mbuild

Release 0.10.10

Mosdef Team

May 06, 2021

Contents

mBuild: *A hierarchical, component based molecule builder*

With just a few lines of mBuild code, you can assemble reusable components into complex molecular systems for molecular simulations.

- mBuild is designed to minimize or even eliminate the need to explicitly translate and orient components when building systems: you simply tell it to connect two pieces!
- mBuild keeps track of the system's topology so you don't have to worry about manually defining bonds when constructing chemically bonded structures from smaller components.

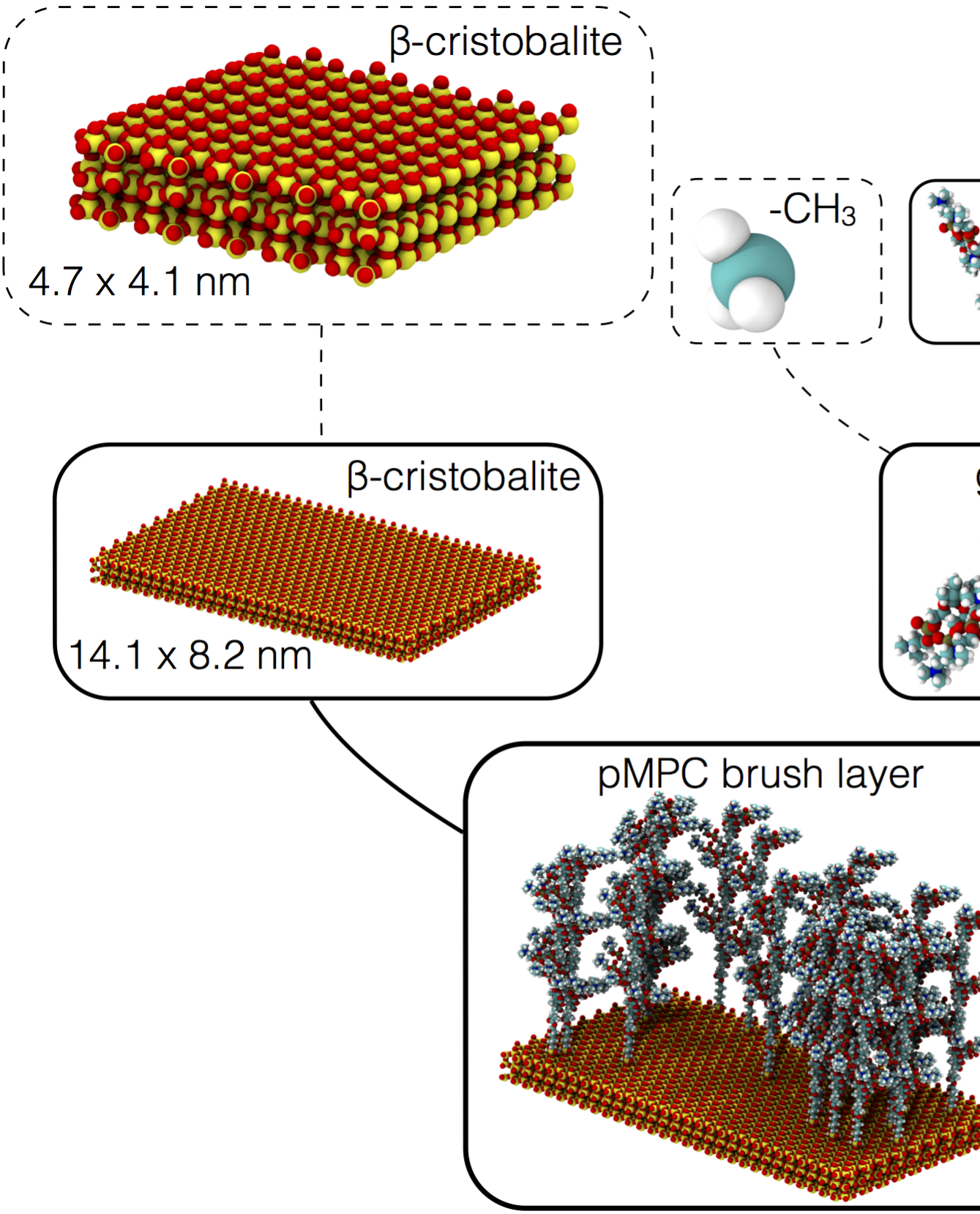
1 Example system

Components in dashed boxes are drawn by hand using, e.g., Avogadro[1] or generated elsewhere. Each component is wrapped as a python class with user defined attachment sites, or ports. That's the hard part! Now mBuild can do the rest. Each component further down the hierarchy is, again, a simple python class that describes which piece should connect to which piece.

Ultimately, complex systems structures can be created with just a line or two of code. Additionally, this approach seamlessly exposes tunable parameters within the hierarchy so you can actually create whole families of structures by adjusting a variable or two:

```
pattern = Random2DPattern(20)  # A random arrangement of 20 pieces on a 2D surface.
brush_layer = BrushLayer(chain_lenth=20, pattern=pattern, tile_x=3, tile_y=2)
```

[1] https://avogadro.cc



[2] Various sub-portions of this library may be independently distributed under different licenses. See those files for their specific terms.

[2] http://opensource.org/licenses/MIT

2 Installation

2.1 Install with conda[3]

```
$ conda install -c conda-forge -c mosdef -c omnia mbuild
```

Alternatively you can add all the required channels to your `.condarc` after which you can simply install without specifying the channels:

```
$ conda config --add channels omnia
$ conda config --add channels mosdef
$ conda config --add channels conda-forge
$ conda install mbuild
```

Note: The order in which channels are added matters: `conda-forge` should be the highest priority as a result of being added last. In your `.condarc` file, it should be listed first.

