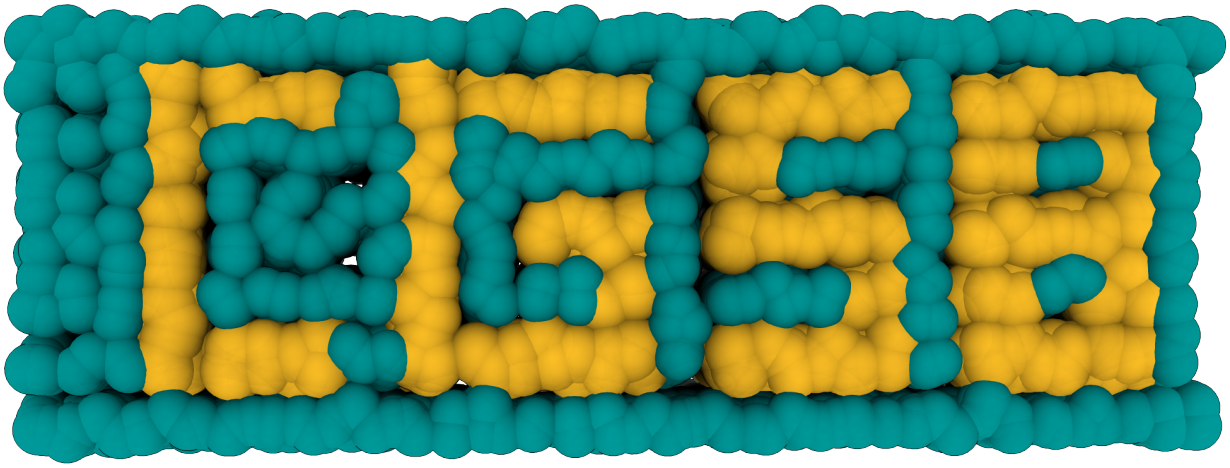
Coarse-grained System Builder (CGSB): Documentation

Mikkel D. Andreasen, Lorena Zuzic



Contents

1 Introduction

Coarse-grained System Builder (CGSB) is a versatile and easy-to-use Python-based program for building coarse-grained complex membranes for use in molecular dynamics simulations. CGSB 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.

CGSB can be used both as a **package within a Python environment**, or as a **command-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, including adding custom lipid types
- 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/CGSB. In principle, one needs to only create a Python environment with correct packages (correct environment is available through the `environment.yml` file).

```
conda env create -f environment.yml
```

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

```
conda activate CGSB
python -m ipykernel install --user --name=CGSB
```

3 Cite us

Coming up...

4 Quickstart

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

```
import sys
sys.path.append("path/to/CGSB")
import CGSB

CGSB.CGSB(
    box = [10, 10, 10],
    membrane = POPC,
    solvation = ""
)
```

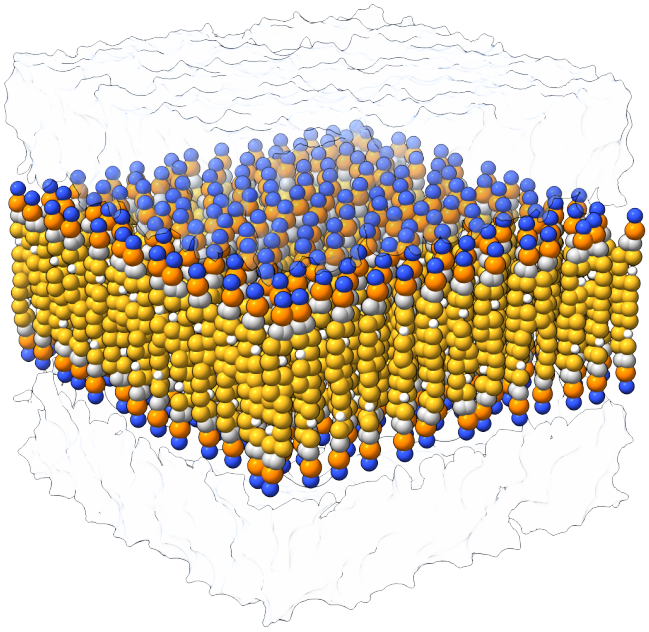


Figure 1: Pure POPC membrane in water with 0.15 M NaCl.

