HOOMD-blue version 3.0 A Modern, Extensible, Flexible, Object-Oriented API for Molecular Simulations

Brandon L. Butler[‡*], Vyas Ramasubramani[‡], Joshua A. Anderson[‡], Sharon C. Glotzer[‡§¶‖]

https://youtu.be/fIFPYZsOVqI

Abstract—HOOMD-blue is a library for running molecular dynamics and hard particle Monte Carlo simulations that uses pybind11 to provide a Python interface to fast C++ internals. The package is designed to scale from a single CPU core to thousands of NVIDIA or AMD GPUs. In developing HOOMD-blue version 3.0, we significantly improve the application protocol interface (API) by making it more flexible, extensible, and Pythonic. We have also striven to provide simpler and more performant entry points to the internal C++ classes and data structures. With these updates, we show how HOOMD-blue users will be able to write completely custom Python classes which integrate directly into the simulation run loop and analyze previously inaccessible data. Throughout this paper, we focus on how these goals have been achieved and explain design decisions through examples of the newly developed API.

Index Terms—molecular dynamics, molecular simulations, Monte Carlo simulations, object-oriented

Introduction

Molecular simulation has been an important technique for studying the equilibrium properties of molecular systems since the 1950s. The two most common methods for this purpose are molecular dynamics and Monte Carlo simulations [MRR+], [AW]. Molecular dynamics (MD) is the application of Newton's laws of motion to molecular system, while Monte Carlo (MC) methods employ a Markov chain to sample from equilibrium configurations. Since their inception these tools have been used to study numerous systems, examples include colloids [DEG], metallic glasses [FIE], and proteins [DZK+], among others.

Today many software packages exist for these purposes. LAMMPS [Pli], GROMACS [BvdSvD], [AMS+], OpenMM [ESC+], ESPResSo [WWS+], [GTK+] and Amber [SCW], [CCD+] are a few examples of popular MD packages, while Cassandra [SMM+] and MCCCS Towhee [Mar] provide MC simulation capabilities. Implementations on high performance GPUs [SMAG], parallel architectures [NBB+], and the greater accessibility of computational power have tremendously improved the length [BCR+] and time [SDS+] scales of simulations from those conducted in the mid 1900s. The flexibility and generality of such tools has dramatically increased the usage of molecular simulations, which has in turn led to demands for even more customizable software packages that can be tailored to very specific simulation requirements. Different tools have taken different approaches to enabling this, such as the text-file scripting in LAMMPS, the command line interface provided by GROMACS, and the Python, C++, C, and Fortran bindings of OpenMM. Recently, programs that have used other interfaces have also added Python bindings such as LAMMPS and GROMACS.

In the development of these tools, the requirements for the software to enable good science became more obvious. Having computational research that is Transferable, Reproducible, Usable (by others), and Extensible (TRUE) [TGM+] is necessary for fully realizing the potential of computational molecular science. HOOMD-blue is part of the MoSDeF project which seeks to bring these traits to the wider computational molecular science community through packages like mbuild [KSJ+] and foyer [KST+] which are Python packages that generalize generating initial particle configurations and force fields respectively across a variety of simulation back ends [CG], [TGM+]. This effort in increased TRUEness is one of many motivating factors for HOOMD-blue version 3.0.

HOOMD-blue [ALT], [GNA+], [AGG], an MD and MC simulations engine with a C++ back end, provides to use a Python API facilitated through pybind11 [JRM]. The package is open-source under the 3-clause BSD license, and the code is hosted on GitHub (https://github.com/glotzerlab/hoomd-blue). HOOMD-blue was initially released in 2008 as the first fully GPU-enabled MD simulation engine using NVIDIA GPUs through CUDA. Since its initial release, HOOMD-blue has remained under active development, adding numerous features over the years that have increased its range of applicability, including adding support for domain decomposition (dividing the simulation box among MPI ranks) in 2014 and recent developments that enable support for AMD in addition to NVIDIA GPUs.

Despite its great flexibility, the package's API still has certain key limitations. In particular, since its inception HOOMD-blue has been designed around some maintenance of global state. The original releases of HOOMD-blue provided Python scripting capabilities based on an imperative programming model, but it required that these scripts be run through HOOMD-blue's mod-

* Corresponding author: butlerbr@umich.edu
‡ University of Michigan, Department of Chemical Engineering
§ University of Michigan, Department of Material Science and Engineering
¶ University of Michigan, Department of Physics
‖ University of Michigan, Biointerfaces Institute

ified interpreter that was responsible for managing this global state. Version 2.0 relaxed this restriction, allowing the use of HOOMD-blue within ordinary Python scripts and introducing the `SimulationContext` object to encapsulate the global state to some degree, thereby allowing multiple largely independent simulations to coexist in a single script. However, this object remained largely opaque to the user, in many ways still behaving like a pseudo-global state, and version 2.0 otherwise made minimal modifications to the HOOMD-blue Python API, which was largely inspired by and reminiscent of the structure of other simulation software, particularly LAMMPS.

In this paper, we describe the upcoming 3.0 release of HOOMD-blue, which is a complete redesign of the API from the ground up to present a more transparent and Pythonic interface for users. Version 3.0 aspires to match the intuitive APIs provided by other Python packages like SciPy [VGO+], NumPy [vdWCV], scikit-learn [PVG+], and matplotlib [Hun], while simultaneously adding seamless interfaces by which such packages may be integrated into simulation scripts using HOOMD-blue. Global state has been completely removed, instead replaced by a highly object-oriented model that gives users explicit and complete control over all aspects of simulation configuration. Where possible, the new version also provides performant, Pythonic interfaces to data stored by the C++ back end. Over the next few sections, we will use examples of HOOMD-blue's version 3.0 API (which is still in development at the time of writing) to highlight the improved extensibility, flexibility, and ease of use of the new HOOMD-blue API.

