GSD Documentation

Release 2.0.0

The Regents of the University of Michigan

Feb 03, 2020

GETTING STARTED

The **GSD** file format is the native file format for HOOMD-blue. **GSD** files store trajectories of the HOOMD-blue system state in a binary file with efficient random access to frames. **GSD** allows all particle and topology properties to vary from one frame to the next. Use the **GSD** Python API to specify the initial condition for a HOOMD-blue simulation or analyze trajectory output with a script. Read a **GSD** trajectory with a visualization tool to explore the behavior of the simulation.

- GitHub Repository: **GSD** source code and issue tracker.
- HOOMD-blue: Simulation engine that reads and writes GSD files.
- hoomd-users Google Group: Ask questions to the **HOOMD-blue** community.
- freud: A powerful set of tools for analyzing trajectories.
- OVITO: The Open Visualization Tool works with GSD files.
- gsd-vmd plugin: VMD plugin to support GSD files.

CHAPTER

ONE

INSTALLATION

gsd binaries are available in the glotzerlab-software Docker/Singularity images and in packages on conda-forge and PyPI. You can also compile **gsd** from source, embed `gsd.c` in your code, or read gsd files with a pure Python reader `pygsd.py`.

1.1 Binaries

1.1.1 Anaconda package

gsd is available on conda-forge. To install, first download and install miniconda. Then add the **conda-forge** channel and install **gsd**:

```
$ conda install -c conda-forge gsd
```

1.1.2 Singularity / Docker images

See the glotzerlab-software documentation for container usage information and cluster specific instructions.

1.1.3 PyPI

Use **pip** to install **gsd**:

```
$ pip install gsd
```

1.2 Compile from source

1.2.1 Obtain the source

Download source releases directly from the web: https://glotzerlab.engin.umich.edu/downloads/gsd

```
curl -O https://glotzerlab.engin.umich.edu/downloads/gsd/gsd-v2.0.0.tar.gz
```

Or, clone using git:

```
$ git clone https://github.com/glotzerlab/gsd
```

1.2.2 Configure a virtual environment

When using a shared Python installation, create a virtual environment where you can install **gsd**:

```
$ python3 -m venv /path/to/environment --system-site-packages
```

Activate the environment before configuring and before executing **gsd** scripts:

```
$ source /path/to/environment/bin/activate
```

Note: Other types of virtual environments (such as *conda*) may work, but are not thoroughly tested.

1.2.3 Install Prerequisites

gsd requires:

- **C compiler** (tested with gcc 4.8-9.0, clang 4-9, vs2017-2019)
- **Python** >= 3.5
- **numpy** >= 1.9.3
- **Cython** >= 0.22

Additional packages may be needed:

- **pytest** >= 3.9.0 (unit tests)
- **Sphinx** (documentation)
- **IPython** (documentation)
- an internet connection (documentation)
- **CMake** (for development builds)

Install these tools with your system or virtual environment package manager. **gsd** developers have had success with `pacman` (arch linux), `apt-get` (ubuntu), Homebrew (macOS), and MacPorts (macOS):

```
$ your-package-manager install ipython python python-pytest python-numpy cmake cython␣
↪python-sphinx python-sphinx_rtd_theme
```

Typical HPC cluster environments provide **Python**, **numpy**, and **cmake** via a module system:

```
$ module load gcc python cmake
```

Note: Packages may be named differently, check your system's package list. Install any `-dev` packages as needed.

Tip: You can install numpy and other python packages into your virtual environment:

```
python3 -m pip install numpy
```

1.2.4 Install with setuptools

Use **pip** to install the python module into your virtual environment:

```
$ python3 -m pip install .
```

1.2.5 Build with CMake for development

You can assemble a functional python module in the build directory. Configure with **CMake** and compile with **make**.

```
$ mkdir build
$ cd build
$ cmake ../
$ make
```

Add the build directory path to your `PYTHONPATH` to test **gsd** or build documentation:

```
$ export PYTHONPATH=$PYTHONPATH:/path/to/build
```

1.2.6 Run tests

Run `pytest` in the source directory to execute all unit tests. This requires that the compiled python module is on the python path.

```
$ cd /path/to/gsd
$ pytest
```

1.2.7 Build user documentation

Build the user documentation with **Sphinx**. **IPython** is required to build the documentation, as is an active internet connection. First, you need to compile and install **gsd**. If you compiled with **CMake**, add **gsd** to your `PYTHONPATH` first:

```
$ export PYTHONPATH=$PYTHONPATH:/path/to/build
```

To build the documentation:

```
$ cd /path/to/gsd
$ cd doc
$ make html
$ open _build/html/index.html
```

1.2.8 Using the C library

gsd is implemented in a single C file. Copy `gsd/gsd.h` and `gsd/gsd.c` into your project.

1.2.9 Using the pure python reader

If you only need to read files, you can skip installing and just extract the module modules `gsd/pygsd.py` and `gsd/hoomd.py`. Together, these implement a pure Python reader for **gsd** and **HOOMD** files - no C compiler required.

CHAPTER

TWO

CHANGE LOG

GSD releases follow semantic versioning.

2.1 v2.x

2.1.1 v2.0.0 (not yet released)

Note

- This release introduces a new file storage format.
- GSD >= 2.0 can read and write to files created by GSD 1.x.
- Files created or upgraded by GSD >= 2.0 can not be opened by GSD < 1.x.

Added

- The `upgrade` method converts a GSD 1.0 file to a GSD 2.0 file in place.
- Support arbitrarily long chunk names (only in GSD 2.0 files).

Changed

- `gsd.fl.open` accepts `None` for `application`, `schema`, and `schema_version` when opening files for reading.
- Improve read latency when accessing files with thousands of chunk names in a frame (only for GSD 2.0 files).
- Buffer small writes to improve write performance.
- Improve performance and reduce memory usage in read/write modes ('rb+', 'wb+' and ('xb+').
- **C API**: functions return error codes from the `gsd_error` enum. v2.x integer error codes differ from v1.x, use the enum to check. For example: `if (retval == GSD_ERROR_IO)`.
- Python, Cython, and C code must follow strict style guidelines.

Removed

- `gsd.fl.create` - use `gsd.fl.open`.
- `gsd.hoomd.create` - use `gsd.hoomd.open`.
- `GSDFile` v1.0 compatibility mode - use `gsd.fl.open`.
- `hoomdxml2gsd.py`.

Fixed

- Allow more than 127 data chunk names in a single GSD file.

2.2 v1.x

2.2.1 v1.10.0 (2019-11-26)

- Improve performance of first frame write.
- Allow pickling of GSD file handles opened in read only mode.
- Removed Cython-generated code from repository. `fl.pyx` will be cythonized during installation.

2.2.2 v1.9.3 (2019-10-04)

- Fixed preprocessor directive affecting Windows builds using setup.py.
- Documentation updates

2.2.3 v1.9.2 (2019-10-01)

- Support chunk sizes larger than 2GiB

2.2.4 v1.9.1 (2019-09-23)

- Support writing chunks wider than 255 from Python.

2.2.5 v1.9.0 (2019-09-18)

- File API: Add `find_matching_chunk_names()`
- `HOOMD` schema 1.4: Add user defined logged data.
- `HOOMD` schema 1.4: Add `type_shapes` specification.
- pytest >= 3.9.0 is required to run unit tests.
- `gsd.fl.open` and `gsd.hoomd.open` accept objects implementing `os.PathLike`.
- Report an error when attempting to write a chunk that fails to allocate a name.
- Reduce virtual memory usage in `rb` and `wb` open modes.
- Additional checks for corrupt GSD files on open.
- Synchronize after expanding file index.

2.2.6 v1.8.1 (2019-08-19)

- Correctly raise `IndexError` when attempting to read frames before the first frame.
- Raise `RuntimeError` when importing `gsd` in unsupported Python versions.

2.2.7 v1.8.0 (2019-08-05)

- Slicing a HOOMDTrajectory object returns a view that can be used to directly select frames from a subset or sliced again.
- raise `IndexError` when attempting to read frames before the first frame.
- Dropped support for Python 2.

2.2.8 v1.7.0 (2019-04-30)

- Add `hpmc/sphere/orientable` to HOOMD schema.
- HOOMD schema 1.3

2.2.9 v1.6.2 (2019-04-16)

- PyPI binary wheels now support numpy>=1.9.3,<2

2.2.10 v1.6.1 (2019-03-05)

- Documentation updates

2.2.11 v1.6.0 (2018-12-20)

- The length of sliced HOOMDTrajectory objects can be determined with the built-in `len()` function.

2.2.12 v1.5.5 (2018-11-28)

- Silence numpy deprecation warnings

2.2.13 v1.5.4 (2018-10-04)

- Add `pyproject.toml` file that defines `numpy` as a proper build dependency (requires pip >= 10)
- Reorganize documentation

2.2.14 v1.5.3 (2018-05-22)

- Revert `setup.py` changes in v1.5.2 - these do not work in most circumstances.
- Include `sys/stat.h` on all architectures.

2.2.15 v1.5.2 (2018-04-04)

- Close file handle on errors in `gsd_open`.
- Always close file handle in `gsd_close`.
- `setup.py` now correctly pulls in the numpy dependency.

2.2.16 v1.5.1 (2018-02-26)

- Documentation fixes.

2.2.17 v1.5.0 (2018-01-18)

- Read and write HPMC shape state data.

2.2.18 v1.4.0 (2017-12-04)

- Support reading and writing chunks with 0 length. No schema changes are necessary to support this.

2.2.19 v1.3.0 (2017-11-17)

- Document `state` entries in the HOOMD schema.
- No changes to the gsd format or reader code in v1.3.

2.2.20 v1.2.0 (2017-02-21)

- Add `gsd.hoomd.open()` method which can create and open hoomd gsd files.
- Add `gsd.fl.open()` method which can create and open gsd files.
- The previous create/class `GSDFile` instantiation is still supported for backward compatibility.

2.2.21 v1.1.0 (2016-10-04)

- Add special pairs section pairs/ to HOOMD schema.
- HOOMD schema version is now 1.1.

2.2.22 v1.0.1 (2016-06-15)

- Fix compile error on more strict POSIX systems.

2.2.23 v1.0.0 (2016-05-24)

Initial release.

CHAPTER

THREE

USER COMMUNITY

3.1 hoomd-users mailing list

GSD primarily exists as a file format for HOOMD-blue, so please use the hoomd-users mailing list. Subscribe for release announcements, to post questions questions for advice on using the software, and discuss potential new features.

3.2 Issue tracker

File bug reports on GSD's issue tracker.

3.3 Contribute

GSD is an open source project. Contributions are accepted via pull request to GSD's github repository. Please review `CONTRIBUTING.MD` in the repository before starting development. You are encouraged to discuss your proposed contribution with the **GSD** user and developer community who can help you design your contribution to fit smoothly into the existing ecosystem.

CHAPTER

FOUR

HOOMD

gsd.hoomd provides high-level access to **HOOMD** schema **GSD** files.

View the page source to find unformatted example code.

4.1 Define a snapshot

```
In [1]: s = gsd.hoomd.Snapshot()

In [2]: s.particles.N = 4

In [3]: s.particles.types = ['A', 'B']

In [4]: s.particles.typeid = [0,0,1,1]

In [5]: s.particles.position = [[0,0,0],[1,1,1], [-1,-1,-1], [1,-1,-1]]

In [6]: s.configuration.box = [3, 3, 3, 0, 0, 0]
```

gsd.hoomd represents the state of a single frame with an instance of the class *gsd.hoomd.Snapshot*. Instantiate this class to create a system configuration. All fields default to `None` and are only written into the file if not `None` and do not match the data in the first frame, or defaults specified in the schema.

4.2 Create a hoomd gsd file

```
In [7]: gsd.hoomd.open(name='test.gsd', mode='wb')
Out[7]: <gsd.hoomd.HOOMDTrajectory at 0x7f13481897b8>
```

4.3 Append frames to a gsd file

```
In [8]: def create_frame(i):
   ...:     s = gsd.hoomd.Snapshot()
   ...:     s.configuration.step = i
   ...:     s.particles.N = 4+i
   ...:     s.particles.position = numpy.random.random(size=(4+i,3))
   ...:     return s
   ...:
```

(continues on next page)

(continued from previous page)

```
In [9]: t = gsd.hoomd.open(name='test.gsd', mode='wb')

In [10]: t.extend( (create_frame(i) for i in range(10)) )

In [11]: t.append( create_frame(10) )

In [12]: len(t)
Out[12]: 11
```

Use *gsd.hoomd.open()* to open a **GSD** file with the high level interface *gsd.hoomd.HOOMDTrajectory*. It behaves like a python `list`, with *gsd.hoomd.HOOMDTrajectory.append()* and *gsd.hoomd.HOOMDTrajectory.extend()* methods.

Note: *gsd.hoomd.HOOMDTrajectory* currently doesn't support files opened in append mode.

Tip: When using *gsd.hoomd.HOOMDTrajectory.extend()*, pass in a generator or generator expression to avoid storing the entire trajectory in memory before writing it out.

