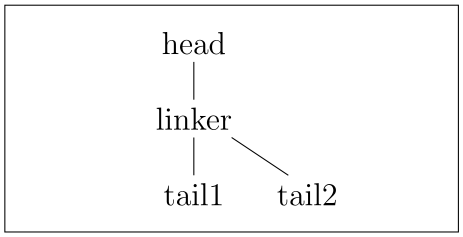
`triglyceride_LTF`

`linker1`	Fragments: GL (default)
`tail1`	Constructed from code containing: C, c, D, T, t, F
`linker2`	Fragments: GL (default)
`tail2`	Constructed from code containing: C, c, D, T, t, F
`linker3`	Fragments: GL (default)
`tail3`	Constructed from code containing: C, c, D, T, t, F



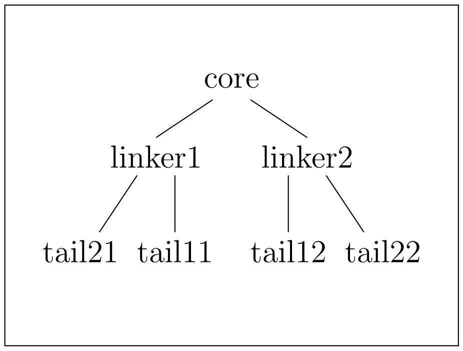
`phospholipid_LTF`

`head`	Fragments: PC, PE, PG, PA, PS, PI, P1, P2, P3, P4, P5, P6, P7
`linker`	Fragments: GL (glycerol) (default), ET (ether), PL (plasmalogen)
`tail1`	Constructed from code containing: C, c, D, T, t, F
`tail2`	Constructed from code containing: C, c, D, T, t, F



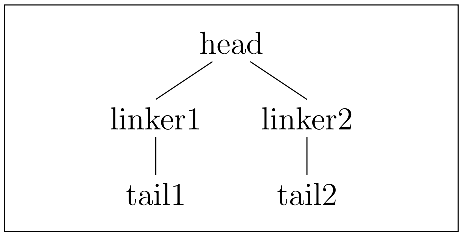
`cardiolipin_LTF`

`core`	Fragments: GLC (default)
`linker1`	Fragments: PhGL (default) (phosphoglycerol)
`tail11`	Constructed from code containing: C, c, D, T, t, F
`tail21`	Constructed from code containing: C, c, D, T, t, F
`linker2`	Fragments: PhGL (default) (phosphoglycerol)
`tail12`	Constructed from code containing: C, c, D, T, t, F
`tail22`	Constructed from code containing: C, c, D, T, t, F



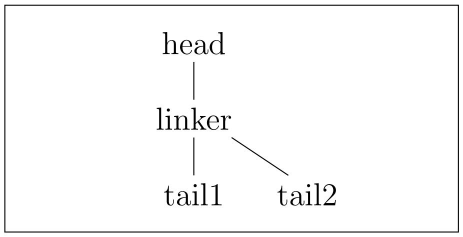
`BMP_LTF`

`head`	Fragments: PO4 (default)
`linker1`	Fragments: 2, 3
`tail1`	Constructed from code containing: C, c, D, T, t, F
`linker2`	Fragments: 2, 3
`tail2`	Constructed from code containing: C, c, D, T, t, F



`sphingolipid_LTF`

`head`	Fragments: PC, PE, PG, PA, PS, PI, P1, P2, P3, P4, P5, P6, P7
`linker`	Fragments: SM (default) (sphingomyelin)
`tail1`	Constructed from code containing: C, c, D, T, t, F
`tail2`	Constructed from code containing: C, c, D, T, t, F



Below is shown an example of a molecule fragment building argument. The `moltype` subargument is mandatory, as it is used to identify which type of molecule is being created. From this, COBY infers a general structure of a molecule, which includes the default and accepted parts, the order in which those parts should appear and how they connect to each other. Similarly, `name` subargument is also mandatory, as it is used to refer to the built molecule in the system building part of the code. Note that the molecule parts inferred from the molecule type (i.e., `linker:GL`) are not specified in the argument.

```
### Building a glycerophospholipid (POPC)
molecule_builder = "moltype:phospholipid head:PC \
    tail1:CDCC tail2:CCCC name:POPC"
```

Below is shown five more examples, this time using the fragment libraries based on the LTF parameters

```
molecule_builder = [
    ### Building a glycerophospholipid (POPC with glycerol linker)
    "moltype:phospholipid_LTF head:PC tail1:CDCC \
        tail2:CCCC name:POPC",

    ### Building a etherphospholipid (POPC with ether linker)
    "moltype:phospholipid_LTF head:PC tail1:CDCC \
        tail2:CCCC linker:ET name:POPC",

    ### Building a sphingolipid (NSM)
    "moltype:sphingolipid_LTF head:PC tail1:tCCC \
        tail2:CCCDCC name:NSM",

    ### Building a cardiolipin (TMCL)
    "moltype:cardiolipin_LTF tail11:CCCC tail21:CCCC \
        tail12:CCCC tail22:CCCC name:TMCL",

    ### Building a BMP (DO3B)
    "moltype:BMP_LTF linker1:2 tail1:CDCC \
        linker2:2 tail2:CDCC name:DO3B",
]
```

By default, molecules built using `molecule_builder` are placed in the MFB parameter library (MFB stands for "Molecule Fragment Builder"). The molecules can then be accessed in `membrane`, `solvation` and `flooding` arguments by using the `params` subargument as shown below.

```
### Both POPC and POPE are read from the MFB library.
membrane = "lipid:POPC lipid:POPE params:MFB"

### POPC is read from the MFB library.
### POPE is read from the default library.
membrane = "lipid:POPC:params:MFB lipid:POPE"
```

All `molecule_builder` arguments, irrespective of molecule type, have standard subarguments (listed below). Note that the charge information is already built into the fragment libraries. If the built molecule can be associated with a topology file, the molecule name (under `[ moleculetype ]`) can be passed onto the `moleculetype` subargument.

Subargument	Effect	Default behaviour
`moltype`	Sets the type of a built molecule.	Required
`name`	Sets the reference name for the molecule.	Required
`params`	Sets the parameter library for the molecule.	`MFB` parameter library.
`resname`	Sets the residue name of the molecule.	Same as `name` subargument.
`moleculetype`	Links the molecule to the topology file.	If explicitly given, uses `resname`. Otherwise uses `name`.

6.13.1 Molecule Fragment Builder and topology files

Instead of passing the molecule fragment builder subarguments directly to the `molecule_builder` argument, one can also specify them directly in the topology file. These subarguments should be specified before or within the `[ moleculetype ]` section of the topology file, where the argument must be prefaced by `;@COBY` (otherwise, COBY will not be able to find it and interpret it).

```
### Example 1: Within the [ moleculetype ] section
[ moleculetype ]
; molname      nrexcl
  POPC          1
;@COBY moltype:phospholipid head:PC tail1:CDCC tail2:CCCC"

### Example 2: Before the [ moleculetype ] section
;@COBY moltype:phospholipid head:PC tail1:CDCC tail2:CCCC"
[ moleculetype ]
; molname      nrexcl
  POPC          1
```

The subargument `moleculetype` is automatically added based on the `[ moleculetype ]` of the topology file. Similarly, the subargument `name` defaults to the `moleculetype` if not explicitly given. Molecules specified in topology files are added to the parameter library called `TOP`, which can be accessed as shown below.

```
itp_input = "include:POPC.itp"
```

```
membrane = "lipid:POPC:params:TOP"
```

6.14 Importing libraries: `import_library`

External libraries can be imported using the argument `import_library`. These libraries can contain lipid scaffolds, lipids, solvent/solutes, ions, as well as fragments that can be used in the lipid fragment builder. The library files follow the Python 3 syntax and have a `.py` extension. The subargument specifier `file:` may be added before the file path but is not necessary for this specific argument as there are no other subarguments available for `import_library`.

```
import_library = [
    "New_LipidScaffolds.py",
    "New_Fragments.py",
    "file:New_Solutes.py",
]
```

6.15 Membrane composition: `membrane`

6.15.1 Basic lipid specification

The `membrane` or `memb` argument can be used to create membranes. Desired lipid types are specified by their name and the number that indicates the ratios between the lipid types within the leaflet. In the case below, only a single lipid type is requested, so the resulting membrane is composed 100% of POPC lipids.

```
membrane = "lipid:POPC:5"
```

`params:LIBRARY` can be added to the `lipid` subargument, which causes the lipid to be looked for in that specific parameter library. In addition, the `params` subargument can be used to set the default parameters for the specific `membrane` argument. If provided, lipid-specific ff designations overwrite the membrane ff designation:

```
membrane = "lipid:POPC:5 lipid:CHOL:1:params:dev18 params:PhosDev13"
```

In this example, POPC would be built from the ”PhosDev13” library, while CHOL would be built from the ”dev18” library.