Note: The MDTraj website[4] makes a nice case for using Python and in particular the Anaconda scientific python distribution[5] to manage your numerical and scientific Python packages.

2.2 Install with pip[6]

```
$ pip install mbuild
```

Note: PACKMOL[7] is not available on pip but can be installed from source or via conda.

2.3 Install an editable version from source

```
$ git clone https://github.com/mosdef-hub/mbuild
$ cd mbuild
$ pip install -e .
```

To make your life easier, we recommend that you use a pre-packaged Python distribution like Continuum's Anaconda[8] in order to get all of the dependencies.

2.4 Supported Python Versions

Python 3.6 and 3.7 are officially supported, including testing during development and packaging. Support for Python 2.7 has been dropped as of August 6, 2019. Other Python versions, such as 3.8 and 3.5 and older, may successfully build and function but no guarantee is made.

[3] https://repo.anaconda.com/miniconda/
[4] http://mdtraj.org/1.9.3/new_to_python.html
[5] https://www.anaconda.com/products/individual
[6] https://pypi.org/project/pip/
[7] http://m3g.iqm.unicamp.br/packmol/
[8] https://www.anaconda.com/products/individual/

2.5 Testing your installation

mBuild uses `py.test` for unit testing. To run them simply type run the following while in the base directory:

```
$ conda install pytest
$ py.test -v
```

3 Tutorials

The following section was generated from `docs/tutorials/tutorial_methane.ipynb`

3.1 Methane: Compounds and bonds

Note: mBuild expects all distance units to be in nanometers.

The primary building block in mBuild is a `Compound`. Anything you construct will inherit from this class. Let's start with some basic imports and initialization:

```
import mbuild as mb

class Methane(mb.Compound):
    def __init__(self):
        super(Methane, self).__init__()
```

Any `Compound` can contain other `Compounds` which can be added using its `add()` method. `Compounds` at the bottom of such a hierarchy are referred to as `Particles`. Note however, that this is purely semantic in mBuild to help clearly designate the bottom of a hierarchy.

```
import mbuild as mb

class Methane(mb.Compound):
    def __init__(self):
        super(Methane, self).__init__()
        carbon = mb.Particle(name='C')
        self.add(carbon, label='C[$]')

        hydrogen = mb.Particle(name='H', pos=[0.11, 0, 0])
        self.add(hydrogen, label='HC[$]')
```

By default a created `Compound/Particle` will be placed at `0, 0, 0` as indicated by its `pos` attribute. The `Particle` objects contained in a `Compound`, the bottoms of the hierarchy, can be referenced via the `particles` method which returns a generator of all `Particle` objects contained below the `Compound` in the hierarchy.

Note: All positions in mBuild are stored in nanometers.

Any part added to a `Compound` can be given an optional, descriptive string label. If the label ends with the characters `[$]`, a list will be created in the labels. Any subsequent parts added to the `Compound` with the same label prefix will be appended to the list. In the example above, we've labeled the hydrogen as `HC[$]`. So this first part, with the label prefix `HC`, is now referenceable via `self['HC'][0]`. The next part added with the label `HC[$]` will be referenceable via `self['HC'][1]`.

Now let's use these styles of referencing to connect the carbon to the hydrogen. Note that for typical use cases, you will almost never have to explicitly define a bond when using mBuild - this is just to show you what's going on under the hood:

```
import mbuild as mb

class Methane(mb.Compound):
    def __init__(self):
        super(Methane, self).__init__()
        carbon = mb.Particle(name='C')
        self.add(carbon, label='C[$]')

        hydrogen = mb.Particle(name='H', pos=[0.11, 0, 0])
        self.add(hydrogen, label='HC[$]')

        self.add_bond((self[0], self['HC'][0]))
```

As you can see, the carbon is placed in the zero index of `self`. The hydrogen could be referenced via `self[1]` but since we gave it a fancy label, it's also referenceable via `self['HC'][0]`.

Alright now that we've got the basics, let's finish building our `Methane` and take a look at it:

```
import mbuild as mb

class Methane(mb.Compound):
    def __init__(self):
        super(Methane, self).__init__()
        carbon = mb.Particle(name='C')
        self.add(carbon, label='C[$]')

        hydrogen = mb.Particle(name='H', pos=[0.1, 0, -0.07])
        self.add(hydrogen, label='HC[$]')

        self.add_bond((self[0], self['HC'][0]))

        self.add(mb.Particle(name='H', pos=[-0.1, 0, -0.07]), label='HC[$]')
        self.add(mb.Particle(name='H', pos=[0, 0.1, 0.07]), label='HC[$]')
        self.add(mb.Particle(name='H', pos=[0, -0.1, 0.07]), label='HC[$]')

        self.add_bond((self[0], self['HC'][1]))
        self.add_bond((self[0], self['HC'][2]))
        self.add_bond((self[0], self['HC'][3]))
```

```
methane = Methane()
methane.visualize()
```

```
# Save to .mol2
methane.save('methane.mol2',overwrite=True)
```

docs/tutorials/tutorial_methane.ipynb ends here.

The following section was generated from docs/tutorials/tutorial_ethane.ipynb

3.2 Ethane: Reading from files, Ports and coordinate transforms

Note: mBuild expects all distance units to be in nanometers.

In this example, we'll cover reading molecular components from files, introduce the concept of `Ports` and start using some coordinate transforms.

