garnett Documentation

Release 0.5.0

The Regents of the University of Michigan

Aug 20, 2019

Getting started

CHAPTER 1

Installation

1.1 Requirements

Installing the **garnett** package requires Python 2.7 or 3.5+, **numpy**, and **rowan**.

1.2 With conda

To install the package with conda, execute

```
$ conda install -c conda-forge garnett
```

To upgrade, execute

```
$ conda update garnett
```

Note: This is the recommended installation method.

1.3 With pip

To install the package with the package manager pip, execute

```
$ pip install git+https://github.com/glotzerlab/garnett.git#egg=garnett --user
```

To upgrade the package, simply execute the same command with the *–upgrade* option.

```
$ pip install git+https://github.com/glotzerlab/garnett.git#egg=garnett --user --
↪upgrade
```

1.4 With git

Alternatively you can clone the git repository and use the `setup.py` to install the package.

```
git clone https://github.com/glotzerlab/garnett.git
cd garnett
python setup.py install --user
```

CHAPTER 2

Quickstart

2.1 Reading and writing of trajectories

2.1.1 Reading and writing with automatic filetype detection

The *garnett.read()* and *garnett.write()* functions will automatically determine the type of a trajectory from its file extension. This can be used to quickly load and save *Trajectory* objects.

```
import garnett
# Load a GSD file...
with garnett.read('dump.gsd') as traj:
    print(len(traj))
    # ...do things with the trajectory, then output a GTAR file
    garnett.write(traj, 'output.tar')
```

2.1.2 Using reader and writer classes

Readers and writers are defined in the `reader` and `writer` modules. The following code uses the *PosFileReader* and *PosFileWriter* as an example.

```
from garnett.reader import PosFileReader
from garnett.writer import PosFileWriter

pos_reader = PosFileReader()
pos_writer = PosFileWriter()

with open('posfile.pos') as file:
    # Access the trajectory
    traj = pos_reader.read(file)

    # Write to standard out:
    pos_writer.write(traj)
```

(continues on next page)

(continued from previous page)

```
    # or directly to a file:
    with open('out.pos', 'w') as posfile:
        pos_writer.write(traj, posfile)
```

2.2 Data access

2.2.1 Indexing and slicing

Once you read a trajectory, access individual frames or sub-trajectories by indexing and slicing:

```
# Select individual frames:
first_frame = traj[0]
last_frame = traj[-1]
n_th_frame = traj[n]
# and so on

# Create a sub-trajectory from the ith frame
# to the (j-1)th frame:
sub_trajectory = traj[i:j]

# We can use advanced slicing techniques:
every_second_frame = traj[::2]
the_last_ten_frames = traj[-10::]
```

The actual trajectory data is then either accessed on a *per trajectory* or *per frame* basis.

2.2.2 Trajectory array access

Access positions, orientations and types as coherent numpy arrays, by calling the *load_arrays()* method. This method will load the complete trajectory into memory and make positions, orientations and types available via properties:

```
traj.load_arrays()
traj.N                # M
traj.positions        # MxNx3
traj.orientations     # MxNx4
traj.velocities       # MxNx3
traj.mass             # MxN
traj.charge           # MxN
traj.diameter         # MxN
traj.moment_inertia   # MxNx3
traj.angmom           # MxNx4
traj.image            # MxNx4
traj.types            # MxN
traj.type_ids         # MxN
traj.type             # list of type names ordered by type_id

# where M=len(traj) is the number of frames and N=max((len(f) for f in traj))
# is the is the maximum number of particles in any frame.
```

2.2.3 Individual frame access

Inidividual frame objects can be accessed via indexing of a (sub-)trajectory object:

```
frame = traj[i]
frame.box               # garnett.trajectory.Box object
frame.types             # N
frame.positions         # Nx3
frame.orientations      # Nx4
frame.velocities        # Nx3
frame.mass              # N
frame.charge            # N
frame.diameter          # N
frame.moment_inertia    # Nx3
frame.angmom            # Nx4
frame.data              # A dictionary of lists for each attribute
frame.data_key          # A list of strings
frame.shapedef          # A ordered dictionary of instances of ShapeDefinition
```

2.2.4 Iterating over trajectories

Iterating over trajectories is the most **memory-efficient** form of data access. Each frame will be loaded *prior* to access and unloaded *post* access, such that there is only one frame loaded into memory at the same time.

```
# Iterate over a trajectory directly for read-only data access
for frame in traj:
    print(frame.positions)
```

2.3 Efficient modification of trajectories

Use a combination of reading, writing, and iteration for **memory-efficient** modification of large trajectory data. This is an example on how to modify frames in-place:

```
import numpy as np
import garnett as gt

def center(frame):
    frame.positions -= np.average(frame.positions, axis=0)
    return frame

with gt.read('in.pos') as traj:
    traj_centered = Trajectory((center(frame) for frame in traj))
    gt.write(traj_centered, 'out.pos')
```

2.4 Loading trajectories into memory

The *Trajectory* class is designed to be *memory-efficient*. This means that loading all trajectory data into memory requires an explicit call of the *load()* or *load_arrays()* methods.

```
# Make trajectory data accessible via arrays:
traj.load_arrays()
traj.positions

# Load all frames:
traj.load()
frame = traj[i]
traj.positions    # load() also loads arrays
```

Note: In general, loading all frames with *load()* is more expensive than just loading arrays with *load_arrays()*. Loading all frames also loads the arrays.

Sub-trajectories inherit already loaded data:

```
traj.load_arrays()
sub_traj = traj[i:j]
sub_traj.positions
```

Tip: If you are only interested in sub-trajectory data, consider to call *load()* or *load_arrays()* only for the sub-trajectory.