4.4 Randomly index frames

```
In [13]: t = gsd.hoomd.open(name='test.gsd', mode='rb')

In [14]: snap = t[5]

In [15]: snap.configuration.step
Out[15]: 5

In [16]: snap.particles.N
Out[16]: 9

In [17]: snap.particles.position
Out[17]:
array([[0.05551339, 0.7118579 , 0.55684733],
       [0.6320372 , 0.52013403, 0.35986692],
       [0.8490573 , 0.45080644, 0.30773836],
       [0.5693308 , 0.5857744 , 0.4568267 ],
       [0.72886753, 0.6121499 , 0.9813543 ],
       [0.5911686 , 0.26352417, 0.48863783],
       [0.417278  , 0.8825609 , 0.12694064],
       [0.9012692 , 0.30470046, 0.62210697],
       [0.61231   , 0.13405989, 0.8273897 ]], dtype=float32)
```

gsd.hoomd.HOOMDTrajectory supports random indexing of frames in the file. Indexing into a trajectory returns a *gsd.hoomd.Snapshot*.

4.5 Slicing and selection

Use the slicing operator to select individual frames or a subset of a trajectory.

```
In [18]: t = gsd.hoomd.open(name='test.gsd', mode='rb')

In [19]: for s in t[5:-2]:
   ....:     print(s.configuration.step, end=' ')
   ....:
5 6 7 8
In [20]: every_2nd_frame = t[::2]  # create a view of a trajectory subset

In [21]: for s in every_2nd_frame[:4]:
   ....:     print(s.configuration.step, end=' ')
   ....:
0 2 4 6
```

Slicing a trajectory creates a trajectory view, which can then be queried for length or sliced again. Selecting individual frames from a view works exactly like selecting individual frames from the original trajectory object.

4.6 Pure python reader

```
In [22]: f = gsd.pygsd.GSDFile(open('test.gsd', 'rb'))

In [23]: t = gsd.hoomd.HOOMDTrajectory(f);

In [24]: t[3].particles.position
Out[24]:
array([[0.5692003 , 0.35402223, 0.04927382],
       [0.03480713, 0.97779214, 0.32133165],
       [0.8875883 , 0.71083546, 0.3328835 ],
       [0.3320165 , 0.18332727, 0.1802465 ],
       [0.28748494, 0.0110937 , 0.09906646],
       [0.4513102 , 0.4319952 , 0.8737437 ],
       [0.81227154, 0.05937914, 0.79250705]], dtype=float32)
```

You can use **GSD** without needing to compile C code to read **GSD** files using *gsd.pygsd.GSDFile* in combination with *gsd.hoomd.HOOMDTrajectory*. It only supports the `rb` mode and does not read files as fast as the C implementation. It takes in a python file-like object, so it can be used with in-memory IO classes, and grid file classes that access data over the internet.

4.7 Access state data

```
In [25]: with gsd.hoomd.open(name='test2.gsd', mode='wb') as t:
   ....:     s = gsd.hoomd.Snapshot()
   ....:     s.particles.types = ['A', 'B']
   ....:     s.state['hpmc/convex_polygon/N'] = [3, 4]
   ....:     s.state['hpmc/convex_polygon/vertices'] = [[-1, -1],
   ....:                                                [1, -1],
   ....:                                                [1, 1],
   ....:                                                [-2, -2],
   ....:                                                [2, -2],
```

(continues on next page)

(continued from previous page)

```
   ....:                                                 [2, 2],
   ....:                                                 [-2, 2]]
   ....:     t.append(s)
   ....:
```

State data is stored in the `state` dictionary as numpy arrays. Place data into this dictionary directly without the 'state/' prefix and gsd will include it in the output. Shape vertices are stored in a packed format. In this example, type 'A' has 3 vertices (the first 3 in the list) and type 'B' has 4 (the next 4).

```
In [26]: with gsd.hoomd.open(name='test2.gsd', mode='rb') as t:
   ....:     s = t[0]
   ....:     print(s.state['hpmc/convex_polygon/N'])
   ....:     print(s.state['hpmc/convex_polygon/vertices'])
   ....:
[3 4]
[[-1. -1.]
 [ 1. -1.]
 [ 1.  1.]
 [-2. -2.]
 [ 2. -2.]
 [ 2.  2.]
 [-2.  2.]]
```

Access read state data in the same way.

4.8 Access logged data

```
In [27]: with gsd.hoomd.open(name='example.gsd', mode='wb') as t:
   ....:     s = gsd.hoomd.Snapshot()
   ....:     s.particles.N = 4
   ....:     s.log['particles/net_force'] = numpy.array([[-1,2,-3],
   ....:                                     [0,2,-4],
   ....:                                     [-3,2,1],
   ....:                                     [1,2,3]], dtype=numpy.float32)
   ....:     s.log['value/potential_energy'] = [1.5]
   ....:     t.append(s)
   ....:
```

Logged data is stored in the `log` dictionary as numpy arrays. Place data into this dictionary directly without the 'log/' prefix and gsd will include it in the output. Store per-particle quantities with the prefix `particles/`. Choose another prefix for other quantities.

```
In [28]: t = gsd.hoomd.open(name='example.gsd', mode='rb')

In [29]: s = t[0]

In [30]: s.log['particles/net_force']
Out[30]:
array([[-1.,  2., -3.],
       [ 0.,  2., -4.],
       [-3.,  2.,  1.],
       [ 1.,  2.,  3.]], dtype=float32)
```

(continues on next page)

(continued from previous page)

```
In [31]: s.log['value/potential_energy']
Out[31]: array([1.5])
```

Read logged data from the `log` dictionary.

Logged data must be a convertible to a numpy array of a supported type:

```
In [32]: with gsd.hoomd.open(name='example.gsd', mode='wb') as t:
   ....:     s = gsd.hoomd.Snapshot()
   ....:     s.particles.N = 4
   ....:     s.log['invalid'] = dict(a=1, b=5)
   ....:     t.append(s)
   ....:
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-32-e8566430e2ad> in <module>
      3     s.particles.N = 4
      4     s.log['invalid'] = dict(a=1, b=5)
----> 5     t.append(s)

~/checkouts/readthedocs.org/user_builds/gsd/conda/v2.0.0/lib/python3.7/site-packages/
↪gsd-2.0.0-py3.7-linux-x86_64.egg/gsd/hoomd.py in append(self, snapshot)
    720         # write log data
    721         for log, data in snapshot.log.items():
--> 722             self.file.write_chunk('log/' + log, data)
    723
    724         self.file.end_frame()

gsd/fl.pyx in gsd.fl.GSDFile.write_chunk()

ValueError: invalid type for chunk: log/invalid
```

4.9 Use multiprocessing with HOOMDTrajectory

```
In [33]: import multiprocessing as mp

In [34]: def cnt_part(args):
   ....:     t, frame = args
   ....:     return len(t[frame].particles.position)
   ....:

In [35]: with gsd.hoomd.open(name='test.gsd', mode='rb') as t:
   ....:     with mp.Pool(processes=mp.cpu_count()) as pool:
   ....:         result = pool.map(cnt_part, [(t, frame) for frame in range(len(t))])
   ....:
```

gsd.hoomd.HOOMDTrajectory and both *gsd.fl.GSDFile* and *gsd.pygsd.GSDFile* can be pickled when in read mode to allow for multiprocessing through pythons native multiprocessing library. Here `cnt_part` finds the number of particles in each frame and appends it to a list. This code would result in a list of all particle numbers throughout the trajectory file.

CHAPTER

FIVE

FILE LAYER

The file layer python module *gsd.fl* allows direct low level access to read and write **GSD** files of any schema. The **HOOMD** reader (*gsd.hoomd*) provides higher level access to **HOOMD** schema files, see *HOOMD*.

View the page source to find unformatted example code.

5.1 Open a gsd file

```
In [1]: f = gsd.fl.open(name="file.gsd",
   ...:                 mode='wb',
   ...:                 application="My application",
   ...:                 schema="My Schema",
   ...:                 schema_version=[1,0])
   ...:

In [2]: f.close()
```

Warning: Opening a gsd file with a 'w' or 'x' mode overwrites any existing file with the given name.

5.2 Write data

```
In [3]: f = gsd.fl.open(name="file.gsd",
   ...:                 mode='wb',
   ...:                 application="My application",
   ...:                 schema="My Schema",
   ...:                 schema_version=[1,0]);
   ...:

In [4]: f.write_chunk(name='chunk1', data=numpy.array([1,2,3,4], dtype=numpy.float32))

In [5]: f.write_chunk(name='chunk2', data=numpy.array([[5,6],[7,8]], dtype=numpy.
↪float32))

In [6]: f.end_frame()

In [7]: f.write_chunk(name='chunk1', data=numpy.array([9,10,11,12], dtype=numpy.
↪float32))
```

(continues on next page)

(continued from previous page)

```
In [8]: f.write_chunk(name='chunk2', data=numpy.array([[13,14],[15,16]], dtype=numpy.
↪float32))

In [9]: f.end_frame()

In [10]: f.close()
```

Call *gsd.fl.open()* to access gsd files on disk. Add any number of named data chunks to each frame in the file with *gsd.fl.GSDFile.write_chunk()*. The data must be a 1 or 2 dimensional numpy array of a simple numeric type (or a data type that will automatically convert when passed to `numpy.array(data)`. Call *gsd.fl.GSDFile.end_frame()* to end the frame and start the next one.

Note: While supported, implicit conversion to numpy arrays creates a copy of the data in memory and adds conversion overhead.

Warning: Make sure to call `end_frame()` before closing the file, or the last frame will be lost.

5.3 Read data

```
In [11]: f = gsd.fl.open(name="file.gsd",
   ....:                 mode='rb',
   ....:                 application="My application",
   ....:                 schema="My Schema",
   ....:                 schema_version=[1,0])
   ....:

In [12]: f.read_chunk(frame=0, name='chunk1')
Out[12]: array([1., 2., 3., 4.], dtype=float32)

In [13]: f.read_chunk(frame=1, name='chunk2')
Out[13]:
array([[13., 14.],
       [15., 16.]], dtype=float32)

In [14]: f.close()
```

gsd.fl.GSDFile.read_chunk() reads the named chunk at the given frame index in the file and returns it as a numpy array.

5.4 Test if a chunk exists

```
In [15]: f = gsd.fl.open(name="file.gsd",
   ....:                 mode='rb',
   ....:                 application="My application",
   ....:                 schema="My Schema",
   ....:                 schema_version=[1,0])
   ....:
```

(continues on next page)

(continued from previous page)

```
In [16]: f.chunk_exists(frame=0, name='chunk1')
Out[16]: True

In [17]: f.chunk_exists(frame=1, name='chunk2')
Out[17]: True

In [18]: f.chunk_exists(frame=2, name='chunk1')
Out[18]: False

In [19]: f.close()
```

gsd.fl.GSDFile.chunk_exists() tests to see if a chunk by the given name exists in the file at the given frame.

5.5 Discover chunk names

```
In [20]: f = gsd.fl.open(name="file.gsd",
   ....:                 mode='rb',
   ....:                 application="My application",
   ....:                 schema="My Schema",
   ....:                 schema_version=[1,0])
   ....:

In [21]: f.find_matching_chunk_names('')
Out[21]: ['chunk1', 'chunk2']

In [22]: f.find_matching_chunk_names('chunk')
Out[22]: ['chunk1', 'chunk2']

In [23]: f.find_matching_chunk_names('chunk1')
Out[23]: ['chunk1']

In [24]: f.find_matching_chunk_names('other')
Out[24]: []
```

gsd.fl.GSDFile.find_matching_chunk_names() finds all chunk names present in a GSD file that start with the given string.

5.6 Read-only access

```
In [25]: f = gsd.fl.open(name="file.gsd",
   ....:                 mode='rb',
   ....:                 application="My application",
   ....:                 schema="My Schema",
   ....:                 schema_version=[1,0])
   ....:

In [26]: if f.chunk_exists(frame=0, name='chunk1'):
   ....:     data = f.read_chunk(frame=0, name='chunk1')
   ....:
```

(continues on next page)

(continued from previous page)

```
In [27]: data
Out[27]: array([1., 2., 3., 4.], dtype=float32)

# Fails because the file is open read only
In [28]: f.write_chunk(name='error', data=numpy.array([1]))
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-28-c9aabea2641a> in <module>
----> 1 f.write_chunk(name='error', data=numpy.array([1]))

gsd/fl.pyx in gsd.fl.GSDFile.write_chunk()

gsd/fl.pyx in gsd.fl.__raise_on_error()

RuntimeError: File must be writable: file.gsd

In [29]: f.close()
```

Writes fail when a file is opened in a read only mode.

5.7 Access file metadata

```
In [30]: f = gsd.fl.open(name="file.gsd",
   ....:                 mode='rb',
   ....:                 application="My application",
   ....:                 schema="My Schema",
   ....:                 schema_version=[1,0])
   ....:

In [31]: f.name
Out[31]: 'file.gsd'

In [32]: f.mode
Out[32]: 'rb'

In [33]: f.gsd_version
Out[33]: (2, 0)

In [34]: f.application
Out[34]: 'My application'

In [35]: f.schema
Out[35]: 'My Schema'

In [36]: f.schema_version
Out[36]: (1, 0)

In [37]: f.nframes
Out[37]: 2

In [38]: f.close()
```

File metadata are available as properties.