First, we need to import the mbuild package:

```
import mbuild as mb
```

As you probably noticed while creating your methane molecule in the last tutorial, manually adding `Particles` and `Bonds` to a `Compound` is a bit cumbersome. The easiest way to create small, reusable components, such as methyls, amines or monomers, is to hand draw them using software like Avogadro[9] and export them as either a .pdb or .mol2 file (the file should contain connectivity information).

Let's start by reading a methyl group from a `.pdb` file:

```
import mbuild as mb

ch3 = mb.load('ch3.pdb')
ch3.visualize()
```

Now let's use our first coordinate transform to center the methyl at its carbon atom:

```
import mbuild as mb

ch3 = mb.load('ch3.pdb')
mb.translate(ch3, -ch3[0].pos)  # Move carbon to origin.
```

Now we have a methyl group loaded up and centered. In order to connect `Compounds` in mBuild, we make use of a special type of `Compound`: the `Port`. A `Port` is a `Compound` with two sets of four "ghost" `Particles`. In addition `Ports` have an `anchor` attribute which typically points to a particle that the `Port` should be associated with. In our methyl group, the `Port` should be anchored to the carbon atom so that we can now form bonds to this carbon:

```
import mbuild as mb

ch3 = mb.load('ch3.pdb')
mb.translate(ch3, -ch3[0].pos)  # Move carbon to origin.

port = mb.Port(anchor=ch3[0])
ch3.add(port, label='up')

# Place the port at approximately half a C-C bond length.
mb.translate(ch3['up'], [0, -0.07, 0])
```

By default, `Ports` are never output from the mBuild structure. However, it can be useful to look at a molecule with the `Ports` to check your work as you go:

```
ch3.visualize(show_ports=True)
```

Now we wrap the methyl group into a python class, so that we can reuse it as a component to build more complex molecules later.

```
import mbuild as mb

class CH3(mb.Compound):
    def __init__(self):
        super(CH3, self).__init__()

        mb.load('ch3.pdb', compound=self)
        mb.translate(self, -self[0].pos)  # Move carbon to origin.

        port = mb.Port(anchor=self[0])
        self.add(port, label='up')
        # Place the port at approximately half a C-C bond length.
        mb.translate(self['up'], [0, -0.07, 0])
```

[9] http://avogadro.cc/wiki/Main_Page

When two `Ports` are connected, they are forced to overlap in space and their parent `Compounds` are rotated and translated by the same amount.

Note: If we tried to connect two of our methyls right now using only one set of four ghost particles, not only would the `Ports` overlap perfectly, but the carbons and hydrogens would also perfectly overlap - the 4 ghost atoms in the `Port` are arranged identically with respect to the other atoms. For example, if a `Port` and its direction is indicated by "<-", forcing the port in <-CH3 to overlap with <-CH3 would just look like <-CH3 (perfectly overlapping atoms).

To solve this problem, every port contains a second set of 4 ghost atoms pointing in the opposite direction. When two `Compounds` are connected, the port that places the anchor atoms the farthest away from each other is chosen automatically to prevent this overlap scenario.

When <->CH3 and <->CH3 are forced to overlap, the CH3<->CH3 is automatically chosen.

Now the fun part: stick 'em together to create an ethane:

```
ethane = mb.Compound()

ethane.add(CH3(), label="methyl_1")
ethane.add(CH3(), label="methyl_2")
mb.force_overlap(move_this=ethane['methyl_1'],
                         from_positions=ethane['methyl_1']['up'],
                         to_positions=ethane['methyl_2']['up'])
```

Above, the `force_overlap()` function takes a `Compound` and then rotates and translates it such that two other `Compounds` overlap. Typically, as in this case, those two other `Compounds` are `Ports` - in our case, `methyl1['up']` and `methyl2['up']`.

```
ethane.visualize()
```

```
ethane.visualize(show_ports=True)
```

Similarly, if we want to make ethane a reusable component, we need to wrap it into a python class.

```
import mbuild as mb

class Ethane(mb.Compound):
    def __init__(self):
        super(Ethane, self).__init__()

        self.add(CH3(), label="methyl_1")
        self.add(CH3(), label="methyl_2")
        mb.force_overlap(move_this=self['methyl_1'],
                         from_positions=self['methyl_1']['up'],
                         to_positions=self['methyl_2']['up'])
```

```
ethane = Ethane()
ethane.visualize()
```

```
# Save to .mol2
ethane.save('ethane.mol2', overwrite=True)
```

docs/tutorials/tutorial_ethane.ipynb ends here.

The following section was generated from `docs/tutorials/tutorial_monolayer.ipynb`

3.3 Monolayer: Complex hierarchies, patterns, tiling and writing to files

Note: mBuild expects all distance units to be in nanometers.

In this example, we'll cover assembling more complex hierarchies of components using patterns, tiling and how to output systems to files. To illustrate these concepts, let's build an alkane monolayer on a crystalline substrate.

First, let's build our monomers and functionalized them with a silane group which we can then attach to the substrate. The `Alkane` example uses the `polymer` tool to combine `CH2` and `CH3` repeat units. You also have the option to cap the front and back of the chain or to leave a `CH2` group with a dangling port. The `Silane` compound is a Si(OH)2 group with two ports facing out from the central Si. Lastly, we combine `alkane` with `silane` and add a label to `AlkylSilane` which points to, `silane['down']`. This allows us to reference it later using `AlkylSilane['down']` rather than `AlkylSilane['silane']['down']`.