2.5 Example use with HOOMD-blue

The **garnett** frames can be used to initialize HOOMD-blue by creating snapshots with the *make_snapshot()* method or by copying the frame data to existing snapshots with the *copyto_snapshot()* methods:

```
from hoomd import init
import garnett as gt

with gt.read('cube.pos') as traj:

    # Initialize from last frame
    snapshot = traj[-1].make_snapshot()
    system = init.read_snapshot(snapshot)

    # Restore last frame
    snapshot = system.take_snapshot()
    traj[-1].copyto_snapshot(snapshot)
```

CHAPTER 3

Readers & Writers

This is the API documentation for all readers and writers provided by **garnett**.

3.1 Automatic reader/writer

garnett.**read**(*filename_or_fileobj*, *template=None*, *fmt=None*)

This function reads a file and returns a trajectory, autodetecting the file format unless `fmt` is specified.

Parameters

- **filename_or_fileobj** (*string or file object*) – Filename to read.
- **template** (*string*) – Optional template for the GSDHOOMDFileReader.
- **fmt** (*string*) – File format, one of 'gsd', 'gtar', 'pos', 'cif', 'dcd', 'xml' (default: None, autodetected from filename_or_fileobj)

Returns Trajectory read from the file.

Return type *Trajectory*

garnett.**write**(*traj*, *filename_or_fileobj*, *fmt=None*)

This function writes a trajectory to a file, autodetecting the file format unless `fmt` is specified.

Parameters

- **traj** (*Trajectory*) – Trajectory to write.
- **filename_or_fileobj** (*string or file object*) – Filename to write.
- **fmt** (*string*) – File format, one of 'gsd', 'gtar', 'pos', 'cif' (default: None, autodetected from filename_or_fileobj)

3.2 General API

Readers and writers are defined in the `reader` and `writer` modules. All readers and writers work with **file-like objects** and use the following API:

```
reader = garnett.reader.Reader()
writer = garnett.writer.Writer()

with open('trajectory_file') as infile:
    traj = reader.read(infile)

    # Dump to a string:
    pos = writer.dump(traj)

    # Write to standard out:
    writer.write(traj)

    # or directly to a file:
    with open('dump', 'w') as outfile:
        writer.write(traj, outfile)
```

Important: Some readers and writers work with **binary** files, which means that when opening those files for reading or writing you need to use the `rb` or `wb` mode. This applies for example to DCD-files:

```
dcd_reader = garnett.reader.DCDFileReader()
with open('dump.dcd', 'rb') as dcdfile:
    dcd_traj = dcd_reader.read(dcdfile)
```

3.3 File Formats

This table outlines the supported properties of each format reader and writer.

Format	Positions	Box	Orientations	Velocities	Shape	Additional Properties (See below)
POS	RW	RW	RW	N/A+	RW	N/A
GSD	RW	RW	RW	RW	RW	RW
GTAR	RW	RW	RW	RW	RW	RW
CIF	RW	RW	N/A	N/A	N/A	N/A
DCD	R	R	R	R	N/A	N/A
XML	R	R	R	R	N/A	N/A

- RW indicates garnett can read and write on this format.
- R indicates garnett can only read.
- N/A indicates the format does not support storing this property.
- **Additional Properties:**
  - Mass
  - Charge
  - Diameter

- Angular momentum
- Moment of inertia
- Image

The following collection of readers and writers is ordered by different file formats.

3.3.1 POS-Files

The *POS*-format is a non-standardized *text-based* format which is human-readable, but very inefficient for storing of trajectory data. The format is used as primary input/output format for the **injavis** visualization tool. HOOMD-blue provides a writer for this format, which is classified as deprecated since version 2.0.

class garnett.reader.**PosFileReader**(*precision=None*)

POS-file reader for the Glotzer Group, University of Michigan.

Authors: Carl Simon Adorf, Richmond Newmann

```
reader = PosFileReader()
with open('a_posfile.pos', 'r', encoding='utf-8') as posfile:
    return reader.read(posfile)
```

Parameters **precision** (*int*) – The number of digits to round floating-point values to.

read(*stream*, *default_type='A'*)

Read text stream and return a trajectory instance.

Parameters

- **stream** (*A file-like textstream.*) – The stream, which contains the posfile.
- **default_type** (*str*) – The default particle type for posfile dialects without type definition.

class garnett.writer.**PosFileWriter**(*rotate=False*)

POS-file writer for the Glotzer Group, University of Michigan.

Author: Carl Simon Adorf

```
writer = PosFileWriter()
with open('a_posfile.pos', 'w', encoding='utf-8') as posfile:
    writer.write(trajectory, posfile)
```

Parameters **rotate** (*bool*) – Rotate the system into the view rotation instead of adding it to the metadata with the 'rotation' keyword.

dump(*trajectory*)

Serialize trajectory into pos-format.

Parameters **trajectory** (*Trajectory*) – The trajectory to serialize.

Return type str

write(*trajectory*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

Serialize a trajectory into pos-format and write it to file.

Parameters

- **trajectory** (*Trajectory*) – The trajectory to serialize

- **file** – A file-like object.

3.3.2 GSD (HOOMD-blue schema)

The *GSD*-format is a highly efficient and versatile *binary* format for storing and reading trajectory data. HOOMD-blue provides a writer for this format.

See also: http://gsd.readthedocs.io

class garnett.reader.**GSDHOOMDFileReader**

Hoomd-GSD-file reader for the Glotzer Group, University of Michigan.

Author: Carl Simon Adorf

This class provides a wrapper for the trajectory reader implementation as part of the gsd package.

