GSD Documentation

Release 1.5.3

The Regents of the University of Michigan

Aug 28, 2018

Contents

Version 1.5.3

See the README GSD's bitbucket page for an overview. Also use the bitbucket project page to report issues.

CHAPTER 1

Examples

1.1 HOOMD examples

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

View the page source to find unformatted example code that can be easily copied.

1.1.1 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.

1.1.2 Create a hoomd gsd file

```
In [7]: gsd.hoomd.open(name='test.gsd', mode='wb')
Out[7]: <gsd.hoomd.HOOMDTrajectory at 0x7f0d0c7daf28>
```

1.1.3 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
   ...:

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(11) )

# length is 12 because extend added 10, and append added 1
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 RAM before writing it out.

1.1.4 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([[ 0.43830729,  0.18432026,  0.63619107],
       [ 0.88168639,  0.84075862,  0.39995787],
       [ 0.11596613,  0.80149883,  0.27557573],
       [ 0.72731137,  0.17001317,  0.9320085 ],
       [ 0.20614301,  0.20308578,  0.67442524],
       [ 0.72466332,  0.35782626,  0.39341748],
       [ 0.29253235,  0.32973558,  0.03689237],
       [ 0.52141386,  0.99802911,  0.37249848],
       [ 0.25337106,  0.04192344,  0.4138754 ]], dtype=float32)
```

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

1.1.5 Slicing

```
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
```

Slicing access works like you would expect it to.

1.1.6 Pure python reader

```
In [20]: f = gsd.pygsd.GSDFile(open('test.gsd', 'rb'))

In [21]: t = gsd.hoomd.HOOMDTrajectory(f);

In [22]: t[3].particles.position
Out[22]:
array([[ 0.5432182 ,  0.64984155,  0.06200188],
       [ 0.62562644,  0.54798734,  0.23615856],
       [ 0.96161157,  0.11727072,  0.91535687],
       [ 0.13533996,  0.64899325,  0.24308661],
       [ 0.28752655,  0.98042023,  0.65371817],
       [ 0.14472696,  0.48237941,  0.4480131 ],
       [ 0.40092716,  0.21247095,  0.80834079]], 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, grid file classes that access data over the internet, etc...

1.1.7 Access state data

```
In [23]: 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],
   ....:                                                [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 [24]: 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.

1.2 File layer examples

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 examples*.

View the page source to find unformatted example code that can be easily copied.

1.2.1 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.

1.2.2 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))
```

(continues on next page)

(continued from previous page)

```
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))

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 2nd 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 is lost.

1.2.3 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.

1.2.4 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])
   ....:

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.

1.2.5 Read-only access

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

In [21]: if f.chunk_exists(frame=0, name='chunk1'):
   ....:     data = f.read_chunk(frame=0, name='chunk1')
   ....:

In [22]: data
Out[22]: array([ 1.,  2.,  3.,  4.], dtype=float32)

# Fails because the file is open read only
In [23]: f.write_chunk(name='error', data=numpy.array([1]))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\-----------------------------------
↪-----------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-23-c9aabea2641a> in <module>()
----> 1 f.write_chunk(name='error', data=numpy.array([1]))

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

RuntimeError: GSD file is opened read only: file.gsd

In [24]: f.close()
```

Files opened in read only (`rb`) mode can be read from, but not written to. The read-only mode is tuned for high performance reads with minimal memory impact and can easily handle files with tens of millions of data chunks.

1.2.6 Access file metadata

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

In [26]: f.name
Out[26]: 'file.gsd'

In [27]: f.mode
\\\\\\\\\\\\\\\\\\\\Out[27]: 'rb'

In [28]: f.gsd_version
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[28]: (1, 0)

In [29]: f.application
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[29]: 'My application'

In [30]: f.schema
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[30]:
↪'My Schema'

In [31]: f.schema_version
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out
↪(1, 0)

In [32]: f.nframes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪2

In [33]: f.close()
```

1.2.7 Open a file in read/write mode

```
In [34]: f = gsd.fl.open(name="file.gsd",
   ....:                 mode='wb+',
   ....:                 application="My application",
   ....:                 schema="My Schema",
   ....:                 schema_version=[1,0])
   ....:

In [35]: f.write_chunk(name='double', data=numpy.array([1,2,3,4], dtype=numpy.
↪float64));