Note: In `Compounds` with multiple `Ports`, by convention, we try to label every `Port` successively as 'up', 'down', 'left', 'right', 'front', 'back' which should roughly correspond to their relative orientations. This is a bit tricky to enforce because the system is so flexible so use your best judgement and try to be consistent! The more components we collect in our library with the same labeling conventions, the easier it becomes to build ever more complex structures.

```
import mbuild as mb

from mbuild.lib.recipes import Alkane
from mbuild.lib.moieties import Silane


class AlkylSilane(mb.Compound):
    """A silane functionalized alkane chain with one Port. """
    def __init__(self, chain_length):
        super(AlkylSilane, self).__init__()

        alkane = Alkane(chain_length, cap_end=False)
        self.add(alkane, 'alkane')
        silane = Silane()
        self.add(silane, 'silane')
        mb.force_overlap(self['alkane'], self['alkane']['down'], self['silane']['up'])

        # Hoist silane port to AlkylSilane level.
        self.add(silane['down'], 'down', containment=False)
```

```
AlkylSilane(5).visualize()
```

Now let's create a substrate to which we can later attach our monomers:

```
import mbuild as mb
from mbuild.lib.surfaces import Betacristobalite

surface = Betacristobalite()
tiled_surface = mb.lib.recipes.TiledCompound(surface, n_tiles=(2, 1, 1))
```

Here we've imported a beta-cristobalite surface from our component library. The `TiledCompound` tool allows you replicate any `Compound` in the x-, y- and z-directions by any number of times - 2, 1 and 1 for our case.

Next, let's create our monomer and a hydrogen atom that we'll place on unoccupied surface sites:

```
from mbuild.lib.atoms import H
alkylsilane = AlkylSilane(chain_length=10)
hydrogen = H()
```

Then we need to tell mBuild how to arrange the chains on the surface. This is accomplished with the "pattern" tools. Every pattern is just a collection of points. There are all kinds of patterns like

spherical, 2D, regular, irregular etc. When you use the `apply_pattern` command, you effectively superimpose the pattern onto the host compound, mBuild figures out what the closest ports are to the pattern points and then attaches copies of the guest onto the binding sites identified by the pattern:

```
pattern = mb.Grid2DPattern(8, 8)  # Evenly spaced, 2D grid of points.

# Attach chains to specified binding sites. Other sites get a hydrogen.
chains, hydrogens = pattern.apply_to_compound(host=tiled_surface, guest=alkylsilane,␣
↪backfill=hydrogen)
```

Also note the `backfill` optional argument which allows you to place a different compound on any unused ports. In this case we want to backfill with hydrogen atoms on every port without a chain.

```
monolayer = mb.Compound([tiled_surface, chains, hydrogens])
monolayer.visualize() # Warning: may be slow in IPython notebooks
```

```
# Save as .mol2 file
monolayer.save('monolayer.mol2', overwrite=True)
```

`lib.recipes.monolayer.py` wraps many these functions into a simple, general class for generating the monolayers, as shown below:

```
from mbuild.lib.recipes import Monolayer

monolayer = Monolayer(fractions=[1.0], chains=alkylsilane, backfill=hydrogen,
                      pattern=mb.Grid2DPattern(n=8, m=8),
                      surface=surface, tile_x=2, tile_y=1)
monolayer.visualize()
```

```

```

docs/tutorials/tutorial_monolayer.ipynb ends here.

The following section was generated from `docs/tutorials/tutorial_simple_LJ.ipynb`

3.4 Point Particles: Basic system initialization

Note: mBuild expects all distance units to be in nanometers.

This tutorial focuses on the usage of basic system initialization operations, as applied to simple point particle systems (i.e., generic Lennard-Jones particles rather than specific atoms).

The code below defines several point particles in a cubic arrangement. Note, the color and radius associated with a Particle name can be set and passed to the visualize command. Colors are passed in hex format (see http://www.color-hex.com/color/bfbfbf).

```
import mbuild as mb

class MonoLJ(mb.Compound):
    def __init__(self):
        super(MonoLJ, self).__init__()
        lj_particle1 = mb.Particle(name='LJ', pos=[0, 0, 0])
        self.add(lj_particle1)

        lj_particle2 = mb.Particle(name='LJ', pos=[1, 0, 0])
        self.add(lj_particle2)

        lj_particle3 = mb.Particle(name='LJ', pos=[0, 1, 0])
```

(continues on next page)

(continued from previous page)

```
        self.add(lj_particle3)

        lj_particle4 = mb.Particle(name='LJ', pos=[0, 0, 1])
        self.add(lj_particle4)

        lj_particle5 = mb.Particle(name='LJ', pos=[1, 0, 1])
        self.add(lj_particle5)

        lj_particle6 = mb.Particle(name='LJ', pos=[1, 1, 0])
        self.add(lj_particle6)

        lj_particle7 = mb.Particle(name='LJ', pos=[0, 1, 1])
        self.add(lj_particle7)

        lj_particle8 = mb.Particle(name='LJ', pos=[1, 1, 1])
        self.add(lj_particle8)


monoLJ = MonoLJ()
monoLJ.visualize()
```

While this would work for defining a single molecule or very small system, this would not be efficient for large systems. Instead, the clone and translate operator can be used to facilitate automation. Below, we simply define a single prototype particle (lj_proto), which we then copy and translate about the system.