A gsd file may not contain all shape information. To provide additional information it is possible to pass a frame object, whose properties are copied into each frame of the gsd trajectory.

The example is given for a HOOMD-blue XML frame:

```
xml_reader = HOOMDXMLFileReader()
gsd_reader = GSDHOOMDFileReader()

with open('init.xml') as xmlfile:
    with open('dump.gsd', 'rb') as gsdfile:
        xml_frame = xml_reader.read(xmlfile)[0]
        traj = gsd_reader.read(gsdfile, xml_frame)
```

About the read_gsd_shape_data parameter: This parameter was removed. By default, shape information is read from a passed frame object, if one provided. Otherwise, shape information is read from the gsd file.

read(*stream*, *frame=None*)

Read binary stream and return a trajectory instance.

Parameters

- **stream** (*A file-like binary stream*) – The stream, which contains the gsd-file.
- **frame** (trajectory.Frame) – A frame containing shape information that is not encoded in the GSD-format. By default, shape information is read from the passed frame object, if one provided. Otherwise, shape information is read from the gsd file.

class garnett.writer.**GSDHOOMDFileWriter**

GSD file writer for the Glotzer Group, University of Michigan.

Author: Vyas Ramasubramani Author: Bradley Dice

```
writer = GSDHOOMDFileWriter()
with open('file.gsd', 'wb') as gsdfile:
    writer.write(trajectory, gsdfile)

# For appending to the file
with open('file.gsd', 'ab') as gsdfile:
    writer.write(trajectory, gsdfile)
```

write(*trajectory*, *stream*)

Serialize a trajectory into gsd-format and write it to a file.

Parameters

- **trajectory** (*Trajectory*) – The trajectory to serialize
- **stream** (*File stream*) – The file to write to.

3.3.3 GeTAR

The *GeTAR*-format is a highly versatile, *binary* format for storing and reading trajectory data. HOOMD-blue provides a writer for this format.

See also: https://libgetar.readthedocs.io

class garnett.reader.**GetarFileReader**

getar-file reader for the Glotzer Group, University of Michigan.

Authors: Matthew Spellings, Carl Simon Adorf

Read GEneric Trajectory ARchive files, a binary format designed for efficient, extensible storage of trajectory data.

This class provides a wrapper for the gtar library.

```
reader = GetarFileReader()
with open('trajectory.tar', 'rb') as file:
    traj = reader.read(file)
```

read(*stream*, *default_type='A'*, *default_box=None*)

Read binary stream and return a trajectory instance.

Parameters

- **stream** (*A file-like binary stream.*) – The stream, which contains the GeTarFile.
- **default_type** (*str*) – The default particle type for posfile dialects without type definition.
- **default_box** (numpy.ndarray) – The default_box value is used if no box is specified in the libgetar file. Defaults to [Lx=Ly=Lz=1.0].

3.3.4 HOOMD-blue XML

The HOOMD-blue XML-format contains topological information about one individual frame. HOOMD-blue provides a writer for this format, which is classified as deprecated since version 2.0.

class garnett.reader.**HOOMDXMLFileReader**

Reader for XML-files containing HOOMD-blue snapshots.

read(*stream*)

Read text stream and return a trajectory instance.

Parameters **stream** (*A file-like textstream.*) – The stream, which contains the xmlfile.

3.3.5 DCD

The *DCD*-format is a very storage efficient *binary* format for storing simple trajectory data. The format contains only data about xyz-positions and the boxes of individual frames.

HOOMD-blue provides a writer for this format with a special dialect for 2-dimensional systems. The *garnett* dcd-reader is capable of reading both the standard and the 2-dim. dialect.

Note: Unlike most other readers, the *DCDFileReader* will return an instance of *DCDTrajectory*, which is optimized for the DCD-format. This special trajectory class provides the *xyz()* method for accessing xyz-coordinates with minimal overhead.

---

class garnett.reader.**DCDFileReader**

DCD-file reader for the Glotzer Group, University of Michigan.

Author: Carl Simon Adorf

A dcd file consists only of positions. To provide additional information it is possible to provide a frame object, whose properties are copied into each frame of the dcd trajectory.

The example is given for a HOOMD-blue xml frame:

```
xml_reader = HOOMDXMLFileReader()
dcd_reader = DCDFileReader()

with open('init.xml') as xmlfile:
    with open('dump.dcd', 'rb') as dcdfile:
        xml_frame = xml_reader.read(xmlfile)[0]
        traj = reader.read(dcdfile, xml_frame)
```

Note: If the topology frame is 2-dimensional, the dcd trajectory positions are interpreted such that the first two values contain the xy-coordinates, the third value is an euler angle.

The euler angle is converted to a quaternion and stored in the orientation of the frame.

To retrieve the euler angles, simply convert the quaternion:

```
alpha = 2 * np.arccos(traj[0].orientations.T[0])
```

---

read(*stream*, *frame=None*, *default_type=None*)

Read binary stream and return a trajectory instance.

Parameters

- **stream** (*A file-like binary stream*) – The stream, which contains the dcd-file.
- **frame** (trajectory.Frame) – A frame containing topological information that cannot be encoded in the dcd-format.
- **default_type** (*str*) – A type name to be used when no first frame is provided, defaults to 'A'.

Returns A trajectory instance.

Return type *DCDTrajectory*

class garnett.dcdfilereader.**DCDTrajectory**(*frames=None*, *dtype=None*)

arrays_loaded()

Returns true if arrays are loaded into memory.

See also: *load_arrays()*

load_arrays()
Load positions, orientations and types into memory.

