MDIO

TGS

Apr 04, 2024

CONTENTS

"MDIO" is a library to work with large multidimensional energy datasets. The primary motivation behind **MDIO** is to represent multidimensional time series data in a format that makes it easier to use in resource assessment, machine learning, and data processing workflows.

See the documentation for more information.

This is not an official TGS product.

Shared Features

- Abstractions for common energy data types (see below).
- Cloud native chunked storage based on Zarr and fsspec.
- Lossy and lossless data compression using Blosc and ZFP.
- Distributed reads and writes using Dask.
- Powerful command-line-interface (CLI) based on Click

Domain Specific Features

- Oil & Gas Data
  - Import and export 2D - 5D seismic data types stored in SEG-Y.
  - Import seismic interpretation, horizon, data. **FUTURE**
  - Optimized chunking logic for various seismic types. **FUTURE**
- Wind Resource Assessment
  - Numerical weather prediction models with arbitrary metadata. **FUTURE**
  - Optimized chunking logic for time-series analysis and mapping. **FUTURE**
  - Xarray interface. **FUTURE**

The features marked as **FUTURE** will be open-sourced at a later date.

CHAPTER

ONE

INSTALLING MDIO

Simplest way to install *MDIO* via pip from PyPI:

```
$ pip install multidimio
```

or install *MDIO* via conda from conda-forge:

```
$ conda install -c conda-forge multidimio
```

Extras must be installed separately on `Conda` environments.

For details, please see the *installation instructions* in the documentation.

CHAPTER

TWO

USING MDIO

Please see the *Command-line Usage* for details.

For Python API please see the *API Reference* for details.

CHAPTER

THREE

REQUIREMENTS

3.1 Minimal

Chunked storage and parallelization: `zarr`, `dask`, `numba`, and `psutil`.
SEG-Y Parsing: `segyio`
CLI and Progress Bars: `click`, `click-params`, and `tqdm`.

3.2 Optional

Distributed computing `[distributed]`: `distributed` and `bokeh`.
Cloud Object Store I/O `[cloud]`: `s3fs`, `gcsfs`, and `adlfs`.
Lossy Compression `[lossy]`: `zfpy`

CHAPTER

FOUR

CONTRIBUTING TO MDIO

Contributions are very welcome. To learn more, see the *Contributor Guide*.

CHAPTER

FIVE

LICENSING

Distributed under the terms of the *Apache 2.0 license*, *MDIO* is free and open source software.

CHAPTER

SIX

ISSUES

If you encounter any problems, please file an issue along with a detailed description.

CHAPTER

SEVEN

CREDITS

This project was established at TGS. Current maintainer is Altay Sansal with the support of many more great colleagues.

This project template is based on @cjolowicz's Hypermodern Python Cookiecutter template.

7.1 Install Instructions

There are different ways to install MDIO:

- Install the latest release via *pip* or *conda*.
- Building package *from source*.

Note: We strongly recommend using a virtual environment `venv` or `conda` to avoid potential conflicts with other Python packages.

7.1.1 Using `pip` and `virtualenv`

Install the 64-bit version of Python 3 from https://www.python.org.

Then we can create a `venv` and install *MDIO*.

```
$ python -m venv mdio-venv
$ mdio-venv/Scripts/activate
$ pip install -U multidimio
```

To check if installation was successful see *checking installation*.

You can also install some optional dependencies (extras) like this:

```
$ pip install multidimio[distributed]
$ pip install multidimio[cloud]
$ pip install multidimio[lossy]
```

`distributed` installs Dask for parallel, distributed processing.
`cloud` installs fsspec backed I/O libraries for AWS' S3, Google's GCS, and Azure ABS.
`lossy` will install the ZFPY library for lossy chunk compression.

7.1.2 Using conda

MDIO can also be installed in a conda environment.

Note: *MDIO* is hosted in the conda-forge channel. Make sure to always provide the -c conda-forge when running conda install or else it won't be able to find the package.

We first run the following to create and activate an environment:

```
$ conda create -n mdio-env
$ conda activate mdio-env
```

Then we can to install with conda:

```
$ conda install -c conda-forge multidimio
```

The above command will install MDIO into your conda environment.

Note: *MDIO* extras must be installed separately when using conda.

7.1.3 Checking Installation

After installing MDIO, run the following:

```
$ python -c "import mdio; print(mdio.__version__)"
```

You should see the version of MDIO printed to the screen.

7.1.4 Building from Source

All dependencies of *MDIO* are Python packages, so the build process is very simple. To install from source, we need to clone the repo first and then install locally via pip.

```
$ git clone https://github.com/TGSAI/mdio-python.git
$ cd mdio-python
$ pip install .
```

We can also install the extras in a similar way, for example:

```
$ pip install .[cloud]
```

If you want an editable version of *MDIO* then we could install it with the command below. This does allow you to make code changes on the fly.

```
$ pip install --editable .[cloud]
```

To check if installation was successful see *checking installation*.

7.2 Get Started in 10 Minutes

In this page we will be showing basic capabilities of MDIO.

For demonstration purposes, we will download the Teapot Dome open-source dataset. The dataset details and licensing can be found at the SEG Wiki.

We are using the 3D seismic stack dataset named `filt_mig.sgy`.

The full link for the dataset (hosted on AWS): http://s3.amazonaws.com/teapot/filt_mig.sgy

Warning: For plotting, the notebook requires Matplotlib as a dependency. Please install it before executing using `pip install matplotlib` or `conda install matplotlib`.

7.2.1 Downloading the SEG-Y Dataset

Let's download this dataset to our working directory. It may take from a few seconds up to a couple minutes based on your internet connection speed. The file is 386 MB in size.

The dataset is irregularly shaped, however it is padded to a rectangle with zero (dead traces). We will see that later at the live mask plotting.

```
from os import path
from urllib.request import urlretrieve

url = "http://s3.amazonaws.com/teapot/filt_mig.sgy"
urlretrieve(url, "filt_mig.sgy")
```

```
('filt_mig.sgy', <http.client.HTTPMessage at 0x7feb830b3b50>)
```

7.2.2 Ingesting to MDIO Format

To do this, we can use the convenient SEG-Y to MDIO converter.

The inline and crossline values are located at bytes `181` and `185`. Note that this is not SEG-Y standard.

```
from mdio import segy_to_mdio

segy_to_mdio(
    segy_path="filt_mig.sgy",
    mdio_path_or_buffer="filt_mig.mdio",
    index_bytes=(181, 185),
    index_names=("inline", "crossline"),
)
```

```
Scanning SEG-Y for geometry attributes:   0%|                                  ␣
↪                    |...
```

```
Ingesting SEG-Y in 6 chunks:   0%|                                             ␣
↪                    |...
```

It only took a few seconds to ingest, since this is a very small file.

However, MDIO scales up to TB (that's 1000 GB) sized volumes!

7.2.3 Opening the Ingested MDIO File

Let's open the MDIO file with the `MDIOReader`.

We will also turn on `return_metadata` function to get the live trace mask and trace headers.

```
from mdio import MDIOReader

mdio = MDIOReader("filt_mig.mdio", return_metadata=True)
```

7.2.4 Querying Metadata

Now let's look at the Textual Header by the convenient `text_header` attribute.

You will notice the text header is parsed as a list of strings that are 80 characters long.

```
mdio.text_header
```

```
['C 1 CLIENT: ROCKY MOUNTAIN OILFIELD TESTING CENTER                              ',
 'C 2 PROJECT: NAVAL PETROLEUM RESERVE #3 (TEAPOT DOME); NATRONA COUNTY, WYOMING  ',
 'C 3 LINE: 3D                                                                    ',
 'C 4                                                                             ',
 'C 5 THIS IS THE FILTERED POST STACK MIGRATION                                   ',
 'C 6                                                                             ',
 'C 7 INLINE 1, XLINE 1:   X COORDINATE: 788937  Y COORDINATE: 938845             ',
 'C 8 INLINE 1, XLINE 188: X COORDINATE: 809501  Y COORDINATE: 939333             ',
 'C 9 INLINE 188, XLINE 1: X COORDINATE: 788039  Y COORDINATE: 976674             ',
 'C10 INLINE NUMBER:    MIN: 1  MAX: 345  TOTAL: 345                              ',
 'C11 CROSSLINE NUMBER: MIN: 1  MAX: 188  TOTAL: 188                              ',
 "C12 TOTAL NUMBER OF CDPS: 64860   BIN DIMENSION: 110' X 110'                    ",
 'C13                                                                             ',
 'C14                                                                             ',
 'C15                                                                             ',
 'C16                                                                             ',
 'C17                                                                             ',
 'C18                                                                             ',
 'C19 GENERAL SEGY INFORMATION                                                    ',
 'C20 RECORD LENGHT (MS): 3000                                                    ',
 'C21 SAMPLE RATE (MS): 2.0                                                       ',
 'C22 DATA FORMAT: 4 BYTE IBM FLOATING POINT                                      ',
 'C23 BYTES  13- 16: CROSSLINE NUMBER (TRACE)                                     ',
 'C24 BYTES  17- 20: INLINE NUMBER (LINE)                                         ',
 'C25 BYTES  81- 84: CDP_X COORD                                                  ',
 'C26 BYTES  85- 88: CDP_Y COORD                                                  ',
 'C27 BYTES 181-184: INLINE NUMBER (LINE)                                         ',
 'C28 BYTES 185-188: CROSSLINE NUMBER (TRACE)                                     ',
 'C29 BYTES 189-192: CDP_X COORD                                                  ',
 'C30 BYTES 193-196: CDP_Y COORD                                                  ',
 'C31                                                                             ',
```

(continues on next page)

(continued from previous page)

```
 'C32                                                                          ',
 'C33                                                                          ',
 'C34                                                                          ',
 'C35                                                                          ',
 'C36 Processed by: Excel Geophysical Services, Inc.                           ',
 'C37               8301 East Prentice Ave. Ste. 402                           ',
 'C38               Englewood, Colorado 80111                                  ',
 'C39               (voice) 303.694.9629 (fax) 303.771.1646                    ',
 'C40 END EBCDIC                                                               ']
```

MDIO parses the binary header into a Python dictionary.

We can easily query it by the `binary_header` attribute and see critical information about the original file.

Since we use `segyio` for parsing the SEG-Y, the field names conform to it.

```
mdio.binary_header
```

```
{'AmplitudeRecovery': 4,
 'AuxTraces': 0,
 'BinaryGainRecovery': 1,
 'CorrelatedTraces': 2,
 'EnsembleFold': 57,
 'ExtAuxTraces': 0,
 'ExtEnsembleFold': 0,
 'ExtSamples': 0,
 'ExtSamplesOriginal': 0,
 'ExtendedHeaders': 0,
 'Format': 1,
 'ImpulseSignalPolarity': 1,
 'Interval': 2000,
 'IntervalOriginal': 0,
 'JobID': 9999,
 'LineNumber': 9999,
 'MeasurementSystem': 2,
 'ReelNumber': 1,
 'SEGYRevision': 0,
 'SEGYRevisionMinor': 0,
 'Samples': 1501,
 'SamplesOriginal': 1501,
 'SortingCode': 4,
 'Sweep': 0,
 'SweepChannel': 0,
 'SweepFrequencyEnd': 0,
 'SweepFrequencyStart': 0,
 'SweepLength': 0,
 'SweepTaperEnd': 0,
 'SweepTaperStart': 0,
 'Taper': 0,
 'TraceFlag': 0,
 'Traces': 188,
 'VerticalSum': 1,
 'VibratoryPolarity': 0}
```

7.2.5 MDIO Grid, Dimensions, and Related Attributes

MDIO also has named dimensions, so we can see which dimension (axis) corresponds to which coordinate.

MDIO has an abstraction for an N-Dimensional grid. We can get the grid, and look at some of its properties.

```
mdio.grid.dim_names
```

```
('inline', 'crossline', 'sample')
```

```
mdio.grid.get_min("inline")
```

```
1
```

```
mdio.grid.get_max("crossline")
```

```
188
```

We can extract a dimension by name, and see its values.

The `Dimension` has `name` and `coords` that returns a string and a numpy array.

```
mdio.grid.select_dim("inline")
```

```
Dimension(coords=array([  1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,  12,  13,
        14,  15,  16,  17,  18,  19,  20,  21,  22,  23,  24,  25,  26,
        27,  28,  29,  30,  31,  32,  33,  34,  35,  36,  37,  38,  39,
        40,  41,  42,  43,  44,  45,  46,  47,  48,  49,  50,  51,  52,
        53,  54,  55,  56,  57,  58,  59,  60,  61,  62,  63,  64,  65,
        66,  67,  68,  69,  70,  71,  72,  73,  74,  75,  76,  77,  78,
        79,  80,  81,  82,  83,  84,  85,  86,  87,  88,  89,  90,  91,
        92,  93,  94,  95,  96,  97,  98,  99, 100, 101, 102, 103, 104,
       105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117,
       118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130,
       131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143,
       144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156,
       157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169,
       170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182,
       183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195,
       196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208,
       209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221,
       222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234,
       235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247,
       248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260,
       261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273,
       274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286,
       287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299,
       300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312,
       313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325,
       326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338,
       339, 340, 341, 342, 343, 344, 345]), name='inline')
```

7.2.6 Fetching Data and Plotting

Now we will demonstrate getting an inline from MDIO.

