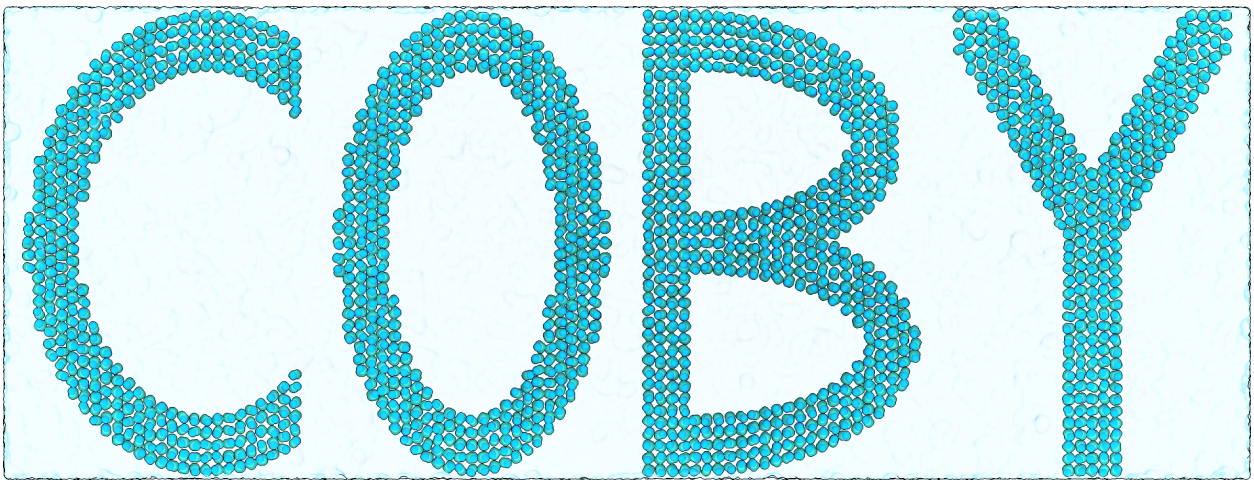
Coarse-Grained System Builder (COBY)

Documentation (for version 0.2.1)

