MagentroPy

Release 0.1.6

Soren Bear

Nov 15, 2023

CONTENTS

References

Please cite the following in any published work that makes use of this package:

[1] J. D. Bocarsly et al., Phys. Rev. B 97, 100404(R) (2018)

[2] J. J. Stickel, Comput. Chem. Eng. 34, 467 (2010)

The first version of the `magentropy` code was included as supplementary material in [1]. The Tikhonov regularization procedure was described in [2] and was originally implemented by Stickel in the package scikit.datasmooth.

CHAPTER

ONE

OVERVIEW

MagentroPy provides a class, *MagentroData*, that can be used to calculate magnetocaloric quantities from DC magnetization data supplied as magnetic moment vs. temperature sweeps (monotonic) taken under several different magnetic fields. The class is set up to work out-of-the-box with `.dat` data files produced by a Quantum Design Vibrating Sample Magnetometer or a Quantum Design MPMS3 SQUID Magnetometer. However, `pandas.DataFrame`s or delimited files such as `.csv` are also acceptable inputs.

During data processing, the magnetic moment is differentiated with respect to temperature and integrated with respect to the magnetic field to calculate the entropy. Smoothing is performed on the magnetization data using Tikhonov regularization in order to reduce noise in the derivative. See [1] and [2] for additional information.

Plotting methods are provided for creating line plots and heat maps. These methods can be used with `matplotlib` for flexible and extensive plotting functionality.

An experimental *bootstrap()* method is implemented to estimate the error in the smoothed moment.

See the *Quickstart* for installation, logging, and usage.

CHAPTER

TWO

JUST-IN-TIME COMPILATION

Certain methods, such as *process_data()*, are accelerated using numba's just-in-time (JIT) compilation. JIT functions will have a noticeable compilation overhead the first time they are run; however, the results are cached on disk, so all subsequent runs will be faster.

If desired, JIT compilation can be disabled globally as follows:

```
from numba import config

config.DISABLE_JIT = True
```

CHAPTER

THREE

CONTENTS

3.1 Quickstart

3.1.1 Installation

Install MagentroPy with `pip`:

```
pip install magentropy
```

Or, with `conda`:

```
conda install -c conda-forge magentropy
```

3.1.2 Logging

The logger can be accessed as follows:

```
import logging
logger = logging.getLogger('magentropy')
```

3.1.3 The MagentroData class

`MagentroData` serves as a representation of magnetoentropic data and provides methods for reading, processing, and plotting the data. Because MagentroPy depends on other packages such as `numpy`, `scipy`, and `matplotlib`, importing can take several seconds.

```
from magentropy import MagentroData
```

3.1.4 Reading data

The easiest and most common way to read in data is with a Quantum Design `.dat` output file, such as that produced by an MPMS3 SQUID Magnetometer. The default settings are such that no additional arguments need to be given to the constructor when this is the case. See *Reading Data* for additional information.

```
magdata = MagentroData('magdata.dat')
```

```
"[Data]" tag detected, assuming QD .dat file.
The sample mass was determined from the QD .dat file: 0.1
```

The sample mass of 0.1 is parsed from the `.dat` file. The default mass units are "mg". We can easily view a summary of the object when using a notebook:

```
magdata
```

```
<magentropy.magentro.MagentroData at 0x7f5983723b10>
```

Above, we see the sample mass, raw data, converted data (SI units), processed data (currently empty), and presets, which are the default data processing settings. These are available individually as the attributes *sample_mass*, *raw_df*, *converted_df*, *processed_df*, and *presets*. For example:

```
magdata.sample_mass
```

```
0.1
```

Raw units can also be obtained using the *get_raw_data_units()* method:

```
magdata.get_raw_data_units()
```

```
{'T': 'K', 'H': 'Oe', 'M': 'emu', 'sample_mass': 'mg'}
```

Additional information about reading data and changing units can be found in *Reading Data* and *Units and Conversions*, respectively.