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

```
python CGSB.py -box 10 10 10 --membrane POPC \
--solvation solv:W pos:NA neg:CL
```

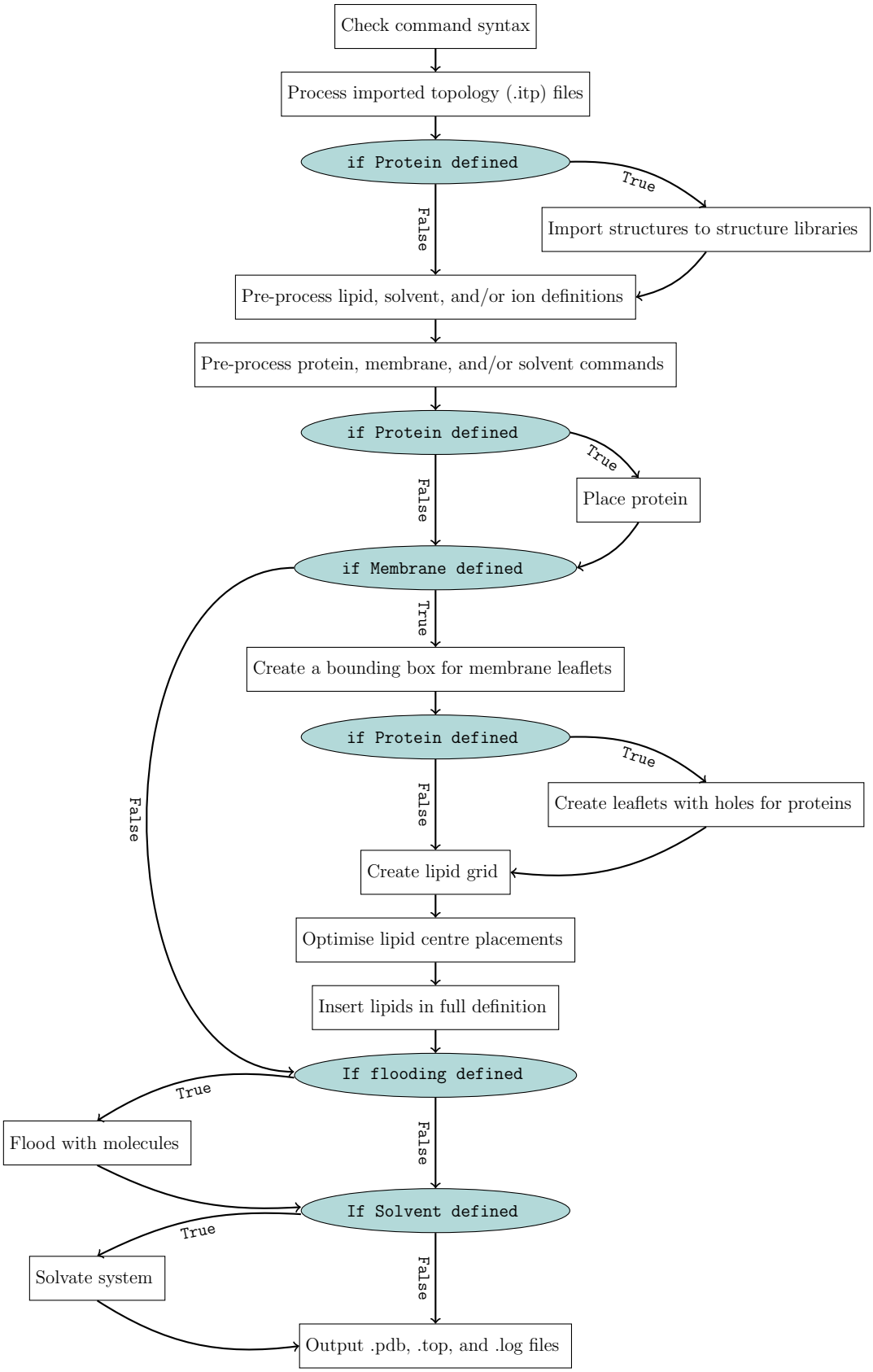
5 General feature overview

The main features of CGSB include:

- It can be imported as a package in a Python environment or run directly from a terminal command-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** (where each segment can feature a different lipid composition).
- It can build **stacked membranes** (where each membrane can be composed of a different lipid composition).
- Optimisation of the positioning of lipid centres within each leaflet to alleviate artefacts.
- Handles cubic, hexagonal, and rhombic dodecahedron system box shapes
- Allows for easy import of new lipid types.
- Handles one or more **membrane proteins** and their placement within the bilayer.
- Performs **system solvation** with a correct handling of ion molarities
- Can perform **partial solvations** within a given volume.
- Can be used to set up **flooding simulations** with one or more molecules of choice.

5.1 Workflow

Steps are run sequentially, and progress further only if the previous steps run successfully. In principle, a minimal executable command requires only a single element of any system: e.g., a lipid (resulting in a homogenous membrane), a solvent (creating a solvent box), or structure (placing the structure in a box). This means that the user has full flexibility in the choice of the building blocks.


Check command syntax
Process imported topology (.itp) files
if Protein defined
True
Import structures to structure libraries
False
Pre-process lipid, solvent, and/or ion definitions
Pre-process protein, membrane, and/or solvent commands
if Protein defined
True
Place protein
False
if Membrane defined
True
Create a bounding box for membrane leaflets
if Protein defined
True
Create leaflets with holes for proteins
False
Create lipid grid
Optimise lipid centre placements
Insert lipids in full definition
If flooding defined
True
Flood with molecules
False
If Solvent defined
True
Solvate system
False
Output .pdb, .top, and .log files
False