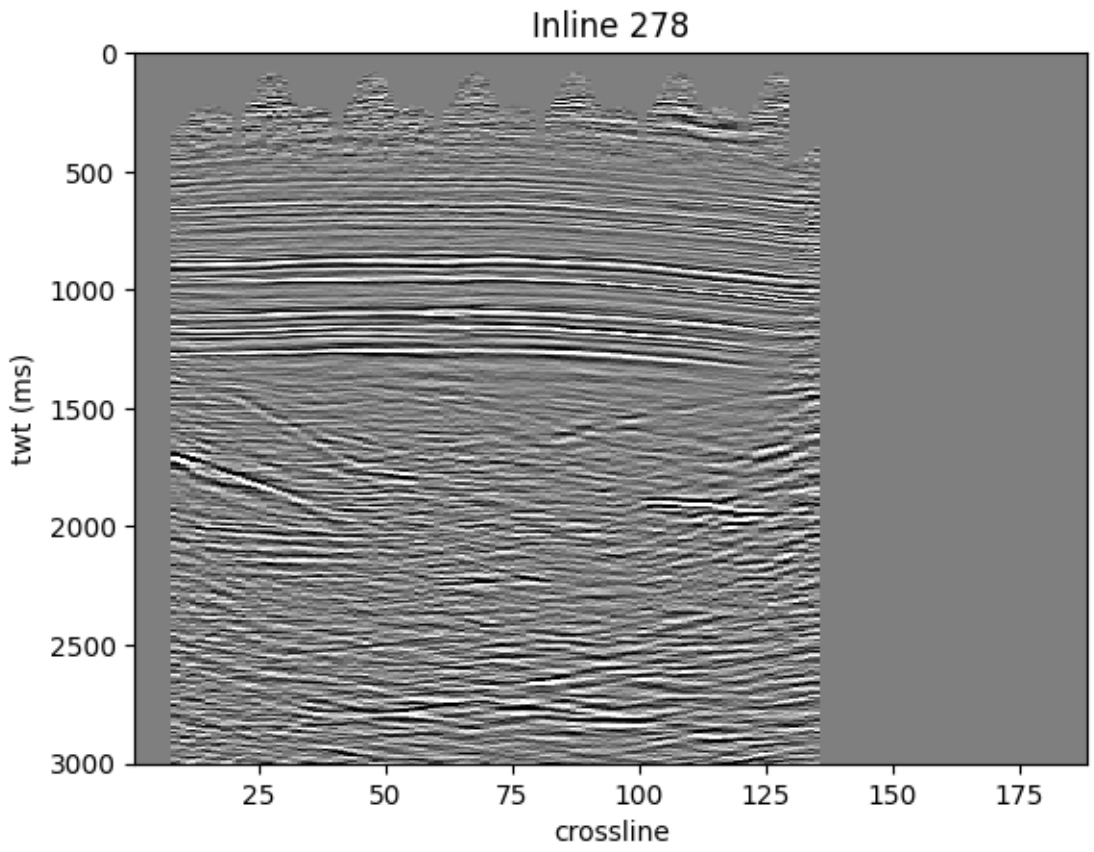
Because MDIO can hold various dimensionality of data, we have to first query the inline location.

Then we can use the queried index to get the data itself.

We will also plot the inline, for this we need the crossline and sample coordinates.

MDIO stores dataset statistics. We can use the standard deviation (std) value of the dataset to adjust the gain.

```
import matplotlib.pyplot as plt


crosslines = mdio.grid.select_dim("crossline").coords
times = mdio.grid.select_dim("sample").coords

std = mdio.stats["std"]

inline_index = int(mdio.coord_to_index(278, dimensions="inline"))
il_mask, il_headers, il_data = mdio[inline_index]

vmin, vmax = -2 * std, 2 * std
plt.pcolormesh(crosslines, times, il_data.T, vmin=vmin, vmax=vmax, cmap="gray_r")
plt.gca().invert_yaxis()
plt.title(f"Inline {278}")
plt.xlabel("crossline")
plt.ylabel("twt (ms)");
```

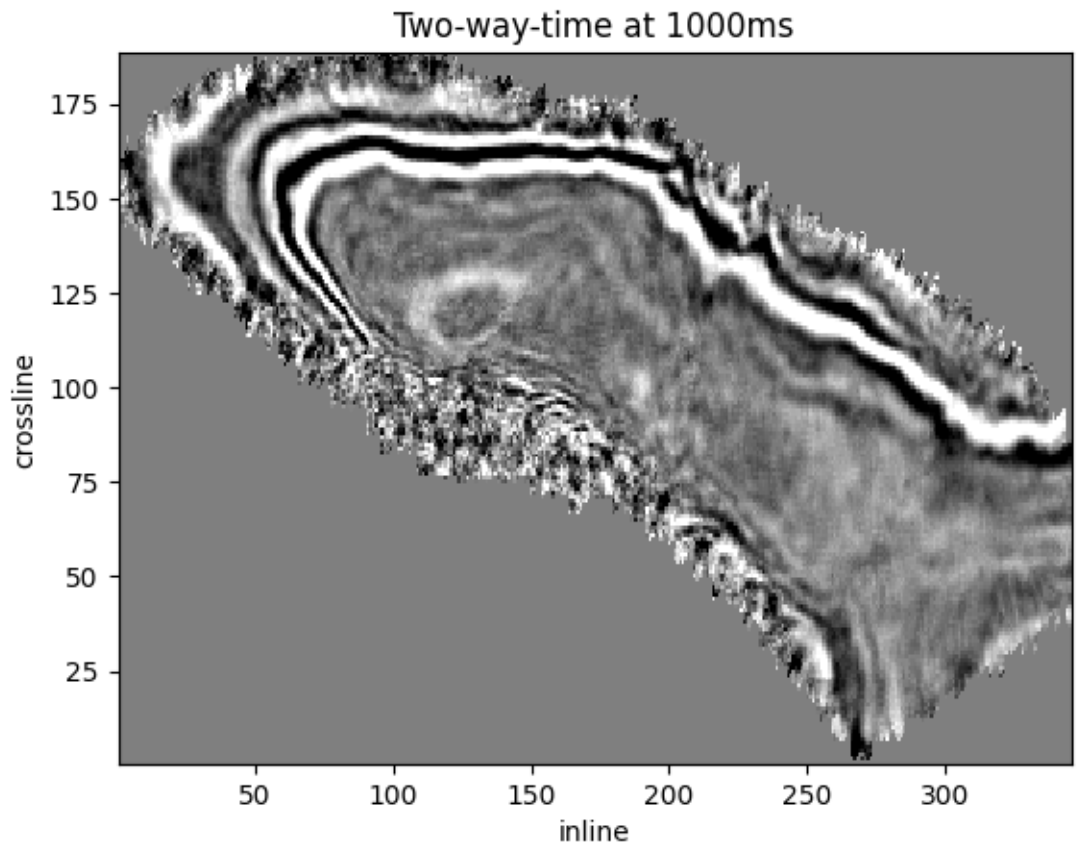


Let's do the same with a time sample.

We already have crossline labels and standard deviation, so we don't have to fetch it again.

We will display two-way-time at 1,000 ms.

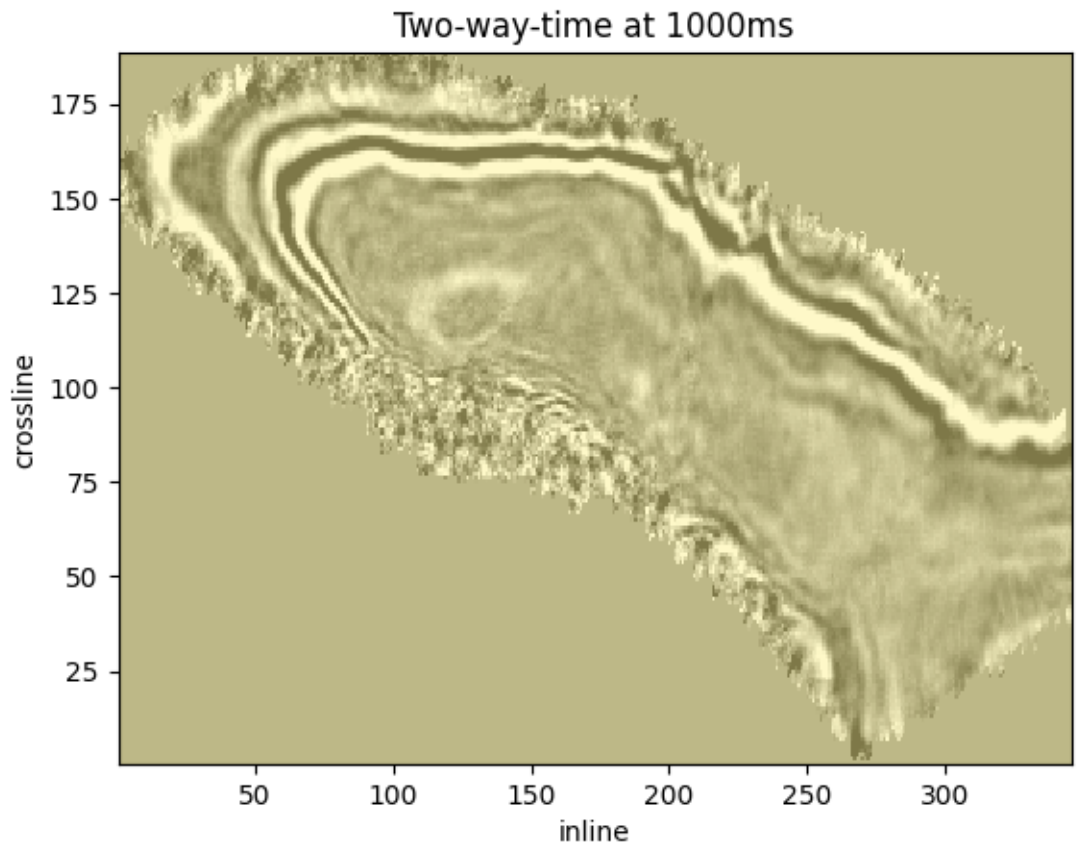
```
inlines = mdio.grid.select_dim("inline").coords

twt_index = int(mdio.coord_to_index(1_000, dimensions="sample"))
z_mask, z_headers, z_data = mdio[:, :, twt_index]

vmin, vmax = -2 * std, 2 * std
plt.pcolormesh(inlines, crosslines, z_data.T, vmin=vmin, vmax=vmax, cmap="gray_r")
plt.title(f"Two-way-time at {1000}ms")
plt.xlabel("inline")
plt.ylabel("crossline");
```



We can also overlay live mask with the time slice. However, in this example dataset is zero padded.

The live mask will always show True.

```
live_mask = mdio.live_mask[:]

plt.pcolormesh(inlines, crosslines, live_mask.T, vmin=0, vmax=1, alpha=0.5)
plt.pcolormesh(inlines, crosslines, z_data.T, vmin=vmin, vmax=vmax, cmap="gray_r",␣
↪alpha=0.5)
plt.title(f"Two-way-time at {1000}ms")
plt.xlabel("inline")
plt.ylabel("crossline");
```



7.2.7 Query Headers

We can query headers for the whole dataset very quickly because they are separated from the seismic wavefield.

Let's get all the headers for byte 189 and 193 (X and Y in this dataset, non-standard).

Note that the header maps will still honor the geometry of the dataset!

```
mdio._headers[:]["189"]
```

```
array([[788937, 789047, 789157, ..., 809282, 809392, 809502],
       [788935, 789045, 789155, ..., 809279, 809389, 809499],
       [788932, 789042, 789152, ..., 809276, 809386, 809496],
       ...,
       [788044, 788154, 788264, ..., 808389, 808499, 808609],
       [788042, 788152, 788262, ..., 808386, 808496, 808606],
       [788039, 788149, 788259, ..., 808383, 808493, 808603]], dtype=int32)
```

```
mdio._headers[:]["193"]
```

```
array([[938846, 938848, 938851, ..., 939329, 939331, 939334],
       [938956, 938958, 938961, ..., 939439, 939441, 939444],
       [939066, 939068, 939071, ..., 939549, 939551, 939554],
       ...,
```

(continues on next page)

(continued from previous page)

```
       [976455, 976458, 976460, ..., 976938, 976941, 976943],
       [976565, 976568, 976570, ..., 977048, 977051, 977053],
       [976675, 976678, 976680, ..., 977158, 977161, 977163]], dtype=int32)
```

or both at he same time:

```
mdio._headers[:][["189", "193"]]
```

```
array([[(788937, 938846), (789047, 938848), (789157, 938851), ...,
        (809282, 939329), (809392, 939331), (809502, 939334)],
       [(788935, 938956), (789045, 938958), (789155, 938961), ...,
        (809279, 939439), (809389, 939441), (809499, 939444)],
       [(788932, 939066), (789042, 939068), (789152, 939071), ...,
        (809276, 939549), (809386, 939551), (809496, 939554)],
       ...,
       [(788044, 976455), (788154, 976458), (788264, 976460), ...,
        (808389, 976938), (808499, 976941), (808609, 976943)],
       [(788042, 976565), (788152, 976568), (788262, 976570), ...,
        (808386, 977048), (808496, 977051), (808606, 977053)],
       [(788039, 976675), (788149, 976678), (788259, 976680), ...,
        (808383, 977158), (808493, 977161), (808603, 977163)]],
      dtype={'names': ['189', '193'], 'formats': ['<i4', '<i4'], 'offsets': [188, 192],
↪'itemsize': 232})
```

As we mentioned before, we can also get specific slices of headers while fetching a slice.

Let's fetch a crossline, we are still using some previous parameters.