General API Design

Rather than beginning with abstract descriptions, we will introduce the new API by example. The script below illustrates a standard MD simulation of a Lennard-Jones fluid using the version 3.0 API. Each of the elements of this script is introduced throughout the rest of this section. We also show a rendering of the particle configuration in Figure (1).

```
import hoomd
import hoomd.md
import numpy as np

device = hoomd.device.Auto()
sim = hoomd.Simulation(device)

# Place particles on simple cubic lattice.
N_per_side = 14
N = N_per_side ** 3
L = 20
xs = np.linspace(0, 0.9, N_per_side)
x, y, z = np.meshgrid(xs, xs, xs)
coords = np.array(
    (x.ravel(), y.ravel(), z.ravel())).T

# One way to define an initial system state is
# by defining a snapshot and using it to
# initialize the system state.
snap = hoomd.Snapshot()
snap.particles.N = N
snap.configuration.box = hoomd.Box.cube(L)
snap.particles.position[:] = (coords - 0.5) * L
snap.particles.types = ['A']

sim.create_state_from_snapshot(snap)

# Create integrator and forces
integrator = hoomd.md.Integrator(dt=0.005)
langevin = hoomd.md.methods.Langevin(
    hoomd.filter.All(), kT=1., seed=42)
```

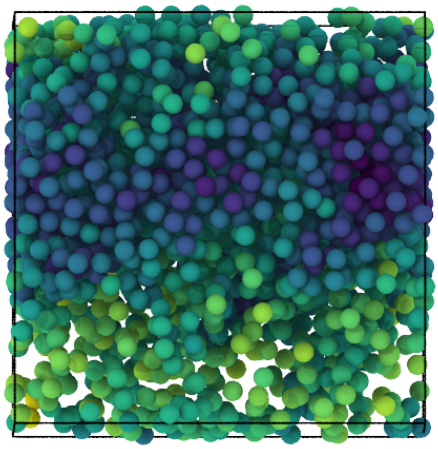

Fig. 1: *A rendering of the Lennard-Jones fluid simulation script output. Particles are colored by the Lennard-Jones potential energy that is logged using the HOOMD-blue `Logger` and `GSD` class objects. Figure is rendered in OVITO [Stu] using the Tachyon [Sto] renderer.*

```
integrator.methods.append(langevin)

nlist = hoomd.md.nlist.Cell()
lj = hoomd.md.pair.LJ(nlist, r_cut=2.5)
lj.params[('A', 'A')] = dict(
    sigma=1., epsilon=1.)
integrator.forces.append(lj)

# Set up output
gsd = hoomd.output.GSD('trajectory.gsd', trigger=100)
log = hoomd.logging.Logger()
log += lj
gsd.log = log

sim.operations.integrator = integrator
sim.operations.analyzers.append(gsd)
sim.run(100000)
```

Simulation, Device, State, Operations

Each simulation in HOOMD-blue is now controlled through three main objects which are joined together by the `Simulation` class: the `Device`, `State`, and `Operations` classes. Figure (2) shows this relationship with some core attributes/methods for each class. Each `Simulation` object holds the requisite information to run a full molecular dynamics or Monte Carlo simulation, thereby circumventing any need for global state information. The `Device` class denotes whether a simulation should be run on CPUs or GPUs and the number of cores/GPUs it should run on. In addition, the device manages custom memory tracebacks, profiler configurations, and the MPI communicator among other things.

The `State` class stores the system data (e.g. particle positions, orientations, velocities, the system box). As shown in our example, the state can be initialized from a snapshot, after which the data can be accessed and modified in two ways. One option is for users to operate on a new `Snapshot` object, which exposes NumPy arrays that store a copy of the system data. To construct a snapshot, all system data distributed across MPI ranks must be gathered and combined by the root rank. Setting the state using the snapshot API requires assigning a modified snapshot to the system state (i.e. all system data is reset upon setting). The advantages to this approach come from the ease of use of working with a

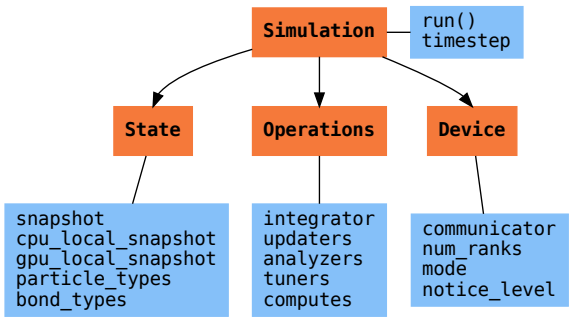

Fig. 2: *Diagram of core objects with some attributes and methods. Classes are in bold and orange; attributes and methods are blue. Figure is made using Graphviz [EGK+], [GKNV].*

single object containing the complete description of the state. The following snippet showcases how this approach can be used to set the z position of all particles to zero.

```
snap = sim.state.snapshot
# snapshot only stores data on rank 0
if snap.exists:
    # set all z positions to 0
    snap.particles.position[:, 2] = 0
sim.state.snapshot = snap
```