5.8 Open a file in read/write mode

```
In [39]: f = gsd.fl.open(name="file.gsd",
   ....:                 mode='wb+',
   ....:                 application="My application",
   ....:                 schema="My Schema",
   ....:                 schema_version=[1,0])
   ....:

In [40]: f.write_chunk(name='double', data=numpy.array([1,2,3,4], dtype=numpy.
↪float64));

In [41]: f.end_frame()

In [42]: f.nframes
Out[42]: 1

In [43]: f.read_chunk(frame=0, name='double')
Out[43]: array([1., 2., 3., 4.])
```

Open a file in read/write mode to allow both reading and writing.

5.9 Write a file in append mode

```
In [44]: f = gsd.fl.open(name="file.gsd",
   ....:                 mode='ab',
   ....:                 application="My application",
   ....:                 schema="My Schema",
   ....:                 schema_version=[1,0])
   ....:

In [45]: f.write_chunk(name='int', data=numpy.array([10,20], dtype=numpy.int16));

In [46]: f.end_frame()

In [47]: f.nframes
Out[47]: 2

# Reads fail in append mode
In [48]: f.read_chunk(frame=2, name='double')
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-48-cab5b10fd02b> in <module>
----> 1 f.read_chunk(frame=2, name='double')

gsd/fl.pyx in gsd.fl.GSDFile.read_chunk()

KeyError: 'frame 2 / chunk double not found in: file.gsd'

In [49]: f.close()
```

Open a file in append mode to write additional chunks to an existing file, but prevent reading.

5.10 Use as a context manager

```
In [50]: with gsd.fl.open(name="file.gsd",
   ....:                  mode='rb',
   ....:                  application="My application",
   ....:                  schema="My Schema",
   ....:                  schema_version=[1,0]) as f:
   ....:     data = f.read_chunk(frame=0, name='double');
   ....:

In [51]: data
Out[51]: array([1., 2., 3., 4.])
```

gsd.fl.GSDFile works as a context manager for guaranteed file closure and cleanup when exceptions occur.

5.11 Store string chunks

```
In [52]: f = gsd.fl.open(name="file.gsd",
   ....:                  mode='wb+',
   ....:                  application="My application",
   ....:                  schema="My Schema",
   ....:                  schema_version=[1,0])
   ....:

In [53]: f.mode
Out[53]: 'wb+'

In [54]: s = "This is a string"

In [55]: b = numpy.array([s], dtype=numpy.dtype((bytes, len(s)+1)))

In [56]: b = b.view(dtype=numpy.int8)

In [57]: b
Out[57]:
array([ 84, 104, 105, 115,  32, 105, 115,  32,  97,  32, 115, 116, 114,
       105, 110, 103,   0], dtype=int8)

In [58]: f.write_chunk(name='string', data=b)

In [59]: f.end_frame()

In [60]: r = f.read_chunk(frame=0, name='string')

In [61]: r
Out[61]:
array([ 84, 104, 105, 115,  32, 105, 115,  32,  97,  32, 115, 116, 114,
       105, 110, 103,   0], dtype=int8)

In [62]: r = r.view(dtype=numpy.dtype((bytes, r.shape[0])));

In [63]: r[0].decode('UTF-8')
Out[63]: 'This is a string'

In [64]: f.close()
```

To store a string in a gsd file, convert it to a numpy array of bytes and store that data in the file. Decode the byte sequence to get back a string.

5.12 Truncate

```
In [65]: f = gsd.fl.open(name="file.gsd",
   ....:                 mode='ab',
   ....:                 application="My application",
   ....:                 schema="My Schema",
   ....:                 schema_version=[1,0])
   ....:

In [66]: f.nframes
Out[66]: 1

In [67]: f.schema, f.schema_version, f.application
Out[67]: ('My Schema', (1, 0), 'My application')

In [68]: f.truncate()

In [69]: f.nframes
Out[69]: 0

In [70]: f.schema, f.schema_version, f.application
Out[70]: ('My Schema', (1, 0), 'My application')
```

Truncating a gsd file removes all data chunks from it, but retains the same schema, schema version, and application name. The file is not closed during this process. This is useful when writing restart files on a Lustre file system when file open operations need to be kept to a minimum.

CHAPTER

SIX

GSD PYTHON PACKAGE

GSD provides a **Python** API intended for most users. Developers, or users not working with the Python language, may want to use the *C API*.

6.1 Submodules

6.1.1 gsd.fl module

GSD file layer API.

Low level access to gsd files. `gsd.fl` allows direct access to create, read, and write `gsd` files. The module is implemented in C and is optimized. See *File layer* for detailed example code.

- `GSDFile` - Class interface to read and write gsd files.
- `open()` - Open a gsd file.

class gsd.fl.**GSDFile**

GSD file access interface.

GSDFile implements an object oriented class interface to the GSD file layer. Use `open()` to open a GSD file and obtain a GSDFile instance. `GSDFile` can be used as a context manager.

name

Name of the open file **(read only)**.

Type str

mode

Mode of the open file **(read only)**.

Type str

gsd_version

GSD file layer version number [major, minor] **(read only)**.

Type tuple[int]

application

Name of the generating application **(read only)**.

Type str

schema

Name of the data schema **(read only)**.

Type str

schema_version
Schema version number [major, minor] **(read only)**.

Type tuple[int]

nframes
Number of frames **(read only)**.

Type int

chunk_exists (*frame*, *name*)
Test if a chunk exists.

Parameters

- **frame** (*int*) – Index of the frame to check
- **name** (*str*) – Name of the chunk

Returns True if the chunk exists in the file. False if it does not.

Return type bool

Example

```
In [1]: with gsd.fl.open(name='file.gsd', mode='wb',
   ...:                  application="My application",
   ...:                  schema="My Schema", schema_version=[1,0]) as f:
   ...:     f.write_chunk(name='chunk1',
   ...:                   data=numpy.array([1,2,3,4],
   ...:                                    dtype=numpy.float32))
   ...:     f.write_chunk(name='chunk2',
   ...:                   data=numpy.array([[5,6],[7,8]],
   ...:                                    dtype=numpy.float32))
   ...:     f.end_frame()
   ...:     f.write_chunk(name='chunk1',
   ...:                   data=numpy.array([9,10,11,12],
   ...:                                    dtype=numpy.float32))
   ...:     f.write_chunk(name='chunk2',
   ...:                   data=numpy.array([[13,14],[15,16]],
   ...:                                    dtype=numpy.float32))
   ...:     f.end_frame()
   ...:

In [2]: f = gsd.fl.open(name='file.gsd', mode='rb',
   ...:                 application="My application",
   ...:                 schema="My Schema", schema_version=[1,0])
   ...:

In [3]: f.chunk_exists(frame=0, name='chunk1')
Out[3]: True

In [4]: f.chunk_exists(frame=0, name='chunk2')
Out[4]: True

In [5]: f.chunk_exists(frame=0, name='chunk3')
Out[5]: False

In [6]: f.chunk_exists(frame=10, name='chunk1')
Out[6]: False
```

close()
Close the file.

Once closed, any other operation on the file object will result in a ValueError. *close()* may be called more than once. The file is automatically closed when garbage collected or when the context manager exits.

Example

```
In [1]: f = gsd.fl.open(name='file.gsd', mode='wb+',
   ...:                 application="My application",
   ...:                 schema="My Schema", schema_version=[1,0])
   ...:

In [2]: f.write_chunk(name='chunk1',
   ...:               data=numpy.array([1,2,3,4], dtype=numpy.float32))
   ...:

In [3]: f.end_frame()

In [4]: data = f.read_chunk(frame=0, name='chunk1')

In [5]: f.close()

# Read fails because the file is closed
In [6]: data = f.read_chunk(frame=0, name='chunk1')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-235a7eaf209c> in <module>
----> 1 data = f.read_chunk(frame=0, name='chunk1')

gsd/fl.pyx in gsd.fl.GSDFile.read_chunk()

ValueError: File is not open
```

end_frame()
Complete writing the current frame. After calling *end_frame()* future calls to *write_chunk()* will write to the **next** frame in the file.

Danger: Call *end_frame()* to complete the current frame **before** closing the file. If you fail to call *end_frame()*, the last frame will not be written to disk.

Example

```
In [1]: f = gsd.fl.open(name='file.gsd', mode='wb',
   ...:                 application="My application",
   ...:                 schema="My Schema", schema_version=[1,0])
   ...:

In [2]: f.write_chunk(name='chunk1',
   ...:               data=numpy.array([1,2,3,4], dtype=numpy.float32))
   ...:

In [3]: f.end_frame()
```

(continues on next page)

(continued from previous page)

```
In [4]: f.write_chunk(name='chunk1',
   ...:               data=numpy.array([9,10,11,12],
   ...:                                dtype=numpy.float32))
   ...:

In [5]: f.end_frame()

In [6]: f.write_chunk(name='chunk1',
   ...:               data=numpy.array([13,14],
   ...:                                dtype=numpy.float32))
   ...:

In [7]: f.end_frame()

In [8]: f.nframes
Out[8]: 3
```

find_matching_chunk_names(*match*)

Find all the chunk names in the file that start with the string *match*.

Parameters **match** (*str*) – Start of the chunk name to match

Returns Matching chunk names

Return type list[str]

Example

```
In [1]: with gsd.fl.open(name='file.gsd', mode='wb',
   ...:                  application="My application",
   ...:                  schema="My Schema", schema_version=[1,0]) as f:
   ...:     f.write_chunk(name='data/chunk1',
   ...:                   data=numpy.array([1,2,3,4],
   ...:                                    dtype=numpy.float32))
   ...:     f.write_chunk(name='data/chunk2',
   ...:                   data=numpy.array([[5,6],[7,8]],
   ...:                                    dtype=numpy.float32))
   ...:     f.write_chunk(name='input/chunk3',
   ...:                   data=numpy.array([9, 10],
   ...:                                    dtype=numpy.float32))
   ...:     f.end_frame()
   ...:     f.write_chunk(name='input/chunk4',
   ...:                   data=numpy.array([11, 12, 13, 14],
   ...:                                    dtype=numpy.float32))
   ...:     f.end_frame()
   ...:

In [2]: f = gsd.fl.open(name='file.gsd', mode='rb',
   ...:                 application="My application",
   ...:                 schema="My Schema", schema_version=[1,0])
   ...:

In [3]: f.find_matching_chunk_names('')
Out[3]: ['data/chunk1', 'data/chunk2', 'input/chunk3', 'input/chunk4']
```

(continues on next page)

(continued from previous page)

```
In [4]: f.find_matching_chunk_names('data')
Out[4]: ['data/chunk1', 'data/chunk2']

In [5]: f.find_matching_chunk_names('input')
Out[5]: ['input/chunk3', 'input/chunk4']

In [6]: f.find_matching_chunk_names('other')
Out[6]: []
```

read_chunk (*frame*, *name*)

Read a data chunk from the file and return it as a numpy array.

Parameters

- **frame** (*int*) – Index of the frame to read
- **name** (*str*) – Name of the chunk

Returns Data read from file. `type` is determined by the chunk metadata. If the data is NxM in the file and M > 1, return a 2D array. If the data is Nx1, return a 1D array.

Return type `numpy.ndarray[type, ndim=?, mode='c']`

Tip: Each call invokes a disk read and allocation of a new numpy array for storage. To avoid overhead, don't call *read_chunk()* on the same chunk repeatedly. Cache the arrays instead.

Example

```
In [1]: with gsd.fl.open(name='file.gsd', mode='wb',
   ...:                  application="My application",
   ...:                  schema="My Schema", schema_version=[1,0]) as f:
   ...:     f.write_chunk(name='chunk1',
   ...:                   data=numpy.array([1,2,3,4],
   ...:                                    dtype=numpy.float32))
   ...:     f.write_chunk(name='chunk2',
   ...:                   data=numpy.array([[5,6],[7,8]],
   ...:                                    dtype=numpy.float32))
   ...:     f.end_frame()
   ...:     f.write_chunk(name='chunk1',
   ...:                   data=numpy.array([9,10,11,12],
   ...:                                    dtype=numpy.float32))
   ...:     f.write_chunk(name='chunk2',
   ...:                   data=numpy.array([[13,14],[15,16]],
   ...:                                    dtype=numpy.float32))
   ...:     f.end_frame()
   ...:

In [2]: f = gsd.fl.open(name='file.gsd', mode='rb',
   ...:                 application="My application",
   ...:                 schema="My Schema", schema_version=[1,0])
   ...:

In [3]: f.read_chunk(frame=0, name='chunk1')
Out[3]: array([1., 2., 3., 4.], dtype=float32)
```

(continues on next page)

(continued from previous page)

```
In [4]: f.read_chunk(frame=1, name='chunk1')
Out[4]: array([ 9., 10., 11., 12.], dtype=float32)

In [5]: f.read_chunk(frame=2, name='chunk1')
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-5-f2a5b71c0390> in <module>
----> 1 f.read_chunk(frame=2, name='chunk1')

gsd/fl.pyx in gsd.fl.GSDFile.read_chunk()

KeyError: 'frame 2 / chunk chunk1 not found in: file.gsd'
```

truncate()
Truncate all data from the file. After truncation, the file has no frames and no data chunks. The application, schema, and schema version remain the same.