6.15.2 Area per lipid designation: `apl`

The `apl` subargument, specifying area per lipid, can be used to control how tightly a membrane/leaflet is packed.

`apl:0.6`	default
`apl:0.7`	more sparsely packed leaflets
`apl:0.5`	more tightly packed leaflets

6.15.3 Membrane types: `type`

The `type` subargument can be used to specify symmetric bilayers, asymmetric bilayers with individual leaflet composition, as well as monolayers.

`type:bilayer`	symmetric bilayer (default)
`type:mono`	upwards-facing monolayer
`type:mono_upper`	upwards-facing monolayer
`type:mono_lower`	downwards-facing monolayer
`type:upper`	upper leaflet (in isolation works the same as `type:mono_upper`)
`type:lower`	lower leaflet (in isolation works the same as `type:mono_lower`)

6.15.4 Leaflet specification: `leaflet`

The `leaflet` subargument takes three possible values: `upper`, `lower` or `membrane/both`. If only one of `upper` or `lower` is specified, the script creates a monolayer. In a sense, this argument offers an overlapping functionality with the `type` subargument. Asymmetry can be created in three different ways.

Version 1: by using subsequent `leaflet` subarguments within the same argument string. In this case, lipid property subarguments will be applicable to the last called `leaflet`. Subarguments given before the first `leaflet` argument or after `leaflet:both` will apply to both leaflets.

```
membrane = " ".join([
    ### applies to the whole membrane
    "apl:0.5",

    ### applies only to the upper leaflet
    "leaflet:upper lipid:POPC:5 lipid:CHOL:1",
    ### applies only to the lower leaflet
    "leaflet:lower lipid:POPC:3 lipid:CHOL:2",

    ### applies to the whole membrane
    "leaflet:both params:Dev18",
])
```

Version 2: By using multiple strings in the membrane argument. This technically creates two separate membranes, both of which are monolayers, though it has the same effect as creating a single asymmetric membrane. Note that the `apl` and `params` subarguments must be given in each individual string in this case:

```
membrane = [
    "leaflet:upper lipid:POPC:5 lipid:CHOL:1 apl:0.5 params:Dev18",
    "leaflet:lower lipid:POPC:3 lipid:CHOL:2 apl:0.5 params:Dev18",
]
```

Version 3: by using multiple `membrane` arguments. Remember that multiple calls to the same argument requires adding a number after it (if not run from the terminal). Note that the `apl` and `params` subarguments must be given in each individual string in this case (note that the backslash (\) specifies a continuation of the string on the next line):

```
membrane1 = "leaflet:upper lipid:POPC:5 lipid:CHOL:1 \
             apl:0.5 params:Dev18",
membrane2 = "leaflet:lower lipid:POPC:3 lipid:CHOL:2 \
             apl:0.5 params:Dev18",
```

6.15.5 Membrane size and placement: `xlength`, `ylength`, `center`, `cx`, `cy`, `cz`

By default, the membrane will fill the entire *xy*-plane. However, this can be modified by using the `xlength` and `ylength` subarguments, which define *xy* lengths of the membrane patch, and the `center`, `cx`, `cy`, and `cz` subarguments, which are used to specify the placement of the patch in the coordinate system. This way, one can precisely modify the size and placement of each membrane (or leaflet) segment. Each segment can be specified with different lipid compositions, resulting in a highly-customisable phase-separated membrane. Note that the system is centrosymmetric during calculations, so `center:X:Y:Z` subarguments should be specified with that in mind.

Note that each leaflet is treated independently in calculations, so when creating complex phase-separated systems, the APLs might end up being slightly off due to multiple rounding error accumulations.

An example of a phase-separated membrane with symmetric leaflets (`type` is unspecified, meaning it is set to "bilayer"):

```
membrane = [
    "lipid:POPC:5 lipid:CHOL:1 xlength:5 center:2.5:0:0",
    "lipid:POPC:4 lipid:CHOL:2 xlength:5 center:-2.5:0:0",
]
```

Because the centering of the membrane patches is only dependent on the x coordinate, the `cx` subargument can be used instead of the `center` subargument:

```
membrane = [
    "lipid:POPC:5 lipid:CHOL:1 xlength:5 cx:2.5",
    "lipid:POPC:4 lipid:CHOL:2 xlength:5 cx:-2.5",
]
```

Here is an example of how to build an asymmetric phase-separated membrane:

```
COBY.COBY(
    box = [10,10,10],
    membrane = [
        ### The first membrane
        " ".join([
            "xlength:5 cx:2.5",
            "leaflet:upper lipid:POPC:5 lipid:CHOL:1",
            "leaflet:lower lipid:POPC:3 lipid:CHOL:2",
        ]),

        ### The second membrane
        " ".join([
            "xlength:5 cx:-2.5",
            "leaflet:upper lipid:POPC:5 lipid:POPE:2 lipid:CHOL:1",
            "leaflet:lower lipid:POPC:5 lipid:CHOL:1",
        ]),
    ],
)
```

6.15.6 Treatment of lipid ratios: `lipid_optim`

There are several different methods available for converting lipid ratios to the actual number of lipids, and they can be chosen using the `lipid_optim` subargument. This set of arguments applies only to intra-leaflet lipid ratios. It is based upon assessing the maximum and minimum number of lipids allowed in a leaflet:

$$N_{\text{max lipids}} = \left\lfloor \frac{A_{\text{free}}}{APL} \right\rfloor \tag{5}$$

where A_{free} is the leaflet area with excluded obstructions (e.g., holes or proteins).

By dividing the $N_{\text{max lipids}}$ with specified ratios and rounding them down, we can calculate the minimum number of lipids in a leaflet:

$$N_{\text{min lipids}} = \sum_{i=1}^{N_{\text{types}}} \left\lfloor \frac{w_i}{\sum_{j=1}^{N_{\text{types}}} w_j} \cdot N_{\text{max lipids}} \right\rfloor \tag{6}$$

where N_{types} is the number of lipid types, w is the inter-lipid ratio of a given lipid, and i and j are the indices of the lipid types.

Based on these values, several lipid optimisation methods are available to the user:

`lipid_optim:force_fill`	(default) Fills the leaflet up to $N_{\text{max lipids}}$ starting from $N_{\text{min lipids}}$ (prioritises APL over ratio).
`lipid_optim:fill`	Same as `lipid_optim:force_fill`, but stops if a perfect lipid ratio is reached (prioritises ratio over APL).
`lipid_optim:avg_optimal`	Chooses a lipid distribution between $N_{\text{min lipids}}$ and $N_{\text{max lipids}}$ that results in lipid ratios closest to the requested ratios.
`lipid_optim:abs_val`	Treats lipid ratios as actual number of lipids. Decimal values are rounded to nearest integer value.
`lipid_optim:no`	Does not attempt to optimise the lipid ratios.
`lipid_optim:insane`	Uses the same algorithm as `insane.py` to calculate the number of lipids to be inserted. Result may not be 100% identical if proteins/holes are present.

6.15.7 Membrane patches and holes: `patch` and `hole`

Membrane patches and artificial holes can be created using the subarguments `patch` and `hole`, respectively. The syntax for both arguments is identical, but `hole` will be used in the examples below. It is possible to use both the `hole` and `patch` subarguments within then same membrane argument, whereby a hole will be created within the patch. A total of five different shape types can be created, with the primary one being a polygon. The remaining four shapes are effectively shorthand ways to create specific shapes: circles, ellipses, squares and rectangles.

A series of (sub)subarguments can be supplied to each shape, some being unique to the specific shape type, and others being applicable to all shapes. Below is a list of all the arguments and where they can be applied.

Can be used with all shapes:

`rotate:[float]`	Rotates the shape a given number of degrees counter-clockwise. Negative values rotate the shape clockwise.
`buffer:[float]`	Adds a buffer with a given size to the shape. A negative value will function as a negative buffer, reducing the size of the object. Imported from **shapely**.
`buffer_cap:[int]`	Sets the buffer cap style. Value can be either 1 (round) - default, 2 (flat) or 3 (square). Imported from **shapely**; click **here** for more info.
`buffer_join:[int]`	Sets the buffer join style. Value can be either 1 (round) - default , 2 (mitre) or 3 (bevel). Imported from **shapely**; click **here** for more info.

Can only be used with `polygon`:

`point:[float]:[float]` or `p:[float]:[float]`	Designates the x and y-values of a point in a polygon. The number of `points` needs to be equal to the number of polygon points.
`scaling:[float]`	Scales the size of the object by the given value along both x and y coordinates.
`xscaling:[float]`	Scales the size of the object along the x axis by the given amount.
`yscaling:[float]`	Scales the size of the object along the y axis by the given amount.

Can be used with `circle`, `ellipse`, `square` and `rectangle`:

`cx:[float]`	Designates the x-coordinate of a centre of the shape.
`cy:[float]`	Designates the y-coordinate of a centre of the shape.