Note, mBuild provides two different translate operations, "translate" and "translate_to". "translate" moves a particle by adding the vector the original position, whereas "translate_to" move a particle to the specified location in space. Note, "translate_to" maintains the internal spatial relationships of a collection of particles by first shifting the center of mass of the collection of particles to the origin, then translating to the specified location. Since the lj_proto particle in this example starts at the origin, these two commands produce identical behavior.

```
import mbuild as mb

class MonoLJ(mb.Compound):
    def __init__(self):
        super(MonoLJ, self).__init__()
        lj_proto = mb.Particle(name='LJ', pos=[0, 0, 0])

        for i in range(0,2):
            for j in range(0,2):
                for k in range(0,2):
                    lj_particle = mb.clone(lj_proto)
                    pos = [i,j,k]
                    lj_particle.translate(pos)
                    self.add(lj_particle)

monoLJ = MonoLJ()
monoLJ.visualize()
```

To simplify this process, mBuild provides several build-in patterning tools, where for example, Grid3DPattern can be used to perform this same operation. Grid3DPattern generates a set of points, from 0 to 1, which get stored in the variable "pattern". We need only loop over the points in pattern, cloning, translating, and adding to the system. Note, because Grid3DPattern defines points between 0 and 1, they must be scaled based on the desired system size, i.e., pattern.scale(2).

```
import mbuild as mb

class MonoLJ(mb.Compound):
    def __init__(self):
        super(MonoLJ, self).__init__()
        lj_proto = mb.Particle(name='LJ', pos=[0, 0, 0])

        pattern = mb.Grid3DPattern(2, 2, 2)
        pattern.scale(2)

        for pos in pattern:
            lj_particle = mb.clone(lj_proto)
            lj_particle.translate(pos)
            self.add(lj_particle)

monoLJ = MonoLJ()
monoLJ.visualize()
```

Larger systems can therefore be easily generated by toggling the values given to Grid3DPattern. Other patterns can also be generated using the same basic code, such as a 2D grid pattern:

```
import mbuild as mb

class MonoLJ(mb.Compound):
    def __init__(self):
        super(MonoLJ, self).__init__()
        lj_proto = mb.Particle(name='LJ', pos=[0, 0, 0])

        pattern = mb.Grid2DPattern(5, 5)
        pattern.scale(5)

        for pos in pattern:
            lj_particle = mb.clone(lj_proto)
            lj_particle.translate(pos)
            self.add(lj_particle)

monoLJ = MonoLJ()
monoLJ.visualize()
```

Points on a sphere can be generated using SpherePattern. Points on a disk using DisKPattern, etc.

Note to show both simultaneously, we shift the x-coordinate of Particles in the sphere by -1 (i.e., pos[0]-=1.0) and +1 for the disk (i.e, pos[0]+=1.0).

```
import mbuild as mb

class MonoLJ(mb.Compound):
    def __init__(self):
        super(MonoLJ, self).__init__()
        lj_proto = mb.Particle(name='LJ', pos=[0, 0, 0])

        pattern_sphere = mb.SpherePattern(200)
        pattern_sphere.scale(0.5)

        for pos in pattern_sphere:
            lj_particle = mb.clone(lj_proto)
            pos[0]-=1.0
            lj_particle.translate(pos)
            self.add(lj_particle)
```

(continues on next page)

(continued from previous page)

```
        pattern_disk = mb.DiskPattern(200)
        pattern_disk.scale(0.5)
        for pos in pattern_disk:
            lj_particle = mb.clone(lj_proto)
            pos[0]+=1.0
            lj_particle.translate(pos)
            self.add(lj_particle)

monoLJ = MonoLJ()
monoLJ.visualize()
```

We can also take advantage of the hierachical nature of mBuild to accomplish the same task more cleanly. Below we create a component that corresponds to the sphere (class SphereLJ), and one that corresponds to the disk (class DiskLJ), and then instantiate and shift each of these individually in the MonoLJ component.

```
import mbuild as mb

class SphereLJ(mb.Compound):
    def __init__(self):
        super(SphereLJ, self).__init__()
        lj_proto = mb.Particle(name='LJ', pos=[0, 0, 0])

        pattern_sphere = mb.SpherePattern(200)
        pattern_sphere.scale(0.5)

        for pos in pattern_sphere:
            lj_particle = mb.clone(lj_proto)
            lj_particle.translate(pos)
            self.add(lj_particle)

class DiskLJ(mb.Compound):
    def __init__(self):
        super(DiskLJ, self).__init__()
        lj_proto = mb.Particle(name='LJ', pos=[0, 0, 0])

        pattern_disk = mb.DiskPattern(200)
        pattern_disk.scale(0.5)
        for pos in pattern_disk:
            lj_particle = mb.clone(lj_proto)
            lj_particle.translate(pos)
            self.add(lj_particle)


class MonoLJ(mb.Compound):
    def __init__(self):
        super(MonoLJ, self).__init__()

        sphere = SphereLJ();
        pos=[-1, 0, 0]
        sphere.translate(pos)
        self.add(sphere)

        disk = DiskLJ();
        pos=[1, 0, 0]
        disk.translate(pos)
        self.add(disk)
```

(continues on next page)