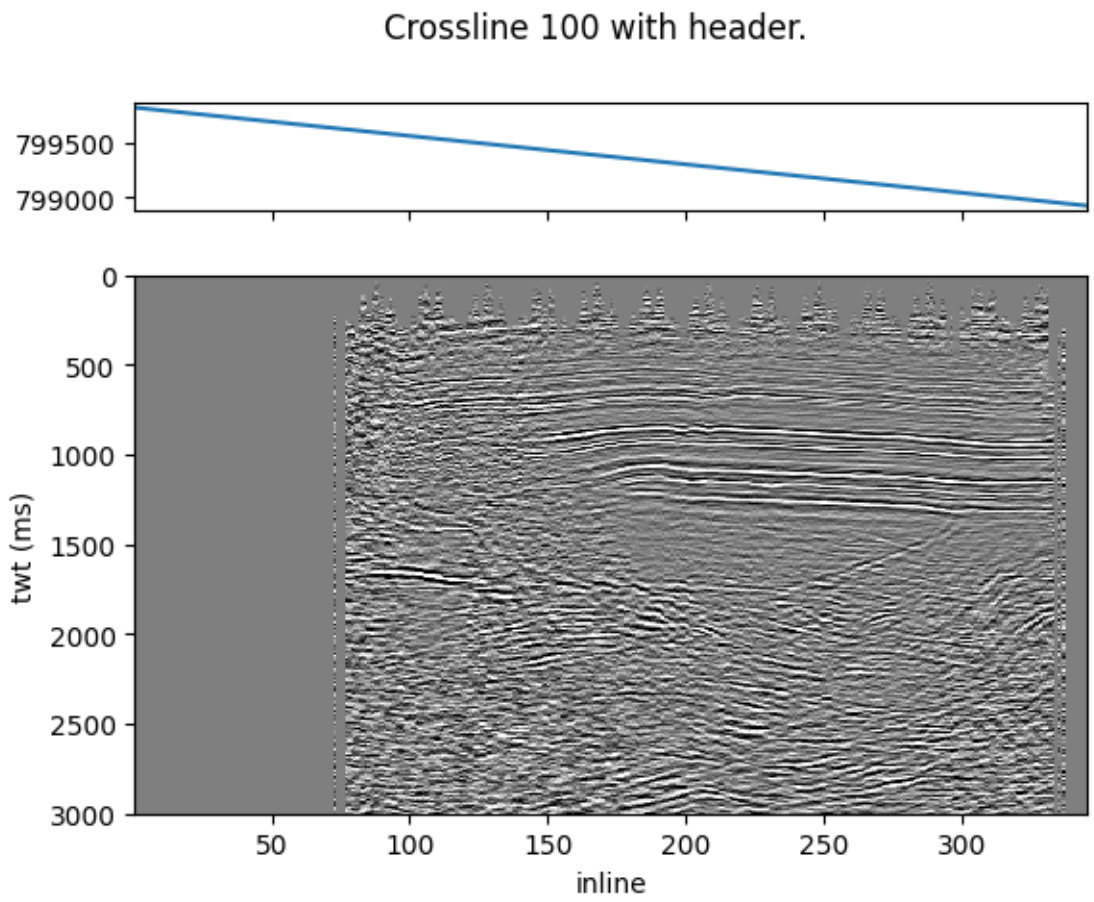
Since crossline is our second dimension, we can put the index in the second `mdio[...]` axis.

Since MDIO returns the headers as well, we can plot the headers on top of the image.

All headers will be returned, so we can select the X-coordinate at byte `189`.

Full headers can be mapped and plotted as well, but we won't demonstrate that here.

```
crossline_index = int(mdio.coord_to_index(100, dimensions="crossline"))
xl_mask, xl_headers, xl_data = mdio[:, crossline_index]

vmin, vmax = -2 * std, 2 * std

gs_kw = dict(height_ratios=(1, 5))
fig, ax = plt.subplots(2, 1, gridspec_kw=gs_kw, sharex="all")

ax[0].plot(inlines, xl_headers["189"])

ax[1].pcolormesh(inlines, times, xl_data.T, vmin=vmin, vmax=vmax, cmap="gray_r")
ax[1].invert_yaxis()
ax[1].set_xlabel("inline")
ax[1].set_ylabel("twt (ms)")

plt.suptitle(f"Crossline {100} with header.");
```



7.2.8 MDIO to SEG-Y Conversion

Finally, let's demonstrate going back to SEG-Y.

We will use the convenient `mdio_to_segy` function and write it out as a round-trip file.

```
from mdio import mdio_to_segy

mdio_to_segy(
    mdio_path_or_buffer="filt_mig.mdio",
    output_segy_path="filt_mig_roundtrip.sgy",
)
```

```
Array shape is (345, 188, 1501)
Setting (dask) chunks from (128, 128, 128) to (128, 128, 1501)
```

```
Step 1 / 2 Writing Blocks:   0%|                                                ␣
↪                   | ...
```

```
Step 2 / 2 Concat Blocks: 0block [00:00, ?block/s]
```

7.2.9 Validate Round-Trip SEG-Y File

We can validate if the round-trip SEG-Y file is matching the original using `segyio`.

Step by step:

- Open original file
- Open round-trip file
- Compare text headers
- Compare binary headers
- Compare 100 random headers and traces

```
import numpy as np
import segyio

original_fp = segyio.open("filt_mig.sgy", iline=181, xline=185)
roundtrip_fp = segyio.open("filt_mig_roundtrip.sgy", iline=181, xline=185)

# Compare text header
assert original_fp.text[0] == roundtrip_fp.text[0]

# Compare bin header
assert original_fp.bin == roundtrip_fp.bin

# Compare 100 random trace headers and traces
rng = np.random.default_rng()
rand_indices = rng.integers(low=0, high=original_fp.tracecount, size=100)
for idx in rand_indices:
    np.testing.assert_equal(original_fp.header[idx], roundtrip_fp.header[idx])
    np.testing.assert_equal(original_fp.trace[idx], roundtrip_fp.trace[idx])

original_fp.close()
roundtrip_fp.close()
```

7.3 Seismic Data Compression

In this page we will be showing compression performance of *MDIO*.

For demonstration purposes, we will use one of the Volve dataset stacks. The dataset is licensed by Equinor and Volve License Partners under Equinor Open Data Licence. License document and further information can be found here.

We are using the 3D seismic stack dataset named `ST10010ZC11_PZ_PSDM_KIRCH_FAR_D.MIG_FIN.POST_STACK.3D.JS-017536.segy`.

However, for convenience, we renamed it to `volve.segy`.

Warning: The examples below need the following extra dependencies:

1. Matplotlib for plotting.
2. Scikit-image for calculating metrics.

Please install them before executing using `pip` or `conda`.

Note: Even though we demonstrate with Volve here, this notebook can be run with any seismic dataset.

If you are new to *MDIO* we recommend you first look at our *quick start guide*

```
from mdio import segy_to_mdio, MDIOReader
```

7.3.1 Ingestion

We will ingest three files:

1. Lossless mode
2. Lossy mode (with default tolerance)
3. Lossy mode (with more compression, more relaxed tolerance)

Lossless (Default)

```
segy_to_mdio(
    "volve.segy",
    "volve.mdio",
    (189, 193),
    # lossless=True,
    # compression_tolerance=0.01,
)

print("Done.")
```

```
Scanning SEG-Y for geometry attributes:   0%|          | 0/6 [00:00<?, ?block/s]
```

```
Ingesting SEG-Y in 24 chunks:   0%|          | 0/24 [00:00<?, ?block/s]
```

```
Done.
```

Lossy Default

Equivalent to `tolerance = 0.01`.

```
segy_to_mdio(
    "volve.segy",
    "volve_lossy.mdio",
    (189, 193),
    lossless=False,
    # compression_tolerance=0.01,
)

print("Done.")
```

```
Scanning SEG-Y for geometry attributes:   0%|          | 0/6 [00:00<?, ?block/s]
```

```
Ingesting SEG-Y in 24 chunks:   0%|          | 0/24 [00:00<?, ?block/s]
```

```
Done.
```

Lossy+ (A Lot of Compression)

Here we set `tolerance = 1`. This means all our errors will be comfortably under 1.0.

```
segy_to_mdio(
    "volve.segy",
    "volve_lossy_plus.mdio",
    (189, 193),
    lossless=False,
    compression_tolerance=1,
)

print("Done.")
```

```
Scanning SEG-Y for geometry attributes:   0%|          | 0/6 [00:00<?, ?block/s]
```

```
Ingesting SEG-Y in 24 chunks:   0%|          | 0/24 [00:00<?, ?block/s]
```

```
Done.
```

Observe Sizes

Since *MDIO* uses a hierarchical directory structure, we provide a convenience function to get size of it using directory recursion and getting size.

```
import os


def get_dir_size(path: str) -> int:
    """Get size of a directory recursively."""
    total = 0
    with os.scandir(path) as it:
        for entry in it:
            if entry.is_file():
                total += entry.stat().st_size
            elif entry.is_dir():
                total += get_dir_size(entry.path)
    return total


def get_size(path: str) -> int:
    """Get size of a folder or a file."""
    if os.path.isfile(path):
```

(continues on next page)

(continued from previous page)

```
        return os.path.getsize(path)

    elif os.path.isdir(path):
        return get_dir_size(path)


print(f"SEG-Y:\t{get_size('volve.segy') / 1024 / 1024:.2f} MB")
print(f"MDIO:\t{get_size('volve.mdio') / 1024 / 1024:.2f} MB")
print(f"Lossy:\t{get_size('volve_lossy.mdio') / 1024 / 1024:.2f} MB")
print(f"Lossy+:\t{get_size('volve_lossy_plus.mdio') / 1024 / 1024:.2f} MB")
```

```
SEG-Y:        1305.02 MB
MDIO:        998.80 MB
Lossy:        263.57 MB
Lossy+:        52.75 MB
```

7.3.2 Open Files, and Get Raw Statistics

```
lossless = MDIOReader("volve.mdio")
lossy = MDIOReader("volve_lossy.mdio")
lossy_plus = MDIOReader("volve_lossy_plus.mdio")

stats = lossless.stats
std = stats["std"]
min_ = stats["min"]
max_ = stats["max"]
```

7.3.3 Plot Images with Differences

Let's define some plotting functions for convenience.

Here, we will make two plots showing data for lossy and lossy+ versions.

We will be showing the following subplots for each dataset:

1. Lossless Inline
2. Lossy Inline
3. Difference
4. 1,000x Gained Difference

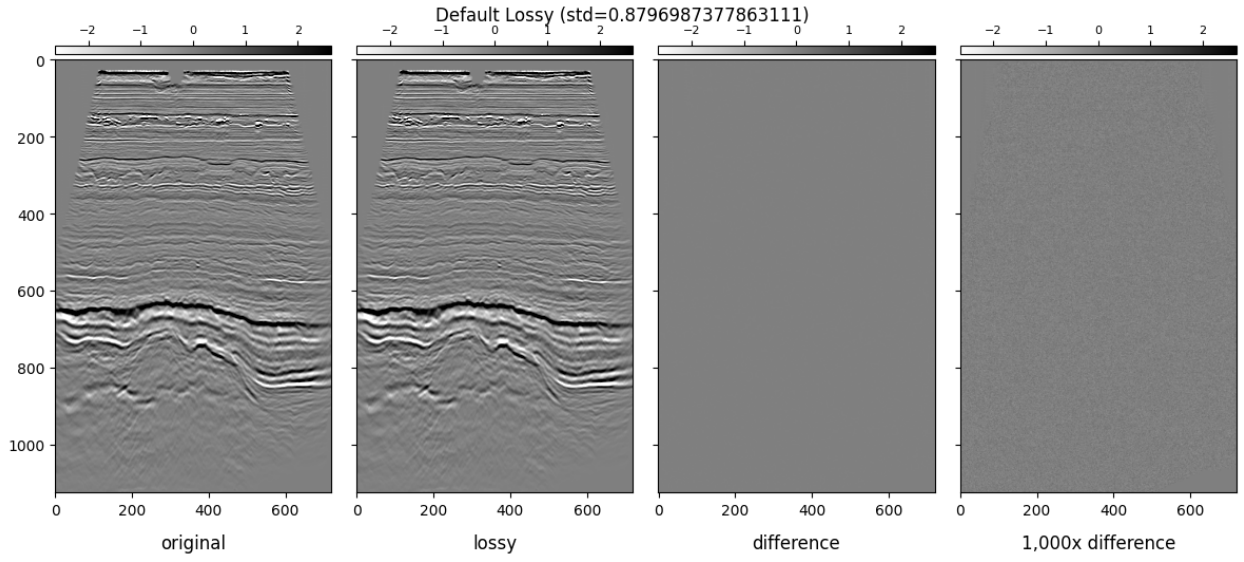
We will be using $\pm$ 3 * standard_deviation of the colorbar ranges.

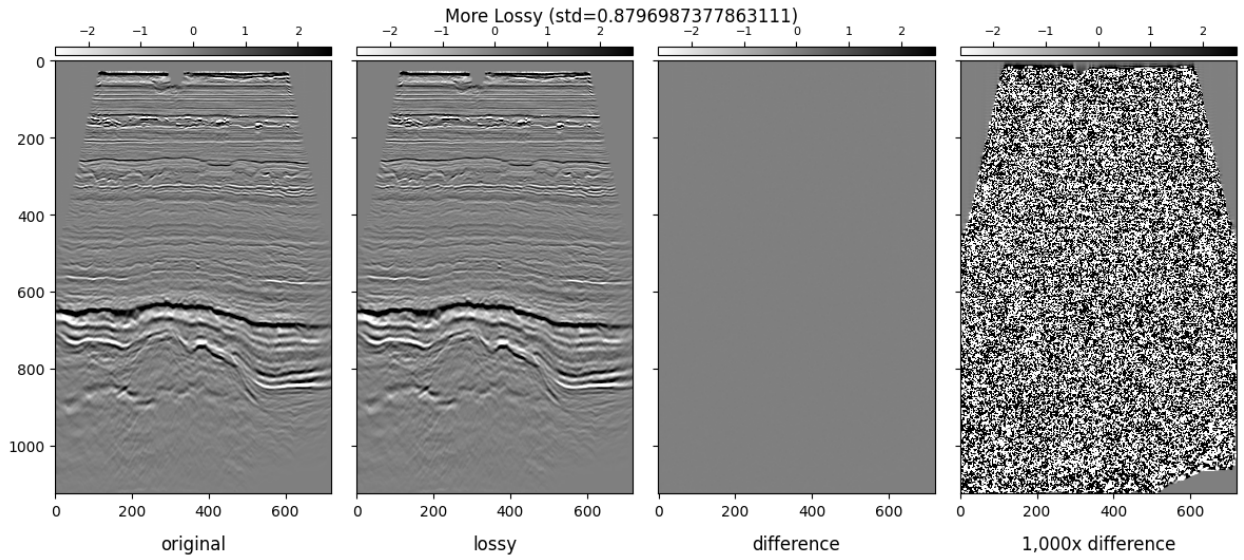
```
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable

vmin = -3 * std
vmax = 3 * std

imshow_kw = dict(vmin=vmin, vmax=vmax, cmap="gray_r", interpolation="bilinear")
```

(continues on next page)

(continued from previous page)

```
def attach_colorbar(image, axis):
    divider = make_axes_locatable(axis)
    cax = divider.append_axes("top", size="2%", pad=0.05)
    plt.colorbar(image, cax=cax, orientation="horizontal")
    cax.xaxis.set_ticks_position("top")
    cax.tick_params(labelsize=8)


def plot_image_and_cbar(data, axis, title):
    image = axis.imshow(data.T, **imshow_kw)
    attach_colorbar(image, axis)
    axis.set_title(title, y=-0.15)


def plot_inlines_with_diff(orig, compressed, title):
    fig, ax = plt.subplots(1, 4, sharey="all", sharex="all", figsize=(12, 5))

    diff = orig[200] - compressed[200]

    plot_image_and_cbar(orig[200], ax[0], "original")
    plot_image_and_cbar(compressed[200], ax[1], "lossy")
    plot_image_and_cbar(diff, ax[2], "difference")
    plot_image_and_cbar(diff * 1_000, ax[3], "1,000x difference")

    plt.suptitle(f"{title} ({std=})")
    fig.tight_layout()

    plt.show()


plot_inlines_with_diff(lossless, lossy, "Default Lossy")
plot_inlines_with_diff(lossless, lossy_plus, "More Lossy")
```





7.3.4 Calculate Metrics

For image quality, there are some metrics used by the broader image compression community.