Can be used with `circle`:

`radius:[float]`	Designates the radius of the circle.

Can be used with `ellipse`:

`xradius:[float]`	Designates the x-radius of the ellipse.
`yradius:[float]`	Designates the y-radius of the ellipse.

Can be used with `square`:

`length:[float]`	Designates the side-length of the square.

Can be used with `rectangle`:

`xlength:[float]`	Designates the x-side-length of the rectangle.
`ylength:[float]`	Designates the y-side-length of the rectangle.

Here are some examples of hole/patch subarguments and explanations of what they do. For reference, the centre of a membrane in COBY is at the coordinate (0, 0) and the full-size membrane spans from −pbc/2 to +pbc/2 (unless this has been changed by the user).

Example 1: a polygon shaped like a triangle

- point 1: (0,0)
- point 2: (2,3)
- point 3: (4,0)

Both lines produce the same result, but the second line uses the abbreviated version of `point` as `p`.

```
hole:polygon:point:0:0:point:2:3:point:4:0

hole:polygon:p:0:0:p:2:3:p:4:0
```

Example 2: a rotated rectangle

- centre: (3,4)
- xlength: 5
- ylength: 2
- rotation: 45°

Note that the arguments after `hole:rectangle:` can be written in any order.

```
hole:rectangle:xradius:5:yradius:2:cx:3:cy:4:rotate:45
```

By default, if solvation has been requested, then a hole (and the part of a membrane, that has been removed when creating a patch) will be filled with solvent. This can be turned off (resulting in no solvent within the hole) by adding `solvate_hole:False` to the membrane argument, as shown below.

```
COBY.COBY(
    box = [10,10,10],
    membrane = " ".join([
        "lipid:POPC",
        "hole:circle:5",
        "solvate_hole:False",
    ]),
    solvation = "default",
)
```

6.15.8 Kick: `kick` / `kickxy` / `kickz`

A kick is a small push that is applied to a molecule during insertion in order to prevent molecules from being positioned along straight grid lines. The universal kick value can be set using `kick`, or be independently set for the xy-plane and the z-axis using `kickxy` and `kickz`. The default value for both is 0.025 nm.

`kick:[float]`	Sets both planar (xy) and vertical (z) kick values. The unit is [nm].
`kickxy:[float]`	Sets the planar (xy) kick value. The unit is [nm].
`kickz:[float]`	Sets the vertical (z) kick value. The unit is [nm].

6.15.9 Lipid rotation: `rotate_lipids`

Lipids are by default rotated a random amount around their z-axis. This can be turned off by giving the subargument `rotate_lipids:False`, which results in all lipids facing the same direction.

```
membrane = "lipid:POPC rotate_lipids:False"
```

6.15.10 Leaflet splitting: `gridsplits`

COBY will, by default, split leaflets into multiple ”subleaflets”, in order to improve performance and precision of certain algorithms. This behavior can be modified in three different ways.

The subargument `gridsplits` can be used to set whether large leaflets should be split into multiple subleaflets and whether this splitting is manually designated or automatically done.

Allowed `gridsplits` settings

`auto:[val]` (default, [val]=500)	Automatically splits leaflet into multiple subleaflets if their axial lengths are above [val]. The number of splits along an axis is calculated as the axial-length divided by [val] rounded up.
`[xval]:[yval]`	Sets that [xval] and [yval] number of subleaflets should be made along the x- and y-axis, respectively. [xval] and [yval] can be integers, floats (rounded down) or "False" (disables splitting).
`[val]`	Sets that [val] number of subleaflets should be made along both the x- and y-axis. [val] can be an integer, a float (rounded down) or "False" (disables splitting).

6.15.11 Splitting disconnected leaflet parts: `split_bbox`

It can happen through the introduction of proteins or by using the `patch` or `hole` subarguments that a leaflet/subleaflet contains multiple disconnected parts. COBY will, by default, split this leaflet/subleaflet into multiple smaller subleaflets each constituting only a single of the previously disconnected parts. This ensures that local lipid densities are maintained in the disconnected parts and helps the lipid insertion algorithm properly place the lipids. This behavior can be changed by using the `split_bbox` subargument, whose default value is True.

```
COBY.COBY(
    box = [20, 10, 10]
    membrane = " ".join([
        "lipid:POPC",
        ### Creates a hole which cuts the membrane in half
        "hole:rectangle:xlength:2:ylength:10:cx:0:cy:0",
        ### Causes both parts to be processed jointly
        "split_bbox:False"
    ])
)
```

6.15.12 Readjusting leaflet border values: `readjust_bbox`

Values representing the actual leaflet/subleaflet edges are, by default, readjusted if holes, patches or leaflet/subleaflet splitting causes them to change. This can be disabled using the subargument `readjust_bbox`. This changes how some things are treated in the lipid insertion algorithm and should generally not be changed as it is mainly used for debugging purposes. The available values are True (default) and False.

6.15.13 Modifying the lipid insertion algorithm

6.15.13.1 Understanding the algorithm

First we need som context about how the algorithms actually work, and please note that these processes are run on individual leaflets, not on the membrane as a whole. The lipid placement process can be split into two sections, the initial lipid insertion and the following position optimization.

The lipid insertion algorithm will by default separate the lipids into groups based on their size (radius) and insert those groups from largest to smallest. If the circular area of a group alongside all previously inserted groups comprise less than than a certain percentage value (called area portion limiter) of the circular area of all lipids, then a simple algorithm is used (called `2D_grid`) else a more complex algorithm is used (called `LineStrings`).

The `2D_grid` algorithm functions by creating a tight 2-dimensional grid where points on the grid are separated by the mean radius of lipids within the current lipid size group. Grid points that are too close to holes, proteins or previously inserted lipids are removed. The needed number of lipids are then inserted into pseudo-randomly chosen points on the grid.

The `LineStrings` algorithm functions by creating a series of lines along the y-axis. The parts of the lines that are too close to holes, proteins or previously inserted lipids are removed after which points are spread out across all the lines based on an estimated inter-point spacing based on the mean radius of the lipid size group. This process is iteratively done until there are enough points to insert the required number of lipids. The iterative process changes the number of lines created and the spacing between points on the line in order to fit in more lipids. This algorithm allows for smaller lipids to be more easily "fitted in" between the larger ones and provides a more "rounded" look to the edges.

Following the initial lipid algorithm the program checks if any lipids are placed too close to each other, in which case it initiates the lipid optimization algorithm. The optimization algorithm functions somewhat similarly to a very rough 2-dimensional minimization, though we use the word "optimization" for this algorithm because there are no energies involved and all lipids are approximated to be circles (meaning you still need to do your own energy minimization after the system has been created). The optimization algorithm iteratively causes lipids to push each other around on the membrane until the lipids have mostly stopped moving. Furthermore, the lipids are also repulsed by the membrane borders, whether those are the outer edges of the membrane, the edges of artificially created holes or the approximated 2-dimensional edges of proteins overlapping with the leaflet. This algorithm helps ensure that even very closely packed membranes with widely different lipid sizes can still be created without lipid-lipid overlaps.

6.15.13.2 Modifying the initial lipid insertion algorithm

The used algorithm can be changed with the subargument `grid_maker_algorithm` / `gm_alg`.

Allowed `grid_maker_algorithm` / `gm_alg` settings

`iterative_groups / ig` (default)	Lipids are grouped based on their radii. Radius groups are inserted from largest to smallest.
`no_groups / ng`	Lipids are not grouped. All lipids are inserted in the same iteration. Might cause problems if lipids have different sizes.
`3d_matrix / 3d`	Experimental insertion method. Not guaranteed to work.

The area portion limiter (which dictates when the `2D_grid` and `LineStrings` algorithms should be used as explained in section 6.15.13.1) can be changed with the subargument `grid_maker_area_portion_limiter` / `gm_apl`, which by default is 0.7 (70%).

A buffer is created around the outer edges of the membrane, the edges of artificially created holes and the approximated 2D edges of proteins overlapping with the leaflet and previously inserted lipids. This buffer can be modified via a multiplier accessible with the subargument `grid_maker_multiplier` / `gm_mult` with the default value being 1.

The separator value, used to determined what size groups lipids are placed in, can be changed with the `grid_maker_separator` / `gm_sep` subargument. The default value is 0.4 (0.4 nm / 4 Å).

```
membrane = " ".join([
    "lipid:POPC:8:",
    "lipid:GIPC:2", # GIPC's are usually wider than POPC
    ### Sets that lipids should be grouped by their radii.
    "gm_alg:ig",
    ### Sets the grouping to be done with 0.1 nm intervals
    "gm_sep:0.1",
    ### Sets area portion limiter to 30%
    "gm_apl:0.3",
    ### Doubles the buffer around lipids and edges
    "gm_mult:2",
])
```

Lipids can either be distributed evenly or randomly over the available grid points. This can set by using the subargument `grid_maker_lipid_distribution` / `gm_ld` with either `random` (default) or `evenly` as value.