3.1.5 Processing data

The various settings in *presets* indicate the default arguments for the *process_data()* method. These are explored further in *Processing Data*. Here, we smooth the magnetic moment, differentiate with respect to temperature, and integrate with respect to the magnetic field to fill the missing columns in each data attribute.

```
magdata.process_data()
```

```
The data contains the following 5 magnetic field strengths and observations per field:
20.0     100
40.0     100
60.0     100
80.0     100
100.0    100
Name: T, dtype: int64

Processing data using the following settings:
{
        npoints: 1000,
        temp_range: [-inf  inf],
        fields: [],
        decimals: 5,
        max_diff: inf,
        min_sweep_len: 10,
```

(continues on next page)

(continued from previous page)

```
        d_order: 2,
        lmbds: [nan],
        lmbd_guess: 0.0001,
        weight_err: True,
        match_err: False,
        min_kwargs: {'method': 'Nelder-Mead', 'bounds': ((-inf, inf),), 'options': {
↪'maxfev': 50, 'xatol': 0.01, 'fatol': 1e-06}},
        add_zeros: False
}

scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 20.0
scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 40.0
scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 60.0
scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 80.0
scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 100.0
Calculated raw derivative and entropy.

last_presets set to:
{
        npoints: 1000,
        temp_range: [  0.99999934 100.00000083],
        fields: [ 20.  40.  60.  80. 100.],
        decimals: 5,
        max_diff: inf,
        min_sweep_len: 10,
        d_order: 2,
        lmbds: [0.00091728 0.00054639 0.00072862 0.00091728 0.00095775],
        lmbd_guess: 0.0001,
        weight_err: True,
        match_err: False,
        min_kwargs: {'method': 'Nelder-Mead', 'bounds': ((-inf, inf),), 'options': {
↪'maxfev': 50, 'xatol': 0.01, 'fatol': 1e-06}},
        add_zeros: False
}

Finished.
```

The field groups and regularization (smoothing) parameters are determined automatically by default. We can see that there are five different magnetic field strengths with 100 temperature points each. The *processed_df* attribute now contains the results:

```
magdata.processed_df
```

```
              T      H         M  M_err  M_per_mass  M_per_mass_err  \
0      0.999999  0.002  0.000002    NaN   19.847003             NaN
1      1.099098  0.002  0.000002    NaN   19.837267             NaN
2      1.198198  0.002  0.000002    NaN   19.827533             NaN
```

(continues on next page)

(continued from previous page)

```
3       1.297297  0.002  0.000002    NaN   19.817803             NaN
4       1.396396  0.002  0.000002    NaN   19.808082             NaN
...          ...    ...       ...    ...         ...             ...
4995   99.603604  0.010  0.000001    NaN   11.673323             NaN
4996   99.702704  0.010  0.000001    NaN   11.591120             NaN
4997   99.801803  0.010  0.000001    NaN   11.508924             NaN
4998   99.900902  0.010  0.000001    NaN   11.426733             NaN
4999  100.000001  0.010  0.000001    NaN   11.344545             NaN

         dM_dT  Delta_SM
0    -0.098265 -0.000098
1    -0.098240 -0.000098
2    -0.098202 -0.000098
3    -0.098138 -0.000098
4    -0.098050 -0.000098
...        ...       ...
4995 -0.829550 -0.004619
4996 -0.829466 -0.004620
4997 -0.829407 -0.004620
4998 -0.829371 -0.004620
4999 -0.829347 -0.004620

[5000 rows x 8 columns]
```

Similarly, one could check that the derivative and entropy have been calculated for the raw data.

