A Hierarchical, Component Based Approach to Screening Properties of Soft Matter

Christoph Klein, János Sallai, Trevor J. Jones, Christopher R. Iacovella, Clare McCabe and Peter T. Cummings

Abstract In prior work, Sallai, et al. introduced the concept and algorithms of building molecular topologies through the use of a hierarchical data structure and the use of an affine coordinate transformation to connect molecular components. In this work, we expand upon the original concept and present a refined version of this software, termed `mBuild`, which is a general tool for constructing arbitrarily complex input configurations for molecular simulation in a programmatic fashion. Basic molecular components are connected using an equivalence operator which reduces and often removes the need for users to explicitly rotate and translate components as they assemble systems. Additionally, the programmatic nature of this approach and integration with the scientific Python ecosystem seamlessly exposes high-level variables that users can tune to alter the chemical composition of their systems, such as mixtures of polymers of different chain lengths and surface patterning. Leveraging these features, we demonstrate how `mBuild` serves as a stepping stone towards screening and performing optimizations in chemical

C. Klein (✉) · T.J. Jones · C.R. Iacovella · C. McCabe · P.T. Cummings
Department of Chemical and Biomolecular Engineering, Vanderbilt University, Nashville, TN 37235, USA
e-mail: christoph.klein@vanderbilt.edu

P.T. Cummings
e-mail: peter.cummings@vanderbilt.edu

J. Sallai
Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN 37235, USA

C. Klein · C.R. Iacovella · C. McCabe · P.T. Cummings
Vanderbilt Multiscale Modeling and Simulation (MuMS) Facility, Vanderbilt University, Nashville, TN 37235, USA

C. McCabe
Department of Chemistry, Vanderbilt University, Nashville, TN 37235, USA

R.Q. Snurr et al. (eds.), *Foundations of Molecular Modeling and Simulation*, Molecular Modeling and Simulation, DOI 10.1007/978-981-10-1128-3_5

parameter space of complex materials by performing automated screening studies of monolayer systems as a function of graft type, degree of polymerization, and surface density.

Keywords Molecular dynamics · Software · System construction

1 Introduction

The biophysics simulation community has put considerable effort into creating tools and databases for building and parameterizing biological molecules with minimal effort, e.g. the Protein Data Bank [1], VMD [2], AmberTools [3], the Omnia suite [4]. Such toolchains allow researchers to generate input files for complex structures, such as proteins and DNA, that can run on most molecular dynamics simulation engines with little to no manual intervention. However, while the biophysics community's tools provide excellent functionality for biological system setup, they do not allow one to easily generate arbitrary structures found outside the biophysics community. For example, surface bound brushes or tethered nanoparticles, which often feature semi-infinite substrates and/or irregular surface bonding sites, require a less specialized approach. These systems may not be regular and thus defining a small unit cell and replicating it is not always possible. Additionally, many tools are tied to a specific simulation environment [3] or are operated via a custom language that complicates integration with a broader scientific ecosystem of tools for performing tasks not specific to the domain of molecular simulation, such as statistical analysis and visualization.

In prior work [5], we introduced the preliminary concepts underpinning `mBuild`'s functionality. Since then, `mBuild` has evolved into a Python package designed to simplify the construction of complex, regular and irregular structures and topologies as well as integrate seamlessly with the Python scientific stack and more recently developed Python tools in the area of molecular simulation [6–10]. `mBuild` adopts a hierarchical approach to system construction that relies on equivalence relations to connect chemical building blocks (components). Every component can recursively contain particles and other components to generate arbitrary, hierarchical structures where every particle represents a leaf in the hierarchy. Low-level components, such as an alkyl group or a monomer, can be hand-drawn using software like Avogadro [11] and then connected using an equivalence operator which matches defined attachment sites between two components—the operator forces two sets of points in space to overlap thus translating and rotating components into the desired positions. This approach minimizes and often even eliminates the need for users to explicitly translate or rotate components while constructing initial configurations—users simply specify which components should be connected. Additionally, the hierarchical nature of this approach allows for complex families of chemical structures to be encapsulated in a single

component class which exposes user defined, tunable parameters that adjust the structural properties of the system (e.g. chain length, surface coverage). By providing a more natural avenue to express such structures, where the requirement for mental visualization of spatial arrangements is minimized, `mBuild` provides a stepping stone towards the goals outlined by the Materials Genome Initiative [12], by enabling screening of and optimizations in chemical parameter space of complex, soft-materials.