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"
)
```

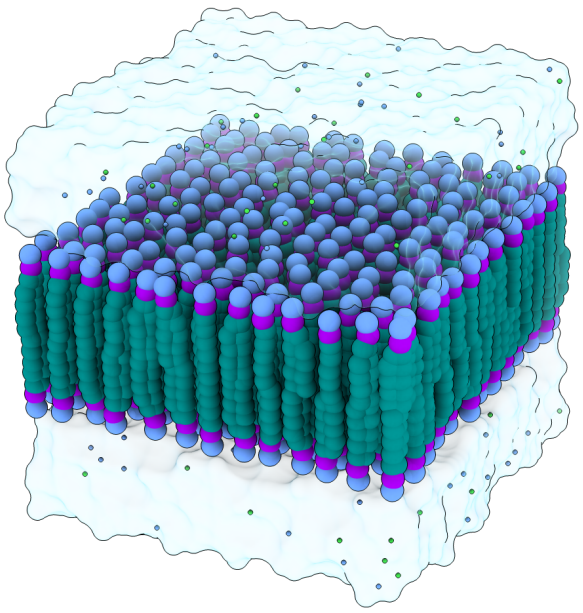


Figure 1: Pure POPC membrane in water with 0.15 M NaCl.

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.

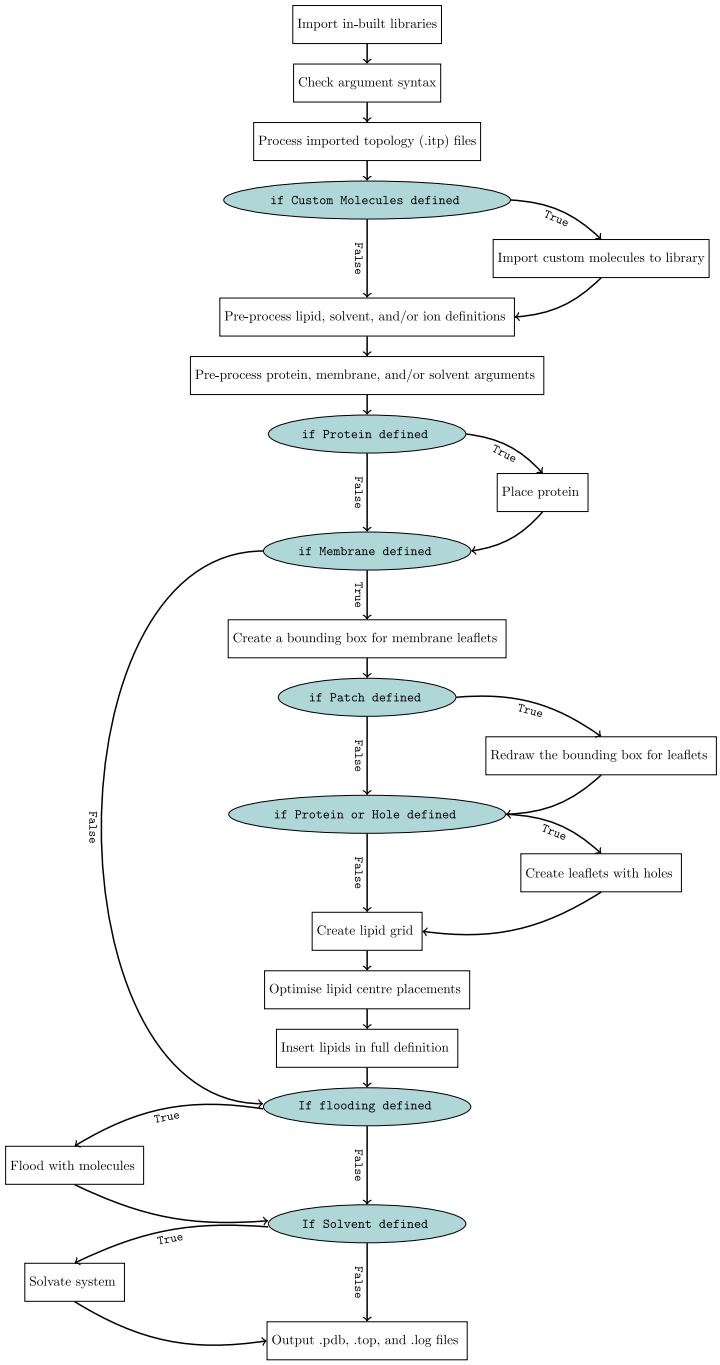


Figure 2: **COBY workflow.**

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.

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

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.

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

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

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

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 ]` and 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, 10],
    membrane1 = "lipid:POPC apl:0.5",
    membrane2 = "lipid:POPE apl:0.4"
)
```

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 (in `pdb_unitcell`), or as in .gro files (in `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 the 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 = [v1(x), v2(y), v3(z)]
```

Non-rectangular boxes:

```
gro_unitcell = [v1(x), v2(y), v3(z), v1(y), v1(z), v2(x), v2(z), v3(x), v3(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.

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 or .gro), and a truncated topology file, which contains `[ system ]` and `[ molecules ]` section. If the output names are unspecified, the files are saved under output.gro, output.pdb, 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_sys = path/to/file,      # saves both .pdb and .gro 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` and `out_gro`, respectively.

Alternatively, if one wants identical file names for all output files (.pdb, .gro, .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 into the .log file, as long as `out_log` or `out_all` has been specified.

```
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` (`#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) must be specified with the `file` subargument.

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

Version 1: If molecule topology exists:

```
molecule_import = " ".join([
    "file:[structure.pdb]",
    "moleculetype:[molname]",
])
```

In this instance, molecule charges are read from the topology file.

Version 2: If molecule topology does not exist:

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

In this instance, molecule names 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 `name` subarguments.

`moleculetype:[str]`	Uses the string both as a reference name and as the `[ moleculetype ]` for charge determination.
`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 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"
```