In [36]: f.end_frame()

In [37]: f.nframes
Out[37]: 1

In [38]: f.read_chunk(frame=0, name='double')
\\\\\\\\\\\Out[38]: array([ 1.,  2.,  3.,  4.])
```

Files in read/write mode (`'wb+' or 'rb+'`) are inefficient. Only use this mode if you **must** read and write to the

same file, and only if you are working with relatively small files with fewer than a million data chunks. Prefer append mode for writing and read-only mode for reading.

1.2.8 Write a file in append mode

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

In [40]: f.write_chunk(name='int', data=numpy.array([10,20], dtype=numpy.int16));

In [41]: f.end_frame()

In [42]: f.nframes
Out[42]: 2

# Reads fail in append mode
In [43]: f.read_chunk(frame=2, name='double')
\\\\\\\\\\\----------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-43-cab5b10fd02b> in <module>()
----> 1 f.read_chunk(frame=2, name='double')

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

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

In [44]: f.close()
```

Append mode is extremely frugal with memory. It only caches data chunks for the frame about to be committed and clears the cache on a call to *gsd.fl.GSDFile.end_frame()*. This is especially useful on supercomputers where memory per node is limited, but you may want to generate gsd files with millions of data chunks.

1.2.9 Use as a context manager

```
In [45]: 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 [46]: data
Out[46]: array([ 1.,  2.,  3.,  4.])
```

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

1.2.10 Store string chunks

```
In [47]: f = gsd.fl.open(name="file.gsd",
   ....:                 mode='wb+',
   ....:                 application="My application",
   ....:                 schema="My Schema",
   ....:                 schema_version=[1,0])
   ....:

In [48]: f.mode
Out[48]: 'wb+'

In [49]: s = "This is a string"

In [50]: b = numpy.array([s], dtype=numpy.dtype((bytes, len(s)+1)))

In [51]: b = b.view(dtype=numpy.int8)

In [52]: b
Out[52]:
array([ 84, 104, 105, 115,  32, 105, 115,  32,  97,  32, 115, 116, 114,
       105, 110, 103,   0], dtype=int8)

In [53]: f.write_chunk(name='string', data=b)

In [54]: f.end_frame()

In [55]: r = f.read_chunk(frame=0, name='string')

In [56]: r
Out[56]:
array([ 84, 104, 105, 115,  32, 105, 115,  32,  97,  32, 115, 116, 114,
       105, 110, 103,   0], dtype=int8)

In [57]: r = r.view(dtype=numpy.dtype((bytes, r.shape[0])));

In [58]: r[0].decode('UTF-8')
Out[58]: 'This is a string'

In [59]: 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.

1.2.11 Truncate

```
In [60]: f = gsd.fl.open(name="file.gsd",
   ....:                 mode='ab',
   ....:                 application="My application",
   ....:                 schema="My Schema",
   ....:                 schema_version=[1,0])
   ....:

In [61]: f.nframes
Out[61]: 1
```

(continues on next page)

(continued from previous page)

```
In [62]: f.schema, f.schema_version, f.application
\\\\\\\\\\\Out[62]: ('My Schema', (1, 0), 'My application')

In [63]: f.truncate()

In [64]: f.nframes
Out[64]: 0

In [65]: f.schema, f.schema_version, f.application
\\\\\\\\\\\Out[65]: ('My Schema', (1, 0), 'My application')
```

Truncating a gsd file removes all data chunks from it, but retains the same schema, schema version, and applicaiton 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 2

gsd python package

GSD provides an optional python API. This is the most convenient way for users to read and write GSD files. Developers, or users not working with the python language, may want to use the *C API*.

2.1 Submodules

2.1.1 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 examples* for detailed example code.

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

class gsd.fl.**GSDFile**(*name*, *mode*, *application*, *schema*, *schema_version*)

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.

Changed in version 1.2: For new code, use `open()` instead of constructing GSDFile directly. GSDFile.__init__ is backwards compatible with the old open syntax used in GSD versions 1.0.x and 1.1.x.

name

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

mode

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

gsd_version

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

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

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

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

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

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')

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 may 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();

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));
```

(continues on next page)

(continued from previous page)

```
In [7]: f.end_frame();

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

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 to 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)

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>()
```

(continues on next page)

(continued from previous page)

```
----> 1 f.read_chunk(frame=2, name='chunk1')

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')
```

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.**create**(*name*, *application*, *schema*, *schema_version*)

Create an empty GSD file on the filesystem.