(continued from previous page)

```
monoLJ = MonoLJ()
monoLJ.visualize()
```

Again, since mBuild is hierarchical, the pattern functions can be used to generate large systems of any arbitary component. For example, we can replicate the SphereLJ component on a regular array.

```
import mbuild as mb

class SphereLJ(mb.Compound):
    def __init__(self):
        super(SphereLJ, self).__init__()
        lj_proto = mb.Particle(name='LJ', pos=[0, 0, 0])

        pattern_sphere = mb.SpherePattern(13)
        pattern_sphere.scale(0.1)

        for pos in pattern_sphere:
            lj_particle = mb.clone(lj_proto)
            lj_particle.translate(pos)
            self.add(lj_particle)
class MonoLJ(mb.Compound):
    def __init__(self):
        super(MonoLJ, self).__init__()
        sphere = SphereLJ();

        pattern = mb.Grid3DPattern(3, 3, 3)
        pattern.scale(2)

        for pos in pattern:
            lj_sphere = mb.clone(sphere)
            lj_sphere.translate_to(pos)
            #shift the particle so the center of mass
            #of the system is at the origin
            lj_sphere.translate([-5,-5,-5])

            self.add(lj_sphere)

monoLJ = MonoLJ()
monoLJ.visualize()
```

Several functions exist for rotating compounds. For example, the spin command allows a compound to be rotated, in place, about a specific axis (i.e., it considers the origin for the rotation to lie at the compound's center of mass).

```
import mbuild as mb
import random
from numpy import pi


class CubeLJ(mb.Compound):
    def __init__(self):
        super(CubeLJ, self).__init__()
        lj_proto = mb.Particle(name='LJ', pos=[0, 0, 0])

        pattern = mb.Grid3DPattern(2, 2, 2)
        pattern.scale(0.2)
```

(continues on next page)

(continued from previous page)

```
        for pos in pattern:
            lj_particle = mb.clone(lj_proto)
            lj_particle.translate(pos)
            self.add(lj_particle)

class MonoLJ(mb.Compound):
    def __init__(self):
        super(MonoLJ, self).__init__()
        cube_proto = CubeLJ();

        pattern = mb.Grid3DPattern(3, 3, 3)
        pattern.scale(2)
        rnd = random.Random()
        rnd.seed(123)

        for pos in pattern:
            lj_cube = mb.clone(cube_proto)
            lj_cube.translate_to(pos)
            #shift the particle so the center of mass
            #of the system is at the origin
            lj_cube.translate([-5,-5,-5])
            lj_cube.spin( rnd.uniform(0, 2 * pi), [1, 0, 0])
            lj_cube.spin(rnd.uniform(0, 2 * pi), [0, 1, 0])
            lj_cube.spin(rnd.uniform(0, 2 * pi), [0, 0, 1])

            self.add(lj_cube)

monoLJ = MonoLJ()
monoLJ.visualize()
```

Configurations can be dumped to file using the save command; this takes advantage of MDTraj and supports a range of file formats (see http://MDTraj.org).

```
#save as xyz file
monoLJ.save('output.xyz')
#save as mol2
monoLJ.save('output.mol2')
```

docs/tutorials/tutorial_simple_LJ.ipynb ends here.

The following section was generated from docs/tutorials/tutorial_polymers.ipynb

3.5 Building a Simple Alkane

The purpose of this tutorial is to demonstrate the construction of an alkane polymer and provide familiarity with many of the underlying functions in mBuild. Note that a robust polymer construction recipe already exists in mBuild, which will also be demonstrated at the end of the tutorial.

Setting up the monomer

The first step is to construct the basic repeat unit for the alkane, i.e., a CH_2 group, similar to the construction of the CH_3 monomer in the prior methane tutorial. Rather than importing the coordinates from a pdb file, as in the previous example, we will instead explicitly define them in the class. Recall that distance units are nm in mBuild.

```
import mbuild as mb

class CH2(mb.Compound):
    def __init__(self):
        super(CH2, self).__init__()
        # Add carbon
        self.add(mb.Particle(name='C', pos=[0,0,0]), label='C[$]')

        # Add hydrogens
        self.add(mb.Particle(name='H', pos=[-0.109, 0, 0.0]), label='HC[$]')
        self.add(mb.Particle(name='H', pos=[0.109, 0, 0.0]), label='HC[$]')

        # Add bonds between the atoms
        self.add_bond((self['C'][0], self['HC'][0]))
        self.add_bond((self['C'][0], self['HC'][1]))

        # Add ports anchored to the carbon
        self.add(mb.Port(anchor=self[0]), label='up')
        self.add(mb.Port(anchor=self[0]), label='down')

        # Move the ports approximately half a C-C bond length away from the carbon
        self['up'].translate([0, -0.154/2, 0])
        self['down'].translate([0, 0.154/2, 0])

monomer = CH2()
monomer.visualize(show_ports=True)
```

This configuration of the monomer is not a particularly realistic conformation. One could use this monomer to construct a polymer and then apply an energy minimization scheme, or, as we will demonstrate here, we can use mBuild's rotation commands to provide a more realistic starting point.

Below, we use the same basic script, but now apply a rotation to the hydrogen atoms. Since the hydrogens start 180° apart and we know they should be ~109.5° apart, each should be rotated half of the difference closer to each other around the y-axis. Note that the rotation angle is given in radians. Similarly, the ports should be rotated around the x-axis by the same amount so that atoms can be added in a realistic orientation.

```
import numpy as np
import mbuild as mb

class CH2(mb.Compound):
    def __init__(self):
        super(CH2, self).__init__()
        # Add carbon
        self.add(mb.Particle(name='C', pos=[0,0,0]), label='C[$]')

        # Add hydrogens
        self.add(mb.Particle(name='H', pos=[-0.109, 0, 0.0]), label='HC[$]')
        self.add(mb.Particle(name='H', pos=[0.109, 0, 0.0]), label='HC[$]')

        # Rotate the hydrogens
        theta = 0.5 * (180 - 109.5) * np.pi / 180
        #mb.rotate(self['HC'][0], theta, around=[0, 1, 0])
        #mb.rotate(self['HC'][1], -theta, around=[0, 1, 0])
        self['HC'][0].rotate( theta, around=[0, 1, 0])
        self['HC'][1].rotate(-theta, around=[0, 1, 0])

        # Add bonds between the atoms
        self.add_bond((self['C'][0], self['HC'][0]))
```