Here, we provide an overview of the algorithms associated with `mBuild` including several recent improvements, and demonstrate its use as a means for automating screening of soft matter systems. We illustrate the construction of basic components, how they can be connected programmatically into complex chemical systems, and finally showcase this functionality by generating and performing parameter sweeping simulations of an ensemble of monolayers constructed of alkanes and polyethylene glycol (PEG) where, through the functionality of `mBuild`, we trivially vary surface density, patterning and chain length in an automated, programmatic way.

2 Software Concept

While the basic concepts and algorithms underlying `mBuild` were outlined in Ref. [5] additional refinement and development has been undertaken, as reported here, in particular to simplify and increase the generality of the data structure and provide enhancements with regards to connecting individual components via equivalence transforms. The primary building blocks of an `mBuild` hierarchy are `Compounds`; every user-created component inherits from this class. Each `Compound` can contain an arbitrary amount of other `Compounds`, allowing for systems to be flexibly built in a hierarchical manner. The programmatic connection of `Compounds` in three dimensional space is facilitated by an equivalence transform. This concept is formalized and implemented via the `Port` class which defines connection sites and orientation. These are each discussed below.

2.1 Data Structure

The hierarchical data structure of `mBuild` is composed of `Compounds`. `Compounds` maintain an ordered set of children which are other `Compounds`. `Compounds` at the bottom of an `mBuild` hierarchy, i.e., the leafs of the tree, are referred to as `Particles` and can be instantiated as, for example, `lj = mb.Particle(name='lennard-jonesium')`. Note however, that this merely serves to illustrate that this `Compound` is at the bottom of the hierarchy; `Particle` is an alias for `Compound` which can be used to clarify the intended role of an object you are creating.

Every `mBuild` hierarchy also maintains a network of bonds between its `Particles` in the form of a graph as provided by the `NetworkX` package [13]. This graph is maintained by the root (top level component) of the given hierarchy. When two `Compounds` with bonds are added together, their bond graphs are composed.

Additionally, `Compounds` have built-in support for copying and deep copying `Compound` hierarchies, enumerating particles or bonds in the hierarchy, proximity based searches, visualization, I/O operations, and a number of other convenience methods that enable complex topologies to be constructed with little user effort.

2.2 Equivalence Transforms

When connecting components in 3D space, their relative orientations must be specified. In `mBuild`, this is accomplished via an equivalence transform. The equivalence operator described here declares points in a component's local coordinate system to be equivalent to points in another component's coordinate system. Using these point pairs, it is possible to compute a rigid transformation, specifically an affine coordinate transformation conserving scaling and orientation (chirality), that, when applied to one component, will transform its designated points to the other component's respective points. Specifying four or more pairs of non-coplanar points is sufficient to compute an unambiguous transformation matrix in 3D space.

Using a rigid transformation F, one can map a point vector v to its image $F(v)$ in a different coordinate system. This operation can be expressed as a multiplication by a rotation matrix $R \in \mathbb{R}^{3\times3}$ and a translation with vector $t \in \mathbb{R}^{3\times1}$.

$$F(v) = Rv + t \tag{1}$$

R and t can be solved for using the singular value decomposition to get the pseudoinverse given four or more points $P_i(x_i, y_i, z_i)$ and their images $P_i'(x_i', y_i', z_i')$ in the target 3-dimensional coordinate system:

$$\begin{bmatrix} x_1' & x_2' & \dots & x_n' \\ y_1' & y_2' & \dots & y_n' \\ z_1' & z_2' & \dots & z_n' \\ 1 & 1 & \dots & 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \dots & x_n \\ y_1 & y_2 & \dots & y_n \\ z_1 & z_2 & \dots & z_n \\ 1 & 1 & \dots & 1 \end{bmatrix} \tag{2}$$

where the lower elements in the transformation matrix (0 and 1) are of dimensions 1×3 and 1×1 respectively.

In `mBuild`, this equivalence transform is used to force four points of one compound to overlap with four points of another. Achieving this generally, requires that the same arrangement of four non-coplanar points must be added to any compound intended to make use of the equivalence transform.