In this example we will be using the following four metrics as comparison.

1. PSNR: Peak signal-to-noise ratio for an image. (higher is better)
2. SSIM: Mean structural similarity index between two images. (higher is better, maximum value is 1.0)
3. MSE: Compute the mean-squared error between two images. (lower is better)
4. NRMSE: Normalized root mean-squared error between two images. (lower is better)

For PSNR or SSIM, we use the global dataset range `(max - min)` as the normalization method.

In image compression community, a PSNR value above 60 dB (decibels) is considered acceptable.

We calculate these metrics on the same inline we show above.

```
import skimage


def get_metrics(image_true, image_test):
    """Get four metrics"""
    psnr = skimage.metrics.peak_signal_noise_ratio(
        image_true[200], image_test[200], data_range=max_ - min_
    )
    ssim = skimage.metrics.structural_similarity(
        image_true[200], image_test[200], data_range=max_ - min_
    )
    mse = skimage.metrics.mean_squared_error(image_true[200], image_test[200])
    nrmse = skimage.metrics.normalized_root_mse(image_true[200], image_test[200])

    return psnr, ssim, mse, nrmse
```

(continues on next page)

(continued from previous page)

```
print("Lossy", get_metrics(lossless, lossy))
print("Lossy+", get_metrics(lossless, lossy_plus))
```

```
Lossy (106.69280984265322, 0.9999999784224242, 9.176027503792131e-08, 0.
↪000330489434736117)
Lossy+ (66.27609586061718, 0.999721336954417, 0.0010100110026414078, 0.0346731484815586)
```

7.4 Optimizing Access Patterns

7.4.1 Introduction

In this page we will be showing how we can take an existing MDIO and add fast access, lossy, versions of the data in X/Y/Z cross-sections (slices).

We can re-use the MDIO dataset we created in the *Quickstart* page. Please run it first.

We will define our compression levels first. We will use this to adjust the quality of the lossy compression.

```
from enum import Enum

class MdioZfpQuality(float, Enum):
    """Config options for ZFP compression."""

    VERY_LOW = 6
    LOW = 3
    MEDIUM = 1
    HIGH = 0.1
    VERY_HIGH = 0.01
    ULTRA = 0.001
```

We will use the lower level `MDIOAccessor` to open the existing file in write mode that allows us to modify its raw metadata. This can be dangerous, we recommend using only provided tools to avoid data corruption.

We specify the original access pattern of the source data `"012"` with some parameters like caching. For the rechunking, we recommend using the single threaded `"zarr"` backend to avoid race conditions.

We also define a `dict` for common arguments in rechunking.

```
from mdio.api.accessor import MDIOAccessor

mdio_path = "filt_mig.mdio"

orig_mdio_cached = MDIOAccessor(
    mdio_path_or_buffer=mdio_path,
    mode="w",
    access_pattern="012",
    storage_options=None,
    return_metadata=False,
    new_chunks=None,
    backend="zarr",
    memory_cache_size=2**28,
```

(continues on next page)

(continued from previous page)

```
    disk_cache=False,
)
```

7.4.2 Compression (Lossy)

Now, let's define our compression level. The compression ratios vary a lot on the data characteristics. However, the compression levels here are good guidelines that are based on standard deviation of the original data.

We use ZFP's fixed accuracy mode with a tolerance based on data standard deviation, as mentioned above. For more ZFP options you can see its documentation.

Empirically, for this dataset, we see the following size reductions (per copy):

- `10 :  1 on VERY_LOW`
- `7.5 :  1 on LOW`
- `4.5 :  1 on MEDIUM`
- `3 :  1 on HIGH`
- `2 :  1 on VERY_HIGH`
- `1.5 :  1 on ULTRA`

```
from numcodecs import ZFPY
from zfpy import mode_fixed_accuracy

std = orig_mdio_cached.stats["std"]  # standard deviation of original data

quality = MdioZfpQuality.LOW
tolerance = quality * std
sample_compressor = ZFPY(mode_fixed_accuracy, tolerance=tolerance)

common_kwargs = {"overwrite": True, "compressor": sample_compressor}
```

7.4.3 Optimizing IL/XL/Z Independently

In this cell, we will demonstrate how to create IL/XL and Z (two-way-time) optimized versions **independently**. In the next section we will do the same with the batch mode where the data only needs to be read into memory once.

In the example below, each rechunking operation will read the data from the original MDIO dataset and discard it. We did enable 256 MB (2^28 bytes) memory cache above, it will help some, but still not ideal.

```
from mdio.api.convenience import rechunk

rechunk(orig_mdio_cached, (4, 512, 512), suffix="fast_il", **common_kwargs)
rechunk(orig_mdio_cached, (512, 4, 512), suffix="fast_xl", **common_kwargs)
rechunk(orig_mdio_cached, (512, 512, 4), suffix="fast_z", **common_kwargs)
```

```
Rechunking to fast_il: 100%|| 3/3 [00:01<00:00,  1.77chunk/s]
Rechunking to fast_xl: 100%|| 3/3 [00:01<00:00,  1.90chunk/s]
Rechunking to fast_z: 100%|| 3/3 [00:01<00:00,  1.97chunk/s]
```

We can now open the original MDIO dataset and the fast access patterns. When printing the `chunks` attribute, we see the original one first, and the subsequent ones show data is rechunked with ZFP compression.

```
from mdio import MDIOReader

orig_mdio = MDIOReader(mdio_path)
il_mdio = MDIOReader(mdio_path, access_pattern="fast_il")
xl_mdio = MDIOReader(mdio_path, access_pattern="fast_xl")
z_mdio = MDIOReader(mdio_path, access_pattern="fast_z")

print(orig_mdio.chunks, orig_mdio._traces.compressor)
print(il_mdio.chunks, il_mdio._traces.compressor)
print(xl_mdio.chunks, xl_mdio._traces.compressor)
print(z_mdio.chunks, z_mdio._traces.compressor)
```

```
(64, 64, 64) Blosc(cname='zstd', clevel=5, shuffle=SHUFFLE, blocksize=0)
(4, 187, 512) ZFPY(mode=4, tolerance=2.7916183359718256, rate=-1, precision=-1)
(345, 4, 512) ZFPY(mode=4, tolerance=2.7916183359718256, rate=-1, precision=-1)
(345, 187, 4) ZFPY(mode=4, tolerance=2.7916183359718256, rate=-1, precision=-1)
```

We can now compare the sizes of the compressed representations to original.

Below commands are for UNIX based operating systems and won't work on Windows.

```
!du -hs {mdio_path}/data/chunked_012
!du -hs {mdio_path}/data/chunked_fast_il
!du -hs {mdio_path}/data/chunked_fast_xl
!du -hs {mdio_path}/data/chunked_fast_z
```

```
149M        filt_mig.mdio/data/chunked_012
 21M        filt_mig.mdio/data/chunked_fast_il
 20M        filt_mig.mdio/data/chunked_fast_xl
 21M        filt_mig.mdio/data/chunked_fast_z
```

Comparing local disk read speeds for inlines:

```
%timeit orig_mdio[175]  # 3d chunked
%timeit il_mdio[175]  # inline optimized
```

```
31.1 ms ± 825 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
3.6 ms ± 52.9 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

For crosslines:

```
%timeit orig_mdio[:, 90]  # 3d chunked
%timeit xl_mdio[:, 90]  # xline optimized
```

```
65.3 ms ± 705 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
8.76 ms ± 353 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

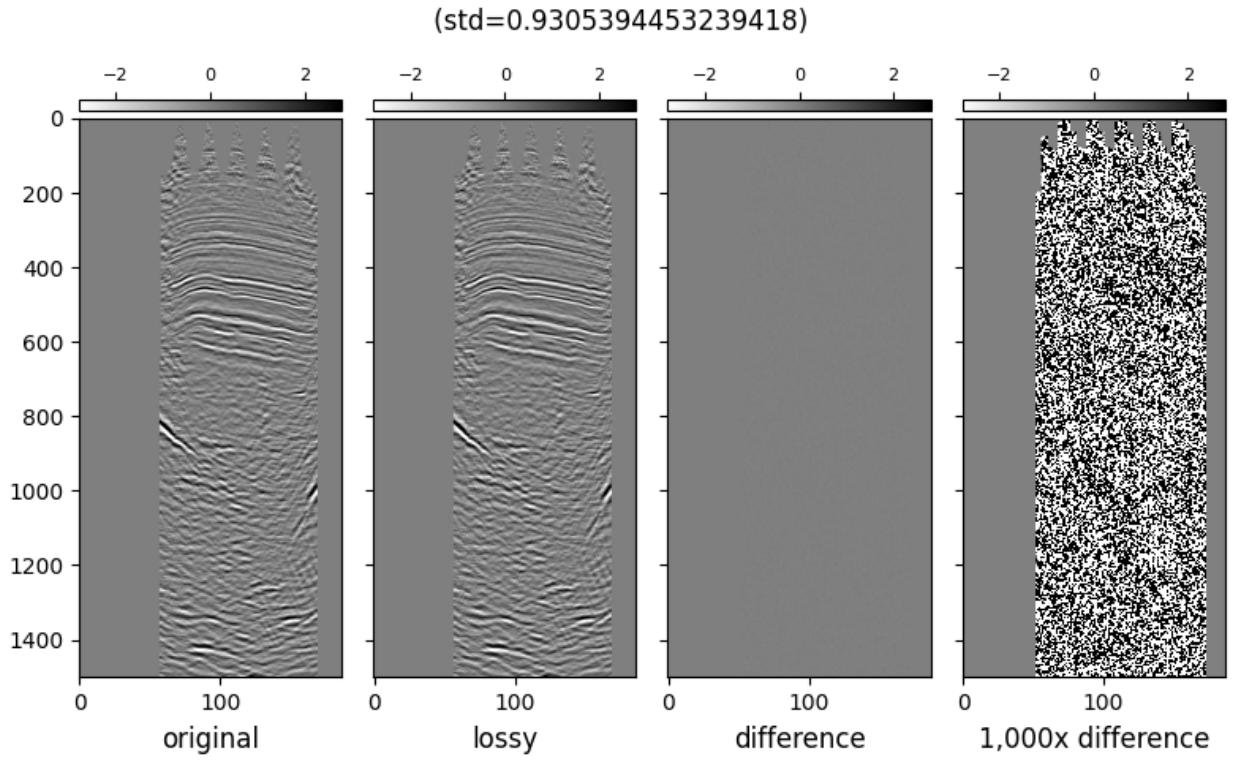
Finally, for Z (time-slices):

```
%timeit orig_mdio[..., 751]  # 3d chunked
%timeit z_mdio[..., 751]  # time-slice optimized
```

```
6.36 ms ± 185 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
872 µs ± 8.24 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

We can check the subjective quality of the compression by visually comparing two inlines. Similar to the example we had in the *Compression* page.

```
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable

vmin = -3 * std
vmax = 3 * std

imshow_kw = dict(vmin=vmin, vmax=vmax, cmap="gray_r", interpolation="bilinear", aspect=
↪"auto")


def attach_colorbar(image, axis):
    divider = make_axes_locatable(axis)
    cax = divider.append_axes("top", size="2%", pad=0.05)
    plt.colorbar(image, cax=cax, orientation="horizontal")
    cax.xaxis.set_ticks_position("top")
    cax.tick_params(labelsize=8)


def plot_image_and_cbar(data, axis, title):
    image = axis.imshow(data.T, **imshow_kw)
    attach_colorbar(image, axis)
    axis.set_title(title, y=-0.15)


def plot_inlines_with_diff(orig, compressed, title):
    fig, ax = plt.subplots(1, 4, sharey="all", sharex="all", figsize=(8, 5))

    diff = orig[200] - compressed[200]

    plot_image_and_cbar(orig[200], ax[0], "original")
    plot_image_and_cbar(compressed[200], ax[1], "lossy")
    plot_image_and_cbar(diff, ax[2], "difference")
    plot_image_and_cbar(diff * 1_000, ax[3], "1,000x difference")

    plt.suptitle(f"{title} ({std=})")
    fig.tight_layout()

    plt.show()


plot_inlines_with_diff(orig_mdio, il_mdio, "")
```



In conclusion, we show that by generating optimized, lossy compressed copies of the data for certain access patterns yield big performance benefits when reading the data.

The differences are orders of magnitude larger on big datasets and remote stores, given available network bandwidth.

7.4.4 Optimizing in Batch

Now that we understand how rechunking and lossy compression works, we will demonstrate how to do this in batches.

The benefit of doing the batched processing is that the dataset gets read once. This is especially important if the original MDIO resides in a remote store like AWS S3, or Google Cloud's GCS.

Note that we not are overwriting the old optimized chunks, just creating new ones with the suffix 2 to demonstrate we can create as many version of the original data as we want.