6.15.13.3 Modifying the optimization algorithm

COBY, by default, checks if it is necessary to run the optimization algorithm before it actually starts it. This can be changed using the `optimize_run` / `optim_run` subargument.

Allowed `optimize_run` / `optim_run` settings

`Auto` (default)	The program checks if it is necessary to run the optimization algorithm.
`True / 1 / yes`	Always runs optimization algorithm.
`False / 0 / no`	Never runs optimization algorithm.

The optimization, by default, runs for a maximum of 100 steps. This can be changed using the subargument `optimize_max_steps` / `optim_ms`.

The ”push forces” exerted by lipids on other lipids and those exerted by edges (edges of the membrane, the edges of artificially created holes and the approximated 2D edges of proteins overlapping with the leaflet) on lipids can be modified using multipliers. The two subarguments are `optimize_lipid_push_multiplier` / `optim_lpm` and `optimize_edge_push_multiplier` / `optim_epm`, with both having default values of 1 (100%).

The push tolerance is a value used to determine when the optimization should stop. If the distance any lipid has moved during a single step is larger than this value, then the optimization continues. Only if all lipids move less than that value, or the optimization reaches the number of steps specified by `optimize_max_steps`, does the optimization stop.

The value can be changed with `optimize_push_tolerance` / `optim_pt` and the default value is 0.5 (0.5 nm / 5 Å).

Below is shown an example with all the optimization subarguments.

```
membrane = " ".join([
    "lipid:POPC",
    "apl:0.5",

    ### Forces the optimization to run
    "optim_run:True",
    ### Sets the maximum number of steps to 500
    "optim_ms:500",
    ### Doubles the push force exerted by lipids upon each other
    "optim_lpm:2",
    ### Halfs the push force exerted by edges upon lipids
    "optim_epm:0.5",
    ### Reduces the push tolerance to 0.1 nm
    "optim_pt:0.1",
])
```

6.15.14 Lipid buffers: `plane_buffer` and `height_buffer`

Buffers are added to all lipids during various calculations. These buffers are split into "plane buffers" (applied to the x-/y-plane) and "height buffers" (applied to the z-axis). These buffers help ensure that there are certain distances between neighboring lipids and other system components. Both buffers are, by default, 0.099 nm (0.99 Å) but can be individually changed with the subarguments `plane_buffer` and `height_buffer` with values given in nanometer.

6.15.15 Protein insertion modification: `protein_buffer` and `alpha_multiplier`

The protein-occupied area of a membrane is initially estimated with an alphashape (see the python ALPHASHAPE package for more details). The alphashape requires an alpha parameter which is the inverse of the minimum distance between two neighboring points on the alphashape surface. In COBY, this parameter is calculated as shown in equation 7, where r is the radius of the largest lipid in the leaflet, M is the alpha multiplier and B is the protein buffer. The radius is automatically calculated from the membrane composition, but the alpha multiplier and protein buffer can be set using the subarguments `alpha_multiplier` and `protein_buffer`, whose default values are 1 and 0.132 nm (1.32 Å), respectively. Values to `protein_buffer` should be given in nanometers.

$$\alpha = \frac{1}{2 \cdot r \cdot M + B} \tag{7}$$

6.15.16 Rounding number of lipids: `lipid_rounding_function`

It often occurs that the number of lipids estimated to be placed within a leaflet ends up as a decimal value. The way this decimal is rounded can be set with the `lipid_rounding_function` subargument, whose options are shown below. Note that the `round` setting does not give the same result as the `int` setting. This is because the python "round()" function uses the "ties to even" rounding rule where values of 5 right after the final digit (after rounding) are always rounded to the even value, while the `int` setting uses the "ties away from zero" rounding rule where values with 5 right after the final digit (after rounding) are always rounded up.

Allowed `lipid_rounding_function` settings

setting	**Description**	**Examples where val is 0.5 and 1.5**
`int` (default)	Rounds up or down using int(val + 0.5).	int(0.5 + 0.5) = 1, int(1.5 + 0.5) = 2
`round`	Rounds up or down using the python "round(val)" function.	round(0.5) = 0, round(1.5) = 2
`floor`	Rounds down using the python "math.floor(val)" function.	math.floor(0.5) = 0, math.floor(1.5) = 1
`ceil`	Rounds up using the python "math.ceil(val)" function.	math.ceil(0.5) = 1, math.ceil(1.5) = 2

6.16 Protein arguments: `protein`

6.16.1 Choosing a file: `file`

The `protein` or `prot` argument is used to insert structures into specific positions within the box. It encompasses several subarguments, the most important being `file`, which requires a string specifying a path to the .pdb, .gro or .cif file containing the structure.

```
protein = "file:protein.pdb"
```

6.16.2 Placement and rotation: cx, cy, cz, rx, ry, rz and rotate

The position where the center of the structure should be placed can be set using cx / cy / cz (in nm). If one wants the structure rotated then one can use rx / ry / rz (in degrees) to rotate around the given axis.

```
protein = "file:protein.pdb cx:3 cz:3 ry:90"
```

One can also rotate a molecule multiple times by instead using the rotate subargument with the rx, ry and rz subarguments as shown below. The rotations are carried out in the order that they are given.

```
protein = " ".join([
    "file:protein.pdb",
    "rotate:rx:50:rz:90",
    "rotate:rx:25:ry:45",
])
```

6.16.3 Protein center designation cen_method

The centering method can be changed using cen_method subargument and has the following uses:

cen_method:cog or cen_method:mean_of_beads	Centers on the center of geometry / mean of all beads (default)
cen_method:axis or cen_method:mean_of_extremes	Centers on the mean of the extremes / axial distance
cen_method:bead:[int]	Centers on a specific bead or a series of beads
cen_method:res:[int]	Centers on a specific residue or a series of residues
cen_method:point:[fl]:[fl]:[fl]	Centers on a specific x:y:z point

The cen_method:res and cen_method:bead settings can be given multiple residue/bead values and residue/bead ranges. cen_method:res is shown in the following examples, but the syntax is identical for cen_method:bead. Note that when cen_method:res is used, it is not the beads within the residues that are used but instead the center of geometry of each residue.

Centers on a single residue

```
cen_method:res:5
```

Centers the mean coordinate of the four residues

```
cen_method:res:5:20:30:40
```

Centers on a series of residues (Including both residue 5 and 20).

```
cen_method:res:5-20
```

Centers on all residues from two series of residues.

```
cen_method:res:5-20:75-90
```

Centering on residues or beads is by default done on their coordinate mean. This can be changed to the axial mean of the selection instead by adding "_axis" to the subargument as shown below.

Centers on all residues from two series of residues. Centering on cog of residue centers is explicitly stated by adding "_cog" after "res"

```
cen_method:res_cog:5-20:75-90
```

Centers on all residues from two series of residues. Centering on axial mean of residue centers is explicitly stated by adding "_axis" after "res".

```
cen_method:res_axis:5-20:75-90
```

6.16.4 Structure topology name: `moleculetype`

If one wants to use topology files to obtain the charges of the protein, then the protein name under `[ moleculetype ]` can be designated with the `moleculetype` subargument (`moleculetypes` is also allowed). If there is no topology file, or if the specified `moleculetype` cannot be found in the topology file, then the program reverts to estimating charges from the amino acid names.

```
protein = "file:protfile.pdb moleculetype:Protein"
```

Should a file contain multiple structures (such as the following example where the file contains 1 Protein and 2 Ligand molecules, then the order of the moleculetypes must be in the same order that they appear in the structure file. There are multiple ways to indicate the presence of multiple structures in a file. The different ways are functionally the same but different ones may be nicer to write under different circumstances.

Each molecule is explicitly named the number of times that it is present:

```
protein = " ".join([
    "file:protfile.pdb",
    "moleculetype:Protein:Ligand:Ligand",
])
```

Each type of molecule is explicitly named with the number of the molecules added after the molecule name. Note that "Protein" does not have a specified number, as 1 is implied if none is given:

```
protein = " ".join([
    "file:protfile.pdb",
    "moleculetype:Protein:Ligand:2",
])
```

Each type of molecule is explicitly named in different `moleculetype` subarguments with the number of the molecules added after the molecule name. Again, "Protein" does not have a specified number, as 1 is implied:

```
protein = " ".join([
    "file:protfile.pdb",
    "moleculetypes:Protein",
    "moleculetypes:Ligand:2",
])
```

6.16.5 Membranes inside proteins: `inside_protein` and `membrane_border`

Normally, membranes are placed around proteins. In some cases, however, lipids need to be placed inside the proteins, such is the case for nanodiscs. This can be done in COBY by supplying two additional subarguments, one to the `protein` argument, and one to the `membrane` argument.

Within the `protein` argument, the subargument `membrane_border` needs to be set to `True` in order to designate the protein as a border to the membrane patch.

