Mesh Intermediary Representation

Release 0.1.1

Petar Marić

Aug 29, 2019

Contents

The Mesh Intermediary Representation (MIR) specification is an open standard describing a platform-neutral data format for long-term storage and interchange of mesh-like data.

It has also been designed to serve as an intermediary step when converting the mesh-like data between different computational mechanics application formats.

You can read the MIR specification online, or download a printable PDF.

This research is based upon the previous work by Peter Iványi *[Ivanyi2012] [Ivanyi2014]*.

CHAPTER 1

MIR concepts

Warning: A valid MIR data file must contain all of the *layers*, *datasets* and *attributes* defined by this specification, even if some of them are empty.

1.1 Identifiers

Identifiers (also referred to as *names*) are described by the following lexical definitions:

```
identifier  ::=  first_word ("_" word)*
first_word  ::=  letter [word]
word        ::=  (letter | digit)+
letter      ::=  "a"..."z"
digit       ::=  "0"..."9"
```

where `letter` has to be a lowercase ASCII character.

1.1.1 Non-standard identifiers

Non-standard or vendor specific identifiers must have their names prefixed with `_x_`:

```
non_standard_identifier  ::=  "_x_" identifier
```

1.2 Data types

MIR specification defines the following atomic data types:

- integer numbers (signed)
- floating point numbers
- Unicode strings

Data elements using non-standard or vendor specific data types must use a *non-standard identifier* for their name and are beyond the scope of MIR specification.

1.3 Attributes

An attribute is a small metadata object attached directly to a *dataset* or a *layer*, used to describe their nature and/or intended usage.

Non-standard or vendor specific attributes must use a *non-standard identifier* for their name and are beyond the scope of MIR specification.

1.4 Datasets

A dataset is a rectangular 2D array (generally known as a *matrix*) of data elements, arranged in rows and columns.

A dataset may use different data types for each column, but all data elements within a single column must be of the same data type.

Non-standard or vendor specific datasets must use a *non-standard identifier* for their name and are beyond the scope of MIR specification.

1.5 Layers

Layers are used to organize individual features and properties of a MIR data file into separate logical groups.

Non-standard or vendor specific layers must use a *non-standard identifier* for their name and are beyond the scope of MIR specification.

A MIR data file is composed of the following layers:

1.5.1 `geometry` layer

The `geometry` layer has the following geometry primitives, each stored in a separate dataset:

Dataset name	Vertices	Edges	Faces	Dimensions
`lines`	2	1	0	1
`triangles`	3	3	1	2
`quads`	4	4	1	2
`hexagons`	6	6	1	2
`tetrahedrons`	4	6	4	3
`cuboids`	8	12	6	3

A vertex is a point in 3D space whose coordinates are defined as a `(x, y, z)` triplet, while a single coordinate is represented by a floating point number.

Each dataset stores a single geometry primitive per row, and has 3 times as many columns as its corresponding geometry primitive has vertices. This is because only the coordinates are stored, and each vertex is defined by 3 coordinates.

For example, the `quads` dataset requires 12 columns to store its coordinates, as it has 4 vertices. A single row of this dataset would be layed out like so:

```
# Vertex "A"    Vertex "B"    Vertex "C"    Vertex "D"
# x   y   z     x   y   z     x   y   z     x   y   z

 1.1 1.2 1.3   2.1 2.2 2.3   3.1 3.2 3.3   4.1 4.2 4.3
```

Each dataset also has the following attributes attached to it:

num_vertices Number of vertices, stored as an integer number.

num_coordinates Number of coordinates, stored as an integer number.

Equal to: `num_vertices * 3`

1.5.2 `metadata` layer

The `metadata` layer has no datasets, and instead uses the following attributes to further describe the MIR data file:

generator_name Name of the program that created the file, stored as a Unicode string.

For example: `foobar`

generator_version Version of the program that created the file, stored as a Unicode string.

For example: `1.0.3`

mir_version Version of MIR specification used when creating the file, stored as a Unicode string.

For example: `1.12.3`

created_at Date and time at which the file was created, stored as a Unicode string in ISO 8601 format.

For example: `2017-08-15T13:51:02+02:00`

CHAPTER 2

MIR file format

MIR is backed up by the HDF5 data file format.