(continues on next page)

(continued from previous page)

```
        self.add_bond((self['C'][0], self['HC'][1]))

        # Add the ports and appropriately rotate them
        self.add(mb.Port(anchor=self[0]), label='up')
        self['up'].translate([0, -0.154/2, 0])
        self['up'].rotate(theta, around=[1, 0, 0])

        self.add(mb.Port(anchor=self[0]), label='down')
        self['down'].translate([0, 0.154/2, 0])
        self['down'].rotate(-theta, around=[1, 0, 0])

monomer = CH2()
monomer.visualize(show_ports=True)
```

Defining the polymerization class

With a basic monomer construct, we can now construct a polymer by connecting the ports together. Here, we first instantiate one instance of the CH2 class as `1ast_monomer`, then use the clone function to make a copy. The `force_overlap()` function is used to connect the `'up'` port from `current_monomer` to the `'down'` port of `last_mononer`.

```
class AlkanePolymer(mb.Compound):
    def __init__(self):
        super(AlkanePolymer, self).__init__()
        last_monomer = CH2()
        self.add(last_monomer)
        for i in range(3):
            current_monomer = CH2()
            mb.force_overlap(move_this=current_monomer,
                             from_positions=current_monomer['up'],
                             to_positions=last_monomer['down'])
            self.add(current_monomer)
            last_monomer = current_monomer

polymer = AlkanePolymer()
polymer.visualize(show_ports=True)
```

Visualization of this structure demonstrates a problem; the polymer curls up on itself. This is a result of the fact that ports not only define the location in space, but also an orientation. This can be trivially fixed, by rotating the down port 180° around the y-axis.

We can also add a variable `chain_length` both to the for loop and `init` that will allow the length of the polymer to be adjusted when the class is instantiated.

```
import numpy as np
import mbuild as mb

class CH2(mb.Compound):
    def __init__(self):
        super(CH2, self).__init__()
         # Add carbons and hydrogens
        self.add(mb.Particle(name='C', pos=[0,0,0]), label='C[$]')
        self.add(mb.Particle(name='H', pos=[-0.109, 0, 0.0]), label='HC[$]')
        self.add(mb.Particle(name='H', pos=[0.109, 0, 0.0]), label='HC[$]')

        # rotate hydrogens
        theta = 0.5 * (180 - 109.5) * np.pi / 180
```

(continues on next page)

(continued from previous page)

```
        self['HC'][0].rotate(theta, around=[0, 1, 0])
        self['HC'][1].rotate(-theta, around=[0, 1, 0])

        # Add bonds between the atoms
        self.add_bond((self['C'][0], self['HC'][0]))
        self.add_bond((self['C'][0], self['HC'][1]))

        # Add ports
        self.add(mb.Port(anchor=self[0]), label='up')
        self['up'].translate([0, -0.154/2, 0])
        self['up'].rotate(theta, around=[1, 0, 0])

        self.add(mb.Port(anchor=self[0]), label='down')
        self['down'].translate([0, 0.154/2, 0])
        self['down'].rotate(np.pi, [0, 1, 0])
        self['down'].rotate(-theta, around=[1, 0, 0])


class AlkanePolymer(mb.Compound):
    def __init__(self, chain_length=1):
        super(AlkanePolymer, self).__init__()
        last_monomer = CH2()
        self.add(last_monomer)
        for i in range (chain_length-1):
            current_monomer = CH2()

            mb.force_overlap(move_this=current_monomer,
                             from_positions=current_monomer['up'],
                             to_positions=last_monomer['down'])
            self.add(current_monomer)
            last_monomer=current_monomer
```

```
polymer = AlkanePolymer(chain_length=10)
polymer.visualize(show_ports=True)
```

Using mBuild's Polymer Class

`mBuild` provides a prebuilt class to perform this basic functionality. Since it is designed to be more general, it takes as an argument not just the chain length, but also the monomer and the port labels (e.g., 'up' and 'down', since these labels are user defined).

```
polymer = mb.lib.recipes.Polymer(CH2(), 10, port_labels=('up', 'down'))
polymer.visualize()
```

Building a System of Alkanes

A system of alkanes can be constructed by simply cloning the polymer constructed above and translating and/or rotating the alkanes in space. `mBuild` provides many routines that can be used to create different patterns, to which the polymers can be shifted.

```
# create the polymer
polymer = mb.lib.recipes.Polymer(CH2(), 10, port_labels=('up', 'down'))

# the pattern we generate puts points in the xy-plane, so we'll rotate the polymer
# so that it is oriented normal to the xy-plane
```

(continues on next page)

(continued from previous page)

```
polymer.rotate(np.pi/2, [1, 0, 0])

# define a compound to hold all the polymers
system = mb.Compound()

# create a pattern of points to fill a disk
# patterns are generated between 0 and 1,
# and thus need to be scaled to provide appropriate spacing
pattern_disk = mb.DiskPattern(50)
pattern_disk.scale(5)

# now clone the polymer and move it to the points in the pattern
for pos in pattern_disk:
    current_polymer = mb.clone(polymer)
    current_polymer.translate(pos)
    system.add(current_polymer)

system.visualize()
```