Example

```
In [1]: with gsd.fl.open(name='file.gsd', mode='wb',
   ...:                  application="My application",
   ...:                  schema="My Schema", schema_version=[1,0]) as f:
   ...:     for i in range(10):
   ...:         f.write_chunk(name='chunk1',
   ...:                       data=numpy.array([1,2,3,4],
   ...:                                        dtype=numpy.float32))
   ...:         f.end_frame()
   ...:

In [2]: f = gsd.fl.open(name='file.gsd', mode='ab',
   ...:                 application="My application",
   ...:                 schema="My Schema", schema_version=[1,0])
   ...:

In [3]: f.nframes
Out[3]: 10

In [4]: f.schema, f.schema_version, f.application
Out[4]: ('My Schema', (1, 0), 'My application')

In [5]: f.truncate()

In [6]: f.nframes
Out[6]: 0

In [7]: f.schema, f.schema_version, f.application
Out[7]: ('My Schema', (1, 0), 'My application')
```

upgrade()
Upgrade a GSD file to the v2 specification in place. The file must be open in a writable mode.

write_chunk(*name*, *data*)
Write a data chunk to the file. After writing all chunks in the current frame, call *end_frame()*.

Parameters

- **name** (*str*) – Name of the chunk

- **data** – Data to write into the chunk. Must be a numpy array, or array-like, with 2 or fewer dimensions.

Warning: *write_chunk()* will implicitly converts array-like and non-contiguous numpy arrays to contiguous numpy arrays with `numpy.ascontiguousarray(data)`. This may or may not produce desired data types in the output file and incurs overhead.

Example

```
In [1]: f = gsd.fl.open(name='file.gsd', mode='wb',
   ...:                 application="My application",
   ...:                 schema="My Schema", schema_version=[1,0])
   ...:

In [2]: f.write_chunk(name='float1d',
   ...:               data=numpy.array([1,2,3,4],
   ...:                                dtype=numpy.float32))
   ...:

In [3]: f.write_chunk(name='float2d',
   ...:               data=numpy.array([[13,14],[15,16],[17,19]],
   ...:                                dtype=numpy.float32))
   ...:

In [4]: f.write_chunk(name='double2d',
   ...:               data=numpy.array([[1,4],[5,6],[7,9]],
   ...:                                dtype=numpy.float64))
   ...:

In [5]: f.write_chunk(name='int1d',
   ...:               data=numpy.array([70,80,90],
   ...:                                dtype=numpy.int64))
   ...:

In [6]: f.end_frame()

In [7]: f.nframes
Out[7]: 1

In [8]: f.close()
```

`gsd.fl.`**open**(*name*, *mode*, *application*, *schema*, *schema_version*)

open() opens a GSD file and returns a *GSDFile* instance. The return value of *open()* can be used as a context manager.

Parameters

- **name** (*str*) – File name to open.
- **mode** (*str*) – File access mode.
- **application** (*str*) – Name of the application creating the file.
- **schema** (*str*) – Name of the data schema.
- **schema_version** (`list[int]`) – Schema version number [major, minor].

Valid values for mode:

mode	description
`'rb'`	Open an existing file for reading.
`'rb+'`	Open an existing file for reading and writing. *Inefficient for large files.*
`'wb'`	Open a file for writing. Creates the file if needed, or overwrites an existing file.
`'wb+'`	Open a file for reading and writing. Creates the file if needed, or overwrites an existing file.
`'xb'`	Create a gsd file exclusively and opens it for writing. Raise an `FileExistsError` exception if it already exists.
`'xb+'`	Create a gsd file exclusively and opens it for reading and writing. Raise an `FileExistsError` exception if it already exists.
`'ab'`	Open an existing file for writing. Does *not* create or overwrite existing files.

When opening a file for reading (`'r' and 'a' modes):` ``application`` and `schema_version` are ignored and may be `None`. When `schema` is not `None`, *`open()`* throws an exception if the file's schema does not match `schema`.

When opening a file for writing (`'w' or 'x'` modes): The given `application`, `schema`, and `schema_version` are saved in the file and must not be None.

Example

```
In [1]: with gsd.fl.open(name='file.gsd', mode='wb',
   ...:                  application="My application", schema="My Schema",
   ...:                  schema_version=[1,0]) as f:
   ...:     f.write_chunk(name='chunk1',
   ...:                   data=numpy.array([1,2,3,4], dtype=numpy.float32))
   ...:     f.write_chunk(name='chunk2',
   ...:                   data=numpy.array([[5,6],[7,8]],
   ...:                                    dtype=numpy.float32))
   ...:     f.end_frame()
   ...:     f.write_chunk(name='chunk1',
   ...:                   data=numpy.array([9,10,11,12],
   ...:                                    dtype=numpy.float32))
   ...:     f.write_chunk(name='chunk2',
   ...:                   data=numpy.array([[13,14],[15,16]],
   ...:                                    dtype=numpy.float32))
   ...:     f.end_frame()
   ...:

In [2]: f = gsd.fl.open(name='file.gsd', mode='rb')

In [3]: if f.chunk_exists(frame=0, name='chunk1'):
   ...:     data = f.read_chunk(frame=0, name='chunk1')
   ...:

In [4]: data
Out[4]: array([1., 2., 3., 4.], dtype=float32)
```

6.1.2 gsd.hoomd module

hoomd schema reference implementation

The main package *gsd.hoomd* is a reference implementation of the GSD schema `hoomd`. It is a simple, but high performance and memory efficient, reader and writer for the schema. See *HOOMD* for full examples.

- *open()* - Open a hoomd schema GSD file.
- *HOOMDTrajectory* - Read and write hoomd schema GSD files.
- *Snapshot* - Store the state of a single frame.
  - *ConfigurationData* - Store configuration data in a snapshot.
  - *ParticleData* - Store particle data in a snapshot.
  - *BondData* - Store topology data in a snapshot.

class gsd.hoomd.**BondData**(*M*)

Store bond data chunks.

Users should not need to instantiate this class. Use the `bonds`, `angles`, `dihedrals`, or `impropers` attribute of a *Snapshot*.

Instances resulting from file read operations will always store per bond quantities in numpy arrays of the defined types. User created snapshots can provide input data as python lists, tuples, numpy arrays of different types, etc… Such input elements will be converted to the appropriate array type by *validate()* which is called when writing a frame.

Note: *M* varies depending on the type of bond. The same python class represents all types of bonds.

Type	*M*
Bond	2
Angle	3
Dihedral	4
Improper	4

N

Number of particles in the snapshot (*bonds/N*, *angles/N*, *dihedrals/N*, *impropers/N*, *pairs/N*).

Type int

types

Names of the particle types (*bonds/types*, *angles/types*, *dihedrals/types*, *impropers/types*, *pairs/types*).

Type list[str]

typeid

N length array defining bond type ids (*bonds/typeid*, *angles/typeid*, *dihedrals/typeid*, *impropers/typeid*, *pairs/types*).

Type `numpy.ndarray` or `array_like` [uint32, ndim=1, mode='c']

group

NxM array defining tags in the particle bonds (*bonds/group*, *angles/group*, *dihedrals/group*, *impropers/group*, *pairs/group*).

Type numpy.ndarray or array_like [uint32, ndim=2, mode='c']

validate()
Validate all attributes.

First, convert every per bond attribute to a numpy array of the proper type. Then validate that all attributes have the correct dimensions.

Ignore any attributes that are None.

> **Warning:** Per bond attributes that are not contiguous numpy arrays will be replaced with contiguous numpy arrays of the appropriate type.

class gsd.hoomd.**ConfigurationData**
Store configuration data.

Users should not need to instantiate this class. Use the configuration attribute of a *Snapshot*.

step
Time step of this frame (*configuration/step*).

Type int

dimensions
Number of dimensions (*configuration/dimensions*).

Type int

box
Box dimensions (*configuration/box*) [lx, ly, lz, xy, xz, yz].

Type numpy.ndarray or array_like [float, ndim=1, mode='c']

validate()
Validate all attributes.

First, convert every array attribute to a numpy array of the proper type. Then validate that all attributes have the correct dimensions.

Ignore any attributes that are None.

> **Warning:** Array attributes that are not contiguous numpy arrays will be replaced with contiguous numpy arrays of the appropriate type.

class gsd.hoomd.**ConstraintData**
Store constraint data chunks.

Users should not need to instantiate this class. Use the constraints, attribute of a *Snapshot*.

Instances resulting from file read operations will always store per constraint quantities in numpy arrays of the defined types. User created snapshots can provide input data as python lists, tuples, numpy arrays of different types, etc... Such input elements will be converted to the appropriate array type by *validate()* which is called when writing a frame.

N
Number of particles in the snapshot (*constraints/N*).

Type int

value
N length array defining constraint lengths (*constraints/value*).

Type numpy.ndarray or array_like [float32, ndim=1, mode='c']

group
Nx2 array defining tags in the particle constraints (*constraints/group*).

Type numpy.ndarray or array_like [int32, ndim=2, mode='c']

validate()
Validate all attributes.

First, convert every per constraint attribute to a numpy array of the proper type. Then validate that all attributes have the correct dimensions.

Ignore any attributes that are None.

Warning: Per bond attributes that are not contiguous numpy arrays will be replaced with contiguous numpy arrays of the appropriate type.

class gsd.hoomd.**HOOMDTrajectory**(*file*)
Read and write hoomd gsd files.

Parameters **file** (*gsd.fl.GSDFile*) – File to access.

Open hoomd GSD files with *open()*.

append(*snapshot*)
Append a snapshot to a hoomd gsd file.

Parameters **snapshot** (*Snapshot*) – Snapshot to append.

Write the given snapshot to the file at the current frame and increase the frame counter. Do not attempt to write any fields that are None. For all non-None fields, scan them and see if they match the initial frame or the default value. If the given data differs, write it out to the frame. If it is the same, do not write it out as it can be instantiated either from the value at the initial frame or the default value.

extend(*iterable*)
Append each item of the iterable to the file.

Parameters **iterable** – An iterable object the provides *Snapshot* instances. This could be another HOOMDTrajectory, a generator that modifies snapshots, or a simple list of snapshots.

read_frame(*idx*)
Read the frame at the given index from the file.

Parameters **idx** (*int*) – Frame index to read.

Returns *Snapshot* with the frame data

Replace any data chunks not present in the given frame with either data from frame 0, or initialize from default values if not in frame 0. Cache frame 0 data to avoid file read overhead. Return any default data as non-writable numpy arrays.

truncate()
Remove all frames from the file.

class gsd.hoomd.**ParticleData**
Store particle data chunks.

Users should not need to instantiate this class. Use the particles attribute of a *Snapshot*.

Instances resulting from file read operations will always store per particle quantities in numpy arrays of the defined types. User created snapshots can provide input data as python lists, tuples, numpy arrays of different types, etc... Such input elements will be converted to the appropriate array type by *validate()* which is called when writing a frame.

N
Number of particles in the snapshot (*particles/N*).

Type int

types
Names of the particle types (*particles/types*).

Type list[str]

position
Nx3 array defining particle position (*particles/position*).

Type numpy.ndarray or array_like [float, ndim=2, mode='c']

orientation
Nx4 array defining particle position (*particles/orientation*).

Type numpy.ndarray or array_like [float, ndim=2, mode='c']

typeid
N length array defining particle type ids (*particles/typeid*).

Type numpy.ndarray or array_like [uint32, ndim=1, mode='c']

mass
N length array defining particle masses (*particles/mass*).

Type numpy.ndarray or array_like [float, ndim=1, mode='c']

charge
N length array defining particle charges (*particles/charge*).

Type numpy.ndarray or array_like [float, ndim=1, mode='c']

diameter
N length array defining particle diameters (*particles/diameter*).

Type numpy.ndarray or array_like [float, ndim=1, mode='c']

body
N length array defining particle bodies (*particles/body*).

Type numpy.ndarray or array_like [int32, ndim=1, mode='c']

moment_inertia (`numpy.ndarray` or `array_like`
[float, ndim=2, mode='c']): Nx3 array defining particle moments of inertia (*particles/moment_inertia*).

velocity
Nx3 array defining particle velocities (*particles/velocity*).

Type numpy.ndarray or array_like [float, ndim=2, mode='c']

angmom
Nx4 array defining particle angular momenta (*particles/angmom*).

Type numpy.ndarray or array_like [float, ndim=2, mode='c']

image
Nx3 array defining particle images (*particles/image*).

Type numpy.ndarray or array_like [int32, ndim=2, mode='c']

type_shapes
Shape specifications for visualizing particle types (*particles/type_shapes*).

Type list[dict]

validate()
Validate all attributes.

First, convert every per particle attribute to a numpy array of the proper type. Then validate that all attributes have the correct dimensions.

Ignore any attributes that are None.

> **Warning:** Per particle attributes that are not contiguous numpy arrays will be replaced with contiguous numpy arrays of the appropriate type.

class gsd.hoomd.**Snapshot**
Top level snapshot container.

configuration
Configuration data.

Type *ConfigurationData*

particles
Particle data snapshot.

Type *ParticleData*

bonds
Bond data snapshot.

Type *BondData*

angles
Angle data snapshot.

Type *BondData*

dihedrals
Dihedral data snapshot.

Type *BondData*

impropers
Improper data snapshot.