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 "minimized" using a custom written algorithm to prevent 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 the 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.

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 handles any given structure (if provided with a coordinate file).

Any number of structures files can be inserted, and they can be moved and rotated based on their center of geometry (COG). Multiple methods available for designating the structure's centre, based on the centering on a coordinate point, axial mean coordinate, residue COG, structure COG, or a bead.

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

5.4 Flooding

If a `flooding` command 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 depends on the solvent type, and for water is equals $c = 55.56$ 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.

5.6 Topology processing

Charge information can be read from the 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 to their `[ moleculetype ]`) specified under their designated commands.

6 Command syntax

CGSB functions on a system of commands and subcommands. A general syntax for both terminal and script use is shown here, with `command_call` being a stand-in for more specific commands that can be used.

Python script:

```
CGSB.CGSB(
    command_call = "cmd"
)
```

Terminal command-line:

```
python CGSB.py --command_call cmd
```

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

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

```
CGSB.CGSB(
    command_call = ["cmd1", "cmd2"]
)
```

or

```
CGSB.CGSB(
```

```
    command_call1 = "cmd1",
    command_call2 = "cmd2"
)
```

In Terminal, the flags can be repeated without the need to number them:

```
python CGSB.py -command_call cmd1 -command_call cmd2
```

Subsequent subcommands are given as `SETTING:VAL` (or `SETTING:VAL1:VAL2:...`), where `SETTING` designates the setting to be changed, while the following `VAL` separated by colons are the values to be given to the setting. The number of values that can be assigned to a setting depends on the specific setting.