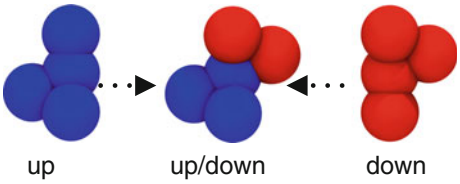


Fig. 1 The spatial arrangement of the particles within a port. Both up and down contain the same arrangement of four non-coplanar particles except that they face opposite directions

2.2.1 Ports

To formalize, simplify, and enable this behavior to function with any compound, `mBuild` provides the `Port` class, which is a simple `Compound` containing four untyped `Particles` in a compact, non-coplanar arrangement (see Fig. 1). Note that for most use cases, it is not desirable to print these untyped, extra `Particles` when outputting the final structure to a file, which is the default behavior of the `Compound.save()` method, but they can be saved if desired, e.g. for visualization purposes.

Instead of having to explicitly define an equivalence relation between four pairs of points, `mBuild` allows for declaring two `Ports`, one in each compound, to be equivalent. When performing an equivalence transform on two `Ports`, one of the `Compounds` that the two `Ports` are a part of is rotated and translated, such that the untyped particles inside their respective ports overlap (see Fig. 2). Since it is common that `Ports` represent bonding sites where molecule fragments need to be attached, `mBuild` allows for defining an *anchor* `Compound` associated with a `Port`. After the affine transformation is applied, `mBuild` will by default create a bond between the two respective anchors, relieving the user from this often tedious task.

Notice that ports have directionality, as well. Consider *Component* C_1 in Fig. 2, representing a methyl group. It is not possible to create an ethane molecule from

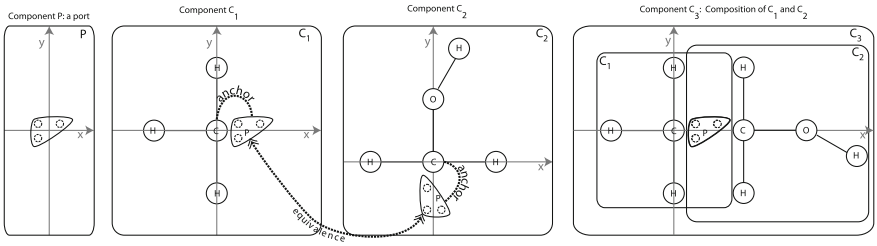


Fig. 2 A `Port` is a compound with two pairs of four `Particles`. Here, one pair of three points is shown to illustrate this 2D example. `Ports` are attached to any other `Compound`, most commonly anchored to a `Particle` where a chemical bond should exist. Compound C1 is a methyl group with a `Port` anchored to the carbon atom. C2 is a methylene bridge already connected to a hydroxyl group. C1 and C2 are then attached using the equivalence relation described in Eq. (2) to create C3, an ethanol molecule. By default, a `Bond` is created between the two anchoring carbons. Adapted with permission from Fig. 2 in Sallai, J. et al. (2013) *Web- and Cloud-based Software Infrastructure for Materials Design*. Procedia Computer Science: Elsevier

two such components, because the equivalence transform would render not just the untyped atoms in the ports, but also the carbon and hydrogen atoms to overlap. While one way of solving this problem would be to have two flavors of each such `Compound` class, one with an "outward pointing" Port, and another one with an "inward pointing" one, `mBuild` takes an alternate approach. The actual implementation of the `Port` class contains not four, but eight untyped atoms: four of them forming an "inward pointing", while the other four comprising an "outward pointing" collection of points. When performing an equivalence transform, `mBuild` computes two affine transformation matrices, and choses the one that avoids the overlap of the compounds' typed atoms. This is achieved by checking which of the two transformations forces the anchor atoms as far away from one another as possible (see Fig. 1 for an illustration of how these quartets of `Particles` are arranged). Figure 2 highlights this procedure via the construction of an ethanol molecule. Additional documentation is included at the development website (http://imodels.github.io/mbuild/) via an interactive IPython notebook [14].

3 Applications

Below, we highlight the basics of assembling low level components into successively more complex structures in `mBuild` and how to programmatically control these workflows to perform automated screening for monolayer systems. All the examples discussed below are also available as tutorials in IPython notebook format where users can seamlessly visualize components as they are constructed from Python code via a widget provided by the `imolecule` package [8]. Static versions of these notebooks are also hosted on our documentation page at http://imodels.github.io/mbuild/. Many additional example systems of varying complexity are provided together with the `mBuild` source code on GitHub.