The other API for accessing `State` data is via a zero-copy, rank-local access to the state's data on either the GPU or CPU. On the CPU, we expose the buffers as `numpy.ndarray`-like objects through provided hooks such as `__array_ufunc__` and `__array_interface__`. Similarly, on the GPU we mock much of the CuPy [zot] `ndarray` class if it is installed; however, at present the CuPy package provides fewer hooks, so our integration is more limited. Whether or not CuPy is installed, we use version 2 of the `__cuda_array_interface__` protocol for GPU access (compatibility with our GPU buffers in Python therefore depends on the support of version 2 of this protocol). This provides support for libraries such as Numba's [LPS] GPU just-in-time compiler and PyTorch [PGM+]. We chose to mock NumPy-like interfaces rather than expose `ndarray` objects directly out of consideration for memory safety. To ensure data integrity, we restrict the data to only be accessible within a specific context manager. This approach is much faster than using the snapshot API because it uses HOOMD-blue's data buffers directly, but the nature of providing zero-copy access requires that users deal directly with the domain decomposition since only data for a MPI rank's local simulation box is stored by a given rank. The example below modifies the previous example to instead use the zero-copy API.

```
with sim.state.cpu_local_snapshot as data:
    data.particles.position[:, 2] = 0

# assumes CuPy is installed
with sim.state.gpu_local_snapshot as data:
    data.particles.position[:, 2] = 0
```

The last of the three classes, `Operations`, holds the different *operations* that will act on the simulation state. Broadly, these consist of 3 categories: updaters, which modify simulation state; analyzers, which observe system state; and tuners, which tune the hyperparameters of other operations for performance. Although updaters and analyzers existed in version 2.x (tuners are a version 3.0 split from updaters), these *operations* have undergone a significant API overhaul for version 3.0 to support one of the more far-reaching changes to HOOMD-blue: the deferred initialization model.

Operations in HOOMD-blue are generally implemented as two classes, a user-facing Python object and an internal C++ object which we denote as the *action* of the operation. On creation, these C++ objects typically require a `Device` and a C++ `State` in order to, for instance, initialize appropriately sized arrays. Unfortunately this requirement restricts the order in which objects may be created since devices and states must exist first. This restriction could create potential confusion for users who forget this ordering and would also limit the composability of modular simulation components by preventing, for instance, the creation of a simple force field without the prior existence of a `Device` and a `State`. To circumvent these difficulties, the new API has moved to a deferred initialization model in which C++ objects are not created until the corresponding Python objects are *attached* to a `Simulation`, a model we discuss in greater detail below.

Deferred C++ Initialization

The core logic for the deferred initialization model is implemented in the `_Operation` class, which is the base class for all operations in Python. This class contains the machinery for attaching/detaching operations to/from their C++ counterparts, and it defines the user interface for setting and modifying operation-specific parameters while guaranteeing that such parameters are synchronized with attached C++ objects as appropriate. Rather than handling these concerns directly, the `_Operation` class manages parameters using specially defined classes that handle the synchronization of attributes between Python and C++: the `ParameterDict` and `TypeParameterDict` classes. In addition to providing transparent `dict`-like APIs for the automatically synchronized setting of parameters, these classes also provide strict validation of input types, ensuring that user inputs are validated regardless of whether or not operations are attached to a simulation.

Each class supports validation of their keys, and they can be used to define the structure and validation of arbitrarily nested dictionaries, lists, and tuples. Likewise, both support default values, but to a varying degree due to their differing purposes. `ParameterDict` acts as a dictionary with additional validation logic. However, the `TypeParameterDict` represents a dictionary in which each entry is validated by the entire defined schema. This distinction occurs often in simulation contexts as simulations with multiple types of particles, bonds, angles, etc. must specify certain parameters for each type. In practice this distinction means that the `TypeParameterDict` class supports default specification with arbitrary nesting, while the `ParameterDict` has defaults but these are equivalent to object attribute defaults. An example `TypeParameterDict` initialization and use of both classes can be seen below.

```
# Specification of Sphere's shape TypeParameterDict
TypeParameterDict(
    diameter=float,
    ignore_statistics=False,
    orientable=False,
    len_keys=1)

from hoomd.hpmc.integrate import Sphere
```

```
sphere = Sphere(seed=42)
# Set nselect parameter using ParameterDict
sphere.nselect = 2
# Set shape for type 'A' using TypeParameterDict
sphere.shape['A'] = {'diameter': 1.}
# Set shape for types 'B', 'C', and 'D'
sphere.shape[['B', 'C', 'D']] = {'diameter': 0.5}
```

The specification defined above sets defaults for `ignore_statistics` and `orientable` (the purpose of these is outside the scope of the paper), but requires the setting of the `diameter` for each type.

To store lists of operations that must be attached to a simulation, the analogous `SyncedList` class transparently handles attaching of operations.

```
import hoomd

ops = hoomd.Operations()
gsd = hoomd.output.GSD('example.gsd')
# Append to the SyncedList ops.analyzers
ops.analyzers.append(gsd)
```

These classes also have the ancillary benefit of improving error messaging and handling. An example error message for trying to set `sigma` for *A*-*A* interactions in the Lennard-Jones pair potential to a string (i.e. `lj.params[('A', 'A')] = {'sigma': 'foo', 'epsilon': 1.}` would provide the error message,

> TypeConversionError: For types [('A', 'A')], error In key sigma: Value foo of type <class 'str'> cannot be converted using OnlyType(float). Raised error: value foo not convertible into type <class 'float'>.

Previously, the equivalent error would be "TypeError: must be real number, not str", the error would not be raised until running the simulation, and the line setting sigma would not be in the stack trace given.

Logging and Accessing Data

Logging simulation data for analysis is a critical feature of molecular simulation software packages. Up to now, HOOMD-blue has supported logging through an analyzer interface that simply accepted a list of quantities to log, where the set of valid quantities was based on what objects had been created at any point and stored to the global state. The creation of the base `_Operation` class has allowed us to simultaneously simplify and increase the flexibility of our logging infrastructure. The `Loggable` metaclass of `_Operation` allows all subclasses to expose their loggable quantities by marking Python properties or methods to query.