After calling this function, positions, orientations and types can be accessed as coherent numpy arrays:

```
traj.load_arrays()
traj.N             # M -- frame sizes
traj.positions     # MxNx3
traj.orientations  # MxNx4
traj.types         # MxN
traj.type_ids      # MxN
```

Note: It is not necessary to call this function again when accessing sub trajectories:

```
traj.load_arrays()
sub_traj = traj[m:n]
sub_traj.positions
```

However, it may be more efficient to call *load_arrays()* only for the sub trajectory if other data is not of interest:

```
sub_traj = traj[m:n]
sub_traj.load_arrays()
sub_traj.positions
```

---

xyz(*xyz=None*)
Return the xyz coordinates of the dcd file.

Use this function to access xyz-coordinates with minimal overhead and maximal performance.

You can provide a reference to an existing numpy.ndarray with shape (Mx3xN), where M is the length of the trajectory and N is the number of particles. Please note that the array needs to be of data type float32 and in-memory contiguous.

Parameters **xyz** (*numpy.ndarray*) – A numpy array of shape (Mx3xN).

Returns A view or a copy of the xyz-array of shape (MxNx3).

Return type numpy.ndarray

class garnett.reader.**PyDCDFileReader**
Pure-python DCD-file reader for the Glotzer Group.

This class is a pure python dcd-reader implementation which should only be used when the more efficient cythonized dcd-reader is not available or you want to work with non-standard file-like objects.

See also:

The API is identical to: *DCDFileReader*

3.3.6 CIF

The *cif*-format is a *text-based* format primarily used in the context of crystallography.

class garnett.reader.**CifFileReader**(*precision=None*, *tolerance=1e-05*)
CIF-file reader for the Glotzer Group, University of Michigan.

Requires the PyCifRW package to parse CIF files.

Authors: Matthew Spellings

```
reader = CifFileReader()
with open('a_ciffile.cif', 'r') as ciffile:
    traj = reader.read(ciffile)
```

Parameters

- **precision** (*int*) – The number of digits to round floating-point values to.
- **tolerance** (*float*) – Floating-point tolerance of particle identity as symmetry operations are applied

read(*stream*, *default_type='A'*)
Read text stream and return a trajectory instance.

Parameters

- **stream** (*A file-like textstream.*) – The stream, which contains the ciffile.
- **default_type** (*str*) – The default particle type for ciffile dialects without type definition.

class garnett.writer.**CifFileWriter**
cif-file writer for the Glotzer Group, University of Michigan.

Authors: Julia Dshemuchadse, Carl Simon Adorf

```
writer = CifFileWriter()

# write to screen:
write.write(trajectory)

# write to file:
with open('a_ciffile.pos', 'w') as ciffile:
    writer.write(trajectory, ciffile)
```

dump(*trajectory*, *data='simulation'*, *occupancy=None*, *fractional=False*, *raw=False*)
Serialize trajectory into cif-format.

Parameters

- **trajectory** (*Trajectory*) – The trajectory to serialize.
- **data** (*str*) – Identifier which be will written to the file, signifying the origin of the data.
- **occupancy** (*numpy.array*) – The default occupancy of individual particles.

Return type str

write(*trajectory*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *data='simulation'*, *occupancy=None*, *fractional=False*, *raw=False*)
Serialize a trajectory into cif-format and write it to file.

Parameters

- **trajectory** (*Trajectory*) – The trajectory to serialize
- **file** (*A file-like object.*) – The file to write the trajectory to.
- **data** (*str*) – Identifier which be will written to the file, signifying the origin of the data.
- **occupancy** (*int*) – The default occupancy of individual particles.

- **fractional** (*bool*) – Whether or not the input coordinates are fractional
- **raw** (*bool*) – Whether or not to write the raw CIF coordinates (with no transformations)

CHAPTER 4

Trajectory API

Instances of *Trajectory* give access to trajectory data stored in files and *file-like* objects. In the simplest case, trajectories are just a sequence of *Frame* instances.

4.1 Trajectories

class garnett.trajectory.**Trajectory**(*frames=None*, *dtype=None*)

Manages a particle trajectory data resource.

A trajectory is basically a sequence of *Frame* instances.

Trajectory data can either be accessed as coherent numpy arrays:

```
traj.load_arrays()
M = len(traj)
traj.N              # M
traj.positions      # MxNx3
traj.orientations   # MxNx4
traj.types          # MxN
traj.type_ids       # MxN
```

or by individual frames:

```
first_frame = traj[0]
last_frame = traj[-1]
n_th_frame = traj[n]

first_frame.positions      # Nx3
first_frame.orientations   # Nx4
first_frame.types          # Nx1
```

You can iterate through individual frames:

```
for frame in trajectory:
    print(frame.positions)
```

and create a sub-trajectory from the *i'th* to the *(j-1)'th* frame:

```
sub_trajectory = traj[i:j]
```

Parameters

- **frames** (*Frame*) – The individual frames of this trajectory.
- **dtype** – The default data type for trajectory data.

N

Access the frame sizes as a numpy array.

Mostly important when the trajectory has varying size.

```
pos_i = traj.positions[i][0:traj.N[i]]
```

Returns frame size as array with length M

Return type numpy.ndarray (dtype= numpy.int_)

Raises **RuntimeError** – When accessed before calling *load_arrays()* or *load()*.

TRAJ_ATTRIBUTES = ['N', 'type', 'types', 'type_ids', 'positions', 'orientations', 'vel

angmom

Access the particle angular momenta as a numpy array.

Returns particle angular momenta quaternions as (Nx4) element array

Return type numpy.ndarray

Raises **RuntimeError** – When accessed before calling *load_arrays()* or *load()*.

arrays_loaded()

Returns true if arrays are loaded into memory.