Type *BondData*

pairs
Special pair interactions snapshot

Type *BondData*

state
Dictionary containing state data

Type dict

log
Dictionary containing logged data (values must be numpy.ndarray or array_like)

Type dict

See the HOOMD schema specification for details on entries in the state dictionary. Entries in this dict are the chunk name without the state prefix. For example, *state/hpmc/sphere/radius* is stored in the dictionary entry `state['hpmc/sphere/radius']`.

validate()
Validate all contained snapshot data.

gsd.hoomd.**open**(*name*, *mode='rb'*)
Open a hoomd schema GSD file.

The return value of *open()* can be used as a context manager.

Parameters

- **name** (*str*) – File name to open.
- **mode** (*str*) – File open mode.

Returns An *HOOMDTrajectory* instance that accesses the file *name* with the given mode.

Valid values for mode:

mode	description
'rb'	Open an existing file for reading.
'rb+'	Open an existing file for reading and writing. *Inefficient for large files.*
'wb'	Open a file for writing. Creates the file if needed, or overwrites an existing file.
'wb+'	Open a file for reading and writing. Creates the file if needed, or overwrites an existing file.
'xb'	Create a gsd file exclusively and opens it for writing. Raise an FileExistsError exception if it already exists.
'xb+'	Create a gsd file exclusively and opens it for reading and writing. Raise an FileExistsError exception if it already exists.
'ab'	Open an existing file for writing. Does *not* create or overwrite existing files.

6.1.3 gsd.pygsd module

GSD reader written in pure python

pygsd.py is a pure python implementation of a GSD reader. If your analysis tool is written in python and you want to embed a GSD reader without requiring C code compilation, then use the following python files from the gsd/ directory to make a pure python reader. It is not as high performance as the C reader, but is reasonable for files up to a few thousand frames.

- gsd/
  - __init__.py
  - pygsd.py
  - hoomd.py

The reader reads from file-like python objects, which may be useful for reading from in memory buffers, and in-database grid files, For regular files on the filesystem, and for writing gsd files, use *gsd.fl*.

The *GSDFile* in this module can be used with the *gsd.hoomd.HOOMDTrajectory* hoomd reader:

```
>>> with gsd.pygsd.GSDFile('test.gsd', 'rb') as f:
...     t = gsd.hoomd.HOOMDTrajectory(f)
...     pos = t[0].particles.position
```

class gsd.pygsd.**GSDFile**(*file*)

GSD file access interface. Implemented in pure python and accepts any python file-like object.

Parameters **file** – File-like object to read.

GSDFile implements an object oriented class interface to the GSD file layer. Use it to open an existing file in a **read-only** mode. For read-write access to files, use the full featured C implementation in *gsd.fl*. Otherwise, this implementation has all the same methods and the two classes can be used interchangeably.

Examples

Open a file in **read-only** mode:

```
f = GSDFile(open('file.gsd', mode='rb'))
if f.chunk_exists(frame=0, name='chunk'):
    data = f.read_chunk(frame=0, name='chunk')
```

Access file **metadata**:

```
f = GSDFile(open('file.gsd', mode='rb'))
print(f.name, f.mode, f.gsd_version)
print(f.application, f.schema, f.schema_version)
print(f.nframes)
```

Use as a **context manager**:

```
with GSDFile(open('file.gsd', mode='rb')) as f:
    data = f.read_chunk(frame=0, name='chunk')
```

file
File-like object opened **(read only)**.

name
file.name **(read only)**.

Type str

mode
Mode of the open file **(read only)**.

Type str

gsd_version
GSD file layer version number [major, minor] **(read only)**.

Type tuple[int]

application
Name of the generating application **(read only)**.

Type str

schema
Name of the data schema **(read only)**.

Type str

schema_version
Schema version number [major, minor] **(read only)**.

Type tuple[int]

nframes
Number of frames **(read only)**.

Type int

chunk_exists(*frame*, *name*)
Test if a chunk exists.

Parameters

- **frame** (*int*) – Index of the frame to check
- **name** (*str*) – Name of the chunk

Returns True if the chunk exists in the file. False if it does not.

Return type bool

Example

Handle non-existent chunks:

```
with GSDFile(open('file.gsd', mode='rb')) as f:
    if f.chunk_exists(frame=0, name='chunk'):
        return f.read_chunk(frame=0, name='chunk')
    else:
        return None
```

close()
Close the file.

Once closed, any other operation on the file object will result in a `ValueError`. *close()* may be called more than once. The file is automatically closed when garbage collected or when the context manager exits.

find_matching_chunk_names(*match*)
Find all the chunk names in the file that start with the string *match*.

Parameters **match** (*str*) – Start of the chunk name to match

Returns Matching chunk names

Return type list[str]

read_chunk(*frame*, *name*)
Read a data chunk from the file and return it as a numpy array.

Parameters

- **frame** (*int*) – Index of the frame to read
- **name** (*str*) – Name of the chunk

Returns

Data read from file. `type` is determined by the chunk metadata. If the data is NxM in the file and M > 1, return a 2D array. If the data is Nx1, return a 1D array.

Return type `numpy.ndarray[type, ndim=?, mode='c']`

Examples

Read a 1D array:

```
with GSDFile(name=filename, mode='rb') as f:
    data = f.read_chunk(frame=0, name='chunk1d')
    # data.shape == [N]
```

Read a 2D array:

```
with GSDFile(name=filename, mode='rb') as f:
    data = f.read_chunk(frame=0, name='chunk2d')
    # data.shape == [N,M]
```

Read multiple frames:

```
with GSDFile(name=filename, mode='rb') as f:
    data0 = f.read_chunk(frame=0, name='chunk')
    data1 = f.read_chunk(frame=1, name='chunk')
    data2 = f.read_chunk(frame=2, name='chunk')
    data3 = f.read_chunk(frame=3, name='chunk')
```

Tip: Each call invokes a disk read and allocation of a new numpy array for storage. To avoid overhead, don't call *read_chunk()* on the same chunk repeatedly. Cache the arrays instead.

6.2 Package contents

The GSD main module

The main package *gsd* is the root package. It holds submodules and does not import them. Users import the modules they need into their python script:

```
import gsd.fl
f = gsd.fl.GSDFile('filename', 'rb');
```

gsd.**__version__**
GSD software version number. This is the version number of the software package as a whole, not the file layer version it reads/writes.

Type str

6.3 Logging

All python modules in **GSD** use the python standard library module `logging` to log events. Use this module to control the verbosity and output destination:

```
import logging
logging.basicConfig(level=logging.INFO)
```

See also:

Module `logging` Documentation of the `logging` standard module.

CHAPTER

SEVEN

C API

The GSD C API consists of a single header and source file. Developers can drop the implementation into any package that needs it.

7.1 Functions

int **gsd_create**(const char **fname*, const char **application*, const char **schema*, *uint32_t schema_version*)

Create an empty gsd file with the given name. Overwrite any existing file at that location. The generated gsd file is not opened. Call *gsd_open()* to open it for writing.

Parameters

- **fname** – File name.
- **application** – Generating application name (truncated to 63 chars).
- **schema** – Schema name for data to be written in this GSD file (truncated to 63 chars).
- **schema_version** – Version of the scheme data to be written (make with *gsd_make_version()*).

Returns

- GSD_SUCCESS (0) on success. Negative value on failure:
- GSD_ERROR_IO: IO error (check errno).

int **gsd_create_and_open**(struct *gsd_handle* handle*, const char **fname*, const char **application*, const char **schema*, *uint32_t schema_version*, *gsd_open_flag flags*, int *exclusive_create*)

Create an empty gsd file with the given name. Overwrite any existing file at that location. Open the generated gsd file in *handle*.

Parameters

- **handle** – Handle to open.
- **fname** – File name.
- **application** – Generating application name (truncated to 63 chars).
- **schema** – Schema name for data to be written in this GSD file (truncated to 63 chars).
- **schema_version** – Version of the scheme data to be written (make with *gsd_make_version()*).
- **flags** – Either `GSD_OPEN_READWRITE`, or `GSD_OPEN_APPEND`.
- **exclusive_create** – Set to non-zero to force exclusive creation of the file.

Returns

- GSD_SUCCESS (0) on success. Negative value on failure:
- GSD_ERROR_IO: IO error (check errno).
- GSD_ERROR_NOT_A_GSD_FILE: Not a GSD file.
- GSD_ERROR_INVALID_GSD_FILE_VERSION: Invalid GSD file version.
- GSD_ERROR_FILE_CORRUPT: Corrupt file.
- GSD_ERROR_MEMORY_ALLOCATION_FAILED: Unable to allocate memory.

int **gsd_open**(struct *gsd_handle** *handle*, const char **fname*, *gsd_open_flag flags*)

Open a GSD file and populates the handle for use by later API calls.

Parameters

- **handle** – Handle to open.
- **fname** – File name to open.
- **flags** – Either `GSD_OPEN_READWRITE`, `GSD_OPEN_READONLY`, or `GSD_OPEN_APPEND`.

Returns

- GSD_SUCCESS (0) on success. Negative value on failure:
- GSD_ERROR_IO: IO error (check errno).
- GSD_ERROR_NOT_A_GSD_FILE: Not a GSD file.
- GSD_ERROR_INVALID_GSD_FILE_VERSION: Invalid GSD file version.
- GSD_ERROR_FILE_CORRUPT: Corrupt file.
- GSD_ERROR_MEMORY_ALLOCATION_FAILED: Unable to allocate memory.

int **gsd_truncate**(*gsd_handle** *handle*)

Truncate a GSD file.

After truncating, a file will have no frames and no data chunks. The file size will be that of a newly created gsd file. The application, schema, and schema version metadata will be kept. Truncate does not close and reopen the file, so it is suitable for writing restart files on Lustre file systems without any metadata access.

Parameters

- **handle** – Open GSD file to truncate.

Returns

- GSD_SUCCESS (0) on success. Negative value on failure:
- GSD_ERROR_IO: IO error (check errno).
- GSD_ERROR_NOT_A_GSD_FILE: Not a GSD file.
- GSD_ERROR_INVALID_GSD_FILE_VERSION: Invalid GSD file version.
- GSD_ERROR_FILE_CORRUPT: Corrupt file.
- GSD_ERROR_MEMORY_ALLOCATION_FAILED: Unable to allocate memory.

int **gsd_close**(*gsd_handle** *handle*)

Close a GSD file.

Parameters

- **handle** – GSD file to close.

Warning: Ensure that all *gsd_write_chunk()* calls are committed with *gsd_end_frame()* before closing the file.

Returns

- GSD_SUCCESS (0) on success. Negative value on failure:
- GSD_ERROR_IO: IO error (check errno).
- GSD_ERROR_INVALID_ARGUMENT: *handle* is NULL.

int **gsd_end_frame** (*gsd_handle* handle*)

Commit the current frame and increment the frame counter.

Parameters

- **handle** – Handle to an open GSD file.

Returns

- GSD_SUCCESS (0) on success. Negative value on failure:
- GSD_ERROR_IO: IO error (check errno).
- GSD_ERROR_INVALID_ARGUMENT: *handle* is NULL.
- GSD_ERROR_FILE_MUST_BE_WRITABLE: The file was opened read-only.
- GSD_ERROR_MEMORY_ALLOCATION_FAILED: Unable to allocate memory.

int **gsd_write_chunk** (struct *gsd_handle* handle*, const char **name*, *gsd_type type*, *uint64_t N*, *uint32_t M*, *uint8_t flags*, const void **data*)

Write a data chunk to the current frame. The chunk name must be unique within each frame. The given data chunk is written to the end of the file and its location is updated in the in-memory index. The data pointer must be allocated and contain at least contains at least `N * M * gsd_sizeof_type(type)` bytes.

Parameters

- **handle** – Handle to an open GSD file.
- **name** – Name of the data chunk.
- **type** – type ID that identifies the type of data in *data*.
- **N** – Number of rows in the data.
- **M** – Number of columns in the data.
- **flags** – Unused, set to 0.
- **data** – Data buffer.

Note: If the GSD file is version 1.0, the chunk name is truncated to 63 bytes. GSD version 2.0 files support arbitrarily long names.

Returns

- GSD_SUCCESS (0) on success. Negative value on failure:
- GSD_ERROR_IO: IO error (check errno).

- GSD_ERROR_INVALID_ARGUMENT: *handle* is NULL, $N == 0$, $M == 0$, *type* is invalid, or *flags* != 0.
- GSD_ERROR_FILE_MUST_BE_WRITABLE: The file was opened read*only.
- GSD_ERROR_NAMELIST_FULL: The file cannot store any additional unique chunk names.
- GSD_ERROR_MEMORY_ALLOCATION_FAILED: failed to allocate memory.

const struct *gsd_index_entry_t** **gsd_find_chunk**(struct *gsd_handle** *handle*, *uint64_t* *frame*, const char **name*)

Find a chunk in the GSD file. The found entry contains size and type metadata and can be passed to `gsd_read_chunk()` to read the data.

Parameters

- **handle** – Handle to an open GSD file.
- **frame** – Frame to look for chunk.
- **name** – Name of the chunk to find.