HDF5 is an extensible and open standard, comprised of platform independent technologies which are available under Open Source licenses. HDF5 format has been designed with the objective of creating data storages that are self-descriptive, flexible, and have extremely fast and efficient access patterns to the stored data.

Note: Although not mandatory, MIR data files should have the `.mir` extension (lowercase), for easier identification on the file system.

2.1 HDF5 key concepts

Although HDF5 key concepts are best explained by the HDF5 User's Guide, a quick overview is given bellow for readers convenience:

Group A collection of HDF5 objects (including other groups).

As suggested by its name "Hierarchical Data Format", an HDF5 file is hierarchically structured, like a tree. This tree structure is very similar to the file system structures employed on UNIX systems, with directories and files. HDF5 groups are analogous to the directories, HDF5 datasets are analogous to the files.

Every HDF5 data file has at least one object, the root group. All other objects (including groups) are either members of the root group or its descendants.

Dataset A multidimensional array of data elements, together with supporting metadata.

An HDF5 dataset is an object composed of a collection of data elements and metadata that stores a description of the data elements, data layout, and all other information necessary to write, read, and interpret the stored data.

Data type A description of a specific class of data element including its storage layout as a pattern of bits.

HDF5 data types implement a flexible, extensible, and portable mechanism for specifying and discovering the storage layout of the data elements, determining how to interpret the elements (for example, as floating point numbers), and for transferring data from different compatible layouts.

Atomic data types are indivisible. Composite data types are composed of multiple elements of atomic data types.

In addition to the standard types, users can define additional custom data types.

Attribute A small metadata object attached to a group or a dataset.

Attributes are a critical part of what makes HDF5 a "self-describing" format. They are small named pieces of data attached directly to group or dataset objects. This is the official way to store metadata in HDF5.

2.2 Mapping MIR concepts onto HDF5

In general, MIR concepts map cleanly onto their HDF5 counterparts:

MIR	HDF5
Data type	Data type
Attribute	Attribute
Dataset	Dataset
Layer	Group

Detailed MIR/HDF5 data type map:

- integer numbers are stored as 32-bit signed integers, using the `H5T_STD_I32LE` HDF5 data type
- floating point numbers are stored as IEEE 754 binary64, using the `H5T_IEEE_F64LE` HDF5 data type
- all Unicode strings are UTF-8 encoded

Additionally, each MIR layer (HDF5 group) is a child of the root HDF5 group.

2.3 Recommended optimizations

2.3.1 Chunked storage layout

What is chunking:

> The storage layout defines how the raw data values in the dataset are physically stored on disk. There are three ways that a dataset can be stored: contiguous, chunked, and compact.
>
> If the storage layout is contiguous, then the raw data values will be stored physically adjacent to each other in the HDF5 file (in one contiguous block). This is the default layout for a dataset.
>
> With a chunked storage layout the data is stored in equal-sized blocks or chunks of a pre-defined size. The HDF5 library always writes and reads the entire chunk. Each chunk is stored as a separate contiguous block in the HDF5 file. There is a chunk index which keeps track of the chunks associated with a dataset.
>
> Chunked storage makes it possible to resize datasets, and because the data is stored in fixed-size chunks, to use compression filters.
>
> —https://support.hdfgroup.org/HDF5/Tutor/layout.html

Although not mandatory, it's still strongly advised to use the chunked storage layout when creating MIR data files to improve processing performance:

> It is commonly used when subsetting very large datasets. Using the chunking layout can greatly improve performance when subsetting large datasets, because only the chunks required will need to be accessed. However, it is easy to use chunking without considering the consequences of the chunk size, which can lead to strikingly poor performance.
>
> If a very small chunk size is specified for a dataset it can cause the dataset to be excessively large and it can result in degraded performance when accessing the dataset. The smaller the chunk size the more chunks that HDF5 has to keep track of, and the more time it will take to search for a chunk.

> An entire chunk has to be read and uncompressed before performing an operation. There can be a performance penalty for reading a small subset, if the chunk size is substantially larger than the subset.
>
> —https://support.hdfgroup.org/HDF5/Tutor/layout.html

2.3.2 Compression and error detection

When an HDF5 dataset is created, optional filters can be specified. These filters are added to the data transfer pipeline when data is read or written. The HDF5 standard library includes filters to implement transparent compression, data shuffling, and error detection code.