Within the `membrane` argument, the subargument `inside_protein` needs to be set to `True` in order to place the membrane patch within the protein, rather than around it. The two separate commands allow for a greater degree of user control — for instance, where there are two proteins in the system, but only one needs to contain a membrane patch.

The example below shows how the command can be used in a system that contains one membrane patch and two proteins (one surrounding the membrane patch, and the other inside the membrane patch).

```
protein = [
    ### The nanodisc
    "file:nanodisc.pdb moleculetypes:nanodisc \
        membrane_border:True",

    ### The protein in the center of the nanodisc
    "file:protein.pdb moleculetypes:protein",
],
### The membrane
membrane = "lipid:POPC inside_protein:True"
```

6.16.6 Buffer for solvation: `buffer`

During solvations, all protein beads are given a buffer distance as their radius to prevent solvent particles from being placed too close to them. This value is set to 0.132 nm (1.32 Å) by default (half the van der Waals radius of a regular bead in Martini 3), but can be changed using the subargument `buffer`, where the value given must be in nanometer.

6.16.7 Periodic boundary checking: `pbc_check`

It is, by default, checked that all beads in inserted proteins are contained within the system box. If they are found outside the box then they will be moved inside on the other side of the box, and a warning will be printed for the user. Should one not want this to happen, then it can disabled using `pbc_check:False`, though we highly recommend against this as it can cause problems with overlapping beads.

6.17 Solvation arguments: `solvation`

6.17.1 Basic solvent and ion specification

The `solvation` or `solv` argument can be used to solvate the system. Water or other solvents can be added using the `solv` subargument. Positive and negative ions can be added using the `pos` and `neg` subarguments, respectively.

```
solvation = "solv:W pos:NA neg:CL"
```

If one supplies the string "default" to the argument, then it will automatically be treated as "solv:W pos:NA neg:CL". Other subarguments can still be added to the argument after "default":

`solvation = "default"` is interpreted as `"solv:W pos:NA neg:CL"`

The `params` subargument can be used to set the default parameters for the specific `solvent` argument. Solvent-specific parameter designations share the same syntax as already seen with lipids.

`solvation = "solv:W:params:DevWater5 pos:NA neg:CL params:DevIons6"`

In this specific example, water parameters are taken from "DevWater5", while ion parameters correspond to "DevIons6".

The molarity (atomistic molarity) of the solvent and ions can be set using `solv_molarity` and `salt_molarity` subarguments, respectively. By default, they are set to:

- `solv_molarity:55.56`
- `salt_molarity:0.15`

```
solvation = " ".join([
    "solv:W pos:NA neg:CL",
    "solv_molarity:55.56 salt_molarity:0.15",
])
```

Different solvents and ions can be added in different ratios, designated by a number after the molecule name:

```
solvation = "solv:W:5 solv:SW:2 pos:NA:5 pos:CA:1 neg:CL"
```

If one wants to specify different parameters for each solvent, then it can be defined by adding `params:[LIBRARY]` to the subargument.

```
solvation = " ".join([
    "solv:W:5:params:DevWater4",
    "solv:SW:2:params:DevWater5",
    "pos:NA neg:CL",
])
```

6.17.2 Partial solvations: `center`, `cx`, `cy`, `cz`, `xlength`, `ylength`, `zlength` or `xmin`, `xmax`, `ymin`, `ymax`, `zmin`, `zmax`

It is possible to solvate only a partial volume. This can be done by either selecting the center of the solvent box (using `center:x:y:z` or `cx`, `cy` and `cz`) alongside its side lengths (using `xlength`, `ylength` and `zlength`) or by setting the minimum and maximum bounds of the box (using `xmin`, `xmax`, `ymin`, `ymax`, `zmin` and `zmax`).

Below are shown two examples, both creating a 10 nm by 10 nm by 10 nm box, where the top half of the box (along the z-axis) has an ion concentration of 0.15 mol/L (default), while the bottom half of the box has an ion concentration of 0.3 mol/L. The first example uses the center-length method, while the second example uses the min-max method of designating the solvent box.

Note that it is only necessary to designate the axes that are being changed as the remaining axes are assumed to how the dimensions of the system box.

```
COBY.COBY(
    box = [10, 10, 10],
    solvation = [
        "solv:W pos:NA neg:CL salt_molarity:0.15 \
        center:0:0:2.5  zlength:5",
        "solv:W pos:NA neg:CL salt_molarity:0.30 \
        center:0:0:-2.5 zlength:5",
    ]
)

COBY.COBY(
    box = [10, 10, 10],
    solvation = [
        "solv:W pos:NA neg:CL salt_molarity:0.15 \
        zmin:0 zmax:5",
        "solv:W pos:NA neg:CL salt_molarity:0.30 \
        zmin:-5 zmax:0",
    ]
)
```

Since the solvent boxes only differ in the z-axis, it is possible to abbreviate the first version to:

```
COBY.COBY(
    box = [10, 10, 10],
    solvation = [
        "solv:W pos:NA neg:CL salt_molarity:0.15 cz:2.5  zlength:5",
        "solv:W pos:NA neg:CL salt_molarity:0.30 cz:-2.5 zlength:5",
    ]
)
```

6.17.3 Charge neutralization: `salt_method`

Charge neutralization can be done in one of three ways using the `salt_method` subargument. The program first adds both positive and negative ions up to the specified concentration, after which one of the following is done:

`salt_method:add`	Adds extra ions to neutralize solvent box (default).
`salt_method:remove`	Removes excess ions to neutralize solvent box.
`salt_method:mean`	Both adds and removes ions. "Mean" of the other two settings.

6.17.4 Inserting an exact number of solvent molecules: `count`

The subargument `count` can be used to control the exact number of molecules that should be placed within a solvent box. If the subargument is set to `True`, then all ratio designations will be used as the absolute number of molecules of a given type of solvent or ion that should be inserted. The following example causes exactly 6000 W, 80 NA and 120 CL to be inserted in the solvent box.

```
solvation = "solv:W:6000 pos:NA:80 neg:CL:120 count:True",
```

6.17.5 Inserting solvent molecules based on the number of lipids

One can also set the number of solvent molecules that should be inserted to be calculated from the number of lipids present within the solvent box, by using the subargument `solv_per_lipid`. Note that an individual lipid will only be counted as being present within a solvent box if at least half of its beads are contained within it. In the example below, the system will have 20 W molecules inserted for each lipid in the solvent box. The number of ions that will be inserted is still calculated based on the volume of the solvent particles.

```
solvation = "solv:W pos:NA neg:CL solv_per_lipid:20",
```

By default it is required that more than half of the beads of a lipid are contained within the solvation box for the lipid to be counted as being "inside" the box. This value can be changed with the `solv_per_lipid_cutoff` subargument. The value must be between 1 and 0 (100% and 0%).

In the following example it is required that 30% of a lipids' beads must be contained for the lipid to be counted.

```
solvation = " ".join([
    "solv:W pos:NA neg:CL",
    "solv_per_lipid:20",
    "solv_per_lipid_cutoff:0.3",
])
```

Note that a lipid is only counted if MORE than the specified percentage of beads are within, meaning that setting the value to 0 will require more than 0% of beads (e.g. at least 1 bead).

6.17.6 Kick: `kick`

A kick is a small push that is applied to a molecule during insertion in order to prevent molecules from being positioned along straight lines. Solvation has a single kick value which can be set using `kick`. The default value is 0.066 nm (a quarter of the regular bead vdW radius).

```
kick:[float]
```

6.17.7 Solvations inside membranes: `solvate_hydrophobic_volume`

Solvent is by default prevented from being placed within the hydrophobic volume of a membrane, but there are be situations where this behavior is not wanted. The subargument `solvate_hydrophobic_volume` can be used both in `membrane` and `solvation` arguments with the following effects:

`membrane`	[True/False]. Sets that the membrane may have solvations place molecules inside the membranes hydrophobic volume. Default value is False.
`solvation`	[True/False]. Sets that the solvation should place molecules inside membranes hydrophobic volume. Default value is False.

Please note, that for the `solvate_hydrophobic_volume` subargument to have any effect, then it must be set to True in both a membrane and a solvation argument.

In the example below, the membrane is created with `solvate_hydrophobic_volume:True`, which designates that solvations may be done within the membranes hydrophobic volume.

The first solvation is set to only solvate with 0.5 water molecules per lipid inside the membrane's volume (with `zlength:2`). The `solvate_hydrophobic_volume:True` subargument allows for the solvent to be placed within the hydrophobic volume of the membrane. The subarguments `kick`, `gridres` and `buffer` are simply there to help the molecules fit in between the lipids during placement. The second solvation argument solvates the whole box, but is not allowed to place molecules inside the membrane as it does not contain a `solvate_hydrophobic_volume:True` subargument.

```
COBY.COBY(
    box = [10,10,10],
    membrane = " ".join([
        "lipid:POPC",
        "solvate_hydrophobic_volume:True",
    ]),
    solvation = [
    " ".join([
            "solv:W",
            "solv_per_lipid:0.5",
            "zlength:2",
            "solvate_hydrophobic_volume:True",
            "kick:0",
            "gridres:0.05",
            "buffer:0.001",
        ])
        "default",
    ],
)
```

6.17.8 Changing volume-related calculations and bead radius: `solvfreevol`, `ionsvol` and `bead_radius`

In order to understand how to select volume types and their effect for the system creation, first we need to explain the difference between various volume types. Parts of this has already been explained in section 5.5.