3.1 Defining and Connecting Basic Components

The simplest way to define a basic component in `mBuild` is to draw the component using software such as Avogadro [11], output it as a `.mol2` or `.pdb` file with defined bonds and then use the `load` function in `mBuild`. Adding a Port to a compound that a user wants to be able to connect to other compounds requires placing the `Port` where a bond could be formed and specifying an anchor particle with which the `Port` is associated. Just as with any other `Compound`, `Ports` can not only be translated but also rotated thus allowing non-linear arrangements to be constructed. This procedure is highlighted in Listing 1; basic components can be stored and reused for future system construction thus minimizing the need for users to place `Ports`, as will be demonstrated as part of the construction of alkane monolayers in the screening application below.

Listing 1 Example code to generate a CH_2 group and attach two ports

```
ch2 = mb.load('ch2.pdb')
mb.translate(ch2, -ch2[0].pos) # Move carbon to origin.

port1 = mb.Port(anchor=ch2[0]) # Anchor the port on the carbon.
mb.translate(port1, [0, 0.07, 0]) # Approx. half a C-C bond length in nm.

port2 = mb.Port(anchor=ch2[0])
mb.translate(port1, [0, -0.07, 0]) # Placed on opposite side of carbon.

ch2.add(port1, label='up')
ch2.add(port2, label='down')
```

Any two `Ports` can be forced to overlap using the equivalence transform. Listing 2 demonstrates how this functionality can be leveraged via the simple yet common use case of creating an alkane polymer chain which will be used for screening—in this example, a CH_2 group with the ports "up" and "down" defined.

Listing 2 Example code for polymerizing CH_2 groups

```
import mbuild as mb
from mbuild.lib.moieties import CH2

polymer = mb.Compound()
last_monomer = CH2()

polymer.add(last_monomer)
for _ in range(10):
    this_monomer = mb.clone(last_monomer)
    mb.equivalence_transform(this_monomer, this_monomer['up'],
        last_monomer['down'])
    polymer.add(this_monomer)
    last_monomer = this_monomer
```

To further simplify the composition of basic components into more complex structures, several classes and functions have been developed to more naturally express many commonly performed tasks. For example, the functionality of the example in Listing 2 is encapsulated within the `Polymer` class which reduces the above for loop to one line for end users. For example, the PEG chains referenced in the following examples, are created with the code in Listing 3.

Listing 3 Using the Polymer class to create PEG chains

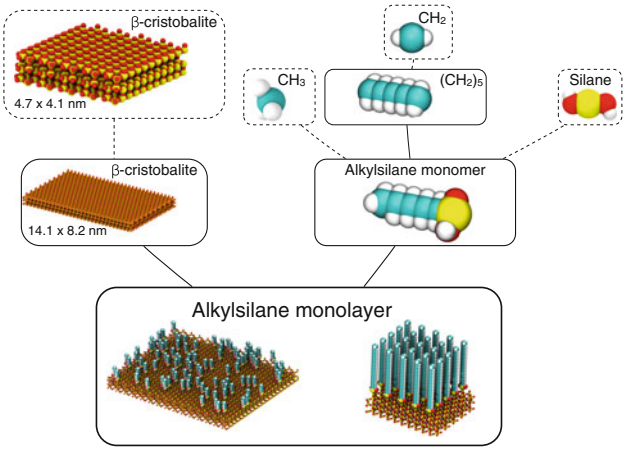
```
import mbuild as mb
from mbuild.lib.moieties import CCO

peg = mb.Polymer(CCO(), n=10)
```

3.2 *Patterning Surfaces*

`mBuild` provides functionality for patterning of surfaces in arbitrary ways. Below, we highlight this feature via the patterning of scientifically relevant 2D and 3D systems.

In the example shown in Fig. 3, the `TiledCompound` class is used to replicate a periodic substrate in the *x*- and *y*-dimensions. This class also internally adjusts periodic bonds. In the final tier of the hierarchy, the patterning functionality, which can be used to create patterns on, for example, substrates or spherical particles, is used to randomly disperse polymer brushes on the substrate. Functionality is provided in `mBuild` for a variety of 2D and 3D patterns including random, grid-like, disks and spherical patterns. Ultimately, a multi-tiered hierarchy of components is assembled, from simple "hand-drawn" monomers, through polymerization and replication of periodic substrates. This functionality is expressed with minimal code via creating a new Python class (shown at the bottom of Fig. 3) to expose the desirable tunable parameters. Here, the number of monomers in the chain, the number of chains on the surface, the pattern on the surface, and the size of the surface can all be trivially modified during screening.