Subcommands given to a command call only affect that specific command string. For example, building of an asymmetric membrane requires lipid composition specification for each individual leaflet. Under these circumstances, properties assigned to one leaflet do not transfer to another.

```
CGSB.CGSB(
    box = [10, 10, 10],
    membrane = [
        "leaflet:upper POPC apl:0.5",
        "leaflet:lower 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.

6.1 Box commands: `box`, `x`, `y`, `z`

The `box` or `pbc` command is used to set the side lengths of the box in x, y, and z direction. The commands `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 one gives only two values using the `box` command or if either the `x` or `y` commands are omitted, while using a rectangular box type, then the box will be made square in the x/y-plane using the first side length value in the list. If only a single value is given using the `box` command or if only one of the `x`, `y` or `z` commands are used, then the box will be made cubic.

There are certain shortcuts one can use to assign the box. If only two values are specified, they are interpreted as a side length of a square xy plane, and the z length.

```
box = [15, 10]   converted to box = [15, 15, 10]
x = 15, z = 10   converted to box = [15, 15, 10]
y = 15, z = 10   converted to box = [15, 15, 10]
```

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

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

6.2 Box types: box_type

The box_type or pbc_type commands can be used to set the type of box. The available box types are rectangular (default), hexagonal, and 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 side length values: x ∧ y ∧ z
box_type = "hexagonal"       2 side length values: ( x ‖ y) ∧ z
box_type = "dodecahedron"    1 side length value: x ‖ y ‖ z
box_type = "optimal"         same as "dodecahedron"
```

6.3 Topology system name: sn

The sn command can be used to set the name of the output system in the topology file.

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

6.4 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 and .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_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 command `out_all` can be used.

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

6.5 Verboseness of terminal printing: `verbose`

The verboseness of the text printed to the terminal can be modified using the `verbose` command. The smaller the number the fewer details are printed to the terminal. Note that the `verbose` command has no impact on the text written to a log file using the `out_log` command.

```
verbose = 6        # default
```

6.6 Backup of overwritten files: `backup`

Command `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]+#`.

6.7 Random seed `rand`

The `rand` command can be used to set the random seed:

```
rand = 5
```

6.8 Topology input: `itp_input`

The `itp_input` command can be used to give the program topology data which will be used to calculate charges.

```
itp_input = "top_for_CGSB.itp"
```

6.9 General parameters: `sys_params`

`sys_params` command can be used to specify a global parameter library that will be used for all lipids, proteins, and solvents, unless specified otherwise. In order to be parsed correctly, the name of the parameter library needs to include at least one alphabetical character.

6.10 Specific parameters: `lipid_params`, `solv_params`, `prot_params`

The `lipid_params`, `solv_params`, and `prot_params` 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 6.11.1).

```
sys_params = "default",
```

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

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

6.11 Membrane composition: `membrane`

6.11.1 Basic lipid specification

The `membrane` or `memb` command 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 = "POPC:5"
```

If another string is added after the ratio number, then the lipid will be looked for in that specific set of parameters. In addition, the `params` subcommand can be used to set the default parameters for the specific `membrane` command. If provided, lipid-specific ff designations overwrite the membrane ff designation:

```
membrane = "POPC:5 POPE:2.5 CHOL:1:dev18 params:PhosDev13"
```

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

6.11.2 Area per lipid designation: `apl`

Subcommand `apl`, 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

6.11.3 Membrane types: `type`

Subcommand `type` 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:upper`	upper leaflet (in isolation works the same as `type:mono`)
`type:lower`	lower leaflet (in isolation creates a downwards-facing monolayer)

6.11.4 Leaflet specification: `leaflet`

`leaflet` subcommand takes two possible values: `upper` or `lower`. If only one is specified, the script creates a monolayer. In a sense, this command offers an overlapping functionality with the `type` subcommand.

Asymmetry can be created in three different ways:

Version 1: by using subsequent `leaflet` subcommands within the same command string. In this case, lipid property subcommands will be applicable to the last called `leaflet`. " " designates that the string is continued on the next line and is purely shown for readability. subcommands given before the first `leaflet` command or after `leaflet:both` will apply to both leaflets.