Returns A pointer to the found chunk, or NULL if not found.

int **gsd_read_chunk**(*gsd_handle** *handle*, void* *data*, const *gsd_index_entry_t** *chunk*)

Read a chunk from the GSD file. The index entry must first be found by `gsd_find_chunk()`. `data` must point to an allocated buffer with at least `N * M * gsd_sizeof_type(type)` bytes.

Parameters

- **handle** – Handle to an open GSD file.
- **data** – Data buffer to read into.
- **chunk** – Chunk to read.

Returns

0 on success

- GSD_SUCCESS (0) on success. Negative value on failure:
- GSD_ERROR_IO: IO error (check errno).
- GSD_ERROR_INVALID_ARGUMENT: *handle* is NULL, *data* is NULL, or *chunk* is NULL.
- GSD_ERROR_FILE_MUST_BE_READABLE: The file was opened in append mode.
- GSD_ERROR_FILE_CORRUPT: The GSD file is corrupt.

uint64_t **gsd_get_nframes**(*gsd_handle** *handle*)

Get the number of frames in the GSD file.

Parameters

- **handle** – Handle to an open GSD file.

Returns The number of frames in the file, or 0 on error.

size_t **gsd_sizeof_type**(*gsd_type* *type*)

Query size of a GSD type ID.

Parameters

- **type** – Type ID to query

Returns Size of the given type, or 0 for an unknown type ID.

uint32_t **gsd_make_version** (unsigned int *major*, unsigned int *minor*)
Specify a version number.

Parameters

- **major** – major version.
- **minor** – minor version.

Returns a packed version number aaaa.bbbb suitable for storing in a gsd file version entry.

const char ***gsd_find_matching_chunk_name** (struct *gsd_handle** *handle*, const char* *match*, const char **prev*)
Search for chunk names in a gsd file.

Parameters

- **handle** – Handle to an open GSD file.
- **match** – String to match.
- **prev** – Search starting point.

To find the first matching chunk name, pass `NULL` for `prev`. Pass in the previous found string to find the next after that, and so on. Chunk names match if they *begin* with the string in `match`. Chunk names returned by this function may be present in at least one frame.

Returns Pointer to a string, `NULL` if no more matching chunks are found found, or `NULL` if `prev` is invalid.

int **gsd_upgrade** (*gsd_handle** *handle*)
Upgrade a GSD file to the latest specification.

Parameters

- **handle** – Handle to an open GSD file.

Returns

0 on success

- GSD_SUCCESS (0) on success. Negative value on failure:
- GSD_ERROR_IO: IO error (check errno).
- GSD_ERROR_INVALID_ARGUMENT: *handle* is NULL, *data* is NULL, or *chunk* is NULL.
- GSD_ERROR_FILE_MUST_BE_WRITEABLE: The file was opened in the read only mode.

7.2 Constants

7.2.1 Data types

gsd_type **GSD_TYPE_UINT8**
Type ID: 8-bit unsigned integer.

gsd_type **GSD_TYPE_UINT16**
Type ID: 16-bit unsigned integer.

gsd_type **GSD_TYPE_UINT32**
Type ID: 32-bit unsigned integer.

gsd_type **GSD_TYPE_UINT64**
Type ID: 64-bit unsigned integer.

gsd_type **GSD_TYPE_INT8**
Type ID: 8-bit signed integer.

gsd_type **GSD_TYPE_INT16**
Type ID: 16-bit signed integer.

gsd_type **GSD_TYPE_INT32**
Type ID: 32-bit signed integer.

gsd_type **GSD_TYPE_INT64**
Type ID: 64-bit signed integer.

gsd_type **GSD_TYPE_FLOAT**
Type ID: 32-bit single precision floating point.

gsd_type **GSD_TYPE_DOUBLE**
Type ID: 64-bit double precision floating point.

7.2.2 Open flags

gsd_open_flag **GSD_OPEN_READWRITE**
Open file in **read/write** mode.

gsd_open_flag **GSD_OPEN_READONLY**
Open file in **read only** mode.

gsd_open_flag **GSD_OPEN_APPEND**
Open file in **append only** mode.

7.2.3 Error values

gsd_error **GSD_SUCCESS**
Success.

gsd_error **GSD_ERROR_IO**
IO error. Check `errno` for details.

gsd_error **GSD_ERROR_INVALID_ARGUMENT**
Invalid argument passed to function.

gsd_error **GSD_ERROR_NOT_A_GSD_FILE**
The file is not a GSD file.

gsd_error **GSD_ERROR_INVALID_GSD_FILE_VERSION**
The GSD file version cannot be read.

gsd_error **GSD_ERROR_FILE_CORRUPT**
The GSD file is corrupt.

gsd_error **GSD_ERROR_MEMORY_ALLOCATION_FAILED**
GSD failed to allocated memory.

gsd_error **GSD_ERROR_NAMELIST_FULL**
The GSD file cannot store any additional unique data chunk names.

gsd_error **GSD_ERROR_FILE_MUST_BE_WRITABLE**
This API call requires that the GSD file opened in with the mode GSD_OPEN_APPEND or GSD_OPEN_READWRITE.

gsd_error **GSD_ERROR_FILE_MUST_BE_READABLE**
This API call requires that the GSD file opened the mode GSD_OPEN_READ or GSD_OPEN_READWRITE.

7.3 Data structures

gsd_handle
Handle to an open GSD file. All members are **read-only**. Only public members are documented here.

gsd_header_t **header**
File header. Use this field to access the header of the GSD file.

int64_t **file_size**
Size of the open file in bytes.

gsd_open_flag **open_flags**
Flags used to open the file.

gsd_header_t
GSD file header. Access version, application, and schema information.

uint32_t **gsd_version**
GSD file format version from *gsd_make_version()*

char application[64]
Name of the application that generated this file.

char schema[64]
Name of data schema.

uint32_t **schema_version**
Schema version from *gsd_make_version()*.

gsd_index_entry_t
Entry for a single data chunk in the GSD file.

uint64_t **frame**
Frame index of the chunk.

uint64_t **N**
Number of rows in the chunk data.

uint8_t **M**
Number of columns in the chunk.

uint8_t **type**
Data type of the chunk. See *Data types*.

gsd_open_flag
Enum defining the file open flag. Valid values are `GSD_OPEN_READWRITE`, `GSD_OPEN_READONLY`, and `GSD_OPEN_APPEND`.

gsd_type
Enum defining the file type of the GSD data chunk.

gsd_error
Enum defining the possible error return values.

uint8_t
: 8-bit unsigned integer (defined by C compiler).

uint32_t
: 32-bit unsigned integer (defined by C compiler).

uint64_t
: 64-bit unsigned integer (defined by C compiler).

int64_t
: 64-bit signed integer (defined by C compiler).

CHAPTER

EIGHT

SPECIFICATION

8.1 HOOMD Schema

HOOMD-blue supports a wide variety of per particle attributes and properties. Particles, bonds, and types can be dynamically added and removed during simulation runs. The `hoomd` schema can handle all of these situations in a reasonably space efficient and high performance manner. It is also backwards compatible with previous versions of itself, as we only add new additional data chunks in new versions and do not change the interpretation of the existing data chunks. Any newer reader will initialize new data chunks with default values when they are not present in an older version file.

Schema name `hoomd`

Schema version 1.4

8.1.1 Use-cases

The GSD schema `hoomd` provides:

1. Every frame of GSD output is viable for restart from `init.read_gsd`
2. No need for a separate topology file - everything is in one `.gsd` file.
3. Support varying numbers of particles, bonds, etc...
4. Support varying attributes (type, mass, etc...)
5. Support orientation, angular momentum, and other fields that DCD cannot.
6. Simple interface for dump - limited number of options that produce valid files
7. Binary format on disk
8. High performance file read and write
9. Support logging computed quantities

8.1.2 Data chunks

Each frame the `hoomd` schema may contain one or more data chunks. The layout and names of the chunks closely match that of the binary snapshot API in HOOMD-blue itself (at least at the time of inception). Data chunks are organized in categories. These categories have no meaning in the `hoomd` schema specification, and are simply an organizational tool. Some file writers may implement options that act on categories (i.e. write **attributes** out to every frame, or just frame 0).

Values are well defined for all fields at all frames. When a data chunk is present in frame *i*, it defines the values for the frame. When it is not present, the data chunk of the same name at frame 0 defines the values for frame *i* (when *N* is equal between the frames). If the data chunk is not present in frame 0, or *N* differs between frames, values are assumed default. Default values allow files sizes to remain small. For example, a simulation with point particles where orientation is always (1,0,0,0) would not write any orientation chunk to the file.

N may be zero. When *N* is zero, an index entry may be written for a data chunk with no actual data written to the file for that chunk.

Name	Category	Type	Size	Default	Units
Configuration					
configuration/step		uint64	1x1	0	number
configuration/dimensions		uint8	1x1	3	number
configuration/box		float	6x1		*varies*
Particle data					
particles/N	attribute	uint32	1x1	0	number
particles/types	attribute	int8	NTxM	['A']	UTF-8
particles/typeid	attribute	uint32	Nx1	0	number
particles/type_shapes	attribute	int8	NTxM		UTF-8
particles/mass	attribute	float	Nx1	1.0	mass
particles/charge	attribute	float	Nx1	0.0	charge
particles/diameter	attribute	float	Nx1	1.0	length
particles/body	attribute	int32	Nx1	-1	number
particles/moment_inertia	attribute	float	Nx3	0,0,0	mass * length^2
particles/position	property	float	Nx3	0,0,0	length
particles/orientation	property	float	Nx4	1,0,0,0	unit quaternion
particles/velocity	momentum	float	Nx3	0,0,0	length/time
particles/angmom	momentum	float	Nx4	0,0,0,0	quaternion
particles/image	momentum	int32	Nx3	0,0,0	number
Bond data					
bonds/N	topology	uint32	1x1	0	number
bonds/types	topology	int8	NTxM		UTF-8
bonds/typeid	topology	uint32	Nx1	0	number
bonds/group	topology	uint32	Nx2	0,0	number
Angle data					
angles/N	topology	uint32	1x1	0	number
angles/types	topology	int8	NTxM		UTF-8
angles/typeid	topology	uint32	Nx1	0	number
angles/group	topology	uint32	Nx3	0,0,0	number
Dihedral data					
dihedrals/N	topology	uint32	1x1	0	number
dihedrals/types	topology	int8	NTxM		UTF-8
dihedrals/typeid	topology	uint32	Nx1	0	number
dihedrals/group	topology	uint32	Nx4	0,0,0,0	number

Continued on next page

Table 1 – continued from previous page

Name	Category	Type	Size	Default	Units
Improper data					
impropers/N	topology	uint32	1x1	0	number
impropers/types	topology	int8	NTxM		UTF-8
impropers/typeid	topology	uint32	Nx1	0	number
impropers/group	topology	uint32	Nx4	0,0,0,0	number
Constraint data					
constraints/N	topology	uint32	1x1	0	number
constraints/value	topology	float	Nx1	0	length
constraints/group	topology	uint32	Nx2	0,0	number
Special pairs data					
pairs/N	topology	uint32	1x1	0	number
pairs/types	topology	int8	NTxM		utf-8
pairs/typeid	topology	uint32	Nx1	0	number
pairs/group	topology	uint32	Nx2	0,0	number

8.1.3 Configuration

`configuration/step`

Type uint64

Size 1x1

Default 0

Units number

Simulation time step.

`configuration/dimensions`

Type uint8

Size 1x1

Default 3

Units number

Number of dimensions in the simulation. Must be 2 or 3.

`configuration/box`

Type float

Size 6x1

Default [1,1,1,0,0,0]

Units *varies*

Simulation box. Each array element defines a different box property. See the hoomd documentation for a full description on how these box parameters map to a triclinic geometry.

- `box[0:3]`: (l_x, l_y, l_z) the box length in each direction, in length units
- `box[3:]`: (xy, xz, yz) the tilt factors, unitless values

8.1.4 Particle data

Within a single frame, the number of particles *N* and *NT* are fixed for all chunks. *N* and *NT* may vary from one frame to the next. All values are stored in hoomd native units.

Attributes

particles/N

Type uint32

Size 1x1

Default 0

Units number

Define *N*, the number of particles, for all data chunks `particles/*`.

particles/types

Type int8

Size NTxM

Default ['A']

Units UTF-8

Implicitly define *NT*, the number of particle types, for all data chunks `particles/*`. *M* must be large enough to accommodate each type name as a null terminated UTF-8 character string. Row *i* of the 2D matrix is the type name for particle type *i*.

particles/typeid

Type uint32

Size Nx1

Default 0

Units number

Store the type id of each particle. All id's must be less than *NT*. A particle with type *id* has a type name matching the corresponding row in *particles/types*.

particles/type_shapes

Type int8

Size NTxM

Default *empty*

Units UTF-8

Store a per-type shape definition for visualization. A dictionary is stored for each of the *NT* types, corresponding to a shape for visualization of that type. *M* must be large enough to accommodate the shape definition as a null-terminated UTF-8 JSON-encoded string. See: *Shape Visualization* for examples.

particles/mass

Type float (32-bit)

Size Nx1

Default 1.0

Units mass

Store the mass of each particle.

particles/charge

Type float (32-bit)

Size Nx1

Default 0.0

Units charge

Store the charge of each particle.

particles/diameter

Type float (32-bit)

Size Nx1

Default 1.0

Units length

Store the diameter of each particle.

particles/body

Type int32

Size Nx1

Default -1

Units number

Store the composite body associated with each particle. The value -1 indicates no body. The body field may be left out of input files, as hoomd will create the needed constituent particles.

particles/moment_inertia

Type float (32-bit)

Size Nx3

Default 0,0,0

Units mass * length^2

Store the moment_inertia of each particle (I_{xx}, I_{yy}, I_{zz}). This inertia tensor is diagonal in the body frame of the particle. The default value is for point particles.