The actual task of logging data is accomplished by the `Logger` class, which provides an interface for logging most HOOMD-blue objects and custom user quantities. In the example script from the General API Design section above, we show that the `Logger` can add an operation's loggable quantities using the += operator. The utility of this class lies in its intermediate representation of the data. Using the HOOMD-blue namespace as the basis for distinguishing between quantities, the `Logger` maps logged quantities into a nested dictionary. For example, logging the Lennard-Jones pair potentials total energy would produce this dictionary by a `Logger` object `{'md': {'pair': {'LJ': {'energy': (-1.4, 'scalar')}}}}` where `'scalar'` is a flag to make processing the logged output easier. In real use cases, the dictionary would likely be filled with many other quantities.

Version 3.0 of HOOMD-blue uses properties extensively to expose object data such as the total potential energy of all pair potentials, the trial move acceptance rate in MC integrators, and thermodynamic variables like temperature or pressure, all of which can be used directly or stored through the logging interface. To support storing these properties, the logging is quite general and supports scalars, strings, arrays, and even generic Python objects. By separating the data collection from the writing to files, and by providing such a flexible intermediate representation, HOOMD-blue can now support a range of back ends for logging; moreover, it offers users the flexibility to define their own. For instance, while logging data to text files or standard out is supported out of the box, other back ends like MongoDB, Pandas [McK], and Python pickles can now be implemented on top of the existing logging infrastructure. Consistent with the new approach to logging, HOOMD-blue version 3.0 makes simulation output an opt-in feature even for common outputs like performance and thermodynamic quantities. In addition to this improved flexibility in storage possibilities, for HOOMD-blue version 3.0 we have exposed more of an object's data than had previously been available through adding new properties to objects. For example, pair potentials now expose *per-particle* potential energies at any given time (this data is used to color Figure (1)).

In conjunction with the deferred initialization model, the new logging infrastructure also allows us to more easily export an object's state (not to be confused with the simulation state). Due to the switch to deferred initialization, all operation state information is now stored directly in Python, so we have made object state a loggable quantity. All operations also provide a `from_state` factory method that can reconstruct the object from the state, dramatically increasing the restartability of simulations since the state of each object can be saved at the end of a given run and read at the start of the next.

```
from hoomd.hpmc.integrate import Sphere

sphere = Sphere.from_state('example.gsd', frame=-1)
```

This code block would create a `Sphere` object with the parameters stored from the last frame of the gsd file `example.gsd`.

User Customization

A major improvement in HOOMD-blue version 3 is the ease with which users can customize their simulations in previously impossible ways. The changes that enable this improvement generally come in two flavors, the generalization of existing concepts in HOOMD-blue and the introduction of a completely new `Action` class that enables the user to inject arbitrary actions into the simulation loop. In this section, we first discuss how concepts like periods and groups have been generalized from previous iterations of HOOMD-blue and then show how users can inject completely novel routines to actually modify the behavior of simulations.

Triggers

In HOOMD-blue version 2.x, everything that was not run on every timestep had a period and phase associated with it. The timesteps the operation was run on could then be determined by the expression, `timestep % period - phase == 0`. In our refactoring and development, we recognized that this concept could be made much more general and consequently more flexible. Objects do not have to be run on a periodic timescale; they just need some indication of when to run. In other words, the operations needed to be *triggered*. The `Trigger` class encapsulates this

concept, providing a uniform way of specifying when an object should run without limiting options. `Trigger` objects return a Boolean value when called with a timestep (i.e. they are functors). Each operation that requires triggering is now associated with a corresponding `Trigger` instance which informs the simulation when the operation should run. The previous behavior is now available through the `Periodic` class in the `hoomd.trigger` module. However, this approach enables much more sophisticated logic through composition of multiple triggers such as `Before` and `After` which return `True` before or after a given timestep with the `And`, `Or`, and `Not` subclasses that function as logical operators on the return value of the composed `Triggers`.

In addition to the flexibility the `Trigger` class provides by abstracting out the concept of triggering an operation, we use pybind11 to easily allow subclasses of the `Trigger` class in Python. This allows users to create their own triggers in pure Python that will execute in HOOMD-blue's C++ back end. An example of such a subclass that reimplements the functionality of HOOMD-blue version 2.x can be seen below.

```
from hoomd.trigger import Trigger

class CustomTrigger(Trigger):
    def __init__(self, period, phase=0):
        super().__init__()
        self.period = period
        self.phase = phase

    def __call__(self, timestep):
        v = timestep % self.period - self.phase == 0
        return v
```

User-defined subclasses of `Trigger` are not restricted to simple algorithms or even stateless ones; they can implement arbitrarily complex Python code as demonstrated in the Large Examples section's first code snippet.