The surface patterning illustrated in Fig. 3 was limited to a two-dimensional surface; however, the underlying functionality in `mBuild` naturally generalizes to three dimensions as well with essentially no changes to the user-level code.




```
pattern = Random2DPattern(100)
chain_length = 5
n_tiles = [2, 3, 1]
```

```
pattern = Grid2DPattern(5, 5)
chain_length = 20
n_tiles = [1, 1, 1]
```

```
monolayer = AlkaneMonolayer(chain_length, pattern, n_tiles)
```

Fig. 3 Hierarchy of compounds used to generate an alkylsilane monolayer on a *β*-cristobalite substrate. *Dashed boxes* indicate base components for which .mol2 or .pdb files exist, e.g. drawn using software such as Avogadro [11]. The code snippet used to generate the structures with all of the tunable parameters exposed is shown at the *bottom* for two different parameter combinations

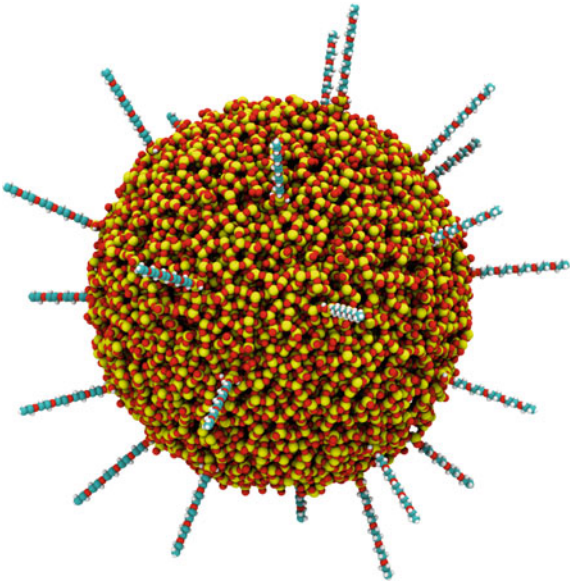


Fig. 4 An 8 nm diameter silica nanoparticle sparsely functionalized with PEG chains bound to the surface with a silane group

Figure 4 and Listing 4 show how this could be used to functionalize a spherical nanoparticle with various polymer chains. The code utilized to attach chemical groups to two-dimensional systems can be reused for three dimensional structures without significant modification or further effort by the end user.

Listing 4 Example code to tether PEG chains to a silica nanoparticle

```
import mbuild as mb
from mbuild.lib.surfaces import SilicaNP
from mbuild.examples import PEGSilane

peg_silane = PEGSilane(peo_units=5) # Length based on number of CCO moieties.
silica_np = SilicaNP()

pattern = mb.SpherePattern(25)
peg_chains, _ = pattern.apply_to_compound(guest=peg_silane, host=silica_np)

tnp = mb.Compound([silica_np, peg_chains])
```

4 Screening Soft Matter Systems: Self-assembled Monolayers

Building upon the prior examples, monolayers are constructed in a programmatic way to demonstrate the use of `mBuild` for screening applications. Monolayers encompass a vast chemical parameter space that can be tuned for applications such

as lubrication [15] and anti-fouling [16], and their behavior and properties often strongly depend on the substrate, binding moiety, chain type, composition of multiple chain types, surface patterning, etc. Sampling more than one or two dimensions of this parameter space using experimental techniques, while technically possible, quickly becomes limited by practical considerations. Molecular dynamics can be used as a screening step to inform subsequent experimental studies and dramatically cut down the relevant search space. Here, using `mBuild` substrate density, chain length, and chain type of monolayer systems are programmatically varied in order to perform a basic screening.

The first step to performing a screening procedure across chemical space involves building the input topologies. Ideally, a user should have seamless access to any variables of interest thus enabling them to adjust these to mimic a statistical distribution. As discussed previously, the hierarchical nature of `mBuild` provides an avenue to expose an arbitrary set of variables to the end user and thus enables users to leverage the scientific Python ecosystem to apply standard optimization techniques and analysis to explore chemical parameter space. As highlighted above, in `mBuild`, the only explicit rotation and translation occurs in the lowest level of the hierarchy when placing ports. Once these simple components have been fitted with ports, they can be stored in the database for future use thus completely eliminating the need for explicit rotation and translation when building many systems; here, we reuse many of the components previously defined in the prior examples. Each higher tier in the hierarchy contains only a few lines of code to express which ports to connect to one another.