Properties

particles/position

Type float (32-bit)

Size Nx3

Default 0,0,0

Units length

Store the position of each particle (*x*, *y*, *z*).

All particles in the simulation are referenced by a tag. The position data chunk (and all other per particle data chunks) list particles in tag order. The first particle listed has tag 0, the second has tag 1, ..., and the last has tag N-1 where N is the number of particles in the simulation.

All particles must be inside the box:

- $x > -l_x/2 + (xz - xy \cdot yz) \cdot z + xy \cdot y$ and $x < l_x/2 + (xz - xy \cdot yz) \cdot z + xy \cdot y$
- $y > -l_y/2 + yz \cdot z$ and $y < l_y/2 + yz \cdot z$
- $z > -l_z/2$ and $z < l_z/2$

particles/orientation

Type float (32-bit)

Size Nx4

Default 1,0,0,0

Units unit quaternion

Store the orientation of each particle. In scalar + vector notation, this is (r, a_x, a_y, a_z), where the quaternion is $q = r + a_x i + a_y j + a_z k$. A unit quaternion has the property: $\sqrt{r^2 + a_x^2 + a_y^2 + a_z^2} = 1$.

Momenta

particles/velocity

Type float (32-bit)

Size Nx3

Default 0,0,0

Units length/time

Store the velocity of each particle (v_x, v_y, v_z).

particles/angmom

Type float (32-bit)

Size Nx4

Default 0,0,0,0

Units quaternion

Store the angular momentum of each particle as a quaternion. See the HOOMD documentation for information on how to convert to a vector representation.

particles/image

Type int32

Size Nx3

Default 0,0,0

Units number

Store the number of times each particle has wrapped around the box (i_x, i_y, i_z). In constant volume simulations, the unwrapped position in the particle's full trajectory is

- $x_u = x + i_x \cdot l_x + xy \cdot i_y \cdot l_y + xz \cdot i_z \cdot l_z$
- $y_u = y + i_y \cdot l_y + yz \cdot i_z * l_z$
- $z_u = z + i_z \cdot l_z$

8.1.5 Topology

bonds/N

Type uint32

Size 1x1

Default 0

Units number

Define *N*, the number of bonds, for all data chunks `bonds/*`.

bonds/types

Type int8

Size NTxM

Default *empty*

Units UTF-8

Implicitly define *NT*, the number of bond types, for all data chunks `bonds/*`. *M* must be large enough to accommodate each type name as a null terminated UTF-8 character string. Row *i* of the 2D matrix is the type name for bond type *i*. By default, there are 0 bond types.

bonds/typeid

Type uint32

Size Nx1

Default 0

Units number

Store the type id of each bond. All id's must be less than *NT*. A bond with type *id* has a type name matching the corresponding row in *bonds/types*.

bonds/group

Type uint32

Size Nx2

Default 0,0

Units number

Store the particle tags in each bond.

angles/N

Type uint32

Size 1x1

Default 0

Units number

Define *N*, the number of angles, for all data chunks `angles/*`.

angles/types

Type int8

Size NTxM

Default *empty*

Units UTF-8

Implicitly define *NT*, the number of angle types, for all data chunks `angles/*`. *M* must be large enough to accommodate each type name as a null terminated UTF-8 character string. Row *i* of the 2D matrix is the type name for angle type *i*. By default, there are 0 angle types.

angles/typeid

Type uint32

Size Nx1

Default 0

Units number

Store the type id of each angle. All id's must be less than *NT*. A angle with type *id* has a type name matching the corresponding row in *angles/types*.

angles/group

Type uint32

Size Nx2

Default 0,0

Units number

Store the particle tags in each angle.

dihedrals/N

Type uint32

Size 1x1

Default 0

Units number

Define *N*, the number of dihedrals, for all data chunks `dihedrals/*`.

dihedrals/types

Type int8

Size NTxM

Default *empty*

Units UTF-8

Implicitly define *NT*, the number of dihedral types, for all data chunks `dihedrals/*`. *M* must be large enough to accommodate each type name as a null terminated UTF-8 character string. Row *i* of the 2D matrix is the type name for dihedral type *i*. By default, there are 0 dihedral types.

dihedrals/typeid

Type uint32

Size Nx1

Default 0

Units number

Store the type id of each dihedral. All id's must be less than *NT*. A dihedral with type *id* has a type name matching the corresponding row in *dihedrals/types*.

dihedrals/group

Type uint32

Size Nx2

Default 0,0

Units number

Store the particle tags in each dihedral.

impropers/N

Type uint32

Size 1x1

Default 0

Units number

Define *N*, the number of impropers, for all data chunks `impropers/*`.

impropers/types

Type int8

Size NTxM

Default *empty*

Units UTF-8

Implicitly define *NT*, the number of improper types, for all data chunks `impropers/*`. *M* must be large enough to accommodate each type name as a null terminated UTF-8 character string. Row *i* of the 2D matrix is the type name for improper type *i*. By default, there are 0 improper types.

impropers/typeid

Type uint32

Size Nx1

Default 0

Units number

Store the type id of each improper. All id's must be less than *NT*. A improper with type *id* has a type name matching the corresponding row in *impropers/types*.

impropers/group

Type uint32

Size Nx2

Default 0,0

Units number

Store the particle tags in each improper.

constraints/N

Type uint32

Size 1x1

Default 0

Units number

Define *N*, the number of constraints, for all data chunks `constraints/*`.

constraints/value

Type float

Size Nx1

Default 0

Units length

Store the distance of each constraint. Each constraint defines a fixed distance between two particles.

constraints/group

Type uint32

Size Nx2

Default 0,0

Units number

Store the particle tags in each constraint.

pairs/N

Type uint32

Size 1x1

Default 0

Units number

Define *N*, the number of special pair interactions, for all data chunks `pairs/*`.

New in version 1.1.

pairs/types

Type int8

Size NTxM

Default *empty*

Units UTF-8

Implicitly define *NT*, the number of special pair types, for all data chunks `pairs/*`. *M* must be large enough to accommodate each type name as a null terminated UTF-8 character string. Row *i* of the 2D matrix is the type name for particle type *i*. By default, there are 0 special pair types.

New in version 1.1.

pairs/typeid

Type uint32

Size Nx1

Default 0

Units number

Store the type id of each special pair interaction. All id's must be less than *NT*. A pair with type *id* has a type name matching the corresponding row in *pairs/types*.

New in version 1.1.

pairs/group

Type uint32

Size Nx2

Default 0,0

Units number

Store the particle tags in each special pair interaction.

New in version 1.1.

8.1.6 State data

HOOMD stores auxiliary state information in `state/*` data chunks. Auxiliary state encompasses internal state to any integrator, updater, or other class that is not part of the particle system state but is also not a fixed parameter. For example, the internal degrees of freedom in integrator. Auxiliary state is useful when restarting simulations.

HOOMD only stores state in GSD files when requested explicitly by the user. Only a few of the documented state data chunks will be present in any GSD file and not all state chunks are valid. Thus, state data chunks do not have default values. If a chunk is not present in the file, that state does not have a well-defined value.

Note: HOOMD-blue versions 3.0 and newer write state data in an application defined format in `log/*`, **not** in `state/*`. See the HOOMD-blue documentation for details on the data chunks it reads and writes.

Name	Type	Size	Units
HPMC integrator state			
state/hpmc/integrate/d	double	1x1	length
state/hpmc/integrate/a	double	1x1	number
state/hpmc/sphere/radius	float	NTx1	length
state/hpmc/sphere/orientable	uint8	NTx1	boolean
state/hpmc/ellipsoid/a	float	NTx1	length
state/hpmc/ellipsoid/b	float	NTx1	length
state/hpmc/ellipsoid/c	float	NTx1	length
state/hpmc/convex_polyhedron/N	uint32	NTx1	number
state/hpmc/convex_polyhedron/vertices	float	sum(N)x3	length
state/hpmc/convex_spheropolyhedron/N	uint32	NTx1	number
state/hpmc/convex_spheropolyhedron/vertices	float	sum(N)x3	length
state/hpmc/convex_spheropolyhedron/sweep_radius	float	NTx1	length
state/hpmc/convex_polygon/N	uint32	NTx1	number
state/hpmc/convex_polygon/vertices	float	sum(N)x2	length
state/hpmc/convex_spheropolygon/N	uint32	NTx1	number
state/hpmc/convex_spheropolygon/vertices	float	sum(N)x2	length
state/hpmc/convex_spheropolygon/sweep_radius	float	NTx1	length
state/hpmc/simple_polygon/N	uint32	NTx1	number
state/hpmc/simple_polygon/vertices	float	sum(N)x2	length

HPMC integrator state

NT is the number of particle types.

state/hpmc/integrate/d

Type double

Size 1x1

Units length

d is the maximum trial move displacement.

New in version 1.2.

state/hpmc/integrate/a

Type double

Size 1x1

Units number

a is the size of the maximum rotation move.

New in version 1.2.

state/hpmc/sphere/radius

Type float

Size NTx1

Units length

Sphere radius for each particle type.

New in version 1.2.

state/hpmc/sphere/orientable

Type uint8

Size NTx1

Units boolean

Orientable flag for each particle type.

New in version 1.3.

state/hpmc/ellipsoid/a

Type float

Size NTx1

Units length

Size of the first ellipsoid semi-axis for each particle type.

New in version 1.2.

state/hpmc/ellipsoid/b

Type float

Size NTx1

Units length

Size of the second ellipsoid semi-axis for each particle type.

New in version 1.2.

state/hpmc/ellipsoid/c

Type float

Size NTx1

Units length

Size of the third ellipsoid semi-axis for each particle type.

New in version 1.2.

state/hpmc/convex_polyhedron/N

Type uint32

Size NTx1

Units number

Number of vertices defined for each type.

New in version 1.2.

state/hpmc/convex_polyhedron/vertices

Type float

Size sum(N)x3

Units length

Position of the vertices in the shape for all types. The shape for type 0 is the first N[0] vertices, the shape for type 1 is the next N[1] vertices, and so on...

New in version 1.2.

state/hpmc/convex_spheropolyhedron/N

Type uint32

Size NTx1

Units number

Number of vertices defined for each type.

New in version 1.2.

state/hpmc/convex_spheropolyhedron/vertices

Type float

Size sum(N)x3

Units length

Position of the vertices in the shape for all types. The shape for type 0 is the first N[0] vertices, the shape for type 1 is the next N[1] vertices, and so on...

New in version 1.2.

state/hpmc/convex_spheropolyhedron/sweep_radius

Type float

Size NTx1

Units length

Sweep radius for each type.

New in version 1.2.

state/hpmc/convex_polygon/N

Type uint32

Size NTx1

Units number

Number of vertices defined for each type.

New in version 1.2.

state/hpmc/convex_polygon/vertices

Type float

Size sum(N)x2

Units length

Position of the vertices in the shape for all types. The shape for type 0 is the first N[0] vertices, the shape for type 1 is the next N[1] vertices, and so on...

New in version 1.2.

state/hpmc/convex_spheropolygon/N

Type uint32

Size NTx1

Units number

Number of vertices defined for each type.

New in version 1.2.

state/hpmc/convex_spheropolygon/vertices

Type float

Size sum(N)x2

Units length

Position of the vertices in the shape for all types. The shape for type 0 is the first N[0] vertices, the shape for type 1 is the next N[1] vertices, and so on...

New in version 1.2.

state/hpmc/convex_spheropolygon/sweep_radius

Type float

Size NTx1

Units length

Sweep radius for each type.

New in version 1.2.

state/hpmc/simple_polygon/N

Type uint32

Size NTx1

Units number

Number of vertices defined for each type.

New in version 1.2.

state/hpmc/simple_polygon/vertices

Type float

Size sum(N)x2

Units length

Position of the vertices in the shape for all types. The shape for type 0 is the first N[0] vertices, the shape for type 1 is the next N[1] vertices, and so on...

New in version 1.2.

8.1.7 Logged data

Users may store logged data in `log/*` data chunks. Logged data encompasses values computed at simulation time that are too expensive or cumbersome to re-compute in post processing. This specification does not define specific chunk names or define logged data. Users may select any valid name for logged data chunks as appropriate for their workflow.

For any named logged data chunks present in any frame frame the file: If a chunk is not present in a given frame i != 0, the implementation should provide the quantity as read from frame 0 for that frame. GSD files that include a logged data chunk only in some frames i != 0 and not in frame 0 are invalid.

By convention, per-particle and per-bond logged data should have a chunk name starting with `log/particles/` and `log/bonds`, respectively. Scalar, vector, and string values may be stored under a different prefix starting with `log/`. This specification may recognize additional conventions in later versions without invalidating existing files.