The first volume type is the "box volume" which is quite simply defined as the total volume of a solvation box, without taking into account whether parts of the box is occupied, as shown in equation 8.

$$V_{box} = x_{solvent\ box} \cdot y_{solvent\ box} \cdot z_{solvent\ box} \tag{8}$$

The second type is the "free volume", which is calculated by taking the box volume and subtracting the volume of all protein, lipid and solute beads placed within the solvent box, as shown in equation 9, where all beads are treated as being spherical with a radius of 0.264 nm (2.64 Å) (by default).

$$V_{free} = V_{box} - V_{proteins} - V_{lipids} - V_{solutes} \tag{9}$$

Without iterating too much from section 5.5; then the number of solvent particles that should be placed within a given volume can be calculated via equation 10, where N_A is Avogadro's constant, $K_{solvent}$ is an atomistic-to-CG mapping (e.g., $K_{RW} = 4$) and c is the molarity (which by default is 55.56 mol L^{-1}).

$$N_{solvent} = \frac{N_A \cdot V \cdot c_{solvent}}{K_{solvent}} \tag{10}$$

The third volume type, called the "solvent volume", can then be calculated from the number of solvent particles to be placed in the box as shown in equation 11, where M is the molar mass of the solvent and ρ is the density.

$$V_{solvent} = \frac{N_{solvent} \cdot K_{solvent} \cdot M_{solvent}}{N_A \cdot \rho_{solvent}} \tag{11}$$

The number of ion particles is calculated using equation 12.

$$V_{ion} = \frac{N_A \cdot V \cdot c_{ion}}{K_{ion}} \tag{12}$$

Which of these volumes are used for in the two number of particles equations (10 and 12) can be changed with the subarguments `solvfreevol` and `ionsvol`. The subargument `solvfreevol` must be either `True` or `False` (with it being `True` by default), which determines whether the the number of solvent particles will be calculated from the free volume (if `True`) or from the box volume (if `False`). The subargument `ionsvol` must be either `solv` (default), `free` or `box`, which sets the number of ions to be calculated either from the solvent volume, free volume or box volume, respectively.

The subargument `bead_radius` can be used to determine bead radius used when calculating the free volume. The value must be given in nanometers.

6.17.9 AA-to-CG Mapping and ratio calculations: `ratio_method` and `mapping`

Course-grained force fields like Martini 3 are fundamentally built on a concept of mapping, wherein a number of atomistic non-hydrogen atoms are mapped to 1 coarse-grained bead. For Martini 3 this is usually a 4-to-1 mapping but there are exceptions, such as for small and tiny water whose mappings are 3-to-1 and 2-to-1. This causes a contention with how solvent ratios should be calculated as the ratio between a number of CG beads is not the same as the ratio between the underlying atoms represented by the beads. Whether the atomistic or coarse-grained mapping ratio should be used can be set with `ratio_method:coarse` (for course-grained mapping) and `ratio_method:atomistic` (for atomistic mapping). Coarse-grained mapping is the default. The following example is adapted from the advanced tutorial titled "Solvation 4: Mixed Solvent".

In this example, a system is created with 5:3 ratios between regular water "W" and small water "SW". Table 6.17.9 shows the resulting number of both coarse-grained and atomistic W and SW molecules inserted and the coarse-grained and atomistic ratios.

	`ratio_method:coarse`	`ratio_method:atomistic`
Number of coarse-grained "W" water	4061	3680
Number of coarse-grained "SW" water	2436	2944
Total number of coarse-grained water	6497	6624
Number of atomistic "W" water	16244	14720
Number of atomistic "SW" water	7308	8832
Total number of atomistic water	23552	23552
Coarse-grained W:SW ratio	5.0 : 3.0	4.4 : 3.6
Atomistic W:SW ratio	5.5 : 2.5	5.0 : 3.0

```
COBY.COBY(
    box = [10, 10, 10], # [nm]
    membrane = "lipid:POPC",
    solvation = " ".join([
        "solv:W:5 solv:SW:3 pos:NA neg:CL",
        "ratio_method:coarse", # default
    ])
)

COBY.COBY(
    box = [10, 10, 10], # [nm]
    membrane = "lipid:POPC",
    solvation = " ".join([
        "solv:W:5 solv:SW:3 pos:NA neg:CL",
        "ratio_method:atomistic",
    ])
)
```

The mapping used for a given solvent, solute or ion is defined in its parameter file or specified when importing structures using the `molecule_import` argument. However, there may be occasions where one wishes to disable mapping from having any effect. This can be done by using the `mapping:False` subargument (the default value is `True`), which, effectively, sets the mapping of all used solvents, solutes and ions to be 1.

6.17.10 Modifying the solvent insertion algorithm: `gridres` and `buffer`

In order to understand how we can change the solvent insertion algorithm, we must first go through how it works. The solvent insertion algorithm functions by first creating a 3-dimensional grid with the corners of the grid corresponding to the corners of the solvation box. This grid is a 3-dimensional boolean array, where the "boolean" part simply means that all values in the array are either 1 or 0 (True or False). All values initially start out as True indicating that they are valid positions for solvent placement. The algorithm then iterates over the beads of all proteins, lipids and prior inserted solvent present within the solvation box, whereby it sets all grid points that are within a certain distance from the bead as being False, indicating the the grid point is occupied.

The distance between grid points can be set with the subargument `gridres`, which by default is 0.264 nm (2.64 Å), and the value given must be in nanometer. The algorithm automatically adjusts the `gridres` value if the diameter of the largest molecule in the current solvation plus twice the `kick` value is larger than than current `gridres` value. The new value becomes 1.2 multiplied by the diameter of the largest molecule and twice the `kick` value.

The distance used to set grid points as false is called a buffer and is unique for proteins, lipids and solvents/solutes. There is a default buffer value of 0.2 nm which is applied to all beads and can be changed with the `buffer` subargument. Extra buffers can be added to each of the three molecule types, using the subarguments `protein_extra_buffer`, `lipid_extra_buffer` and `solute_extra_buffer` for proteins, lipids and solvents/solutes, respectively. The values must be given in nanometer. The default values for `protein_extra_buffer`, `lipid_extra_buffer` and `solute_extra_buffer` are 0.2 nm, 0 nm and 0 nm (meaning that only proteins have extra buffers applied to their beads by default).

6.17.11 Solvating multiple volumes: `extra_insertion_box` and `extra_volume_box`

In some occasions, such as when dealing with solvent boxes spanning the PBC, it may be useful to be able to solvate multiple volumes with a single solvation argument. This can be done using the subargument `extra_insertion_box / eib`. The subargument can take six axis-direction:value pairs (axes: x, y, z, directions: max, min, value: [float/int], such as xmin:7) as subsubarguments. Note that if an axis-direction has not been given, then it is assumed to be the appropriate value for the system box. Multiple extra solvent boxes can be added with extra_insertion_box and all appropriate calculations are done on all of the added solvent boxes.

The example below effectively solvates the entire system, though the solvation algorithm processes each solvation box separately for solvent placement.

```
COBY.COBY(
    box = [10, 10, 10],
    membrane = "lipid:POPC",
    solvation = " ".join([
        "solv:W pos:NA neg:CL",
        ### Main solvation box
        "zmin:0 zmax:5",
        ### Extra solvation box
        "extra_insertion_box:zmin:-5:zmax:0",
    ])
)
```

Alternatively, if one does not want solvent to be placed within an extra solvent box, but only want to include the present volume and particles in occupancy calculations, then one can use the subargument extra_volume_box, which has identical subsubargument syntax to extra_insertion_box.

In the example below only the top half of the box is solvated, but it is solvated with a number of particles equivalent to the number that would have been used if the entire box was included in the main solvent box.

```
COBY.COBY(
    box = [10, 10, 10],
    membrane = "lipid:POPC",
    solvation = " ".join([
        "solv:W pos:NA neg:CL",
        ### Main solvation box
        "zmin:0 zmax:5",
        ### Extra solvation box
        "extra_volume_box:zmin:-5:zmax:0",
    ])
)
```

6.18 Flooding arguments: `flooding`

Flooding the system box with a specified molecule of choice can be done with the `flooding` argument, where the name and the number of requested molecules can be specified either directly or by using the `solute` subargument. In this example, 30 molecules of sucrose (SUCR) are added to the system.

```
flooding = "solute:SUCR:30"
```

Note that this works only if the topologies of the requested flooding molecule already exist in the solvent/ion libraries, which is likely not the case. Therefore, flooding molecule properties need to be defined in the `molecule_import` argument, explained previously.