Deprecated since version 1.2: As of version 1.2, you can create and open GSD files in the same call to *open()*. *create()* is kept for backwards compatibility.

Parameters

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

Example

Create a gsd file:

```
In [1]: gsd.fl.create(name="file.gsd",
   ...:               application="My application",
   ...:               schema="My Schema",
   ...:               schema_version=[1,0]);
   ...:
```

Danger: The file is overwritten if it already exists.

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. *Inefficient for large files.*
`'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. *Inefficient for large files.*
`'ab'`	Open an existing file for writing. Does *not* create or overwrite existing files.

The '+' read/write modes are inefficient at handling large files, as they read the entire file index into memory. Prefer the appropriate read or write only modes.

When opening a file for reading (`'r' or 'a'` modes): `application` and `schema_version` are ignored. *`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.

New in version 1.2.

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.GSDFile(name='file.gsd', mode='rb');

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

2.1.2 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, in-database grid files, etc... 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 – File-like object opened **(read only)**.

name
str – file.name **(read only)**.

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

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

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

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

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

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

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.

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 to 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.

2.1.3 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 examples* for full examples.

- *create()* - Create a hoomd schema GSD file (deprecated).
- *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

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

types

list[str] – Names of the particle types (*bonds/types*, *angles/types*, *dihedrals/types*, *impropers/types*, *pairs/types*).

typeid

numpy.ndarray[uint32, ndim=1, mode='c'] – N length array defining bond type ids (*bonds/typeid*, *angles/typeid*, *dihedrals/typeid*, *impropers/typeid*, *pairs/types*).

group

numpy.ndarray[uint32, ndim=2, mode='c'] – NxM array defining tags in the particle bonds (*bonds/group*, *angles/group*, *dihedrals/group*, *impropers/group*, *pairs/group*).

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

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

dimensions

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

box
numpy.ndarray[float, ndim=1, mode='c'] – Box dimensions (*configuration/box*) - [lx, ly, lz, xy, xz, yz].

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

value
numpy.ndarray[float32, ndim=1, mode='c'] – N length array defining constraint lengths (*constraints/value*).

group
numpy.ndarray[uint32, ndim=2, mode='c'] – Nx2 array defining tags in the particle constraints (*constraints/group*).

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.

Create hoomd GSD files with *create()*.

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

types
list[str] – Names of the particle types (*particles/types*).

position
numpy.ndarray[float, ndim=2, mode='c'] – Nx3 array defining particle position (*particles/position*).

orientation
numpy.ndarray[float, ndim=2, mode='c'] – Nx4 array defining particle position (*particles/orientation*).

typeid
numpy.ndarray[uint32, ndim=1, mode='c'] – N length array defining particle type ids (*particles/typeid*).

mass
numpy.ndarray[float, ndim=1, mode='c'] – N length array defining particle masses (*particles/mass*).

charge
numpy.ndarray[float, ndim=1, mode='c'] – N length array defining particle charges (*particles/charge*).

diameter
numpy.ndarray[float, ndim=1, mode='c'] – N length array defining particle diameters (*particles/diameter*).

body
numpy.ndarray[int32, ndim=1, mode='c'] – N length array defining particle bodies (*particles/body*).

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

velocity
numpy.ndarray[float, ndim=2, mode='c'] – Nx3 array defining particle velocities (*particles/velocity*).

angmom
numpy.ndarray[float, ndim=2, mode='c'] – Nx4 array defining particle angular momenta (*particles/angmom*).

image
numpy.ndarray[int32, ndim=2, mode='c'] – Nx3 array defining particle images (*particles/image*).

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
ConfigurationData – Configuration data.

particles
ParticleData – Particle data snapshot.

bonds
BondData – Bond data snapshot.

angles
BondData – Angle data snapshot.

dihedrals
BondData – Dihedral data snapshot.

impropers
BondData – Improper data snapshot.

pairs (
py:class: *BondData*): Special pair interactions snapshot

state
dict – Dictionary containing state data

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.**create**(*name*, *snapshot=None*)
Create a hoomd gsd file from the given snapshot.