Variants

`Variant` objects are used in HOOMD-blue to specify quantities like temperature, pressure, and box size which can vary over time. Similar to `Trigger`, we generalized our ability to linearly interpolate values across timesteps (`hoomd.variant.linear_interp` in HOOMD-blue version 2.x) to a base class `Variant` which generalizes the concept of functions in the semi-infinite domain of timesteps $t \in \mathbb{Z}_0^+$. This allows sinusoidal cycling, non-uniform ramps, and other behaviors. Like `Trigger`, `Variant` can be a direct subclass of the C++ class. An example of a sinusoidal cycling variant is shown below.

```
from math import sin
from hoomd.variant import Variant

class SinVariant(Variant):
    def __init__(self, frequency, amplitude,
                 phase=0, center=0):
        super().__init__()
        self.frequency = frequency
        self.amplitude = amplitude
        self.phase = phase
        self.center = center

    def __call__(self, timestep):
        tmp = self.frequency * timestep
        tmp = sin(tmp + self.phase)
        return self.amplitude * tmp + self.center

    def _min(self):
        return self.center - self.amplitude

    def _max(self):
        return self.center + self.amplitude
```

ParticleFilters

Unlike `Trigger` or `Variant`, `ParticleFilter` is not a generalization of an existing concept but the splitting of one class into two. However, this split is also targeted at increasing flexibility and extensibility. In HOOMD-blue version 2.x, the `ParticleGroup` class and subclasses served to provide a subset of particles within a simulation for file output, application of thermodynamic integrators, and other purposes. The class hosted both the logic for storing the subset of particles and filtering them out from the system. After the refactoring, `ParticleGroup` is only responsible for the logic to store and perform some basic operations on a set of particle tags (a means of identifying individual particles), while the new class `ParticleFilter` implements the selection logic. This choice makes `ParticleFilter` objects lightweight and provides a means of implementing a `State` instance-specific cache of `ParticleGroup` objects. The latter ensures that we do not create multiples of the same `ParticleGroup` which can occupy large amounts of memory. The caching also allows the creation of many of the same `ParticleFilter` object without needing to worry about memory constraints.

`ParticleFilter` can be subclassed (like `Trigger` and `Variant`), but only through the `CustomParticleFilter` class. This is necessary to prevent some internal details from leaking to the user. An example of a `CustomParticleFilter` that selects only particles with positive charge is given below.

```
from hoomd.filter import CustomParticleFilter

class PositiveCharge(CustomParticleFilter):
    def __init__(self, state):
        super().__init__(state)

    def __hash__(self):
        return hash(self.__class__.__name__)

    def __eq__(self, other):
        return type(self) == type(other)

    def find_tags(self, state):
        with state.cpu_local_snapshot as data:
            mask = data.particles.charge > 0
            return data.particles.tag[mask]
```

Custom Actions

In HOOMD-blue, we distinguish between the objects that perform an action on the simulation state (called *Actions*) and their containing objects that deal with setting state and the user interface (called *Operations*). Through composition, HOOMD-blue offers the ability to create custom actions in Python and wrap them in our `_CustomOperation` subclasses (divided on the type of action performed) allowing the execution of the action in the `Simulation` run loop. The feature makes user created actions behave indistinguishably from native C++ actions. Through custom actions, users can modify state, tune hyperparameters for performance, or observe parts of the simulation. In addition, we are adding a signal for Actions to send that would stop a `Simulation.run` call. This would allow actions to stop the simulation when they complete, which could be useful for tasks like tuning MC trial move sizes. With respect to performance,

with zero copy access to the data on the CPU or GPU, custom actions can also achieve high performance using standard Python libraries like NumPy, SciPy, Numba, CuPy and others. Below we show an example of an `Action` that switches particles of type `initial_type` to type `final_type` with a specified `rate` each time it is run. This action could be refined to implement a reactive MC move reminiscent of [GSJ] or to have a variable switch rate. These exercises are left to the reader.

```
import hoomd
from hoomd.filter import (
    Intersection, All, Type)
from hoomd.custom import Action

class SwapType(Action):
    def __init__(self, initial_type,
                 final_type, rate, filter=All()):
        self.final_type = final_type
        self.rate = rate
        self.filter = Intersection(
            [Type(initial_type), filter])

    def act(self, timestep):
        state = self._state
        final_type_id = state.particle_types.index(
            self.final_type)
        tags = self.filter(state)
        with state.cpu_local_snapshot as snap:
            tags = np.intersect1d(
                tags, snap.particles.tag, True)
            part = data.particles
            filtered_index = part.rtags[tags]
            N_swaps = int(len(tags) * self.rate)
            mask = np.random.choice(filtered_index,
                                    N_swaps,
                                    replace=False)
            part.typeid[mask] = final_type_id
```

Conclusion

With modern simulation analysis packages such as freud [RDH+], MDTraj [MBH+], and MDAnalysis [GLB+], [MDWB], initialization tools such as mbuild and foyer, and visualization packages like OVITO and plato [SD] using Python APIs, HOOMD-blue, built from the ground up with Python in mind, fits in seamlessly. Version 3.0 improves upon this and presents a Pythonic API that encourages customization. Through enabling Python subclassing of C++ classes, introducing custom actions, and exposing data in zero-copy arrays/buffers, we allow HOOMD-blue users to utilize the full potential of Python and the scientific Python community.

Acknowledgements

This research was supported by the National Science Foundation, Division of Materials Research Award # DMR 1808342 (HOOMD-blue algorithm and performance development) and by the National Science Foundation, Office of Advanced Cyberinfrastructure Award # OAC 1835612 (Pythonic architecture for MoSDeF). Hardware provided by NVIDIA Corp. is gratefully acknowledged. This research was supported in part through computational resources and services supported by Advanced Research Computing at the University of Michigan, Ann Arbor.

Appendix

In the appendix, we will provide more substantial applications of features new to HOOMD-blue.