```
membrane = [
    "apl:0.5 \                      # applies to whole membrane
     leaflet:upper POPC:5 CHOL:1 \ # applies only to upper leaflet
     leaflet:lower POPC:3 CHOL:2 \ # applies only to lower leaflet
     leaflet:both params:Dev18"    # applies to whole membrane
]
```

Version 2: By using multiple strings in the membrane command. 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` subcommands must be given in each individual string in this case:

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

Version 3: by using multiple `membrane` commands. Remember that multiple calls to the

same command requires adding a number after it. Note that the `apl` and `params` subcommands must be given in each individual string in this case:

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

6.11.5 Phase-separated membranes

By default, the membrane will fill the entire xy-plane. However, this can be modified by using the `x`, `y`, which define xy lengths of the membrane patch, and `center`, `cx`, `cy`, and `cz` subcommands, 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` subcommands 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 = [
    "POPC:5 CHOL:1 x:5 center:2.5:0:0",
    "POPC:4 CHOL:2 x:5 center:-2.5:0:0"
]
```

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

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

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

```
CGSB.CGSB(
    box = [10,10,10],
    membrane = [
        "x:5 cx:2.5 \
        leaflet:upper POPC:5 CHOL:1 \
```

```
        leaflet:lower POPC:3 CHOL:2",

        "x:5 cx:-2.5 \
        leaflet:upper POPC:5 POPE:2 CHOL:1 \
        leaflet:lower POPC:5 CHOL:1"
    ]
)
```

6.11.6 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 `lipid_optim` subcommand.

`lipid_optim:avg_optimal`	(default) Optimisation based on the mean lipid deviation from the expected ratios
`lipid_optim:abs_val`	Treats lipid ratios as actual number of lipids
`lipid_optim:fill`	Attempts to fill the leaflet according to the apl regardless of how skewed the ratios would become. Stops if perfect ratio is reached.
`lipid_optim:force_fill`	Same as `fill`, but forces the leaflet to be filled completely.
`lipid_optim:no`	Does not attempt to optimise the lipid ratios. Works the same as `insane.py`

6.12 Protein commands: `protein`

The `protein` or `prot` command is used to insert structures into specific position within the box. It encompasses several subcommands, the most important being `prot_file` or `file`, which requires a string specifying a path to the structure .pdb file.

```
protein = "prot_file:protein.pdb"
```

The structure can be translated using `tx` / `ty` / `tz` (in nm) and rotated using `rx` / `ry` / `rz` (in degrees).

```
protein = "prot_file:protein.pdb" tx:3 tz:3 ry:90"
```

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

`cen_method:cog`	Centre of geometry (COG) of all structure beads (default)
`cen_method:axis`	Centers on the axial mean coordinate
`cen_method:bead:beadnr`	Centers on a specific bead number
`cen_method:res:resnr`	Centers on the COG of a specific residue
`cen_method:point:x:y:z`	Centers on a specific point in the coordinate system

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`.

Example 1: Centers on a single residue

```
cen_method:res:5
```

Example 2: Centers the mean coordinate of the four residues

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

Example 3: Centers on a series of residues.

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

Example 4: Centers on all residues from two series of residues.

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

6.12.1 Structure topology name: `mol_names`

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

```
protein = "protein.pdb mol_names:Protein1:Ligand1:Ligand1"
```

In this example, the "Protein.pdb" file, which contains one protein structure and two ligands of the same type, will be matched with the appropriate topologies as specified under `mol_names` subcommand.

6.13 Solvation commands: `solvation`

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

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

If one supplies an empty string to the command, then it will automatically be treated as "solv:W pos:NA neg:CL":

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

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

```
solvation = "solv:W: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` (can be abbreviated to `salt`) subcommands, respectively. By default, they are set to:
`solv_molarity:55.56` and `salt_molarity:0.15`.