Listing 5 shows the `mBuild` code that generates configurations for a simple screening procedure of alkane and PEG monolayers on silica substrates. This code varies the chain length and the number of chains on the surface for both molecules types. It is important to note that the code to generate both monolayer types are nearly identical due to the hierarchical nature of `mBuild`; the `Monolayer` function is generic, as it simply expects a `Compound` with a `Port` defined for attachment. Thus it can readily accept either the `Alkane` or `PEG Compounds` (or mixture thereof) that have previously been define, where each of these `Compounds` accepts an argument to define the length of the desired polymer chain. As such, this example can be trivially extended by creating a different molecule `Compound`, and substituting this in place of either the `Alkane` or `PEG Compound`.

Figure 5 illustrates two of the systems created using this procedure post-equilibration. In this example, the monolayers were patterned in a 2D grid but the patterning of the surface is also tunable if desired, as shown previously. Each monolayer that was created was sampled for 10 ns using GROMACS [17] and the OPLS-aa forcefield [18] with modifications as described by Lorentz et al. [19].

Listing 5 Example code to generate alkane and PEG monolayers differing in both chain length and number of surface grafted chains. Note that most of the code can be reused to create both the PEG and alkane monolayer; the only difference is the chain class that is instantiated

```
import mbuild as mb
from mbuild.examples import Alkane, PEG
from mbuild.lib.atoms import H
from mbuild.lib.surfaces import Betacristobalite

hydrogen = H()
surface = Betacristobalite()

for sqrt_n_chains in range(4, 11): # Amount of chains on surface.
    pattern = mb.Grid2DPattern(sqrt_n_chains, sqrt_n_chains)

    for chain_length in range(6, 22, 3): # Length of chains on surface.
        chain = Alkane(chain_length) # Use alkane chains.
        monolayer = mb.Monolayer(surface, chain, pattern, backfill=hydrogen)
        monolayer.save('{}_{}_alkane.mol2'.format(sqrt_n_chains**2, cl))

        chain = PEG(peo_units=chain_length / 3) # Use PEG chains.
        monolayer = mb.Monolayer(surface, chain, pattern, backfill=hydrogen)
        monolayer.save('{}_{}_peg.mol2'.format(sqrt_n_chains**2, cl))
```

Figure 6 shows the average nematic order parameter, $S2$, of the chains on monolayer [20, 21]. $S2$ measures the orientational ordering of the chains, where for monolayers, values below 0.7 indicate a fluid-like state (i.e., low order) whereas values that approach unity indicate a high degree of crystalline orientational ordering. It has been shown that $S2$ influences the frictional properties of monolayers, where lower values of $S2$ for monolayers tend to be correlated with higher frictional forces when the monolayers are brought together in sliding contact [22]. Thus $S2$ serves as a useful surrogate for rapidly screening monolayers to determine which regimes are likely to produce high/low coefficients of frictions. While additional simulations and sampling are required to draw more robust conclusions, several regimes are readily apparent. A clear transition from disordered, fluid-like monolayer states to ordered states occurs for both systems. This transition occurs at lower surface coverages for alkane chains as compared to PEG chains. That is, PEG systems appear to have a smaller regime of well order states, which can be

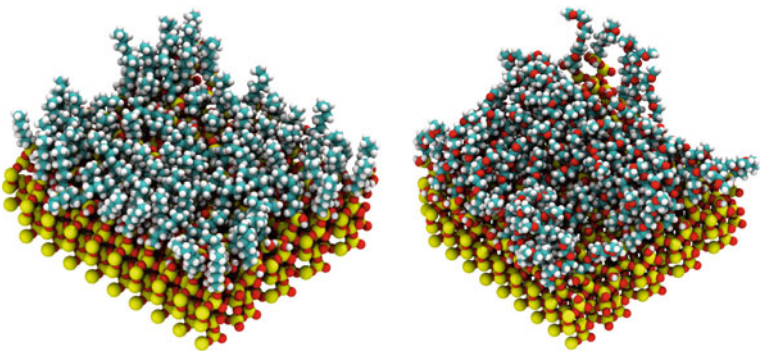


Fig. 5 An alkane system with 81 chains with 7 carbons each (*left*) and a PEG system with 64 chains and 13 carbons/oxygens (*right*). Both shown post-equilibration