See also: *load_arrays()*

charge

Access the particle charge as a numpy array.

Returns particle charge as (N) element array

Return type numpy.ndarray

Raises **RuntimeError** – When accessed before calling *load_arrays()* or *load()*.

diameter

Access the particle diameter as a numpy array.

Returns particle diameter as (N) element array

Return type numpy.ndarray

Raises **RuntimeError** – When accessed before calling *load_arrays()* or *load()*.

image

Access the particle periodic images as a numpy array.

Returns particle periodic images as (Nx3) element array

Return type numpy.ndarray

Raises **RuntimeError** – When accessed before calling *load_arrays()* or *load()*.

load()
Load all frames into memory.

By default, only frames which are accessed are loaded into memory. Using this function, all frames are loaded at once.

This can be useful, e.g., if the trajectory resource cannot remain open, however in all other cases this should be avoided.

See also: *load_arrays()*

load_arrays()
Load positions, orientations and types into memory.

After calling this function, positions, orientations and types can be accessed as coherent numpy arrays:

```
traj.load_arrays()
traj.N              # M -- frame sizes
traj.positions      # MxNx3
traj.orientations   # MxNx4
traj.types          # MxN
traj.type_ids       # MxN
```

Note: It is not necessary to call this function again when accessing sub trajectories:

```
traj.load_arrays()
sub_traj = traj[m:n]
sub_traj.positions
```

However, it may be more efficient to call *load_arrays()* only for the sub trajectory if other data is not of interest:

```
sub_traj = traj[m:n]
sub_traj.load_arrays()
sub_traj.positions
```

---

loaded()
Returns True if all frames are loaded into memory.

See also: *load()*

mass
Access the particle mass as a numpy array.

Returns particle mass as (N) element array

Return type numpy.ndarray

Raises **RuntimeError** – When accessed before calling *load_arrays()* or *load()*.

moment_inertia
Access the particle principal moment of inertia components as a numpy array.

Returns particle principal moment of inertia components as (Nx3) element array

Return type numpy.ndarray

Raises **RuntimeError** – When accessed before calling *load_arrays()* or *load()*.

orientations
Access the particle orientations as a numpy array.

Orientations are stored as quaternions.

Returns particle orientations as (Nx4) array

Return type numpy.ndarray

Raises **RuntimeError** – When accessed before calling *load_arrays()* or *load()*.

positions
Access the particle positions as a numpy array.

Returns particle positions as (Nx3) array

Return type numpy.ndarray

Raises **RuntimeError** – When accessed before calling *load_arrays()* or *load()*.

set_dtype(*value*)
Change the data type of this trajectory.

This function cannot be called if any frame is already loaded.

Parameters **value** – The new data type value.

type
List of type names ordered by type_id.

Use the type list to map between type_ids and type names:

```
type_name = traj.type[type_id]
```

See also: *type_ids*

Returns particle types in order of type id.

Return type list

Raises **RuntimeError** – When accessed before calling *load_arrays()* or *load()*.

type_ids
Access the particle type ids as a numpy array.

See also: *type*

Returns particle type ids as (MxN) array.

Return type numpy.ndarray (dtype= numpy.int_)

Raises **RuntimeError** – When accessed before calling *load_arrays()* or *load()*.

types
Access the particle types as a numpy array.

Returns particles types as (MxN) array

Return type numpy.ndarray (dtype= numpy.str_)

Raises **RuntimeError** – When accessed before calling *load_arrays()* or *load()*.

velocities
Access the particle velocities as a numpy array.

Returns particle velocities as (Nx3) array

Return type numpy.ndarray

Raises **RuntimeError** – When accessed before calling *load_arrays()* or *load()*.

4.2 Frames

Trajectory data can be accessed via individual frames.

class garnett.trajectory.**Frame**(*dtype=None*)
A frame is a container object for the actual frame data.

The frame data is read from the origin stream whenever accessed.

Parameters **dtype** – The data type for frame data.

angmom
Nx4 array of angular momenta for N particles represented as quaternions.

box
Instance of *Box*

charge
Nx1 array of charges for N particles.

copyto_snapshot(*snapshot*)
Copy this frame to a HOOMD-blue snapshot.

data
A dictionary of lists for each attribute.

data_keys
A list of strings, where each string represents one attribute.

diameter
Nx1 array of diameters for N particles.

dtype
Return the data type for frame data.

image
Nx3 array of periodic images for N particles in 3 dimensions.

load()
Load the frame into memory.

loaded()
Returns True if the frame is loaded into memory.

make_snapshot()
Create a HOOMD-blue snapshot object from this frame.

mass
Nx1 array of masses for N particles.

moment_inertia
Nx3 array of principal moments of inertia for N particles in 3 dimensions.

orientations
Nx4 array of rotational coordinates for N particles represented as quaternions.

positions
Nx3 array of coordinates for N particles in 3 dimensions.

shapedef
A ordered dictionary of instances of `ShapeDefinition`.

types
Nx1 array of types represented as strings.

unload()
Unload the frame from memory.

Use this method carefully. This method removes the frame reference to the frame data, however any other references that may still exist, will prevent a removal of said data from memory.

velocities
Nx3 array of velocities for N particles in 3 dimensions.

view_rotation
A quaternion specifying a rotation that should be applied for visualization.

4.3 Box

The box instance gives access to the box of individual frames.

class garnett.trajectory.**Box**(*Lx*, *Ly*, *Lz*, *xy=0.0*, *xz=0.0*, *yz=0.0*, *dimensions=3*)
A triclinical box class.