6.12.2 Parameter libraries 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"
```

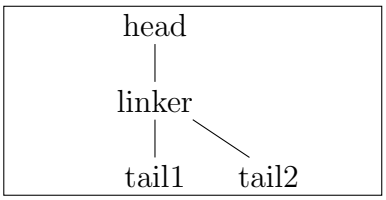
6.13 Molecule Fragment Builder: `molecule_builder`

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). Currently, the only available molecule type that can be built using the in-built fragment library are **phosphoglycerides** (phospholipids). However, if provided by custom fragment libraries, COBY can build molecules with an arbitrary number of molecular groups (for lipids, this is most relevant in terms of building lipids with $N_{\text{tails}} \neq 2$).

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 created 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



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"
```

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 a 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.

```
import_library = [
    "New_LipidScaffolds.py",
    "New_Fragments.py",
    "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 two possible values: `upper` or `lower`. If only one 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`. "
" designates that the string is continued on the next line and is purely shown for readability.

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:

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

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 $N_{\text{max lipids}}$ (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"
```

6.16 Protein arguments: `protein`

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 or .gro file containing the structure.

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

6.16.1 Placement and rotation `cx`, `cy`, `cz`, `rx`, `ry`, `rz`

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"
```

6.16.2 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
```

6.16.3 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",
])
```

6.16.4 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"
```

6.17 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

It is possible to solvate only a partial volume by setting the solvent centre using either `center:x:y:z` or one of the axis-specific arguments, `cx`, `cy` and `cz`.

This example creates 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.

```
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",
    ]
)
```

Since the solvent boxes only differ in the z-axis, it is possible to abbreviate the argument 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",
```

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]
```

6.18 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`

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

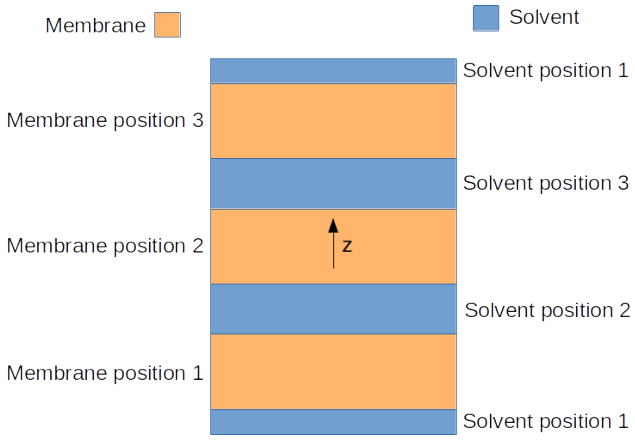


Figure 3: A schematic of a stacked membrane system.

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.

If no `number` subargument is given, then it will be determined based on the number of membrane and solvation arguments given.

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.
`distance_type:mean_surface`	Distance is calculated from an averaged membrane surface z-coordinate. First, the code finds the bead furthest from the center for each lipid, and then takes the mean of all found lipid beads in the membrane. The distance values for each lipid is weighted based on the inter-lipid ratios.

Example:

Assume we have a membrane consisting of two lipids, (L1 and L2) in a 5:2 ratio. The lipid beads furthest from the membrane centre are 2 nm and 1.5 nm from the membrane centre for L1 and L2, respectively. This results in a following calculation:

$$\mathrm{s} = \frac{\sum w_{\mathrm{lipid}} \cdot d_{\mathrm{lipid}}}{\sum w_{\mathrm{lipid}}} \tag{7}$$

$$\mathrm{s} = \frac{w_{\mathrm{L1}} \cdot d_{\mathrm{L1}} + w_{\mathrm{L2}} \cdot d_{\mathrm{L2}}}{w_{\mathrm{L1}} + w_{\mathrm{L2}}} \tag{8}$$

$$\mathrm{s} = \frac{5 \cdot 2 \text{ nm} + 2 \cdot 1.5 \text{ nm}}{5 + 2} = 1.86 \text{ nm} \tag{9}$$

where w are weights/ratios, d is the largest distance from the centre of a given lipid, and s is the resulting mean distance.

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, 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.

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_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 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
```

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
```

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 the `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
```