6.19 Stacked membrane generator: `stacked_membranes`

6.19.1 Syntax via an example

COBY has a special argument called `stacked_membranes` which allows for the systemic creation of stacked membranes. Note that while the argument uses the `membrane` and `solvation arguments`, the syntax is slightly different. This argument is best explained using an example (the one shown here is the same one used in the Advanced tutorial notebook titled "Stacked Membranes 1: Three Bilayers").

Here is an example of a `stacked_membranes` argument:

```
COBY.COBY(
    x = 20,
    y = 10,
    stacked_membranes = " ".join([
        "number:3",
        "distance:5:3:3",
        "distance_type:surface",

        "membrane_subarg:positions:1:3 lipid:POPC:5 lipid:CHOL:1",
        "membrane_subarg:positions:2   lipid:POPE:3 lipid:CHOL:2",

        "solvation_subarg:positions:1   default solv_per_lipid:20",
        "solvation_subarg:positions:2:3 default solv_per_lipid:10",
    ]),
)
```

Figure 6.19.1 shows how the membrane and solvent spaces are numbered in relation to the z-axis.

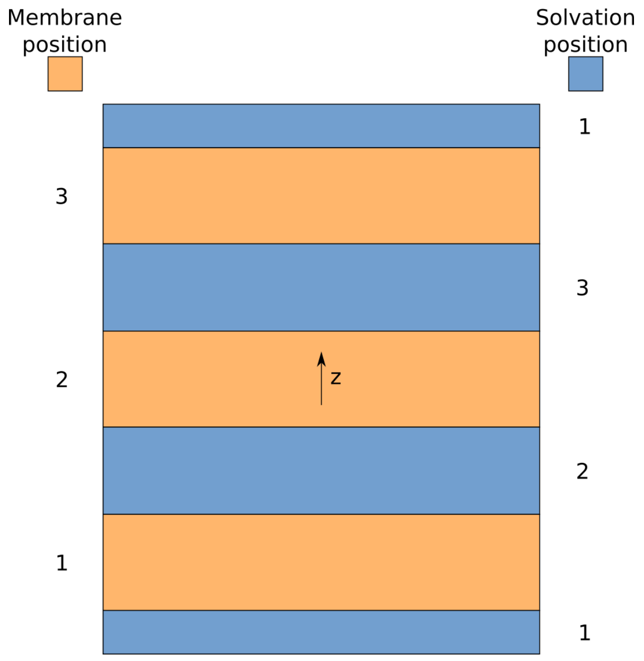


Figure 3: A schematic of a stacked membrane system.

When creating stacked bilayer systems, we do not need to provide a z-component to the box size. Instead, it is calculated directly from the `stacked_membranes` argument, as it will depend on the number of membranes and the distances between them. Therefore, the xy plane size is indicated by using the `x` and `y` arguments.

Next, the `stacked_membranes` argument takes a series of subarguments, those being `number`, `distance`, and `distance_type`, followed by the `membrane_argument` and `solvation_argument`.

`number:[int]`	Number of stacked membranes
`distance:[int]:[int]:[int]`	Distances between neighbouring membranes
`distance_type:[string]`	Inter-membrane distance calculation method.

The `distance` and `distance_type` subarguments must contain at least one value, in which case it is applied to all inter-membrane distances. Otherwise, it needs to match the number of requested membranes in `number`.

The `distance_type` subargument has three different options:

`distance_type:surface`	(default) Distance is calculated from the membrane bead that is the furthest away from the membrane center.
`distance_type:center`	Distance is calculated from the center of a membrane.

Example:

The `stacked_membranes` argument requires `membrane_argument` and `solvation_argument` to be specified as subarguments. Both need to be a specified as a part of a single string passed onto the `stacked_membranes` (which is handled by the `" ".join()` statement at the start of the argument).

Both `membrane_subarg` and `solvation_subarg` need a `positions` subsubargument, which designates integers associated with a specific membrane or solvent space within the stack. If no positions have been given, then it will be applied to all the positions. Membranes and solvent spaces are numbered bottom-up: e.g. Solvent 1 spans across the PBC by default (see subsection 6.20), followed by Membrane 1 as the lowermost membrane. The `positions` subargument designates which membranes or solvents should reflect the specified properties. The rest of the `membrane_argument` and `solvent_argument` (sub)subarguments and syntax are the same as described in subsection 6.15 and subsection 6.17.

If we return to the previous example, subargument `number:3` specifies that the stack contains three membranes. `positions` within the `membrane_argument` and `solvation_argument` refer to the specific membrane/solvent spaces. Note that multiple numbers can be given to the `positions` subarguments, which will result in all listed membranes/solvents to share the same following properties.

Finally, while the `solv_per_lipid` subargument is not unique to `stacked_membranes`, it can be used to specify how many solvent beads should be placed within a solvent space with regards to the number of lipids. In this instance, only lipids bordering a solvent space are taken into account.

6.20 Changing how the pbc is handled: `pbc`

The example above shows the default pbc behaviour though it may sometimes be useful to not split the first solvation position in half. This can be changed with the `pbc` subargument. The subargument can take one of three values with those being `split` (default), `bottom` / `bot` or `top` with the effect of each being showcased in figure 6.20.

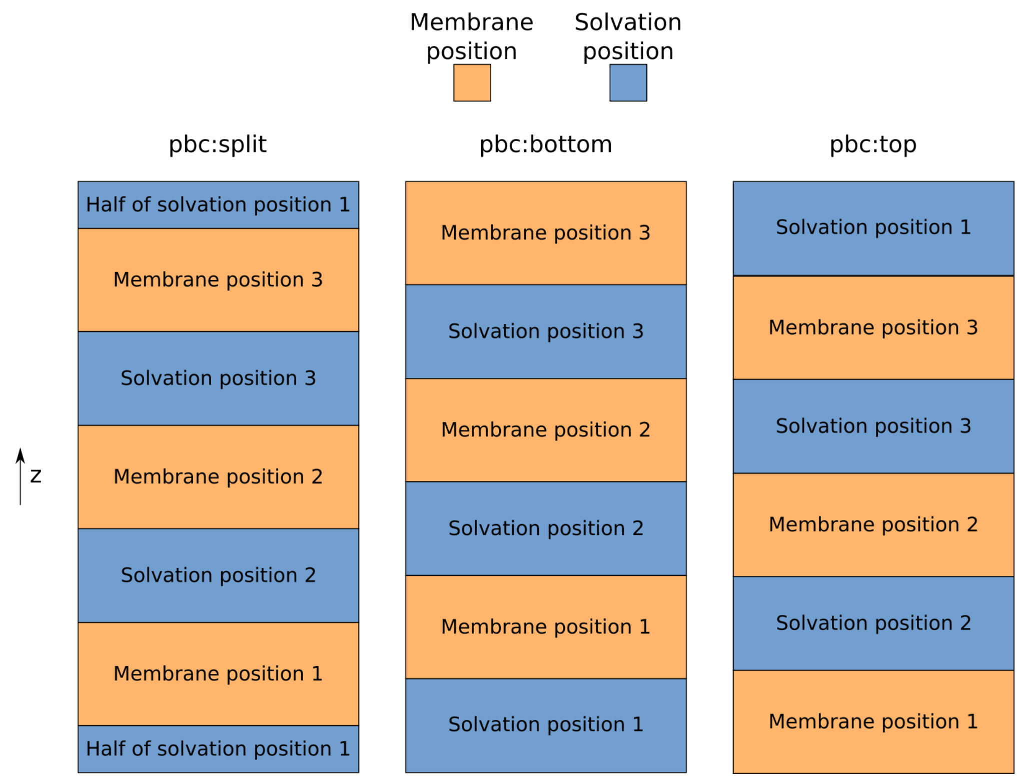


Figure 4: A schematic of a stacked membrane system.

6.21 pdbx/mmCIF files

Some structure files, like .pdb and .gro files, have specific types of atom information written in specific columns, which ensures that all .pdb and all .gro files follow the exact same structure. This is not true for .cif (pdbx/mmCIF) files, because they are built on a system of key:value pairing, wherein keys are initially defined after which the information for each atom must be written in the exact same order as the keys, though without there being any strict adherence to column positions. The values in .cif files are instead space separated.