You can access the box size and tilt factors via attributes:

```
# Reading
length_x = box.Lx
tilt_xy = box.xy
# etc.

# Setting
box.lx = 10.0
box.ly = box.lz = 5.0
box.xy = box.xz = box.yz = 0.01
# etc.
```

See also:

http://hoomd-blue.readthedocs.io/en/stable/box.html

Lx = None
The box length in x-direction.

Ly = None
The box length in y-direction.

Lz = None
The box length in z-direction.

dimensions = None
The number of box dimensions (2 or 3).

get_box_array()
Returns the box parameters as a 6-element list.

get_box_matrix()
Returns the box matrix (3x3).

The dimensions (Lx,Ly,Lz) are the diagonal.

round(*decimals=0*)
Return box instance with all values rounded up to the given precision.

xy = None
The box tilt factor in the xy-plane.

xz = None
The box tilt factor in the xz-plane.

yz = None
The box tilt factor in the yz-plane.

CHAPTER 5

Shape Definitions

Shape definitions contain information about the shape of individual particles. Some shapes define a `shape_dict` property, which returns a `dict` for consumption by visualization tools, in the format of `hoomd.hpmc.integrate.mode_hpmc.get_type_shapes()`.

class `garnett.shapes.`**FallbackShape**
This shape definition class is used when no specialized Shape class can be applied.

The fallback shape definition is a string containing the definition.

class `garnett.shapes.`**Shape**(*shape_class=None*, *color=None*)
Parent class of all shape objects.

Parameters

- **shape_class** (*str*) – Shape class directive, used for POS format (default: `None`).
- **color** (*str*) – Hexadecimal color string in format `RRGGBBAA` (default: `None`).

class `garnett.shapes.`**SphereShape**(*diameter*, *orientable=False*, *color=None*)
Shape class for spheres of a specified diameter.

Parameters

- **diameter** (*float*) – Diameter of the sphere.
- **orientable** (*bool*) – Set to True for spheres with orientation (default: `False`).
- **color** (*str*) – Hexadecimal color string in format `RRGGBBAA` (default: `None`).

shape_dict
Shape as dictionary. Example:

```
>>> SphereShape(2.0).shape_dict
{'type': 'Sphere', 'diameter': 2.0, 'orientable': False}
```

class `garnett.shapes.`**ArrowShape**(*thickness=0.1*, *color=None*)
Shape class for arrows of a specified thickness.

Parameters

- **thickness** (*float*) – Thickness of the arrow.
- **color** (*str*) – Hexadecimal color string in format RRGGBBAA (default: None).

class garnett.shapes.**SphereUnionShape**(*diameters*, *centers*, *colors=None*)
Shape class for sphere unions, such as rigid bodies of many spheres.

Parameters

- **diameters** (*list*) – List of sphere diameters.
- **centers** (*list*) – List of 3D center vectors.
- **colors** (*list*) – List of hexadecimal color strings in format RRGGBBAA (default: None).

class garnett.shapes.**PolygonShape**(*vertices*, *color=None*)
Shape class for polygons in a 2D plane.

Parameters

- **vertices** (*list*) – List of 2D vertex vectors.
- **color** (*str*) – Hexadecimal color string in format RRGGBBAA (default: None).

shape_dict
Shape as dictionary. Example:

```
>>> PolygonShape([[-0.5, -0.5], [0.5, -0.5], [0.5, 0.5]]).shape_dict
{'type': 'Polygon', 'rounding_radius': 0,
 'vertices': [[-0.5, -0.5], [0.5, -0.5], [0.5, 0.5]]}
```

class garnett.shapes.**SpheropolygonShape**(*vertices*, *rounding_radius=0*, *color=None*)
Shape class for rounded polygons in a 2D plane.

Parameters

- **vertices** (*list*) – List of 2D vertex vectors.
- **rounding_radius** (*float*) – Rounding radius applied to the spheropolygon (default: 0).
- **color** (*str*) – Hexadecimal color string in format RRGGBBAA (default: None).

shape_dict
Shape as dictionary. Example:

```
>>> SpheropolygonShape([[-0.5, -0.5], [0.5, -0.5], [0.5, 0.5]], 0.1).shape_
↪dict
{'type': 'Polygon', 'rounding_radius': 0.1,
 'vertices': [[-0.5, -0.5], [0.5, -0.5], [0.5, 0.5]]}
```

class garnett.shapes.**ConvexPolyhedronShape**(*vertices*, *color=None*)
Shape class for convex polyhedra.

Parameters

- **vertices** (*list*) – List of 3D vertex vectors.
- **color** (*str*) – Hexadecimal color string in format RRGGBBAA (default: None).

shape_dict
Shape as dictionary. Example:

```
>>> ConvexPolyhedronShape([[0.5, 0.5, 0.5], [0.5, -0.5, -0.5],
                           [-0.5, 0.5, -0.5], [-0.5, -0.5, 0.5]]).shape_dict
{'type': 'ConvexPolyhedron', 'rounding_radius': 0,
 'vertices': [[0.5, 0.5, 0.5], [0.5, -0.5, -0.5],
              [-0.5, 0.5, -0.5], [-0.5, -0.5, 0.5]]}
```

class garnett.shapes.**ConvexPolyhedronUnionShape**(*vertices*, *centers*, *orientations*, *colors=None*)

Shape class for unions of convex polyhedra.

Parameters

- **vertices** (*list*) – List of lists of 3D vertex vectors in particle coordinates (each polyhedron, each vertex).
- **centers** (*list*) – List of 3D polyhedra center vectors.
- **orientations** (*list*) – Orientations of the polyhedra, as a list of quaternions.
- **colors** (*list*) – List of hexadecimal color strings in format RRGGBBAA (default: None).

class garnett.shapes.**ConvexSpheropolyhedronShape**(*vertices*, *rounding_radius=0*, *color=None*)

Shape class for a convex polyhedron extended by a rounding radius.