Other patterns can be used, e.g., the `Grid3DPattern`. We can also use the rotation commands to randomize the orientation.

```
import random

polymer = mb.lib.recipes.Polymer(CH2(), 10, port_labels=('up', 'down'))
system = mb.Compound()
polymer.rotate(np.pi/2, [1, 0, 0])

pattern_disk = mb.Grid3DPattern(5, 5, 5)
pattern_disk.scale(8.0)

for pos in pattern_disk:
    current_polymer = mb.clone(polymer)
    for around in [(1, 0, 0), (0, 1, 0), (0, 0, 1)]:  # rotate around x, y, and z
        current_polymer.rotate(random.uniform(0, np.pi), around)
    current_polymer.translate(pos)
    system.add(current_polymer)

system.visualize()
```

`mBuild` also provides an interface to `PACKMOL`, allowing the creation of a randomized configuration.

```
polymer = mb.lib.recipes.Polymer(CH2(), 5, port_labels=('up', 'down'))
system = mb.fill_box(polymer, n_compounds=100, overlap=1.5, box=[10,10,10])
system.visualize()
```

Variations

Rather than a linear chain, the `Polymer` class we wrote can be easily changed such that small perturbations are given to each port. To avoid accumulation of deviations from the equilibrium angle, we will clone an unperturbed monomer each time (i.e., `monomer_proto`) before applying a random variation.

We also define a variable `delta`, which will control the maximum amount of perturbation. Note that large values of `delta` may result in the chain overlapping itself, as `mBuild` does not currently include routines to exclude such overlaps.

```
import mbuild as mb

import random

class AlkanePolymer(mb.Compound):
    def __init__(self, chain_length=1, delta=0):
        super(AlkanePolymer, self).__init__()
        monomer_proto = CH2()
        last_monomer = CH2()
        last_monomer['down'].rotate(random.uniform(-delta,delta), [1, 0, 0])
        last_monomer['down'].rotate(random.uniform(-delta,delta), [0, 1, 0])
        self.add(last_monomer)
        for i in range(chain_length-1):
            current_monomer = mb.clone(monomer_proto)
            current_monomer['down'].rotate(random.uniform(-delta,delta), [1, 0, 0])
            current_monomer['down'].rotate(random.uniform(-delta,delta), [0, 1, 0])
            mb.force_overlap(move_this=current_monomer,
                             from_positions=current_monomer['up'],
                             to_positions=last_monomer['down'])
            self.add(current_monomer)
            last_monomer=current_monomer

polymer = AlkanePolymer(chain_length = 200, delta=0.4)
polymer.visualize()
```

```
```

docs/tutorials/tutorial_polymers.ipynb ends here.

4 Data Structures

The primary building blocks in an mBuild hierarchy inherit from the `Compound` class. `Compounds` maintain an ordered set of `children` which are other `Compounds`. In addition, an independent, ordered dictionary of `labels` is maintained through which users can reference any other `Compound` in the hierarchy via descriptive strings. Every `Compound` knows its parent `Compound`, one step up in the hierarchy, and knows which `Compounds` reference it in their `labels`. `Ports` are a special type of `Compound` which are used internally to connect different `Compounds` using the equivalence transformations described below.

`Compounds` at the bottom of an mBuild hierarchy, the leafs of the tree, are referred to as `Particles` and can be instantiated as `foo = mb.Particle(name='bar')`. Note however, that this merely serves to illustrate that this `Compound` is at the bottom of the hierarchy; `Particle` is simply an alias for `Compound` which can be used to clarify the intended role of an object you are creating. The method `Compound.particles()` traverses the hierarchy to the bottom and yields those `Compounds`. `Compound.root()` returns the compound at the top of the hierarchy.

4.1 Compound

4.2 Port

4.3 Box

4.4 Lattice

5 Coordinate transformations

The following utility functions provide mechanisms for spatial transformations for mbuild compounds:

6 Recipes

6.1 Monolayer

6.2 Polymer

6.3 Tiled Compound

6.4 Silica Interface

6.5 Packing

6.6 Pattern

7 Citing mBuild

If you use mBuild for your research, please cite our paper[10]:

ACS

> Klein, C.; Sallai, J.; Jones, T. J.; Iacovella, C. R.; McCabe, C.; Cummings, P. T. A Hierarchical, Component Based Approach to Screening Properties of Soft Matter. In *Foundations of Molecular Modeling and Simulation. Molecular Modeling and Simulation (Applications and Perspectives)*; Snurr, R. Q., Adjiman, C. S., Kofke, D. A., Eds.; Springer, Singapore, 2016; pp 79-92.

BibTeX

```
@Inbook{Klein2016mBuild,
    author      = "Klein, Christoph and Sallai, János and Jones, Trevor J. and Iacovella,␣
↪Christopher R. and McCabe, Clare and Cummings, Peter T.",
    editor      = "Snurr, Randall Q and Adjiman, Claire S. and Kofke, David A.",
    title       = "A Hierarchical, Component Based Approach to Screening Properties of␣
↪Soft Matter",
    bookTitle   = "Foundations of Molecular Modeling and Simulation: Select Papers from␣
↪FOMMS 2015",
    year        = "2016",
    publisher   = "Springer Singapore",
```

(continues on next page)

[10] http://doi.org/10.1007%2F978-981-10-1128-3_5

(continued from previous page)

```
    address     = "Singapore",
    pages       = "79--92",
    isbn        = "978-981-10-1128-3",
    doi         = "10.1007/978-981-10-1128-3_5",
    url         = "https://doi.org/10.1007/978-981-10-1128-3_5"
}
```

Download as BibTeX or RIS