Trigger that detects nucleation

This example demonstrates a `Trigger` that returns true when a threshold Q_6 Steinhardt order parameter [SNR] (as calculated by freud) is reached. Such a `Trigger` could be used for BCC nucleation detection which could trigger a decrease in cooling rate, a more frequent output of simulation trajectories, or any other desired action. Also, in this example we showcase the use of the zero-copy rank-local data access. This example also requires the use of ghost particles, which are a subset of particles bordering a MPI rank's local box. Ghost particles are known by a rank, but the rank is not responsible for updating them. In this case, ghost particles are required for computing the Q_6 value for particles near the edges of the current rank's local simulation box.

```
import numpy as np
import freud
from mpi4py import MPI
from hoomd.trigger import Trigger

class Q6Trigger(Trigger):
    def __init__(self, simulation, threshold,
                 mpi_comm=None):
        super().__init__()
        self.threshold = threshold
        self.state = simulation.state
        nr = simulation.device.num_ranks
        if nr > 1 and mpi_comm is None:
            raise RuntimeError()
        elif nr > 1:
            self.comm = mpi_comm
        else:
            self.comm = None
        self.q6 = freud.order.Steinhardt(l=6)

    def __call__(self, timestep):
        with self.state.cpu_local_snapshot as data:
            part = data.particles
            box = data.box
            aabb_box = freud.locality.AABBQuery(
                box, part.positions_with_ghosts)
            nlist = aabb_box.query(
                part.position,
                {'num_neighbors': 12,
                 'exclude_ii': True})
            Q6 = np.nanmean(self.q6.compute(
                (box, part.positions_with_ghosts),
                nlist).particle_order)
            if self.comm:
                return self.comm.allreduce(
                    Q6 >= self.threshold,
                    op=MPI.LOR)
            else:
                return Q6 >= self.threshold
```

Pandas Logger Back-end

Here we highlight the ability to use the `Logger` class to create a Pandas back end for simulation data. It will store the scalar and string quantities in a single `pandas.DataFrame` object while each array-like object is stored in a separate `DataFrame` object. All `DataFrame` objects are stored in a single dictionary.

```
import pandas as pd
from hoomd.custom import Action
from hoomd.util import (
    dict_flatten, dict_filter, dict_map)

def is_flag(flags):
    def func(v):
        return v[1] in flags
    return func
```

```
def not_none(v):
    return v[0] is not None

def hnd_2D_arrays(v):
    if v[1] in ['scalar', 'string', 'state']:
        return v
    elif len(v[0].shape) == 2:
        return {
            str(i): col
            for i, col in enumerate(v[0].T)}


class DataFrameBackEnd(Action):
    def __init__(self, logger):
        self.logger = logger

    def act(self, timestep):
        log_dict = self.logger.log()
        is_scalar = is_flag(['scalar', 'string'])
        sc = dict_flatten(dict_map(dict_filter(
            log_dict,
            lambda x: not_none(x) and is_scalar(x)),
            lambda x: x[0]))
        rem = dict_flatten(dict_map(dict_filter(
            log_dict,
            lambda x: not_none(x) \
                and not is_scalar(x)),
            hnd_2D_arrays))

        if not hasattr(self, 'data'):
            self.data = {
                'scalar': pd.DataFrame(
                    columns=[
                        '.'.join(k) for k in sc]),
                'array': {
                    '.'.join(k): pd.DataFrame()
                    for k in rem}}

        sdf = pd.DataFrame(
            {'.'.join(k): v for k, v in sc.items()},
            index=[timestep])
        rdf = {'.'.join(k): pd.DataFrame(
                    v, columns=[timestep]).T
            for k,v in rem.items()}
        data = self.data
        data['scalar'] = data['scalar'].append(sdf)
        data['array'] = {
            k: v.append(rdf[k])
            for k, v in data['array'].items()}
```

REFERENCES

[AGG] Joshua A. Anderson, Jens Glaser, and Sharon C. Glotzer. HOOMD-blue: A Python package for high-performance molecular dynamics and hard particle Monte Carlo simulations. 173:109363. URL: http://www.sciencedirect.com/science/article/pii/S0927025619306627, doi:10.1016/j.commatsci.2019.109363.

[ALT] Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. 227(10):5342–5359. URL: http://www.sciencedirect.com/science/article/pii/S0021999108000818, doi:10.1016/j.jcp.2008.01.047.

[AMS+] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C. Smith, Berk Hess, and Erik Lindahl. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. 1-2:19–25. URL: http://www.sciencedirect.com/science/article/pii/S2352711015000059, doi:10.1016/j.softx.2015.06.001.

[AW] B. J. Alder and T. E. Wainwright. Studies in Molecular Dynamics. I. General Method. 31(2):459–466. URL: https://aip.scitation.org/doi/abs/10.1063/1.1730376, doi:10.1063/1.1730376.

[BCR+] Surendra Byna, Jerry Chou, Oliver Rubel, Prabhat, Homa Karimabadi, William S. Daughter, Vadim Roytershteyn, E. Wes Bethel, Mark Howison, Ke-Jou Hsu, Kuan-Wu Lin, Arie Shoshani, Andrew Uselton, and Kesheng Wu. Parallel I/O, analysis, and visualization of a trillion particle simulation. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. doi:10.1109/SC.2012.92.

[BvdSvD] H. J. C. Berendsen, D. van der Spoel, and R. van Drunen. GROMACS: A message-passing parallel molecular dynamics implementation. 91(1):43–56. URL: http://www.sciencedirect.com/science/article/pii/001046559500042E, doi:10.1016/0010-4655(95)00042-E.

[CCD+] David A. Case, Thomas E. Cheatham, Tom Darden, Holger Gohlke, Ray Luo, Kenneth M. Merz, Alexey Onufriev, Carlos Simmerling, Bing Wang, and Robert J. Woods. The Amber biomolecular simulation programs. 26(16):1668–1688. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.20290, doi:10.1002/jcc.20290.