Parameters

- **vertices** (*list*) – List of 3D vertex vectors.
- **rounding_radius** (*float*) – Rounding radius applied to the spheropolyhedron (default: 0).
- **color** (*str*) – Hexadecimal color string in format RRGGBBAA (default: None).

shape_dict

Shape as dictionary. Example:

```
>>> ConvexSpheropolyhedronShape([[0.5, 0.5, 0.5], [0.5, -0.5, -0.5],
                                 [-0.5, 0.5, -0.5], [-0.5, -0.5, 0.5]], 0.1).
↪shape_dict
{'type': 'ConvexPolyhedron', 'rounding_radius': 0.1,
 'vertices': [[0.5, 0.5, 0.5], [0.5, -0.5, -0.5],
              [-0.5, 0.5, -0.5], [-0.5, -0.5, 0.5]]}
```

class garnett.shapes.**GeneralPolyhedronShape**(*vertices*, *faces*, *color=None*, *facet_colors=None*)

Shape class for general polyhedra, such as arbitrary meshes.

Parameters

- **vertices** (*list*) – List of 3D vertex vectors.
- **faces** (*list*) – List of lists of integers representing vertex indices for each face.
- **color** (*str*) – Hexadecimal color string in format RRGGBBAA (default: None).
- **facet_colors** (*list*) – List of hexadecimal color strings in format RRGGBBAA for each facet (default: None).

shape_dict

Shape as dictionary. Example:

```
>>> GeneralPolyhedronShape([[0.5, 0.5, 0.5], [0.5, -0.5, -0.5],
                            [-0.5, 0.5, -0.5], [-0.5, -0.5, 0.5]]).shape_dict
{'type': 'Mesh',
 'vertices': [[0.5, 0.5, 0.5], [0.5, -0.5, -0.5],
              [-0.5, 0.5, -0.5], [-0.5, -0.5, 0.5]],
 'indices': [[1, 2, 3], [1, 2, 4], [1, 3, 4], [2, 3, 4]]}
```

class garnett.shapes.**EllipsoidShape**(*a*, *b*, *c*, *color=None*)

Shape class for ellipsoids of with principal axes a, b, and c.

Parameters

- **a** (*float*) – Principal axis a of the ellipsoid (radius in the x direction).
- **b** (*float*) – Principal axis b of the ellipsoid (radius in the y direction).
- **c** (*float*) – Principal axis c of the ellipsoid (radius in the z direction).
- **color** (*str*) – Hexadecimal color string in format RRGGBBAA (default: None).

shape_dict

Shape as dictionary. Example:

```
>>> EllipsoidShape(7.0, 5.0, 3.0).shape_dict
{'type': 'Ellipsoid',
'a': 7.0,
'b': 5.0,
'c': 3.0}
```

CHAPTER 6

Changelog

The **garnett** package follows semantic versioning.

6.1 Version 0.5

6.1.1 [0.5.0] – 2019-08-20

Added

- Added rowan as a dependency.
- Add GETAR file reader/writer.
- Add `shape_dict` representation to `Shape` classes.
- Add support for particle properties:
  - mass
  - charge
  - diameter
  - image
  - moment of inertia
  - angular momentum
- Add support for reading/writing shapes in GSD via HOOMD-HPMC state.
- Add universal reader/writer with format detection.
- Add orientable attribute to spheres.
- Extend list of supported shape classes:
  - ellipsoid

  - polygon
  - spheropolygon
  - convex polyhedron
  - convex spheropolyhedron

Changed

- Raise `AttributeError` if accessing a frame or trajectory property not defined in the file.
- Rename several existing shape classes.
- Improve unit test coverage.
- Revise documentation.
- Move shape definitions to separate module.

Deprecated

- Tests for Python 2 are no longer updated (Python 2 support will be dropped in the next minor release).

Removed

- Remove acceleration as supported property.
- Remove the `read_gsd_shape_data` flag from GSD reader.

6.2 Version 0.4

6.2.1 [0.4.1] – 2017-08-23

Fixed

- Fix minor bug related to QR check for 2d boxes.

6.2.2 [0.4.0] – 2017-06-26

Added

- Add readers/writers:
  - CIF reader
  - GSD writer
- Support shape definitions:
  - spheropolyhedron
  - polyunion
  - convex polyhedron union

- Add `gf2pos` script - convert to pos-file from any supported format.
- Add shape definitions to `GetarFileReader`.
- Interpret the pos-file rotation key word.

Changed

- `GetarFileReader` skips records that have a non-empty group field.
- Improve algorithm for the normalization of frames with non-standard box.
- Various documentation updates.

6.3 Version 0.3

6.3.1 [0.3.9] – 2017-01-30

Added

- The `GSDReader` now reads velocities.
- Support `PolyV` shape definitions.

Changed

- Update documentation concerning the conversion of rotations from quaternions to euler angles.

Fixed

- Fix bug related to trajectory arrays when slicing the array.

6.3.2 [0.3.8] – 2016-12-21

Fixed

- Hot fix: Negative euler angles were not read correctly in skewed boxes using the `DCDFileReader`.

6.3.3 [0.3.7] – 2016-11-07

Added

- Add the `whence` argument to the file format's seek method.

Fixed

- Fix bug in `DCDfilereader` leading to incorrect box dimensions to be read for skewed boxes. Cubic or squared boxes are not affected.

6.3.4 [0.3.6] – 2016-10-20

Fixed

- Fix quaternion to euler angle conversion example in the DCD file reader documentation.