```
from mdio.api.convenience import rechunk_batch

rechunk_batch(
    orig_mdio_cached,
    chunks_list=[(4, 512, 512), (512, 4, 512), (512, 512, 4)],
    suffix_list=["fast_il2", "fast_xl2", "fast_z2"],
    **common_kwargs,
)
```

```
Rechunking to fast_il2,fast_xl2,fast_z2: 100%|| 3/3 [00:05<00:00,  1.84s/chunk]
```

```
from mdio import MDIOReader
```

(continues on next page)

(continued from previous page)

```
orig_mdio = MDIOReader(mdio_path)
il2_mdio = MDIOReader(mdio_path, access_pattern="fast_il2")
xl2_mdio = MDIOReader(mdio_path, access_pattern="fast_xl2")
z2_mdio = MDIOReader(mdio_path, access_pattern="fast_z2")

print(orig_mdio.chunks, orig_mdio._traces.compressor)
print(il_mdio.chunks, il2_mdio._traces.compressor)
print(xl_mdio.chunks, xl2_mdio._traces.compressor)
print(z_mdio.chunks, z2_mdio._traces.compressor)
```

```
(64, 64, 64) Blosc(cname='zstd', clevel=5, shuffle=SHUFFLE, blocksize=0)
(4, 187, 512) ZFPY(mode=4, tolerance=2.7916183359718256, rate=-1, precision=-1)
(345, 4, 512) ZFPY(mode=4, tolerance=2.7916183359718256, rate=-1, precision=-1)
(345, 187, 4) ZFPY(mode=4, tolerance=2.7916183359718256, rate=-1, precision=-1)
```

7.5 Usage

7.5.1 Ingestion and Export

The following example shows how to minimally ingest a 3D seismic stack into a local ***MDIO*** file. Only one lossless copy will be made.

There are many more options, please see the *CLI Reference*.

```
$ mdio segy import \
    path_to_segy_file.segy \
    path_to_mdio_file.mdio \
    -loc 181,185 \
    -names inline,crossline
```

To export the same file back to SEG-Y format, the following command should be executed.

```
$ mdio segy export \
    path_to_mdio_file.mdio \
    path_to_segy_file.segy
```

7.5.2 Cloud Connection Strings

MDIO supports I/O on major cloud service providers. The cloud I/O capabilities are supported using the fsspec and its specialized version for:

- Amazon Web Services (AWS S3) - s3fs
- Google Cloud Provider (GCP GCS) - gcsfs
- Microsoft Azure (Datalake Gen2) - adlfs

Any other file-system supported by `fsspec` will also be supported by ***MDIO***. However, we will focus on the major providers here.

The protocols that help choose a backend (i.e. `s3://`, `gs://`, or `az://`) can be passed prepended to the ***MDIO*** path.

The connection string can be passed to the command-line-interface (CLI) using the `-storage, --storage-options` flag as a JSON string or the Python API with the `storage_options` keyword argument as a Python dictionary.

Warning: On Windows clients, JSON strings are passed to the CLI with a special escape character.

For instance a JSON string:

```
{"key": "my_super_private_key", "secret": "my_super_private_secret"}
```

must be passed with an escape character \ for inner quotes as:

```
"{\"key\": \"my_super_private_key\", \"secret\": \"my_super_private_secret\"}"
```

whereas, on Linux bash this works just fine:

```
'{"key": "my_super_private_key", "secret": "my_super_private_secret"}'
```

If this done incorrectly, you will get an invalid JSON string error from the CLI.

Amazon Web Services

Credentials can be automatically fetched from pre-authenticated AWS CLI. See here for the order `s3fs` checks them. If it is not pre-authenticated, you need to pass `--storage-options`.

Prefix:
`s3://`

Storage Options:
`key`: The auth key from AWS
`secret`: The auth secret from AWS

Using UNIX:

```
mdio segy import \
  path/to/my.segy \
  s3://bucket/prefix/my.mdio \
  --header-locations 189,193 \
  --storage-options '{"key": "my_super_private_key", "secret": "my_super_private_secret"}
↪'
```

Using Windows (note the extra escape characters \):

```
mdio segy import \
  path/to/my.segy \
  s3://bucket/prefix/my.mdio \
  --header-locations 189,193 \
  --storage-options "{\"key\": \"my_super_private_key\", \"secret\": \"my_super_private_
↪secret\"}"
```

Google Cloud Provider

Credentials can be automatically fetched from pre-authenticated `gcloud` CLI. See here for the order `gcsfs` checks them. If it is not pre-authenticated, you need to pass `--storage-options`.

GCP uses service accounts to pass authentication information to APIs.

Prefix:
`gs://` or `gcs://`

Storage Options:
`token`: The service account JSON value as string, or local path to JSON

Using a service account:

```
mdio segy import \
  path/to/my.segy \
  gs://bucket/prefix/my.mdio \
  --header-locations 189,193 \
  --storage-options '{"token": "~/.config/gcloud/application_default_credentials.json"}'
```

Using browser to populate authentication:

```
mdio segy import \
  path/to/my.segy \
  gs://bucket/prefix/my.mdio \
  --header-locations 189,193 \
  --storage-options '{"token": "browser"}'
```

Microsoft Azure

There are various ways to authenticate with Azure Data Lake (ADL). See here for some details. If ADL is not pre-authenticated, you need to pass `--storage-options`.

Prefix:
`az://` or `abfs://`

Storage Options:
`account_name`: Azure Data Lake storage account name
`account_key`: Azure Data Lake storage account access key

```
mdio segy import \
  path/to/my.segy \
  az://bucket/prefix/my.mdio \
  --header-locations 189,193 \
  --storage-options '{"account_name": "myaccount", "account_key": "my_super_private_key"}
↪'
```

Advanced Cloud Features

There are additional functions provided by `fsspec`. These are advanced features and we refer the user to read `fsspec` documentation. Some useful examples are:

- Caching Files Locally
- Remote Write Caching
- File Buffering and random access
- Mount anything with FUSE

Note: When combining advanced protocols like `simplecache` and using a remote store like `s3` the URL can be chained like `simplecache::s3://bucket/prefix/file.mdio`. When doing this the `--storage-options` argument must explicitly state parameters for the cloud backend and the extra protocol. For the above example it would look like this:

```
{
  "s3": {
    "key": "my_super_private_key",
    "secret": "my_super_private_secret"
  },
  "simplecache": {
    "cache_storage": "/custom/temp/storage/path"
  }
}
```

In one line:

```
{"s3": {"key": "my_super_private_key", "secret": "my_super_private_secret"}, "simplecache
↪": {"cache_storage": "/custom/temp/storage/path"}
```

7.5.3 CLI Reference

MDIO provides a convenient command-line-interface (CLI) to do various tasks.

For each command / subcommand you can provide `--help` argument to get information about usage.

mdio

Welcome to MDIO!

MDIO is an open source, cloud-native, and scalable storage engine for various types of energy data.

MDIO supports importing or exporting various data containers, hence we allow plugins as subcommands.

From this main command, we can see the MDIO version.

```
mdio [OPTIONS] COMMAND [ARGS]...
```

Options

--version
Show the version and exit.

copy

Copy a MDIO dataset to anpther MDIO dataset.

Can also copy with empty data to be filled later. See *excludes* and *includes* parameters.

More documentation about *excludes* and *includes* can be found in Zarr's documentation in *zarr.convenience.copy_store*.

```
mdio copy [OPTIONS] SOURCE_MDIO_PATH TARGET_MDIO_PATH
```

Options

-access, --access-pattern <access_pattern>
Access pattern of the file

Default
012

-exc, --excludes <excludes>
Data to exclude during copy, like *chunked_012*. The data values won't be copied but an empty array will be created. If blank, it copies everything.

-inc, --includes <includes>
Data to include during copy, like *trace_headers*. If not specified, and certain data is excluded, it will not copy headers. To preserve headers, specify trace_headers. If left blank, it will copy everything except what is specified in the 'excludes' parameter.

-storage, --storage-options <storage_options>
Custom storage options for cloud backends

-overwrite, --overwrite
Flag to overwrite if mdio file if it exists

Default
False

Arguments

SOURCE_MDIO_PATH
Required argument

TARGET_MDIO_PATH
Required argument

info

Provide information on a MDIO dataset.

By default, this returns human-readable information about the grid and stats for the dataset. If output-format is set to json then a json is returned to facilitate parsing.

```
mdio info [OPTIONS] MDIO_PATH
```

Options

-access, --access-pattern <access_pattern>

Access pattern of the file

Default
012

-format, --output-format <output_format>

Output format. Pretty console or JSON.

Default
pretty

Options
pretty | json

Arguments

MDIO_PATH

Required argument

segy

MDIO and SEG-Y conversion utilities. Below is general information about the SEG-Y format and MDIO features. For import or export specific functionality check the import or export modules:

mdio segy import –help
mdio segy export –help

MDIO can import SEG-Y files to a modern, chunked format.

The SEG-Y format is defined by the Society of Exploration Geophysicists as a data transmission format and has its roots back to 1970s. There are currently multiple revisions of the SEG-Y format.

MDIO can unravel and index any SEG-Y file that is on a regular grid. There is no limitation to dimensionality of the data, as long as it can be represented on a regular grid. Most seismic surveys are on a regular grid of unique shot/receiver IDs or are imaged on regular CDP or INLINE/CROSSLINE grids.

The SEG-Y headers are used as identifiers to take the flattened SEG-Y data and convert it to the multi-dimensional tensor representation. An example of ingesting a 3-D Post-Stack seismic data can be though as the following, per the SEG-Y Rev1 standard:

–header-names inline,crossline
–header-locations 189,193
–header-types int32,int32

Our recommended chunk sizes are:
(Based on GCS benchmarks)
3D: 64 x 64 x 64
2D: 512 x 512

The 4D+ datasets chunking recommendation depends on the type of 4D+ dataset (i.e. SHOT vs CDP data will have different chunking).

MDIO also import or export big and little endian coded IBM or IEEE floating point formatted SEG-Y files. MDIO can also build a grid from arbitrary header locations for indexing. However, the headers are stored as the SEG-Y Rev 1 after ingestion.

```
mdio segy [OPTIONS] COMMAND [ARGS]...
```

export

Export MDIO file to SEG-Y.

SEG-Y format is explained in the "segy" group of the command line interface. To see additional information run:

mdio segy –help

MDIO allows exporting multidimensional seismic data back to the flattened seismic format SEG-Y, to be used in data transmission.

The input headers are preserved as is, and will be transferred to the output file.

The user has control over the endianness, and the floating point data type. However, by default we export as Big-Endian IBM float, per the SEG-Y format's default.

The input MDIO can be local or cloud based. However, the output SEG-Y will be generated locally.

```
mdio segy export [OPTIONS] MDIO_FILE SEGY_PATH
```

Options

-access, --access-pattern <access_pattern>

Access pattern of the file

Default
012

-format, --segy-format <segy_format>

SEG-Y sample format

Default
ibm32

Options
ibm32 | ieee32

-storage, --storage-options <storage_options>

Custom storage options for cloud backends.

-endian, --endian <endian>

Endianness of the SEG-Y file

Default
big

Options
little | big

Arguments

MDIO_FILE

Required argument

SEGY_PATH

Required argument

import

Ingest SEG-Y file to MDIO.

SEG-Y format is explained in the "segy" group of the command line interface. To see additional information run:

mdio segy –help

MDIO allows ingesting flattened seismic surveys in SEG-Y format into a multidimensional tensor that represents the correct geometry of the seismic dataset.

The SEG-Y file must be on disk, MDIO currently does not support reading SEG-Y directly from the cloud object store.

The output MDIO file can be local or on the cloud. For local files, a UNIX or Windows path is sufficient. However, for cloud stores, an appropriate protocol must be provided. Some examples:

File Path Patterns:

If we are working locally: –input_segy_path local_seismic.segy –output-mdio-path local_seismic.mdio

If we are working on the cloud on Amazon Web Services: –input_segy_path local_seismic.segy –output-mdio-path s3://bucket/local_seismic.mdio

If we are working on the cloud on Google Cloud: –input_segy_path local_seismic.segy –output-mdio-path gs://bucket/local_seismic.mdio

If we are working on the cloud on Microsoft Azure: –input_segy_path local_seismic.segy –output-mdio-path abfs://bucket/local_seismic.mdio

The SEG-Y headers for indexing must also be specified. The index byte locations (starts from 1) are the minimum amount of information needed to index the file. However, we suggest giving names to the index dimensions, and if needed providing the header types if they are not standard. By default, all header entries are assumed to be 4-byte long (int32).

The chunk size depends on the data type, however, it can be chosen to accommodate any workflow's access patterns. See examples below for some common use cases.

By default, the data is ingested with LOSSLESS compression. This saves disk space in the range of 20% to 40%. MDIO also allows data to be compressed using the ZFP compressor's fixed accuracy lossy compression. If lossless parameter is set to False and MDIO was installed using the lossy extra; then the data will be compressed to approximately

30% of its original size and will be perceptually lossless. The compression amount can be adjusted using the option compression_tolerance (float). Values less than 1 gives good results. The higher the value, the more compression, but will introduce artifacts. The default value is 0.01 tolerance, however we get good results up to 0.5; where data is almost compressed to 10% of its original size. NOTE: This assumes data has amplitudes normalized to have approximately standard deviation of 1. If dataset has values smaller than this tolerance, a lot of loss may occur.

Usage:

Below are some examples of ingesting standard SEG-Y files per the SEG-Y Revision 1 and 2 formats.

3D Seismic Post-Stack: Chunks: 128 inlines x 128 crosslines x 128 samples –header-locations 189,193 –header-names inline,crossline

3D Seismic Imaged Pre-Stack Gathers: Chunks: 16 inlines x 16 crosslines x 16 offsets x 512 samples –header-locations 189,193,37 –header-names inline,crossline,offset –chunk-size 16,16,16,512

2D Seismic Shot Data (Byte Locations Vary): Chunks: 16 shots x 256 channels x 512 samples –header-locations 9,13 –header-names shot,chan –chunk-size 16,256,512

3D Seismic Shot Data (Byte Locations Vary): Let's assume streamer number is at byte 213 as a 2-byte integer field. Chunks: 8 shots x 2 cables x 256 channels x 512 samples –header-locations 9,213,13 –header-names shot,cable,chan –header-types int32,int16,int32 –chunk-size 8,2,256,512

We can override the dataset grid by the *grid_overrides* parameter. This allows us to ingest files that don't conform to the true geometry of the seismic acquisition.

For example if we are ingesting 3D seismic shots that don't have a cable number and channel numbers are sequential (i.e. each cable doesn't start with channel number 1; we can tell MDIO to ingest this with the correct geometry by calculating cable numbers and wrapped channel numbers. Note the missing byte location and type for the "cable" index.

Usage:

3D Seismic Shot Data (Byte Locations Vary): Let's assume streamer number does not exist but there are 800 channels per cable. Chunks: 8 shots x 2 cables x 256 channels x 512 samples –header-locations 9,None,13 –header-names shot,cable,chan –header-types int32,None,int32 –chunk-size 8,2,256,512 –grid-overrides '{"ChannelWrap": True, "ChannelsPerCable": 800,

"CalculateCable": True}'

If we do have cable numbers in the headers, but channels are still sequential (aka. unwrapped), we can still ingest it like this. –header-locations 9,213,13 –header-names shot,cable,chan –header-types int32,int16,int32 –chunk-size 8,2,256,512 –grid-overrides '{"ChannelWrap":True, "ChannelsPerCable": 800}' For shot gathers with channel numbers and wrapped channels, no grid overrides are necessary.

In cases where the user does not know if the input has unwrapped channels but desires to store with wrapped channel index use: –grid-overrides '{"AutoChannelWrap": True}'

For cases with no well-defined trace header for indexing a NonBinned grid override is provided.This creates the index and attributes an incrementing integer to the trace for the index based on first in first out. For example a CDP and Offset keyed file might have a header for offset as real world offset which would result in a very sparse populated index. Instead, the following override will create a new index from 1 to N, where N is the number of offsets within a CDP ensemble. The index to be auto generated is called "trace". Note the required "chunksize" parameter in the grid override. This is due to the non-binned ensemble chunksize is irrelevant to the index dimension chunksizes and has to be specified in the grid override itself. Note the lack of offset, only indexing CDP, providing CDP header type, and chunksize for only CDP and Sample dimension. The chunksize for non-binned dimension is in the grid overrides as described above. The below configuration will yield 1MB chunks. –header-locations 21 –header-names cdp –header-types int32 –chunk-size 4,1024 –grid-overrides '{"NonBinned": True, "chunksize": 64}'

A more complicated case where you may have a 5D dataset that is not binned in Offset and Azimuth directions can be ingested like below. However, the Offset and Azimuth dimensions will be combined to "trace" dimension.

The below configuration will yield 1MB chunks. –header-locations 189,193 –header-names inline,crossline –header-types int32,int32 –chunk-size 4,4,1024 –grid-overrides '{"NonBinned": True, "chunksize": 16}'

For dataset with expected duplicate traces we have the following parameterization. This will use the same logic as NonBinned with a fixed chunksize of 1. The other keys are still important. The below example allows multiple traces per receiver (i.e. reshoot). –header-locations 9,213,13 –header-names shot,cable,chan –header-types int32,int16,int32 –chunk-size 8,2,256,512 –grid-overrides '{"HasDuplicates": True}'