Parameters

- **name** (*str*) – File name.
- **snapshot** (*Snapshot*) – Snapshot to write to frame 0. No frame is written if snapshot is `None`.

Deprecated since version 1.2: As of version 1.2, you can create and open hoomd GSD files in the same call to *open()*. *create()* is kept for backwards compatibility.

Danger: The file is overwritten if it already exists.

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. *Inefficient for large files.*
`'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. *Inefficient for large files.*
`'ab'`	Open an existing file for writing. Does *not* create or overwrite existing files.

New in version 1.2.

2.2 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__**
str – GSD software version number. This is the version number of the software package as a whole, not the file layer version it reads/writes.

2.3 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` Documenation of the `logging` standard module.

CHAPTER 3

C API

The GSD C API consists of a single header and source file (less than 1k lines of code). It does not build as a shared library. Instead, it is intended that developers simply drop the implementation into any package that needs it.

3.1 Functions

int **gsd_create** (const char **fname*, const char **application*, const char **schema*, *uint32_t schema_version*)
Create an empty gsd file in a file of 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 0 on success, -1 on a file IO failure - see errno for details

int **gsd_open** (struct *gsd_handle_t* handle*, const char **fname*, const *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`.

Prefer the modes `GSD_OPEN_APPEND` for writing and `GSD_OPEN_READONLY` for reading. These modes are optimized to only load as much of the index as needed. `GSD_OPEN_READWRITE` needs to store the entire index in memory: in files with millions of chunks, this can add up to GiB.

Returns

0 on success. Negative value on failure:

- -1: IO error (check errno)
- -2: Not a GSD file
- -3: Invalid GSD file version
- -4: Corrupt file
- -5: Unable to allocate memory

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

Create an empty gsd file in a file of 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

0 on success. Negative value on failure:

- -1: IO error (check errno)
- -2: Not a GSD file
- -3: Invalid GSD file version
- -4: Corrupt file
- -5: Unable to allocate memory
- -6: Invalid argument

int **gsd_truncate**(*gsd_handle_t** *handle*)

Truncate a GSD file opened by `gsd_open()`.

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** – Handle to truncate.

Returns

0 on success. Negative value on failure:

- -1: IO error (check errno)

- -2: Not a GSD file
- -3: Invalid GSD file version
- -4: Corrupt file
- -5: Unable to allocate memory

int **gsd_close**(*gsd_handle_t* handle*)

Close a GSD file opened by *gsd_open()*. Call *gsd_end_frame()* after the last call to *gsd_write_chunk()* **before** closing the file.

Parameters

- **handle** – Handle to close.

Warning: Do not write chunks to the file with gsd_write_chunk() and then immediately close the file with *gsd_close()*. This will result in data loss. Data chunks written by *gsd_write_chunk()* are not updated in the index until *gsd_end_frame()* is called. This is by design to prevent partial frames in files.

Returns 0 on success, -1 on a file IO failure - see errno for details, and -2 on invalid input

int **gsd_end_frame**(*gsd_handle_t* handle*)

Move on to the next frame after writing 1 or more chunks with *gsd_write_chunk()*. Increase the frame counter by 1 and flush the cached index to disk.

Parameters

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

Returns 0 on success, -1 on a file IO failure - see errno for details, and -2 on invalid input

int **gsd_write_chunk**(struct *gsd_handle_t* 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 (truncated to 63 chars).
- **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.

Returns 0 on success, -1 on a file IO failure - see errno for details, and -2 on invalid input

const struct *gsd_index_entry_t** **gsd_find_chunk**(struct *gsd_handle_t* 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_t* 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

- -1 on a file IO failure - see errno for details
- -2 on invalid input
- -3 on invalid file data

uint64_t **gsd_get_nframes** (*gsd_handle_t* 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 1 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.

3.2 Constants

3.2.1 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.

3.2.2 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.

3.3 Data structures

gsd_handle_t
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**
File format version: 0xaaaabbbb => aaaa.bbbb

char application[64]
Name of the application that wrote the file.

char schema[64]
Name of schema defining the stored data.

uint32_t **schema_version**
Schema version: 0xaaaabbbb => aaaa.bbbb

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. Vaild values are `GSD_OPEN_READWRITE`, `GSD_OPEN_READONLY`, and `GSD_OPEN_APPEND`.