[CG] Peter T Cummings and Justin B Gilmer. Open-source molecular modeling software in chemical engineering. 23:99–105. URL: http://www.sciencedirect.com/science/article/pii/S2211339819300073, doi:10.1016/j.coche.2019.03.008.

[DEG] Pablo F. Damasceno, Michael Engel, and Sharon C. Glotzer. Predictive Self-Assembly of Polyhedra into Complex Structures. 337(6093):453–457. URL: https://science.sciencemag.org/content/337/6093/453, arXiv:22837525, doi:10.1126/science.1220869.

[DZK+] Gregory L. Dignon, Wenwei Zheng, Young C. Kim, Robert B. Best, and Jeetain Mittal. Sequence determinants of protein phase behavior from a coarse-grained model. 14(1):e1005941. URL: https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1005941, doi:10.1371/journal.pcbi.1005941.

[EGK+] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz and dynagraph – static and dynamic graph drawing tools. In *Graph Drawing Software*, pages 127–148. Springer-Verlag.

[ESC+] Peter Eastman, Jason Swails, John D. Chodera, Robert T. McGibbon, Yutong Zhao, Kyle A. Beauchamp, Lee-Ping Wang, Andrew C. Simmonett, Matthew P. Harrigan, Chaya D. Stern, Rafal P. Wiewiora, Bernard R. Brooks, and Vijay S. Pande. OpenMM 7: Rapid development of high performance algorithms for molecular dynamics. 13(7):e1005659. URL: https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1005659, doi:10.1371/journal.pcbi.1005659.

[FIE] Yue Fan, Takuya Iwashita, and Takeshi Egami. How thermally activated deformation starts in metallic glass. 5(1):1–7. URL: https://www.nature.com/articles/ncomms6083, doi:10.1038/ncomms6083.

[GKNV] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-phong Vo. A Technique for Drawing Directed Graphs. 19(3):214–230.

[GLB+] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. pages 98–105. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html, doi:10.25080/Majora-629e541a-00e.

[GNA+] Jens Glaser, Trung Dac Nguyen, Joshua A. Anderson, Pak Lui, Filippo Spiga, Jaime A. Millan, David C. Morse, and Sharon C. Glotzer. Strong scaling of general-purpose molecular dynamics simulations on GPUs. 192:97–107. URL: http://www.sciencedirect.com/science/article/pii/S0010465515000867, doi:10.1016/j.cpc.2015.02.028.

[GSJ] Sharon C. Glotzer, Dietrich Stauffer, and Naeem Jan. Monte Carlo simulations of phase separation in chemically reactive binary mixtures. 72(26):4109–4112. URL: https://link.aps.org/doi/10.1103/PhysRevLett.72.4109, doi:10.1103/PhysRevLett.72.4109.

[GTK+] Horacio V. Guzman, Nikita Tretyakov, Hideki Kobayashi, Aoife C. Fogarty, Karsten Kreis, Jakub Krajniak, Christoph Junghans, Kurt Kremer, and Torsten Stuehn. ESPResSo++ 2.0: Advanced methods for multiscale molecular simulation. 238:66–76. URL: http://www.sciencedirect.com/science/article/pii/S0010465518304399, doi:10.1016/j.cpc.2018.12.017.

[Hun] John D. Hunter. Matplotlib: A 2D Graphics Environment. 9(3):90–95. doi:10.1109/MCSE.2007.55.

[JRM] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. Pybind11

– Seamless operability between C++11 and Python. URL: https://github.com/pybind/pybind11.

[KSJ+] Christoph Klein, János Sallai, Trevor J. Jones, Christopher R. Iacovella, Clare McCabe, and Peter T. Cummings. A Hierarchical, Component Based Approach to Screening Properties of Soft Matter. In Randall Q Snurr, Claire S. Adjiman, and David A. Kofke, editors, *Foundations of Molecular Modeling and Simulation: Select Papers from FOMMS 2015*, Molecular Modeling and Simulation, pages 79–92. Springer. URL: https://doi.org/10.1007/978-981-10-1128-3_5, doi:10.1007/978-981-10-1128-3_5.

[KST+] Christoph Klein, Andrew Z. Summers, Matthew W. Thompson, Justin B. Gilmer, Clare McCabe, Peter T. Cummings, Janos Sallai, and Christopher R. Iacovella. Formalizing atom-typing and the dissemination of force fields with foyer. 167:215–227. URL: http://www.sciencedirect.com/science/article/pii/S0927025619303040, doi:10.1016/j.commatsci.2019.05.026.

[LPS] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 1–6. Association for Computing Machinery. URL: https://doi.org/10.1145/2833157.2833162, doi:10.1145/2833157.2833162.

[Mar] Marcus G. Martin. MCCCS Towhee: A tool for Monte Carlo molecular simulation. 39(14-15):1212–1222. URL: https://doi.org/10.1080/08927022.2013.828208, doi:10.1080/08927022.2013.828208.

[MBH+] Robert T. McGibbon, Kyle A. Beauchamp, Matthew P. Harrigan, Christoph Klein, Jason M. Swails, Carlos X. Hernández, Christian R. Schwantes, Lee-Ping Wang, Thomas J. Lane, and Vijay S. Pande. MDTraj: A Modern Open Library for the Analysis of Molecular Dynamics Trajectories. 109(8):1528–1532. URL: http://www.sciencedirect.com/science/article/pii/S0006349515008267, doi:10.1016/j.bpj.2015.08.015.

[McK] Wes McKinney. Data Structures for Statistical Computing in Python. pages 56–61. URL: https://conference.scipy.org/proceedings/scipy2010/mckinney.html, doi:10.25080/Majora-92bf1922-00a.