```
mdio segy import [OPTIONS] SEGY_PATH MDIO_PATH
```

Options

-loc, --header-locations <header_locations>

Required Byte locations of the index attributes in SEG-Y trace header.

-types, --header-types <header_types>

Data types of the index attributes in SEG-Y trace header.

-names, --header-names <header_names>

Names of the index attributes

-chunks, --chunk-size <chunk_size>

Custom chunk size for bricked storage

-endian, --endian <endian>

Endianness of the SEG-Y file

Default
`big`

Options
little | big

-lossless, --lossless <lossless>

Toggle lossless, and perceptually lossless compression

Default
`True`

-tolerance, --compression-tolerance <compression_tolerance>

Lossy compression tolerance in ZFP.

Default
`0.01`

-storage, --storage-options <storage_options>

Custom storage options for cloud backends

-overwrite, --overwrite

Flag to overwrite if mdio file if it exists

Default
`False`

-grid-overrides, --grid-overrides <grid_overrides>

Option to add grid overrides.

Arguments

SEGY_PATH

Required argument

MDIO_PATH

Required argument

7.6 Reference

7.6.1 Readers / Writers

MDIO accessor APIs.

class mdio.api.accessor.**MDIOAccessor**(*mdio_path_or_buffer*, *mode*, *access_pattern*, *storage_options*, *return_metadata*, *new_chunks*, *backend*, *memory_cache_size*, *disk_cache*)

Accessor class for MDIO files.

The accessor can be used to read and write MDIO files. It allows you to open an MDIO file in several *mode* and *access_pattern* combinations.

Access pattern defines the dimensions that are chunked. For instance if you have a 3-D array that is chunked in every direction (i.e. a 3-D seismic stack consisting of inline, crossline, and sample dimensions) its access pattern would be "012". If it was only chunked in the first two dimensions (i.e. seismic inline and crossline), it would be "01".

By default, MDIO will try to open with "012" access pattern, and will raise an error if that pattern doesn't exist.

After dataset is opened, when the accessor is sliced it will either return just seismic trace data as a Numpy array or a tuple of live mask, headers, and seismic trace in Numpy based on the parameter *return_metadata*.

Regarding object store access, if the user credentials have been set system-wide on local machine or VM; there is no need to specify credentials. However, the *storage_options* option allows users to specify credentials for the store that is being accessed. Please see the *fsspec* documentation for configuring storage options.

MDIO currently supports *Zarr* and *Dask* backends. The Zarr backend is useful for reading small amounts of data with minimal overhead. However, by utilizing the *Dask* backend with a larger chunk size using the *new_chunks* argument, the data can be read in parallel using a Dask LocalCluster or a distributed Cluster.

The accessor also allows users to enable *fsspec* caching. These are particularly useful when we are accessing the data from a high-latency store such as object stores, or mounted network drives with high latency. We can use the *disk_cache* option to fetch chunks the local temporary directory for faster repetitive access. We can also turn on the Least Recently Used (LRU) cache by using the *memory_cache* option. It has to be specified in bytes.

Parameters

- **mdio_path_or_buffer** (*str*) – Store URL for MDIO file. This can be either on a local disk, or a cloud object store.
- **mode** (*str*) – Read or read/write mode. The file must exist. Options are in {'r', 'r+', 'w'}. 'r' is read only, 'r+' is append mode where only existing arrays can be modified, 'w' is similar to 'r+' but rechunking or other file-wide operations are allowed.
- **access_pattern** (*str*) – Chunk access pattern, optional. Default is "012". Examples: '012', '01', '01234'.

- **storage_options** (*dict | None*) – Options for the storage backend. By default, system-wide credentials will be used. If system-wide credentials are not set and the source is not public, an authentication error will be raised by the backend.
- **return_metadata** (*bool*) – Flag for returning live mask, headers, and traces or just the trace data. Default is False, which means just trace data will be returned.
- **new_chunks** (*tuple[int, ...] | None*) – Chunk sizes used in Dask backend. Ignored for Zarr backend. By default, the disk-chunks will be used. However, if we want to stream groups of chunks to a Dask worker, we can rechunk here. Then each Dask worker can asynchronously fetch multiple chunks before working.
- **backend** (*str*) – Backend selection, optional. Default is "zarr". Must be in {'zarr', 'dask'}.
- **memory_cache_size** (*int*) – Maximum, in memory, least recently used (LRU) cache size in bytes.
- **disk_cache** (*bool*) – Disk cache implemented by *fsspec*, optional. Default is False, which turns off disk caching. See *simplecache* from *fsspec* documentation for more details.

Raises

MDIONotFoundError – If the MDIO file can not be opened.

Notes

The combination of the *Dask* backend and caching schemes are experimental. This configuration may cause unexpected memory usage and duplicate data fetching.

Examples

Assuming we ingested *my_3d_seismic.segy* as *my_3d_seismic.mdio* we can open the file in read-only mode like this.

```
>>> from mdio import MDIOReader
>>>
>>>
>>> mdio = MDIOReader("my_3d_seismic.mdio")
```

This will open the file with the lazy *Zarr* backend. To access a specific inline, crossline, or sample index we can do:

```
>>> inline = mdio[15]  # get the 15th inline
>>> crossline = mdio[:, 15]  # get the 50th crossline
>>> samples = mdio[..., 250]  # get the 250th sample slice
```

The above will variables will be Numpy arrays of the relevant trace data. If we want to retreive the live mask and trace headers for our sliding we need to open the file with the *return_metadata* option.

```
>>> mdio = MDIOReader("my_3d_seismic.mdio", return_metadata=True)
```

Then we can fetch the data like this (for inline):

```
>>> il_live, il_headers, il_traces = mdio[15]
```

Since MDIOAccessor returns a tuple with these three Numpy arrays, we can directly unpack it and use it further down our code.

Accessor initialization function.

coord_to_index(**args*, *dimensions=None*)

Convert dimension coordinate to zero-based index.

The coordinate labels of the array dimensions are converted to zero-based indices. For instance if we have an inline dimension like this:

[10, 20, 30, 40, 50]

then the indices would be:

[0, 1, 2, 3, 4]

This method converts from coordinate labels of a dimension to equivalent indices.

Multiple dimensions can be queried at the same time, see the examples.

Parameters

- ***args** – Variable length argument queries. # noqa: RST213
- **dimensions** (*str | list[str] | None*) – Name of the dimensions to query. If not provided, it will query all dimensions in the grid and will require *len(args) == grid.ndim*

Returns

Zero-based indices of coordinates. Each item in result corresponds to indicies of that dimension

Raises

- **ShapeError** – if number of queries don't match requested dimensions.
- **ValueError** – if requested coordinates don't exist.

Return type

tuple[*ndarray*[*Any*, *dtype*[int]], ...]

Examples

Opening an MDIO file.

```
>>> from mdio import MDIOReader
>>>
>>>
>>> mdio = MDIOReader("path_to.mdio")
>>> mdio.coord_to_index([10, 7, 15], dimensions='inline')
array([ 8,  5, 13], dtype=uint16)
```

```
>>> ils, xls = [10, 7, 15], [5, 10]
>>> mdio.coord_to_index(ils, xls, dimensions=['inline', 'crossline'])
(array([ 8,  5, 13], dtype=uint16), array([3, 8], dtype=uint16))
```

With the above indices, we can slice the data:

```
>>> mdio[ils]  # only inlines
>>> mdio[:, xls]  # only crosslines
>>> mdio[ils, xls]  # intersection of the lines
```

Note that some fancy-indexing may not work with Zarr backend. The Dask backend is more flexible when it comes to indexing.

If we are querying all dimensions of a 3D array, we can omit the *dimensions* argument.

```
>>> mdio.coord_to_index(10, 5, [50, 100])
(array([8], dtype=uint16),
 array([3], dtype=uint16),
 array([25, 50], dtype=uint16))
```

copy(*dest_path_or_buffer*, *excludes=''*, *includes=''*, *storage_options=None*, *overwrite=False*)

Makes a copy of an MDIO file with or without all arrays.

Refer to mdio.api.convenience.copy for full documentation.

Parameters

- **dest_path_or_buffer** (*str*) – Destination path. Could be any FSSpec mapping.
- **excludes** (*str*) – Data to exclude during copy. i.e. *chunked_012*. The raw data won't be copied, but it will create an empty array to be filled. If left blank, it will copy everything.
- **includes** (*str*) – Data to include during copy. i.e. *trace_headers*. If this is not specified, and certain data is excluded, it will not copy headers. If you want to preserve headers, specify *trace_headers*. If left blank, it will copy everything except specified in *excludes* parameter.
- **storage_options** (*dict | None*) – Storage options for the cloud storage backend. Default is None (will assume anonymous).
- **overwrite** (*bool*) – Overwrite destination or not.

property binary_header: dict

Get seismic binary header metadata.

property chunks: tuple[int, ...]

Get dataset chunk sizes.

property live_mask: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | Array

Get live mask (i.e. not-null value mask).

property n_dim: int

Get number of dimensions for dataset.

property shape: tuple[int, ...]

Get shape of dataset.

property stats: dict

Get global statistics like min/max/rms/std.

property text_header: list

Get seismic text header.

property trace_count: int

Get trace count from seismic MDIO.

class mdio.api.accessor.**MDIOReader**(*mdio_path_or_buffer*, *access_pattern='012'*, *storage_options=None*, *return_metadata=False*, *new_chunks=None*, *backend='zarr'*, *memory_cache_size=0*, *disk_cache=False*)

Read-only accessor for MDIO files.

For detailed documentation see MDIOAccessor.

Parameters

- **mdio_path_or_buffer** (*str*) – Store URL for MDIO file. This can be either on a local disk, or a cloud object store.
- **access_pattern** (*str*) – Chunk access pattern, optional. Default is "012". Examples: '012', '01', '01234'.
- **storage_options** (*dict*) – Options for the storage backend. By default, system-wide credentials will be used. If system-wide credentials are not set and the source is not public, an authentication error will be raised by the backend.
- **return_metadata** (*bool*) – Flag for returning live mask, headers, and traces or just the trace data. Default is False, which means just trace data will be returned.
- **new_chunks** (*tuple[int, ...]*) – Chunk sizes used in Dask backend. Ignored for Zarr backend. By default, the disk-chunks will be used. However, if we want to stream groups of chunks to a Dask worker, we can rechunk here. Then each Dask worker can asynchronously fetch multiple chunks before working.
- **backend** (*str*) – Backend selection, optional. Default is "zarr". Must be in {'zarr', 'dask'}.
- **memory_cache_size** (*int*) – Maximum, in memory, least recently used (LRU) cache size in bytes.
- **disk_cache** (*bool*) – Disk cache implemented by *fsspec*, optional. Default is False, which turns off disk caching. See *simplecache* from *fsspec* documentation for more details.

Initialize super class with *r* permission.

class mdio.api.accessor.**MDIOWriter**(*mdio_path_or_buffer*, *access_pattern='012'*, *storage_options=None*, *return_metadata=False*, *new_chunks=None*, *backend='zarr'*, *memory_cache_size=0*, *disk_cache=False*)

Writable accessor for MDIO files.

For detailed documentation see MDIOAccessor.

Parameters

- **mdio_path_or_buffer** (*str*) – Store URL for MDIO file. This can be either on a local disk, or a cloud object store.
- **access_pattern** (*str*) – Chunk access pattern, optional. Default is "012". Examples: '012', '01', '01234'.
- **storage_options** (*dict*) – Options for the storage backend. By default, system-wide credentials will be used. If system-wide credentials are not set and the source is not public, an authentication error will be raised by the backend.
- **return_metadata** (*bool*) – Flag for returning live mask, headers, and traces or just the trace data. Default is False, which means just trace data will be returned.
- **new_chunks** (*tuple[int, ...]*) – Chunk sizes used in Dask backend. Ignored for Zarr backend. By default, the disk-chunks will be used. However, if we want to stream groups of chunks to a Dask worker, we can rechunk here. Then each Dask worker can asynchronously fetch multiple chunks before working.

- **backend** (*str*) – Backend selection, optional. Default is "zarr". Must be in {'zarr', 'dask'}.
- **memory_cache_size** (*int*) – Maximum, in memory, least recently used (LRU) cache size in bytes.
- **disk_cache** (*bool*) – Disk cache implemented by *fsspec*, optional. Default is False, which turns off disk caching. See *simplecache* from *fsspec* documentation for more details.

Initialize super class with *r+* permission.

7.6.2 Data Converters

Seismic Data

Note: By default, the SEG-Y ingestion tool uses Python's multiprocessing to speed up parsing the data. This almost always requires a `__main__` guard on any other Python code that is executed directly like `python file.py`. When running inside Jupyter, this is **NOT** needed.