Functionalities that import structures (such as the `file` subargument in the `protein` argument) can read the following atom keys. All other keys present are simply ignored.

```
'_atom_site.id'            (Atom number)
'_atom_site.auth_atom_id'  (Atom name)
'_atom_site.auth_comp_id'  (Residue name)
'_atom_site.auth_seq_id'   (Residue number)
'_atom_site.Cartn_x'       (X-value)
'_atom_site.Cartn_y'       (Y-value)
'_atom_site.Cartn_z'       (Z-value)
```

Written .cif files contain the following atom keys. The last four values are written because ChimeraX requires them to be present in order for it to read a .cif file. The last value is not very relevant for coarse-grained systems so `COBY` just writes the first letter of the atom name.

```
'_atom_site.id'              (Atom number)
'_atom_site.auth_atom_id'    (Atom name)
'_atom_site.auth_comp_id'    (Residue name)
'_atom_site.auth_seq_id'     (Residue number)
'_atom_site.Cartn_x'         (X-value)
'_atom_site.Cartn_y'         (Y-value)
'_atom_site.Cartn_z'         (Z-value)
'_atom_site.label_asym_id'   (Chain identifier. Always writes 'A')
'_atom_site.label_atom_id'   (Atom name)
'_atom_site.label_comp_id'   (Residue name)
'_atom_site.label_seq_id'    (Residue number)
'_atom_site.type_symbol'     (Atomic symbol. First letter of atom name)
```

7 Molecule System Builder: `COBY.Crafter`

7.1 Accessing the command

While all previously presented functionalities were linked to the `COBY.COBY` program, we implemented a second program, `COBY.Crafter` in the `COBY` package, which can be used to create individual molecules. In other words, `COBY.Crafter` skips the system building steps and instead creates a structure file of a single molecule.

The molecule can be obtained from the in-built libraries (lipid, solvent/solute, and ion libraries), or built from fragments using the Molecule Fragment Builder functionality.

`Crafter` can be accessed from within the Python environment using the following syntax:

```
import COBY
COBY.Crafter()
```

or

```
from COBY import Crafter
Crafter()
```

`Crafter` can also be accessed from a terminal by using the `--program` flag. If not specified, the `--program` flag defaults to COBY.

```
python -m COBY --program Crafter
```

7.2 Shared arguments

COBY.Crafter shares some of the arguments with COBY.COBY. The shared arguments are listed below:

Argument	Description in Section
out_all	6.6
out_sys	6.6
out_pdb	6.6
out_gro	6.6
out_cif	6.6
out_log	6.6
verbose	6.7
backup	6.8
randseed	6.9
itp_input	6.10
molecule_builder	6.13
import_library	6.14

While COBY.Crafter accepts the itp_input argument, the charge information from the topology file is not considered. Instead, COBY.Crafter only searches for the line addressing the Molecule Fragment Builder ;@COBY and, if the line exists, builds the requested molecule and places it into the TOP library.

The only argument that is unique to COBY.Crafter is molecule.

7.3 Molecule creation: molecule

The argument molecule can be used to instruct COBY.Crafter which molecules should be written to files. molecule takes three subarguments: moltype, name, and params.

Subargument	Effect	Default behaviour
moltype	Sets the type of a molecule (lipid, solvent/solute, pos_ion or neg_ion)	Required
name	Sets the name of a molecule.	Required
params	The parameter library of the molecule.	"default"

Lipids created with the molecule_builder argument are automatically added to the list of molecules that should be written, and therefore should not be explicitly specified with the molecule argument. The subargument molecule_builder was retained to keep internal consistency with the rest of COBY code. It should be noted that molecules built from topology files, by embedding a ;@COBY molecule_builder argument into the files, are not automatically added to the list of molecule arguments, and as such must be explicitly specified with molecule arguments.

The file names of the output structure files are generated by combining the file name prefix and the molecule name:

prefix_name.pdb/.gro/.cif

If the same molecule exists in other libraries, the parameter library name is also added to the file name. If there are multiple molecules with the same name and within the same library, the molecule type information is also added to the file name. The most verbose file-naming scheme is as follows:

prefix_name_params_moltype.pdb/.gro/.cif

The following example demonstrates a conventional use of COBY.Crafter. It creates structure files for water (W), phenylalanine (PHE), potassium (NA), chloride (CL), cholesterol (CHOL) and POPC. Another POPC is built using the Molecule Fragment Builder and is therefore automatically added to the list of output molecules.

```
import COBY
COBY.Crafter(
    molecule_builder = "moltype:phospholipid head:PC \
        tail1:CDCC tail2:CCCC name:POPC",

    molecule = [
        "moltype:solvent name:W",
        "moltype:solute name:PHE",
        "moltype:pos_ion name:NA",
        "moltype:neg_ion name:CL",

        "moltype:lipid name:CHOL",
        "moltype:lipid name:POPC",
    ],

    out_pdb = out.pdb,
    out_log = log.log,
)
```

This command call results in eight output files: one log file, and seven pdb files. Note that the POPC files have their `params` as a part of their file output names in order to differentiate the two outputs.

```
log.log         out_CL.pdb   out_PHE.pdb            out_POPC_LFB.pdb
out_CHOL.pdb    out_NA.pdb   out_POPC_default.pdb   out_W.pdb
```

8 Parameter library explorer: `COBY.Library`

Finding the correct lipid, solvent or ion can sometimes be difficult without looking through the definition source files that COBY uses. The method `COBY.Library` solves this problem by allowing the user to interactively navigate through the parameter libraries for lipids, solvents/solutes, ions and protein residues (only used for automatic charge determination). The method can easily be used by simply replacing the method `COBY.COBY` with `COBY.Library`. All arguments given to `COBY.Library` that are not specifically related to definition processing (such as importing custom definitions or creating custom molecules using the molecule fragment builder) are completely ignored, as shown in the example below.

Before:

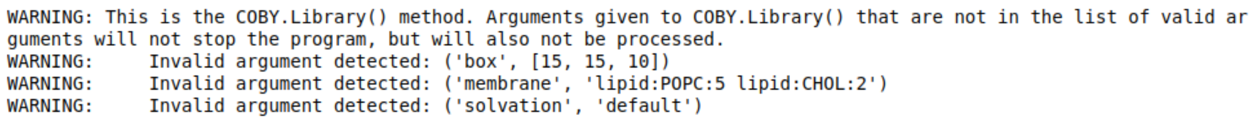
```
### In python
COBY.COBY(
    box = [15, 15, 10],
    membrane = "lipid:POPC:5 lipid:CHOL:2",
    solvation = "default",
)
```

```
### In terminal
python -m COBY \
    -program COBY \ # Default, shown for clarity
    -box 15 15 10 \
    -membrane lipid:POPC:5 lipid:CHOL:2 \
    -solvation default
```

After:

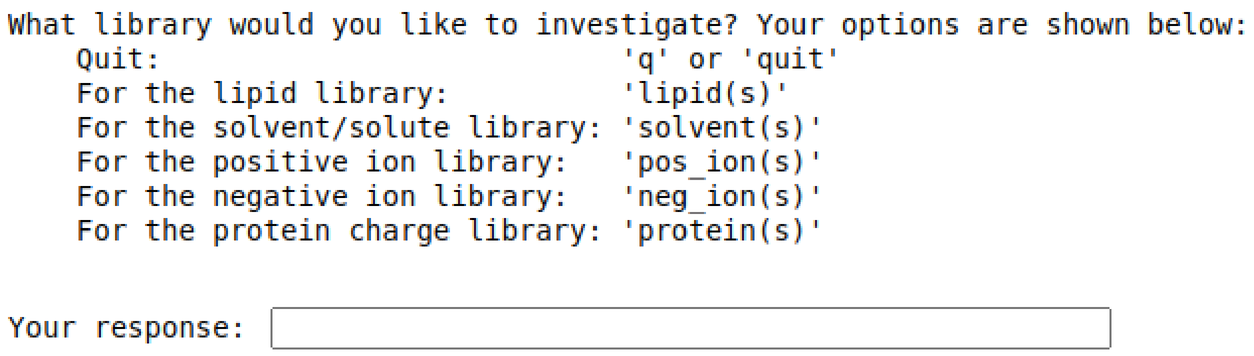
```
### In python
COBY.Library(
    box = [15, 15, 10],
    membrane = "lipid:POPC:5 lipid:CHOL:2",
    solvation = "default",
)

### In terminal
python -m COBY \
    -program Library \
    -box 15 15 10 \
    -membrane lipid:POPC:5 lipid:CHOL:2 \
    -solvation default
```

Running the "After" code generates the following warnings simply to inform the user that the given arguments are not processed:



```
WARNING: This is the COBY.Library() method. Arguments given to COBY.Library() that are not in the list of valid ar
guments will not stop the program, but will also not be processed.
WARNING:     Invalid argument detected: ('box', [15, 15, 10])
WARNING:     Invalid argument detected: ('membrane', 'lipid:POPC:5 lipid:CHOL:2')
WARNING:     Invalid argument detected: ('solvation', 'default')
```

A prompt then appears in the terminal with a list of instructions. The method should be self-explanatory from here on.



```
What library would you like to investigate? Your options are shown below:
    Quit:                          'q' or 'quit'
    For the lipid library:         'lipid(s)'
    For the solvent/solute library: 'solvent(s)'
    For the positive ion library:   'pos_ion(s)'
    For the negative ion library:   'neg_ion(s)'
    For the protein charge library: 'protein(s)'

Your response:
```