> **Warning:** To apply a filter to an HDF5 dataset it must be created with a *chunked storage layout*.

Although not mandatory, it's still strongly advised to apply the following filters when creating MIR data files to minimize disk storage requirements and improve processing performance:

Compression Enable the transparent compression filter to save storage space.

Data is compressed on the way to disk, and automatically decompressed when read. Once the dataset is created with a particular compression filter applied, data may be read and written as normal with no special steps required.

Although many interesting 3rd party compression filters are supported, HDF5 itself provides only 2 predefined filters for compression by default: ZLIB and SZIP. SZIP can't be used freely due to licensing issues, therefore ZLIB is recommended for maximum portability.

For best overall ZLIB performance the PyTables optimization guide advises the lowest compression level (1) to be used.

Shuffle Enable the shuffle filter to improve the compression ratio.

Block-oriented compressors work better when presented with runs of similar values. The shuffle filter rearranges the bytes in the chunk and may improve compression ratio. No significant speed penalty.

Fletcher32 Adds an error detection checksum to each chunk to detect data corruption.

Attempts to read corrupted chunks will fail with an error. No significant speed penalty.

CHAPTER 3

Example programs and data files

An example MIR data file, filled with random data, is available on our GitHub repository, under `examples/random-data.mir`.

3.1 Creating a MIR data file

The following Python program creates a MIR data file, filled with random data:

```
#!/usr/bin/env python
from datetime import datetime
import os

import numpy as np # $ pip install numpy
import tables as tb # $ pip install tables
from tzlocal import get_localzone # $ pip install tzlocal


filename = 'random-data.mir'

sample_num_rows = 100
geometry_primitives = [
    # dataset_name , num_vertices
    ('lines'       , 2),
    ('triangles'   , 3),
    ('quads'       , 4),
    ('hexagons'    , 6),
    ('tetrahedrons', 4),
    ('cuboids'     , 8),
]


filters = tb.Filters(complib='zlib', complevel=1, shuffle=True, fletcher32=True)
with tb.open_file(filename, 'w', filters=filters) as fp:
    geometry_layer = fp.create_group(where='/', name='geometry')
    for dataset_name, num_vertices in geometry_primitives:
        num_coordinates = num_vertices * 3 # num_coordinates_per_vertex

        dataset = fp.create_earray(
```

```
            where=geometry_layer,
            name=dataset_name,
            atom=tb.Float64Atom(),
            shape=(0, num_coordinates),
        )
        dataset._v_attrs.num_vertices = num_vertices
        dataset._v_attrs.num_coordinates = num_coordinates

        dataset.append(np.random.rand(sample_num_rows, num_coordinates))

    metadata_layer = fp.create_group(where='/', name='metadata')
    metadata_layer._v_attrs.generator_name = __file__
    metadata_layer._v_attrs.generator_version = '1.0'
    metadata_layer._v_attrs.mir_version = '0.1'
    metadata_layer._v_attrs.created_at = datetime.now(get_localzone()).replace(
        microsecond=0
    ).isoformat()
```

Note: The `geometry` layer, while structurally valid, may not contain valid geometry primitives due to randomly generated coordinates.

3.2 Reading from a MIR data file

The following Python program reads some data from a MIR data file:

```
#!/usr/bin/env python
import tables as tb # $ pip install tables


filename = 'random-data.mir'


with tb.open_file(filename) as fp:
    print 'Loading the `/geometry/quads` dataset...'
    dataset = fp.root.geometry.quads

    print '\nFirst 5 geometry primitives in the dataset:'
    print dataset[:5]

    print '\nLast vertex of the first 5 geometry primitives in the dataset:'
    print dataset[:5, -3:]

    print '\n`num_vertices` dataset attribute:',
    print dataset._v_attrs.num_vertices
```

Bibliography

[Ivanyi2012] Iványi P. Finite element mesh conversion based on regular expressions. Advances in Engineering Software. 2012;51:20–39.

[Ivanyi2014] Iványi P. A Technique for the Long Term Preservation of Finite Element Meshes. In: 2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS). IEEE; 2014. p. 116–120.