```
if __name__ == "__main__":
    segy_to_mdio(...)
```

When the CLI is invoked, this is already handled.

See the official `multiprocessing` documentation here and here.

Conversion from SEG-Y to MDIO.

`mdio.converters.segy.segy_to_mdio`(*segy_path*, *mdio_path_or_buffer*, *index_bytes*, *index_names=None*, *index_types=None*, *chunksize=None*, *endian='big'*, *lossless=True*, *compression_tolerance=0.01*, *storage_options=None*, *overwrite=False*, *grid_overrides=None*)

Convert SEG-Y file to MDIO format.

MDIO allows ingesting flattened seismic surveys in SEG-Y format into a multidimensional tensor that represents the correct geometry of the seismic dataset.

The SEG-Y file must be on disk, MDIO currently does not support reading SEG-Y directly from the cloud object store.

The output MDIO file can be local or on the cloud. For local files, a UNIX or Windows path is sufficient. However, for cloud stores, an appropriate protocol must be provided. See examples for more details.

The SEG-Y headers for indexing must also be specified. The index byte locations (starts from 1) are the minimum amount of information needed to index the file. However, we suggest giving names to the index dimensions, and if needed providing the header lengths if they are not standard. By default, all header entries are assumed to be 4-byte long.

The chunk size depends on the data type, however, it can be chosen to accommodate any workflow's access patterns. See examples below for some common use cases.

By default, the data is ingested with LOSSLESS compression. This saves disk space in the range of 20% to 40%. MDIO also allows data to be compressed using the ZFP compressor's fixed rate lossy compression. If lossless parameter is set to False and MDIO was installed using the lossy extra; then the data will be compressed to approximately 30% of its original size and will be perceptually lossless. The compression ratio can be adjusted using the option compression_ratio (integer). Higher values will compress more, but will introduce artifacts.

Parameters

- **segy_path** (*str*) – Path to the input SEG-Y file
- **mdio_path_or_buffer** (*str*) – Output path for MDIO file
- **index_bytes** (*Sequence[int]*) – Tuple of the byte location for the index attributes
- **index_names** (*Sequence[str] | None*) – Tuple of the index names for the index attributes
- **index_types** (*Sequence[str] | None*) – Tuple of the data-types for the index attributes. Must be in {"int16, int32, float16, float32, ibm32"} Default is 4-byte integers for each index key.
- **chunksize** (*Sequence[int] | None*) – Override default chunk size, which is (64, 64, 64) if 3D, and (512, 512) for 2D.
- **endian** (*str*) – Endianness of the input SEG-Y. Rev.2 allows little endian. Default is 'big'. Must be in *{"big", "little"}*
- **lossless** (*bool*) – Lossless Blosc with zstandard, or ZFP with fixed precision.
- **compression_tolerance** (*float*) – Tolerance ZFP compression, optional. The fixed accuracy mode in ZFP guarantees there won't be any errors larger than this value. The default is 0.01, which gives about 70% reduction in size. Will be ignored if *lossless=True*.
- **storage_options** (*dict[str, Any] | None*) – Storage options for the cloud storage backend. Default is *None* (will assume anonymous)
- **overwrite** (*bool*) – Toggle for overwriting existing store
- **grid_overrides** (*dict | None*) – Option to add grid overrides. See examples.

Raises

- **GridTraceCountError** – Raised if grid won't hold all traces in the SEG-Y file.
- **ValueError** – If length of chunk sizes don't match number of dimensions.
- **NotImplementedError** – If can't determine chunking automatically for 4D+.

Return type
None

Examples

If we are working locally and ingesting a 3D post-stack seismic file, we can use the following example. This will ingest with default chunks of 128 x 128 x 128.

```
>>> from mdio import segy_to_mdio
>>>
>>>
>>> segy_to_mdio(
...     segy_path="prefix1/file.segy",
...     mdio_path_or_buffer="prefix2/file.mdio",
...     index_bytes=(189, 193),
...     index_names=("inline", "crossline")
... )
```

If we are on Amazon Web Services, we can do it like below. The protocol before the URL can be *s3* for AWS, *gcs* for Google Cloud, and *abfs* for Microsoft Azure. In this example we also change the chunk size as a demonstration.

```
>>> segy_to_mdio(
...     segy_path="prefix/file.segy",
...     mdio_path_or_buffer="s3://bucket/file.mdio",
...     index_bytes=(189, 193),
...     index_names=("inline", "crossline"),
...     chunksize=(64, 64, 512),
... )
```

Another example of loading a 4D seismic such as 3D seismic pre-stack gathers is below. This will allow us to extract offset planes efficiently or run things in a local neighborhood very efficiently.

```
>>> segy_to_mdio(
...     segy_path="prefix/file.segy",
...     mdio_path_or_buffer="s3://bucket/file.mdio",
...     index_bytes=(189, 193, 37),
...     index_names=("inline", "crossline", "offset"),
...     chunksize=(16, 16, 16, 512),
... )
```

We can override the dataset grid by the *grid_overrides* parameter. This allows us to ingest files that don't conform to the true geometry of the seismic acquisition.

For example if we are ingesting 3D seismic shots that don't have a cable number and channel numbers are sequential (i.e. each cable doesn't start with channel number 1; we can tell MDIO to ingest this with the correct geometry by calculating cable numbers and wrapped channel numbers. Note the missing byte location and word length for the "cable" index.

```
>>> segy_to_mdio(
...     segy_path="prefix/shot_file.segy",
...     mdio_path_or_buffer="s3://bucket/shot_file.mdio",
...     index_bytes=(17, None, 13),
...     index_lengths=(4, None, 4),
...     index_names=("shot", "cable", "channel"),
...     chunksize=(8, 2, 128, 1024),
...     grid_overrides={
...         "ChannelWrap": True, "ChannelsPerCable": 800,
...         "CalculateCable": True
...     },
... )
```

If we do have cable numbers in the headers, but channels are still sequential (aka. unwrapped), we can still ingest it like this.

```
>>> segy_to_mdio(
...     segy_path="prefix/shot_file.segy",
...     mdio_path_or_buffer="s3://bucket/shot_file.mdio",
...     index_bytes=(17, 137, 13),
...     index_lengths=(4, 2, 4),
...     index_names=("shot_point", "cable", "channel"),
...     chunksize=(8, 2, 128, 1024),
...     grid_overrides={"ChannelWrap": True, "ChannelsPerCable": 800},
... )
```

For shot gathers with channel numbers and wrapped channels, no grid overrides are necessary.

In cases where the user does not know if the input has unwrapped channels but desires to store with wrapped channel index use: >>> grid_overrides={“AutoChannelWrap”: True,

“AutoChannelTraceQC”: 1000000}

For ingestion of pre-stack streamer data where the user needs to access/index *common-channel gathers* (single gun) then the following strategy can be used to densely ingest while indexing on gun number:

```
>>> segy_to_mdio(
...     segy_path="prefix/shot_file.segy",
...     mdio_path_or_buffer="s3://bucket/shot_file.mdio",
...     index_bytes=(133, 171, 17, 137, 13),
...     index_lengths=(2, 2, 4, 2, 4),
...     index_names=("shot_line", "gun", "shot_point", "cable", "channel"),
...     chunksize=(1, 1, 8, 1, 128, 1024),
...     grid_overrides={
...         "AutoShotWrap": True,
...         "AutoChannelWrap": True,
...         "AutoChannelTraceQC":  1000000
...     },
... )
```

For AutoShotWrap and AutoChannelWrap to work, the user must provide “shot_line”, “gun”, “shot_point”, “cable”, “channel”. For improved common-channel performance consider modifying the chunksize to be (1, 1, 32, 1, 32, 2048) for good common-shot and common-channel performance or (1, 1, 128, 1, 1, 2048) for common-channel performance.

For cases with no well-defined trace header for indexing a NonBinned grid override is provided.This creates the index and attributes an incrementing integer to the trace for the index based on first in first out. For example a CDP and Offset keyed file might have a header for offset as real world offset which would result in a very sparse populated index. Instead, the following override will create a new index from 1 to N, where N is the number of offsets within a CDP ensemble. The index to be auto generated is called “trace”. Note the required “chunksize” parameter in the grid override. This is due to the non-binned ensemble chunksize is irrelevant to the index dimension chunksizes and has to be specified in the grid override itself. Note the lack of offset, only indexing CDP, providing CDP header type, and chunksize for only CDP and Sample dimension. The chunksize for non-binned dimension is in the grid overrides as described above. The below configuration will yield 1MB chunks:

```
>>> segy_to_mdio(
...     segy_path="prefix/cdp_offset_file.segy",
...     mdio_path_or_buffer="s3://bucket/cdp_offset_file.mdio",
...     index_bytes=(21,),
...     index_types=("int32",),
...     index_names=("cdp",),
...     chunksize=(4, 1024),
...     grid_overrides={"NonBinned": True, "chunksize": 64},
... )
```

A more complicated case where you may have a 5D dataset that is not binned in Offset and Azimuth directions can be ingested like below. However, the Offset and Azimuth dimensions will be combined to “trace” dimension. The below configuration will yield 1MB chunks.

```
>>> segy_to_mdio(
...     segy_path="prefix/cdp_offset_file.segy",
...     mdio_path_or_buffer="s3://bucket/cdp_offset_file.mdio",
```

(continues on next page)

(continued from previous page)

```
...     index_bytes=(189, 193),
...     index_types=("int32", "int32"),
...     index_names=("inline", "crossline"),
...     chunksize=(4, 4, 1024),
...     grid_overrides={"NonBinned": True, "chunksize": 64},
... )
```

For dataset with expected duplicate traces we have the following parameterization. This will use the same logic as NonBinned with a fixed chunksize of 1. The other keys are still important. The below example allows multiple traces per receiver (i.e. reshoot).

```
>>> segy_to_mdio(
...     segy_path="prefix/cdp_offset_file.segy",
...     mdio_path_or_buffer="s3://bucket/cdp_offset_file.mdio",
...     index_bytes=(9, 213, 13),
...     index_types=("int32", "int16", "int32"),
...     index_names=("shot", "cable", "chan"),
...     chunksize=(8, 2, 256, 512),
...     grid_overrides={"HasDuplicates": True},
... )
```

Conversion from to MDIO various other formats.

mdio.converters.mdio.**mdio_to_segy**(*mdio_path_or_buffer*, *output_segy_path*, *endian='big'*, *access_pattern='012'*, *out_sample_format='ibm32'*, *storage_options=None*, *new_chunks=None*, *selection_mask=None*, *client=None*)

Convert MDIO file to SEG-Y format.

MDIO allows exporting multidimensional seismic data back to the flattened seismic format SEG-Y, to be used in data transmission.

The input headers are preserved as is, and will be transferred to the output file.

The user has control over the endianness, and the floating point data type. However, by default we export as Big-Endian IBM float, per the SEG-Y format's default.

The input MDIO can be local or cloud based. However, the output SEG-Y will be generated locally.

A *selection_mask* can be provided (in the shape of the spatial grid) to export a subset of the seismic data.

Parameters

- **mdio_path_or_buffer** (*str*) – Input path where the MDIO is located
- **output_segy_path** (*str*) – Path to the output SEG-Y file
- **endian** (*str*) – Endianness of the input SEG-Y. Rev.2 allows little endian. Default is 'big'.
- **access_pattern** (*str*) – This specificies the chunk access pattern. Underlying zarr.Array must exist. Examples: '012', '01'
- **out_sample_format** (*str*) – Output sample format. Currently support: {'ibm32', 'float32'}. Default is 'ibm32'.
- **storage_options** (*dict*) – Storage options for the cloud storage backend. Default: None (will assume anonymous access)
- **new_chunks** (*tuple[int, ...]*) – Set manual chunksize. For development purposes only.

- **selection_mask** (*np.ndarray*) – Array that lists the subset of traces
- **client** (*distributed.Client*) – Dask client. If *None* we will use local threaded scheduler. If *auto* is used we will create multiple processes (with 8 threads each).

Raises

- **ImportError** – if distributed package isn't installed but requested.
- **ValueError** – if cut mask is empty, i.e. no traces will be written.

Return type
None

Examples

To export an existing local MDIO file to SEG-Y we use the code snippet below. This will export the full MDIO (without padding) to SEG-Y format using IBM floats and big-endian byte order.