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

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 4

Specification

4.1 File layer

Version: 1.0

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

4.1.1 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
  - easy to integrate 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
- **writes**
  - write data to named chunk in the current frame
  - write a single data chunk from multiple MPI ranks
  - end frame and commit to disk

These capabilities should 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.

4.1.2 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, that raw data should not be modifiable.
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.

4.1.3 Specifications

Support:

- Files as large as the underlying filesystem allows (up to 64-bit address limits)
- Data chunk names up to 63 characters
- Reference up to 65536 different chunk names within a file
- Application and scheme 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.

4.1.4 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 in a few hundred lines of code. 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.

Boost will **not** be used so the python API will work on the widest possible number of systems. Instead, the low level C library will be 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.

4.1.5 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 used entries in the index, a pointer to the name list, the size of the name list, and the number of entries used in the name list.
    - 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 header indicates how many slots are used.
    - 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.
    - Each name is a *name_entry* struct, which holds up to 63 characters.
    - The header stores the total number of names available in the list and the number of name slots used.
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];
```

(continues on next page)

(continued from previous page)

```
    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 entries allocated in the index 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 (it rounds the struct size out to 32 bytes).

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.

The frame index must monotonically increase from one index entry to the next. The GSD API ensures this.

Namelist block

An namelist block is made of a number of line items that store the string name of a data chunk entry:

```
struct gsd_namelist_entry
    {
    char name[64];
    };
```

The `id` field of the index entry refers to the index of the name within the namelist entry.

Data block

A data block is just 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.

4.1.6 API and implementation thoughts

The C-level API is object oriented through the use of the handle structure. In the handle, the API will store cached index data in memory and so forth. A pointer to the handle will be passed in to every API call.

- `int gsd_create()` : Create a GSD file on disk, overwriting any existing file.
- `gsd_handle_t* gsd_open()` : Open a GSD file and return an allocated handle.
- `int gsd_close()` : Close a GSD file and free all memory associated with it.
- **`int gsd_end_frame()`** [Complete writing the current frame and flush it to disk. This automatically] starts a new frame.
- `int gsd_write_chunk()` : Write a chunk out to the current frame
- `uint64_t gsd_get_nframes()` : Get the number of frames written to the file
- `int gsd_index_entry_t* gsd_find_chunk()` : Find a chunk with the given name in the given frame.
- `int gsd_read_chunk()` : Read data from a given chunk (must find the chunk first with `gsd_find_chunk`).

`gsd_open` will open the file, read all of the index blocks in to memory, and determine some things it will need later. The index block is stored in memory to facilitate fast lookup of frames and named data chunks in frames.

`gsd_end_frame` increments the current frame counter and writes the current index block to disk.

`gsd_write_chunk` seeks to the end of the file and writes out the chunk. Then it updates the cached index block with a new entry. If the current index block is full, it will create a new, larger one at the end of the file. Normally, `write_chunk` only updates the data in the index cache. Only a call to `gsd_end_frame` writes out the updated index. This facilitates contiguous writes and helps ensure that all frame data blocks are completely written in a self-consistent way.

4.1.7 Failure modes

GSD is resistant to failures. The code aggressively checks for failures in memory allocations, and verifies that `write()` and `read()` return the correct number of bytes after each call. Any time an error condition hits, the current function call aborts.

GSD has a protections against invalid data in files. A specially constructed file may still be able to cause problems, but at GSD tries to stop if corrupt data is present in a variety of ways.

- The header has a magic number. If it is invalid, GSD reports an error on open. This guards against corrupt file headers.
- Before allocating memory for the index block, GSD verifies that the index block is contained within the file.
- When writing chunks, data is appended to the end of the file and the index is updated *in memory*. After all chunks for the current frame are written, the user calls `gsd_end_frame()` which writes out the updated index and header. This way, if the process is killed in the middle of writing out a frame, the index will not contain entries for the partially written data. Such a file could still be appended to safely.
- If an index entry lists a size that goes past the end of the file, `read_chunk` will return an error.

4.2 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 1.2

4.2.1 Use-cases

There are a few problems with XML, DCD, and other dump files that the GSD schema `hoomd` solves.