```
solvation = "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 solvent name:

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

If one also wants to specify different parameters for each solvent, it needs to be specified after the ratio number.

```
solvation = "solv:W:5:DevWater4 solv:SW:2:DevWater5 pos:NA neg:CL"
```

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

This example creates a 10 nm by 10 nm by 10 nm box, wherein 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. Note that the `salt` command is the abbreviated version of `salt_molarity`

```
CGSB.CGSB(
    box = [10, 10, 10]
    solvation = [
        "solv:W pos:NA neg:CL salt:0.15 center:0:0:2.5  z:5",
        "solv:W pos:NA neg:CL salt:0.30 center:0:0:-2.5 z:5",
    ]
)
```

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

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

Charge neutralization can be done one of three ways using the `salt_method` subcommand, with the `add` setting being default. Note that the program first adds 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 settings

6.14 Flooding commands: `flooding`

Flooding the system box with a specified molecule of choice can be done with the `flooding` command, where the name and the number of requested molecules can be specified either directly or by using the `solv` subcommand. In this example, 30 molecules of rhodamine (RHO) are added to the system.

```
flooding = "solv:RHO:30"
flooding = "RHO: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 `solute_input` command, explained in the next section.

6.14.1 Solvent topology: `solute_input`

The `solute_input` command can be used to import solute structures, which are added to the solvent/ion libraries. Therefore, this command can be used to specify a solvent type that is not water, or a flooding molecule. In principle, any number of new solvent/solute molecules can be defined.

There are several subcommands that can be used in assigning the solute topologies, namely:

`solute_file` / `file`	.pdb file that contains solute coordinates
`names`	string specifying solute name. Required unless `n_residues = 1`. In that case, it reads and uses the residue name from the .pdb file.
`n_residues` / `nres`	integer value (default = 1); number of residues in the solute
`charges`	int/float value (correspoding to charge) or string (corresponding to the `moleculetype` in the topology file. Default = 0.

`Solute_input` commands can be given in two different ways that have an equivalent effect.

Version 1: A subcommand written in a format of `mol:name:n_residues:charge`. Each `mol` subcommand is new molecule.

The example shows "ligands.pdb" file that contains one rhodamine (RHO), 1 peptide containined 8 residues (PEP1), and one unspecified molecule (resname) with a charge of +2.

```
solute_input = ["solute_file:ligands.pdb \
                mol:RHO:1:rhodamine \
                mol:PEP1:8:peptide_1 \
                mol:resname:1:2"]
```

Version 2: Subcommand inputs (`name`, `n_residues`, `charges`) are assigned multiple values, where each corresponds to one solute type, respectively. The same number of values must be given to each of the subcommands.

```
solute_input = ["solute_file:ligands.pdb \
                 names:RHO:PEP1:resname \
                 n_residues:1:8:1 \
                 charge:rhodamine:peptide_1:2"]
```

The `names` can be used to refer back to the specific solutes in `flooding` or `solvation` commands.

Here is another example of how to use the `flooding` and `solute_input` commands in tandem to perform flooding. In this example, we are adding 50 molecules of LIG1 and 10 molecules of LIG2 containing 2 and 3 residues, respectively. LIG1 molecule has a total charge of +2, and LIG2 has a total charge specified in the topology file, in which it is called Ligand2 (as specified in the `[ moleculetype ]` section). Both molecules are contained in the `ligands.pdb` file. Note that the order of solutes under `mol` needs to follow the order in the .pdb file.

```
flooding = ["LIG1:50",
            "LIG2:10"],
solute_input = ["file:ligands.pdb \
                 mol:LIG1:2:2 \
                 mol:LIG2:3:Ligand2"]
```

or

```
flooding = ["LIG1:50",
            "LIG2:10"],
solute_input = ["file:LIGANDS.pdb \
                 names:LIG1:LIG2 \
                 n_residues:2:3 \
                 charges:2:Ligand2"]
```

7 Adding new lipid type parameters

Work in progress.