```
>>> from mdio import mdio_to_segy
>>>
>>>
>>> mdio_to_segy(
...     mdio_path_or_buffer="prefix2/file.mdio",
...     output_segy_path="prefix/file.segy",
... )
```

If we want to export this as an IEEE big-endian, using a selection mask, we would run:

```
>>> mdio_to_segy(
...     mdio_path_or_buffer="prefix2/file.mdio",
...     output_segy_path="prefix/file.segy",
...     selection_mask=boolean_mask,
...     out_sample_format="float32",
... )
```

7.6.3 Core Functionality

Dimensions

Dimension (grid) abstraction and serializers.

class mdio.core.dimension.**Dimension**(*coords*, *name*)

Dimension class.

Dimension has a name and coordinates associated with it. The Dimension coordinates can only be a vector.

Parameters

- **coords** (*list | tuple | ndarray[Any, dtype[_ScalarType_co]] | range*) – Vector of coordinates.
- **name** (*str*) – Name of the dimension.

classmethod deserialize(*stream*, *stream_format*)

Deserialize buffer into Dimension.

Parameters

- **stream** (*str*) –
- **stream_format** (*str*) –

Return type
Dimension

classmethod from_dict(*other*)

Make dimension from dictionary.

Parameters
other (*dict[str, Any]*) –

Return type
Dimension

max()

Get maximum value of dimension.

Return type
NDArray[np.float]

min()

Get minimum value of dimension.

Return type
NDArray[np.float]

serialize(*stream_format*)

Serialize the dimension into buffer.

Parameters
stream_format (*str*) –

Return type
str

to_dict()

Convert dimension to dictionary.

Return type
dict[str, *Any*]

property size: int

Size of the dimension.

class mdio.core.dimension.**DimensionSerializer**(*stream_format*)

Serializer implementation for Dimension.

Initialize serializer.

Parameters
stream_format (*str*) – Stream format. Must be in {"JSON", "YAML"}.

deserialize(*stream*)

Deserialize buffer into Dimension.

Parameters
stream (*str*) –

Return type
Dimension

serialize(*dimension*)

Serialize Dimension into buffer.

Parameters
dimension (Dimension) –

Return type
str

Data I/O

(De)serialization factory design pattern.

Current support for JSON and YAML.

class mdio.core.serialization.**Serializer**(*stream_format*)

Serializer base class.

Here we define the interface for any serializer implementation.

Parameters
stream_format (*str*) – Format of the stream for serialization.

Initialize serializer.

Parameters
stream_format (*str*) – Stream format. Must be in {"JSON", "YAML"}.

abstract deserialize(*stream*)

Abstract method for deserialize.

Parameters
stream (*str*) –

Return type
dict

abstract serialize(*payload*)

Abstract method for serialize.

Parameters
payload (*dict*) –

Return type
str

static validate_payload(*payload*, *signature*)

Validate if required keys exist in the payload for a function signature.

Parameters

- **payload** (*dict*) –
- **signature** (*Signature*) –

Return type
dict

mdio.core.serialization.**get_deserializer**(*stream_format*)

Get deserializer based on format.

Parameters
stream_format (*str*) –

Return type
Callable

mdio.core.serialization.**get_serializer**(*stream_format*)

Get serializer based on format.

Parameters
stream_format (*str*) –

Return type
Callable

7.6.4 Convenience Functions

Convenience APIs for working with MDIO files.

mdio.api.convenience.**copy_mdio**(*source*, *dest_path_or_buffer*, *excludes=''*, *includes=''*, *storage_options=None*, *overwrite=False*)

Copy MDIO file.

Can also copy with empty data to be filled later. See *excludes* and *includes* parameters.

More documentation about *excludes* and *includes* can be found in Zarr's documentation in *zarr.convenience.copy_store*.

Parameters

- **source** (MDIOReader) – MDIO reader or accessor instance. Data will be copied from here
- **dest_path_or_buffer** (*str*) – Destination path. Could be any FSSpec mapping.
- **excludes** (*str*) – Data to exclude during copy. i.e. *chunked_012*. The raw data won't be copied, but it will create an empty array to be filled. If left blank, it will copy everything.
- **includes** (*str*) – Data to include during copy. i.e. *trace_headers*. If this is not specified, and certain data is excluded, it will not copy headers. If you want to preserve headers, specify *trace_headers*. If left blank, it will copy everything except specified in *excludes* parameter.
- **storage_options** (*dict | None*) – Storage options for the cloud storage backend. Default is None (will assume anonymous).
- **overwrite** (*bool*) – Overwrite destination or not.

Return type
None

mdio.api.convenience.**rechunk**(*source*, *chunks*, *suffix*, *compressor=None*, *overwrite=False*)

Rechunk MDIO file adding a new variable.

Parameters

- **source** (MDIOAccessor) – MDIO accessor instance. Data will be copied from here.
- **chunks** (*tuple[int, ...]*) – Tuple containing chunk sizes for new rechunked array.
- **suffix** (*str*) – Suffix to append to new rechunked array.
- **compressor** (*Codec | None*) – Data compressor to use, optional. Default is Blosc('zstd').
- **overwrite** (*bool*) – Overwrite destination or not.

Return type
None

Examples

To rechunk a single variable we can do this

```
>>> accessor = MDIOAccessor(...)
>>> rechunk(accessor, (1, 1024, 1024), suffix="fast_il")
```

mdio.api.convenience.**rechunk_batch**(*source*, *chunks_list*, *suffix_list*, *compressor=None*, *overwrite=False*)

Rechunk MDIO file to multiple variables, reading it once.

Parameters

- **source** (MDIOAccessor) – MDIO accessor instance. Data will be copied from here.
- **chunks_list** (*list[tuple[int, ...]]*) – List of tuples containing new chunk sizes.
- **suffix_list** (*list[str]*) – List of suffixes to append to new chunk sizes.
- **compressor** (*Codec | None*) – Data compressor to use, optional. Default is Blosc('zstd').
- **overwrite** (*bool*) – Overwrite destination or not.

Return type
None

Examples

To rechunk multiple variables we can do things like:

```
>>> accessor = MDIOAccessor(...)
>>> rechunk_batch(
>>>     accessor,
>>>     chunks_list=[(1, 1024, 1024), (1024, 1, 1024), (1024, 1024, 1)],
>>>     suffix_list=["fast_il", "fast_xl", "fast_z"],
>>> )
```

7.7 Contributor Guide

Thank you for your interest in improving this project. This project is open-source under the Apache 2.0 license and welcomes contributions in the form of bug reports, feature requests, and pull requests.

Here is a list of important resources for contributors:

- Source Code
- Documentation
- Issue Tracker
- *Code of Conduct*

7.7.1 How to report a bug

Report bugs on the Issue Tracker.

When filing an issue, make sure to answer these questions:

- Which operating system and Python version are you using?
- Which version of this project are you using?
- What did you do?
- What did you expect to see?
- What did you see instead?

The best way to get your bug fixed is to provide a test case, and/or steps to reproduce the issue.

7.7.2 How to request a feature

Request features on the Issue Tracker.

7.7.3 How to set up your development environment

You need Python 3.9+ and the following tools:

- Poetry
- Nox
- nox-poetry

Another alternative is to use a Development Container has been setup to provide an environment with the required dependencies. This facilitates development on different systems.

This should seamlessly enable development for users of VS Code on systems with docker installed.

Known Issues:

- `git config --global --add safe.directory $(pwd)` might be needed inside the container.

7.7.4 How to Install and Run MDIO

Install the package with development requirements:

```
$ poetry install
```

You can now run an interactive Python session, or the command-line interface:

```
$ poetry run python
$ poetry run mdio
```

7.7.5 How to test the project

Run the full test suite:

```
$ nox
```

List the available Nox sessions:

```
$ nox --list-sessions
```

You can also run a specific Nox session. For example, invoke the unit test suite like this:

```
$ nox --session=tests
```

Unit tests are located in the *tests* directory, and are written using the pytest testing framework.

7.7.6 How to submit changes

Open a pull request to submit changes to this project.

Your pull request needs to meet the following guidelines for acceptance:

- The Nox test suite must pass without errors and warnings.
- Include unit tests. This project maintains 100% code coverage.
- If your changes add functionality, update the documentation accordingly.

Feel free to submit early, though—we can always iterate on this.

To run linting and code formatting checks before committing your change, you can install pre-commit as a Git hook by running the following command:

```
$ nox --session=pre-commit -- install
```

It is recommended to open an issue before starting work on anything. This will allow a chance to talk it over with the owners and validate your approach.

7.8 Contributor Covenant Code of Conduct

7.8.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, caste, color, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

7.8.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

7.8.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

7.8.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

7.8.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at opensource@tgs.com. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

7.8.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

7.8.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.1, available at https://www.contributor-covenant.org/version/2/1/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

7.9 License

```
Copyright 2022 TGS

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

------------------------------------------------------------------------

                             Apache License
                       Version 2.0, January 2004
                    http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

  "License" shall mean the terms and conditions for use, reproduction,
  and distribution as defined by Sections 1 through 9 of this document.

  "Licensor" shall mean the copyright owner or entity authorized by
  the copyright owner that is granting the License.

  "Legal Entity" shall mean the union of the acting entity and all
  other entities that control, are controlled by, or are under common
  control with that entity. For the purposes of this definition,
  "control" means (i) the power, direct or indirect, to cause the
  direction or management of such entity, whether by contract or
  otherwise, or (ii) ownership of fifty percent (50%) or more of the
  outstanding shares, or (iii) beneficial ownership of such entity.

  "You" (or "Your") shall mean an individual or Legal Entity
  exercising permissions granted by this License.

  "Source" form shall mean the preferred form for making modifications,
  including but not limited to software source code, documentation
  source, and configuration files.

  "Object" form shall mean any form resulting from mechanical
  transformation or translation of a Source form, including but
  not limited to compiled object code, generated documentation,
  and conversions to other media types.
```

(continues on next page)

(continued from previous page)

```
  "Work" shall mean the work of authorship, whether in Source or
  Object form, made available under the License, as indicated by a
  copyright notice that is included in or attached to the work
  (an example is provided in the Appendix below).

  "Derivative Works" shall mean any work, whether in Source or Object
  form, that is based on (or derived from) the Work and for which the
  editorial revisions, annotations, elaborations, or other modifications
  represent, as a whole, an original work of authorship. For the purposes
  of this License, Derivative Works shall not include works that remain
  separable from, or merely link (or bind by name) to the interfaces of,
  the Work and Derivative Works thereof.

  "Contribution" shall mean any work of authorship, including
  the original version of the Work and any modifications or additions
  to that Work or Derivative Works thereof, that is intentionally
  submitted to Licensor for inclusion in the Work by the copyright owner
  or by an individual or Legal Entity authorized to submit on behalf of
  the copyright owner. For the purposes of this definition, "submitted"
  means any form of electronic, verbal, or written communication sent
  to the Licensor or its representatives, including but not limited to
  communication on electronic mailing lists, source code control systems,
  and issue tracking systems that are managed by, or on behalf of, the
  Licensor for the purpose of discussing and improving the Work, but
  excluding communication that is conspicuously marked or otherwise
  designated in writing by the copyright owner as "Not a Contribution."

  "Contributor" shall mean Licensor and any individual or Legal Entity
  on behalf of whom a Contribution has been received by Licensor and
  subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
  this License, each Contributor hereby grants to You a perpetual,
  worldwide, non-exclusive, no-charge, royalty-free, irrevocable
  copyright license to reproduce, prepare Derivative Works of,
  publicly display, publicly perform, sublicense, and distribute the
  Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
  this License, each Contributor hereby grants to You a perpetual,
  worldwide, non-exclusive, no-charge, royalty-free, irrevocable
  (except as stated in this section) patent license to make, have made,
  use, offer to sell, sell, import, and otherwise transfer the Work,
  where such license applies only to those patent claims licensable
  by such Contributor that are necessarily infringed by their
  Contribution(s) alone or by combination of their Contribution(s)
  with the Work to which such Contribution(s) was submitted. If You
  institute patent litigation against any entity (including a
  cross-claim or counterclaim in a lawsuit) alleging that the Work
  or a Contribution incorporated within the Work constitutes direct
  or contributory patent infringement, then any patent licenses
  granted to You under this License for that Work shall terminate
```

(continues on next page)

(continued from previous page)

```
  as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
  Work or Derivative Works thereof in any medium, with or without
  modifications, and in Source or Object form, provided that You
  meet the following conditions:

  (a) You must give any other recipients of the Work or
      Derivative Works a copy of this License; and

  (b) You must cause any modified files to carry prominent notices
      stating that You changed the files; and

  (c) You must retain, in the Source form of any Derivative Works
      that You distribute, all copyright, patent, trademark, and
      attribution notices from the Source form of the Work,
      excluding those notices that do not pertain to any part of
      the Derivative Works; and

  (d) If the Work includes a "NOTICE" text file as part of its
      distribution, then any Derivative Works that You distribute must
      include a readable copy of the attribution notices contained
      within such NOTICE file, excluding those notices that do not
      pertain to any part of the Derivative Works, in at least one
      of the following places: within a NOTICE text file distributed
      as part of the Derivative Works; within the Source form or
      documentation, if provided along with the Derivative Works; or,
      within a display generated by the Derivative Works, if and
      wherever such third-party notices normally appear. The contents
      of the NOTICE file are for informational purposes only and
      do not modify the License. You may add Your own attribution
      notices within Derivative Works that You distribute, alongside
      or as an addendum to the NOTICE text from the Work, provided
      that such additional attribution notices cannot be construed
      as modifying the License.

  You may add Your own copyright statement to Your modifications and
  may provide additional or different license terms and conditions
  for use, reproduction, or distribution of Your modifications, or
  for any such Derivative Works as a whole, provided Your use,
  reproduction, and distribution of the Work otherwise complies with
  the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
  any Contribution intentionally submitted for inclusion in the Work
  by You to the Licensor shall be under the terms and conditions of
  this License, without any additional terms or conditions.
  Notwithstanding the above, nothing herein shall supersede or modify
  the terms of any separate license agreement you may have executed
  with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
```

(continues on next page)

(continued from previous page)

```
  names, trademarks, service marks, or product names of the Licensor,
  except as required for reasonable and customary use in describing the
  origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
  agreed to in writing, Licensor provides the Work (and each
  Contributor provides its Contributions) on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
  implied, including, without limitation, any warranties or conditions
  of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
  PARTICULAR PURPOSE. You are solely responsible for determining the
  appropriateness of using or redistributing the Work and assume any
  risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
  whether in tort (including negligence), contract, or otherwise,
  unless required by applicable law (such as deliberate and grossly
  negligent acts) or agreed to in writing, shall any Contributor be
  liable to You for damages, including any direct, indirect, special,
  incidental, or consequential damages of any character arising as a
  result of this License or out of the use or inability to use the
  Work (including but not limited to damages for loss of goodwill,
  work stoppage, computer failure or malfunction, or any and all
  other commercial damages or losses), even if such Contributor
  has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
  the Work or Derivative Works thereof, You may choose to offer,
  and charge a fee for, acceptance of support, warranty, indemnity,
  or other liability obligations and/or rights consistent with this
  License. However, in accepting such obligations, You may act only
  on Your own behalf and on Your sole responsibility, not on behalf
  of any other Contributor, and only if You agree to indemnify,
  defend, and hold each Contributor harmless for any liability
  incurred by, or claims asserted against, such Contributor by reason
  of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

PYTHON MODULE INDEX

m

INDEX

Symbols

B

C

N

R

S

T

V