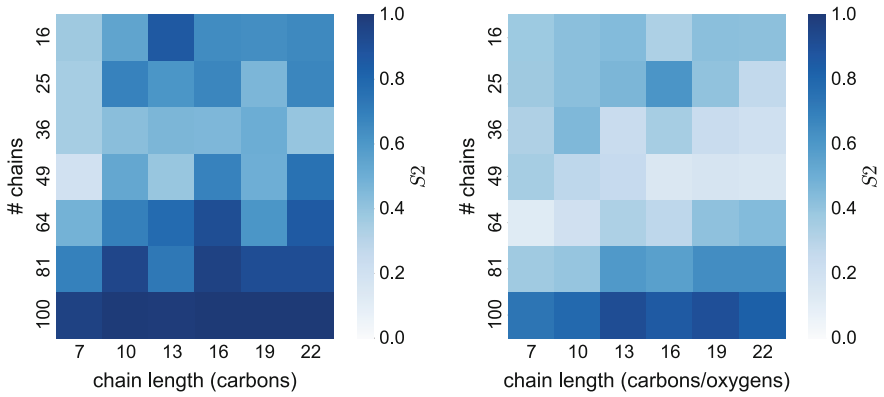


Fig. 6 Average nematic order parameter of every system after 10 ns of sampling. The total process of constructing all 84 systems with `mBuild` takes a few minutes on a modern laptop and the simulations each take approximately 0.5–3 h depending on system size using a GTX980 and 8 CPU cores of an Intel Xeon E5 2600v3

accounted for by the increased flexibility in PEG. In both cases, systems with the highest values of $S2$ tend to occur for higher surface densities and longer chains, and thus one would expect materials in these regimes to demonstrate the most favorable frictional properties. Interestingly, these screening simulations also reveal a second regime for PEG occurring for low surface coverage and short chain length; in this regime moderate values of $S2$ are observed, which, upon visual inspection, appears associated with chains lying flat along the surface. The ability to rapidly screen, evaluate and cross-correlate metrics like the nematic order parameter will accelerate our ability to rationally design soft materials in complex parameter landscapes.

5 Conclusion

`mBuild` provides a programmatic pathway to constructing arbitrary, complex input topologies for molecular simulations. The use of an equivalence operator typically eliminates the need for users to explicitly rotate or translate components while assembling chemical structures. The core data structures of `mBuild` and how the equivalence operator is implemented and used in practice are described and the pathway from basic component creation all the way through constructing several complex example hierarchies illustrated. The format-agnostic nature of `mBuild` allows for flexible interoperability with other tools in the scientific Python and molecular modeling communities, such as packmol [23], polymatic [6], MDTraj [7], imolecule [8], OpenMM [9] and HOOMD-blue [10]. Using monolayers as an

example, the power of this approach is highlighted by performing a small parameter sweeping simulation study, demonstrating clear regimes of highly ordered monolayers which are likely correlated with favorable friction coefficients. This example demonstrates how this approach can be leveraged to more broadly study, design and optimize complex materials. Source code and interactive tutorials in the IPython notebook format, which reinforce the basics of component construction and how to re-use components to assemble more complex systems, are also provided on the `mBuild` website (http://imodels.github.io/mbuild/).

The amount of easily generatable chemical configurations scales dramatically as users contribute components to `mBuild`'s library. As such, we have begun curating a version-controlled library of components such that they can be reused, error-corrected and added to. `mBuild` and its component library are fully open-sourced at https://github.com/imodels/mbuild and user contributions are actively encouraged, which we hope will attract an active user base.

Acknowledgments This material is based upon work supported by the National Science Foundation under Grants No. NSF CBET-1028374 and OCI-1047828.