Name	Type	Size	Units
log/particles/user_defined	n/a	NxM	user-defined
log/bonds/user_defined	n/a	NxM	user-defined
log/user_defined	n/a	NxM	user-defined

log/particles/user_defined

Type user-defined

Size NxM

Units user-defined

This chunk is a place holder for any number of user defined per-particle quantities. *N* is the number of particles in this frame. *M*, the data type, the units, and the chunk name (after the prefix `log/particles/`) are user-defined.

New in version 1.4.

log/bonds/user_defined

Type user-defined

Size NxM

Units user-defined

This chunk is a place holder for any number of user defined per-bond quantities. *N* is the number of bonds in this frame. *M*, the data type, the units, and the chunk name (after the prefix `log/bonds/`) are user-defined.

New in version 1.4.

log/user_defined

Type user-defined

Size NxM

Units user-defined

This chunk is a place holder for any number of user defined quantities. *N*, *M*, the data type, the units, and the chunk name (after the prefix `log/`) are user-defined.

New in version 1.4.

8.2 Shape Visualization

The chunk *particles/type_shapes* stores information about shapes corresponding to particle types. Shape definitions are stored for each type as a UTF-8 encoded JSON string containing key-value pairs. The class of a shape is defined by the `type` key. All other keys define properties of that shape. Keys without a default value are required for a valid shape specification.

8.2.1 Empty (Undefined) Shape

An empty dictionary can be used for undefined shapes. A visualization application may choose how to interpret this, e.g. by drawing nothing or drawing spheres.

Example:

```
{}
```

8.2.2 Spheres

Type: `Sphere`

Spheres' dimensionality (2D circles or 3D spheres) can be inferred from the system box dimensionality.

Key	Description	Type	Size	Default	Units
diameter	Sphere diameter	float	1x1		length

Example:

```
{
    "type": "Sphere",
    "diameter": 2.0
}
```

8.2.3 Ellipsoids

Type: `Ellipsoid`

The ellipsoid class has principal axes a, b, c corresponding to its radii in the x, y, and z directions.

Key	Description	Type	Size	Default	Units
a	Radius in x direction	float	1x1		length
b	Radius in y direction	float	1x1		length
c	Radius in z direction	float	1x1		length

Example:

```
{
    "type": "Ellipsoid",
    "a": 7.0,
    "b": 5.0,
    "c": 3.0
}
```

8.2.4 Polygons

Type: `Polygon`

A simple polygon with its vertices specified in a counterclockwise order. Spheropolygons can be represented using this shape type, through the `rounding_radius` key.

Key	Description	Type	Size	Default	Units
rounding_radius	Rounding radius	float	1x1	0.0	length
vertices	Shape vertices	float	Nx2		length

Example:

```
{
    "type": "Polygon",
    "rounding_radius": 0.1,
    "vertices": [[-0.5, -0.5], [0.5, -0.5], [0.5, 0.5]]
}
```

8.2.5 Convex Polyhedra

Type: `ConvexPolyhedron`

A convex polyhedron with vertices specifying the convex hull of the shape. Spheropolyhedra can be represented using this shape type, through the `rounding_radius` key.

Key	Description	Type	Size	Default	Units
rounding_radius	Rounding radius	float	1x1	0.0	length
vertices	Shape vertices	float	Nx3		length

Example:

```
{
    "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]]
}
```

8.2.6 General 3D Meshes

Type: `Mesh`

A list of lists of indices are used to specify faces. Faces must contain 3 or more vertex indices. The vertex indices must be zero-based. Faces must be defined with a counterclockwise winding order (to produce an "outward" normal).

Key	Description	Type	Size	Default	Units
vertices	Shape vertices	float	Nx3		length
indices	Vertices indices	uint32			number

Example:

```
{
    "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": [[0, 1, 2], [0, 3, 1], [0, 2, 3], [1, 3, 2]]
}
```

8.3 File layer

Version: 2.0

General simulation data (GSD) **file layer** design and rationale. These use cases and design specifications define the low level GSD file format.

Differences from the 1.0 specification are noted.

8.3.1 Use-cases

- capabilities
  - efficiently store many frames of data from simulation runs
  - high performance file read and write
  - support arbitrary chunks of data in each frame (position, orientation, type, etc. . .)
  - variable number of named chunks in each frame
  - variable size of chunks in each frame
  - each chunk identifies data type
  - common use cases: NxM arrays in double, float, int, char types.
  - generic use case: binary blob of N bytes
  - can be integrated into other tools
  - append frames to an existing file with a monotonically increasing frame number
  - resilient to job kills
- queries
  - number of frames
  - is named chunk present in frame *i*
  - type and size of named chunk in frame *i*
  - read data for named chunk in frame *i*
  - read only a portion of a chunk
  - list chunk names in the file
- writes
  - write data to named chunk in the current frame
  - end frame and commit to disk

These capabilities enable a simple and rich higher level schema for storing particle and other types of data. The schema determine which named chunks exist in a given file and what they mean.

8.3.2 Non use-cases

These capabilities are use-cases that GSD does **not** support, by design.

1. Modify data in the file: GSD is designed to capture simulation data.
2. Add chunks to frames in the middle of a file: See (1).
3. Transparent conversion between float and double: Callers must take care of this.
4. Transparent compression: this gets in the way of parallel I/O. Disk space is cheap.

8.3.3 Dependencies

The file layer is implemented in C (*not C++*) with no dependencies to enable trivial installation and incorporation into existing projects. A single header and C file completely implement the entire file layer. Python based projects that need only read access can use `gsd.pygsd`, a pure Python gsd reader implementation.

A Python interface to the file layer allows reference implementations and convenience methods for schemas. Most non-technical users of GSD will probably use these reference implementations directly in their scripts.

The low level C library is wrapped with cython. A Python setup.py file will provide simple installation on as many systems as possible. Cython c++ output is checked in to the repository so users do not even need cython as a dependency.

8.3.4 Specifications

Support:

- Files as large as the underlying filesystem allows (up to 64-bit address limits)
- Data chunk names of arbitrary length (v1.0 limits chunk names to 63 bytes)
- Reference up to 65535 different chunk names within a file
- Application and schema names up to 63 characters
- Store as many frames as can fit in a file up to file size limits
- Data chunks up to (64-bit) x (32-bit) elements

The limits on only 16-bit name indices and 32-bit column indices are to keep the size of each index entry as small as possible to avoid wasting space in the file index. The primary use cases in mind for column indices are Nx3 and Nx4 arrays for position and quaternion values. Schemas that wish to store larger truly n-dimensional arrays can store their dimensionality in metadata in another chunk and store as an Nx1 index entry. Or use a file format more suited to N-dimensional arrays such as HDF5.

8.3.5 File format

There are four types of data blocks in a GSD file.

1. Header block
    - Overall header for the entire file, contains the magic cookie, a format version, the name of the generating application, the schema name, and its version. Some bytes in the header are reserved for future use. Header size: 256 bytes. The header block also includes a pointer to the index, the number of allocated entries, the number of allocated entries in the index, a pointer to the name list, and the size of the name list block.
    - The header is the first 256 bytes in the file.
2. Index block
    - Index the frame data, size information, location, name id, etc...
    - The index contains space for any number of `index_entry` structs
    - The first index in the list with a location of 0 marks the end of the list.
    - When the index fills up, a new index block is allocated at the end of the file with more space and all current index entries are rewritten there.
    - Index entry size: 32 bytes
3. Name list
    - List of string names used by index entries.
    - v1.0 files: Each name is a 64-byte character string.
    - v2.0 files: Names may have any length and are separated by 0 terminators.
    - The first name that starts with the 0 byte marks the end of the list
    - The header stores the total size of the name list block.
4. Data chunk
    - Raw binary data stored for the named frame data blocks.

Header index, and name blocks are stored in memory as C structs (or arrays of C structs) and written to disk in whole chunks.

Header block

This is the header block:

```
struct gsd_header
    {
    uint64_t magic;
    uint64_t index_location;
    uint64_t index_allocated_entries;
    uint64_t namelist_location;
    uint64_t namelist_allocated_entries;
    uint32_t schema_version;
    uint32_t gsd_version;
    char application[64];
    char schema[64];
    char reserved[80];
    };
```

- `magic` is the magic number identifying this as a GSD file (`0x65DF65DF65DF65DF`).
- `gsd_version` is the version number of the gsd file layer (`0xaaaabbbb => aaaa.bbbb`).
- `application` is the name of the generating application.
- `schema` is the name of the schema for data in this gsd file.
- `schema_version` is the version of the schema (`0xaaaabbbb => aaaa.bbbb`).
- `index_location` is the file location f the index block.
- `index_allocated_entries` is the number of 64-byte segments available in the namelist block.
- `namelist_location` is the file location of the namelist block.
- `namelist_allocated_entries` is the number of entries allocated in the namelist block.
- `reserved` are bytes saved for future use.

This structure is ordered so that all known compilers at the time of writing produced a tightly packed 256-byte header. Some compilers may required non-standard packing attributes or pragmas to enforce this.

Index block

An Index block is made of a number of line items that store a pointer to a single data chunk:

```
struct gsd_index_entry
    {
    uint64_t frame;
    uint64_t N;
    int64_t location;
    uint32_t M;
    uint16_t *id*;
    uint8_t type;
    uint8_t flags;
    };
```

- `frame` is the index of the frame this chunk belongs to
- `N` and `M` define the dimensions of the data matrix (NxM in C ordering with M as the fast index).
- `location` is the location of the data chunk in the file
- `id` is the index of the name of this entry in the namelist.
- `type` is the type of the data (char, int, float, double) indicated by index values
- `flags` is reserved for future use.

Many `gsd_index_entry_t` structs are combined into one index block. They are stored densely packed and in the same order as the corresponding data chunks are written to the file.

This structure is ordered so that all known compilers at the time of writing produced a tightly packed 32-byte entry. Some compilers may required non-standard packing attributes or pragmas to enforce this.

In v1.0 files, the frame index must monotonically increase from one index entry to the next. The GSD API ensures this.

In v2.0 files, the entire index block is stored sorted first by frame, then by *id*.

Namelist block

In v2.0 files, the namelist block stores a list of strings separated by 0 terminators.

In v1.0 files, the namelist block stores a list of 0-terminated strings in 64-byte segments.

The first string that starts with 0 marks the end of the list.

Data block

A data block stores raw data bytes on the disk. For a given index entry `entry`, the data starts at location `entry.location` and is the next `entry.N * entry.M * gsd_sizeof_type(entry.type)` bytes.

CHAPTER

NINE

CODE STYLE

All code in GSD must follow a consistent style to ensure readability. We provide configuration files for linters (specified below) so that developers can automatically validate and format files.

9.1 Python

Python code in GSD should follow PEP8 with the following choices:

- 80 character line widths.
- Hang closing brackets.
- Break before binary operators.

9.1.1 Tools

- Linter: flake8 with pep8-naming
- Run: `flake8` to see a list of linter violations.
- Autoformatter: yapf
- Run: `yapf -d -r .` to see needed style changes.
- Run: `yapf -i file.py` to apply style changes to a whole file, or use your IDE to apply **yapf** to a selection.

9.1.2 Documentation

Python code should be documented with docstrings and added to the Sphinx documentation index in `doc/`. Docstrings should follow Google style formatting for use in Napoleon.

9.2 C

- 100 character line width.
- Indent only with spaces.
- 4 spaces per indent level.
- Naming conventions:
  - Functions: lowercase with words separated by underscores `function_name`.

  - Structures: lowercase with words separated by underscores `struct_name`.
  - Constants: all upper-case with words separated by underscores `SOME_CONSTANT`.

9.2.1 Tools

- Autoformatter: clang-format.
- Run: `./run-clang-format.py -r .` to see needed changes.
- Run: `clang-format -i file.c` to apply the changes.
- Linter: clang-tidy
- Compile **gsd** with **CMake** to see **clang-tidy** output.

9.2.2 Documentation

Documentation comments should be in Javadoc format and precede the item they document for compatibility with Doxygen and most source code editors. Multi-line documentation comment blocks start with `/**` and single line ones start with `///`.

See `gsd.h` for an example.

9.3 Restructured Text/Markdown files

- 80 character line width.
- Use spaces to indent.
- Indentation levels are set by the respective formats.

9.4 Other file types

Use your best judgment and follow existing patterns when styling CMake and other files types. The following general guidelines apply:

- 100 character line width.
- 4 spaces per indent level.
- 4 space indent.

9.5 Editor configuration

Visual Studio Code users: Open the provided workspace file (`gsd.code-workspace`) which provides configuration settings for these style guidelines.

CHAPTER

TEN

CREDITS

The following people contributed to GSD.

- Joshua A. Anderson, University of Michigan
- Carl Simon Adorf, University of Michigan
- Bradley Dice, University of Michigan
- Jenny W. Fothergill, Boise State University
- Jens Glaser, University of Michigan
- Vyas Ramasubramani, University of Michigan
- Luis Y. Rivera-Rivera, University of Michigan
- Brandon Butler, University of Michigan

CHAPTER

ELEVEN

LICENSE

GSD is available under the following license.

```
Copyright (c) 2016-2020 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

TWELVE

INDEX

- genindex
- modindex

PYTHON MODULE INDEX

g

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W