6.3.5 [0.3.5] – 2016-09-20

Changed

- `GSDHOOMDFileReader` uses the native GSD library if installed.
- Reduced warning verbosity.

Fixed

- Fix bug that caused the `GSDHOOMDFileReader` to ignore dimensions specified in the GSD file.

6.3.6 [0.3.4] – 2016-09-08

Added

- Support velocities in HOOMD-blue XML files.
- Support `SphereUnionShape` in `PosFileReader`.

Changed

- Support Pos-Files using the keyword 'box' instead of 'boxMatrix'

Fixed

- Fix bug in `PosFileReader` which occured with non-standard pos-file in python 3.5
- Fix bug, which occured when constructing frames from raw frames using box instances instead of a box matrix.

6.3.7 [0.3.3] – 2016-07-19

Fixed

- Fix bug related to 2-dimensional systems and a box z-dimensions not equal to 1.

6.3.8 [0.3.2] – 2016-07-15

Added

- Add `trajectory.N`, `trajectory.type` and `trajectory.type_ids` as an alternative mode to access frame length and type information.

Fixed

- Fix bug in `GSDHOOMDFileReader` when not providing template frame.

6.3.9 [0.3.1] – 2016-07-08

Changed

- Update the GSD hoomd module.

6.3.10 [0.3.0] – 2016-07-06

Added

- Provide a highly optimized cythonized `DCDFileReader`.
- Allow trajectory data acess via coherent numpy arrays.
- Make snapshot creation and copying HOOMD-blue 2.0 compatible.

Changed

- Update the GSD module.
- Improve the `Box` class documentation.
- Overall improvement of the documentation.

Fixed

- Fix and optimize the pure-python `DCDFileReader`.

6.4 Version 0.2

6.4.1 [0.2.1] – 2016-07-10

Fixed

- Fix an issue with injavis pos-files causing parser errors.

6.4.2 [0.2.0] – 2016-04-28

Fixed

- Fix HOOMD-blue snapshot type issue.

6.5 Version 0.1

6.5.1 [0.1.9] – 2016-04-09

Added

- Add `GSDHoomdFileReader`.

Fixed

- Fix type issue in `HoomdBlueXMLFileReader`.

6.5.2 [0.1.8] – 2016-04-04

Added

- Add `HoomdBlueXMLFileReader`.
- Add `DCDFileReader`.
- Add `CifFileWriter`.
- Add `GetarFileReader`.

Fixed

- Fix type issue in DCD.

6.5.3 [0.1.6] – 2016-01-28

Changed

- Extend FileFormat API to increase file-like compatibility.

Fixed

- Fixed `box_matrix` calculation.

6.5.4 [0.1.5] – 2016-01-11

Changed

- Frames only loaded into memory on demand.
- Improved trajectory iteration logic.

6.5.5 No change logs prior to v0.1.5

CHAPTER 7

Developer's Guide

7.1 Trajectory reader implementation

To implement a trajectory reader, first keep in mind that a garnett trajectory is simply a sequence of frames. This means that a trajectory reader needs to generate individual instances of frames.

To implement a new reader:

1. Provide a minimal sample for your format.
2. Create a new module in *garnett* with the name *yourformatreader.py*.
3. Specialize a frame class for your format called *YourFormatFrame*.
4. Implement the *read()* method for your frame, it should return an instance of `garnett.trajectory._RawFrameData`.

```
class YourFormatFrame(trajectory.Frame):

    def read():
        """Read the frame data from the stream.

        :returns: :class:`.trajectory._RawFrameData`
        """
```

5. Define a class *YourFormatReader*, where the constructor may take optional arguments for your reader.
6. The *YourFormatReader* class needs to implement a *read()* method.

```
class YourFormatReader(object):

    def read(stream):
        """Read the trajectory from stream.

        .. code::
```

(continues on next page)

(continued from previous page)

```
            # pseudo-example
            frames = list(self.scan(stream))
            return trajectory.Trajectory(frames)

        :stream: A file-like object.
        :returns: :class:`.trajectory.Trajectory`
        """
```

7. Add your reader class to the *__all__* directive in the *garnett/reader.py* module.
8. Provide a unit test for your reader, that reads a sample and generates a trajectory object accordingly.

For an example, please see the `GSDHOOMDFileReader` implementation.

CHAPTER 8

Credits

8.1 Garnett Developers

The following people have contributed to the `garnett` package.

Carl Simon Adorf, University of Michigan

- Original Author
- Former Maintainer
- Trajectory classes
- Shapes classes
- Testing framework
- CIF file writer
- DCD file reader
- GSD file reader
- HOOMD XML file reader
- POS file reader
- POS file writer

Richmond Newman, University of Michigan

- Original Author
- POS file reader

Matthew Spellings, University of Michigan

- CIF file reader
- GETAR file reader

Julia Dshemuchadse, University of Michigan

- CIF file writer

Vyas Ramasubramani, University of Michigan

- GSD file writer

Bradley Dice, University of Michigan

- Maintainer
- GETAR file writer
- Support for additional frame properties
- Improved support for parsing and writing particle shapes

Sophie Youjung Lee, University of Michigan

- Former Maintainer
- Universal read and write functions

Luis Y. Rivera-Rivera, University of Michigan

- Maintainer

Kelly Wang, University of Michigan

- Maintainer

8.2 Source code

GSD is used to construct trajectory objects from GSD files and is available at https://github.com/glotzerlab/gsd - Used under the BSD license:

```
Copyright (c) 2016-2019 The Regents of the University of Michigan
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice,
   this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Index

M

N

O

P

R

S

T

U

V

W

X

Y