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

4.2.2 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/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
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

Continued on next page

Table 1 – continued from previous page

Name	Category	Type	Size	Default	Units
pairs/types	topology	int8	NTxM		utf-8
pairs/typeid	topology	uint32	Nx1	0	number
pairs/group	topology	uint32	Nx2	0,0	number

4.2.3 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

4.2.4 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/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

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

4.2.5 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.

4.2.6 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.

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/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/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.

CHAPTER 5

Scripts

5.1 hoomdxml2gsd.py

hoomdxml2gsd.py Converts an HOOMD-blue formated XML file to a hoomd schema GSD file.

usage `hoomd2xmlgsd.py input output`

Example:

```
$ hoomdxml2gsd.py init.xml init.gsd
```

input
Input hoomd XML file.

output
Output gsd file.

CHAPTER 6

Benchmarks

The benchmark script `scripts/benchmark-hoomd.py` runs a suite of I/O benchmarks that measure the time it takes to write a file, read frames sequentially, and read frames randomly. This script only runs on linux and requires that the user have no-password *sudo* access (set this only temporarily). It flushes filesystem buffers and clears the cache to provide accurate timings. It is representative of typical use cases, storing position and orientation in a hoomd schema GSD file at each frame. The benchmark runs at fixed file sizes with varying N (and varying number of frames) in order to test small block and large block I/O.

6.1 SSD

Samsung SSD 840 EVO 120GB

Size	N	Open (ms)	Write (MB/s)	Read (MB/s)	Random (MB/s)	Random (ms)
128 MiB	32^2	2.063	45.23	64.77	50.13	0.545
128 MiB	128^2	1.091	175	304.1	226.3	1.93
128 MiB	1024^2	15.56	177.7	366.2	463.8	60.4
1 GiB	32^2	3.119	54.15	73.57	35.79	0.764
1 GiB	128^2	1.703	227	305.2	188.3	2.32
1 GiB	1024^2	8.414	175.8	425.5	474.5	59
16 GiB	32^2	5.401	58.3	70.02	26.22	1.04
16 GiB	128^2	5.286	134.5	330.7	152.4	2.87
16 GiB	1024^2	8.054	130	406.7	465.5	60.1

6.2 NFS

10Gb Ethernet connection (Intel X520) through several 10Gb switches into a 100Gb campus backbone into a modern multi-petabyte Isilon fileserver, mounted with NFSv3.

Size	N	Open (ms)	Write (MB/s)	Read (MB/s)	Random (MB/s)	Random (ms)
128 MiB	32^2	16.34	42.24	84.79	39.24	0.697
128 MiB	128^2	11.14	172.2	192.6	142.7	3.07
128 MiB	1024^2	10.16	163.5	161.1	186.3	150
1 GiB	32^2	18.54	56.64	76.98	18.41	1.49
1 GiB	128^2	10.93	227.6	197.1	70.84	6.18
1 GiB	1024^2	17.35	253.5	166.8	155.6	180
128 GiB	32^2	146.9	55.34	75.62	2.111	13
128 GiB	128^2	29.95	265.3	353.5	27.03	16.2
128 GiB	1024^2	34.83	319.3	225.9	116.7	240

6.3 HDD

RAID 1 (mdadm) on two ST3000NM0033-9ZM178 drives.

Size	N	Open (ms)	Write (MB/s)	Read (MB/s)	Random (MB/s)	Random (ms)
128 MiB	32^2	36.43	12.92	59	11.63	2.35
128 MiB	128^2	29.68	72.22	175.5	48.23	9.07
128 MiB	1024^2	10.82	94.69	161.7	167.6	167
1 GiB	32^2	52.85	43.03	59.43	4.943	5.53
1 GiB	128^2	24.22	115.5	174	33.65	13
1 GiB	1024^2	31.61	123.6	153.7	151.8	184
128 GiB	32^2	113.3	46.26	58.36	2.085	13.1
128 GiB	128^2	90.05	141.8	146.6	21.82	20
128 GiB	1024^2	51.49	139.4	139.6	140.8	199

CHAPTER 7

License

GSD is available under the following license.

```
Copyright (c) 2016-2018 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 8

Index

- genindex
- modindex

Python Module Index

g

Index

Symbols

A

B

C

D

E

F

G

H

I

M

N

O

P

R

S

T

U

V

W