[MDWB] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. 32(10):2319–2327. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[MRR+] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of State Calculations by Fast Computing Machines. 21(6):1087–1092. URL: https://aip.scitation.org/doi/abs/10.1063/1.1699114, doi:10.1063/1.1699114.

[NBB+] Christoph Niethammer, Stefan Becker, Martin Bernreuther, Martin Buchholz, Wolfgang Eckhardt, Alexander Heinecke, Stephan Werth, Hans-Joachim Bungartz, Colin W. Glass, Hans Hasse, Jadran Vrabec, and Martin Horsch. Ls1 mardyn: The Massively Parallel Molecular Dynamics Code for Large Systems. 10(10):4455–4464. URL: https://doi.org/10.1021/ct500169q, doi:10.1021/ct500169q.

[PGM+] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d\ textquotesingle Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8026–8037. Curran Associates, Inc. URL: http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[Pli] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. URL: https://www.osti.gov/biblio/10176421, doi:10.2172/10176421.

[PVG+] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine Learning in Python. 12(85):2825–2830. URL: http://jmlr.org/papers/v12/pedregosa11a.html.

[RDH+] Vyas Ramasubramani, Bradley D. Dice, Eric S. Harper, Matthew P. Spellings, Joshua A. Anderson, and Sharon C. Glotzer. Freud: A software suite for high throughput analysis of particle simulation data. page 107275. URL: http://www.sciencedirect.com/science/article/pii/S0010465520300916, doi:10.1016/j.cpc.2020.107275.

[SCW] Romelia Salomon-Ferrer, David A. Case, and Ross C. Walker. An overview of the Amber biomolecular simulation package. 3(2):198–210. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/wcms.1121, doi:10.1002/wcms.1121.

[SD] Matthew Spellings and Bradley D. Dice. Plato. URL: https://github.com/glotzerlab/plato.

[SDS+] David E. Shaw, Ron O. Dror, John K. Salmon, J. P. Grossman, Kenneth M. Mackenzie, Joseph A. Bank, Cliff Young, Martin M. Deneroff, Brannon Batson, Kevin J. Bowers, Edmond Chow, Michael P. Eastwood, Douglas J. Ierardi, John L. Klepeis, Jeffrey S. Kuskin, Richard H. Larson, Kresten Lindorff-Larsen, Paul Maragakis, Mark A. Moraes, Stefano Piana, Yibing Shan, and Brian Towles. Millisecond-scale molecular dynamics simulations on Anton. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 1–11. Association for Computing Machinery. URL: https://doi.org/10.1145/1654059.1654126, doi:10.1145/1654059.1654126.

[SMAG] Matthew Spellings, Ryan L. Marson, Joshua A. Anderson, and Sharon C. Glotzer. GPU accelerated Discrete Element Method (DEM) molecular dynamics for conservative, faceted particle simulations. 334:460–467. URL: http://www.sciencedirect.com/science/article/pii/S0021999117300244, doi:10.1016/j.jcp.2017.01.014.

[SMM+] Jindal K. Shah, Eliseo Marin-Rimoldi, Ryan Gotchy Mullen, Brian P. Keene, Sandip Khan, Andrew S. Paluch, Neeraj Rai, Lucienne L. Romanielo, Thomas W. Rosch, Brian Yoo, and Edward J. Maginn. Cassandra: An open source Monte Carlo package for molecular simulation. 38(19):1727–1739. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.24807, doi:10.1002/jcc.24807.

[SNR] Paul J. Steinhardt, David R. Nelson, and Marco Ronchetti. Bond-orientational order in liquids and glasses. 28(2):784–805. URL: https://link.aps.org/doi/10.1103/PhysRevB.28.784, doi:10.1103/PhysRevB.28.784.

[Sto] John Edward Stone. An effiicient library for parallel ray tracing and animation. URL: http://jedi.ks.uiuc.edu/~johns/tachyon/papers/thesis.pdf.

[Stu] Alexander Stukowski. Visualization and analysis of atomistic simulation data with OVITO–the Open Visualization Tool. 18(1):015012. URL: https://doi.org/10.1088%2F0965-0393%2F18%2F1%2F015012, doi:10.1088/0965-0393/18/1/015012.

[TGM+] Matthew W. Thompson, Justin B. Gilmer, Ray A. Matsumoto, Co D. Quach, Parashara Shamaprasad, Alexander H. Yang, Christopher R. Iacovella, Clare McCabe, and Peter T. Cummings. Towards molecular simulations that are transparent, reproducible, usable by others, and extensible (TRUE). 118(9-10):e1742938. URL: https://doi.org/10.1080/00268976.2020.1742938, doi:10.1080/00268976.2020.1742938.

[vdWCV] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. 13(2):22–30. doi:10.1109/MCSE.2011.37.

[VGO+] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C. J. Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, and Paul van Mulbregt. SciPy 1.0: Fundamental algorithms for scientific computing in Python. 17(3):261–272. URL: https://www.nature.com/articles/s41592-019-0686-2, doi:10.1038/s41592-019-0686-2.

[WWS+] Florian Weik, Rudolf Weeber, Kai Szuttor, Konrad Breitsprecher, Joost de Graaf, Michael Kuron, Jonas Landsgesell, Henri Menke, David Sean, and Christian Holm. ESPResSo 4.0 – an extensible software package for simulating soft matter systems. 227(14):1789–1816. URL: https://doi.org/10.1140/epjst/e2019-800186-9, doi:10.1140/epjst/e2019-800186-9.

[zot] CuPy. URL: https://cupy.chainer.org/.