References

1. Bernstein, F.C., Koetzle, T.F., Williams, G.J., Meyer, E.F., Brice, M.D., Rodgers, J.R., Kennard, O., Shimanouchi, T., Tasumi, M.: The protein data bank: a computer-based archival file for macromolecular structures. Arch. Biochem. Biophys. **185**, 584–591 (1978)
2. Humphrey, W., Dalke, A., Schulten, K.: VMD: visual molecular dynamics. J. Mol. Graph. **14**, 33–38 (1996)
3. Salomon-Ferrer, R., Case, D.A., Walker, R.C.: An overview of the Amber biomolecular simulation package. Wiley Interd. Rev.: Comput. Mol. Sci. **3**, 198–210 (2013)
4. Omnia: High performance, high usability toolkits for predictive biomolecular simulation. http://www.omnia.md
5. Sallai, J., Varga, G., Toth, S., Iacovella, C.T., Klein, C., McCabe, C., Ledeczi, A., Cummings, P.T.: Web- and cloud-based software infrastructure for materials design. Proc. Comput. Sci. **29**, 2034–2044 (2014)
6. Abbott, L.J., Hart, K.E., Colina, C.M.: Polymatic: a generalized simulated polymerization algorithm for amorphous polymers. Theoret. Chem. Acc. **132**, 1–19 (2013)
7. McGibbon, R.T., Beauchamp, K.A., Schwantes, C.R., Wang, L.-P., Hernández, C.X., Harrigan, M.P., Lane, T.J., Swails, J.M., Pande, V.S.: MDTraj: a modern, open library for the analysis of molecular dynamics trajectories. bioRxiv (2014)
8. Fuller, P.: Imolecule: an embeddable webGL molecule viewer. https://github.com/patrickfuller/imolecule
9. Eastman, P., et al.: OpenMM 4: a reusable, extensible, hardware independent library for high performance molecular simulation. J. Chem. Theory Comput. **9**, 461–469 (2013)
10. Anderson, J.A., Lorenz, C.D., Travesset, A.: General purpose molecular dynamics simulations fully implemented on graphics processing units. J. Comput. Phys. **227**, 5342–5359 (2008)
11. Hanwell, M.D., Curtis, D.E., Lonie, D.C., Vandermeersch, T., Zurek, E., Hutchison, G.R.: Avogadro: an advanced semantic chemical editor, visualization, and analysis platform. J. Cheminform. **4**, 17 (2012)
12. http://www.whitehouse.gov/mgi. Materials genome initiative for global competitiveness

13. Aric Hagberg, P.S., Dan Schult NetworkX: High-productivity software for complex networks. https://networkx.github.io/
14. Pérez, F., Granger, B.E.: IPython: a system for interactive scientific computing. Comput. Sci. Eng. **9**, 21–29 (2007)
15. Bhushan, B., Israelachvili, J.N., Landman, U.: Nanotribology: friction, wear and lubrication at the atomic scale. Nature **374**, 607–616 (1995)
16. Brzoska, J.B., Shahidzadeh, N., Rondelez, F.: Evidence of a transition temperature for the optimum deposition of grafted monolayer coatings. Nature **360**, 719–721 (1992)
17. Pronk, S., Páll, S., Schulz, R., Larsson, P., Bjelkmar, P., Apostolov, R., Shirts, M.R., Smith, J. C., Kasson, P.M., Van Der Spoel, D., Hess, B., Lindahl, E.: GROMACS 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. Bioinformatics **29**, 845–854 (2013)
18. Jorgensen, W.L., Maxwell, D.S., Tirado-Rives, J.: Development and testing of the OPLS all-atom force field on conformational energetics and properties of organic liquids. J. Am. Chem. Soc. **118**, 11225–11236 (1996)
19. Lorenz, C., Webb, E., Stevens, M., Chandross, M., Grest, G.: Frictional dynamics of perfluorinated self-assembled monolayers on amorphous SiO_2. Tribol. Lett. **19**, 93–98 (2005)
20. Lagomarsino, M.C., Dogterom, M., Dijkstra, M.: Isotropic nematic transition of long, thin, hard spherocylinders confined in a quasi-two-dimensional planar geometry. J. Phys. Chem. **119**, 719–721 (2003)
21. Wilson, M.R.: Determination of order parameters in realistic atom-based models of liquid crystal systems. J. Mol. Liq. **68**, 23–31 (1996)
22. Black, J.E., Iacovella, C.R., Cummings, P.T., McCabe, C.: Molecular dynamics study of alkylsilane monolayers on realistic amorphous silica surfaces. Langmuir **31**, 3086–3093 (2015)
23. Martnez, L., Andrade, R., Birgin, E.G., Martnez, J.M.: PACKMOL: a package for building initial configurations for molecular dynamics simulations. J. Comput. Chem. **30**, 2157–2164 (2009)