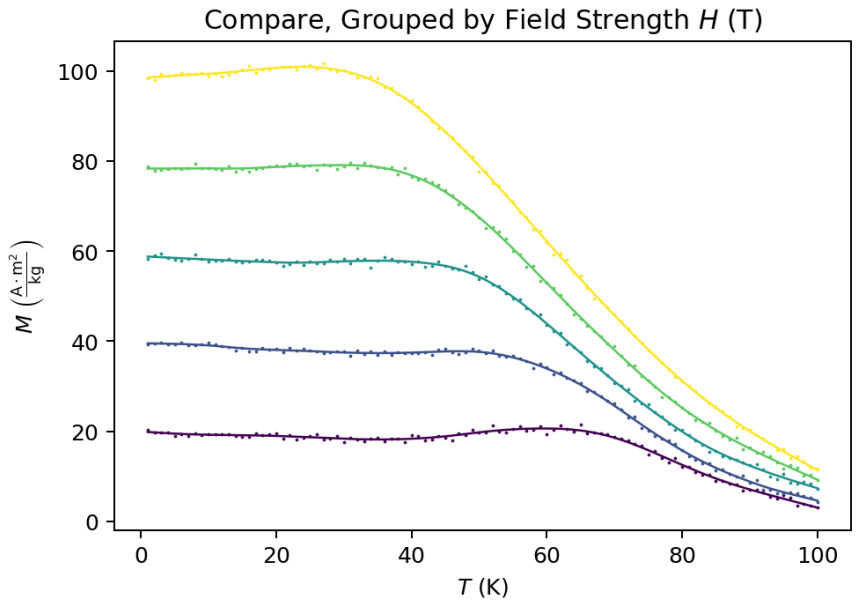
Notice that the error columns in the processed data are empty. An experimental *bootstrap()* method is implemented to estimate the error in the smoothed moment. See *Bootstrap Estimates*.

3.1.6 Plotting Data

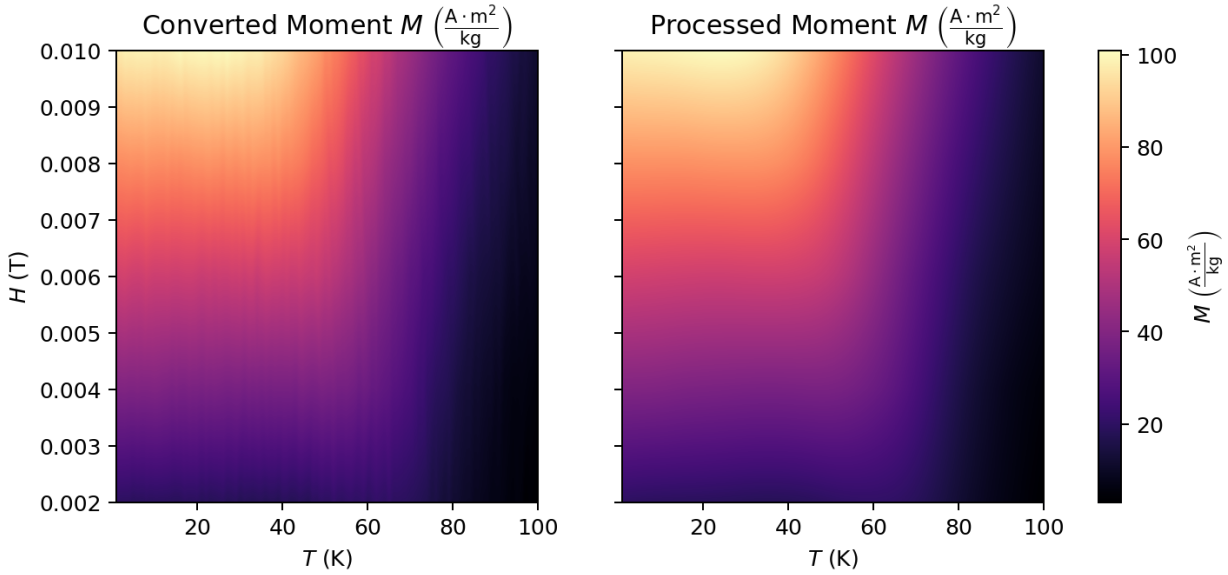
```
import matplotlib.pyplot as plt
```

Line plots and heat maps can be easily created with *plot_lines()* and *plot_map()*.

```
fig, ax = plt.subplots(figsize=(6, 4))
magdata.plot_lines(data_prop='M_per_mass', data_version='compare', ax=ax);
```




```
fig, ax = plt.subplots(1, 2, figsize=(9, 3.75), sharey=True)

magdata.plot_map(data_prop='M_per_mass', data_version='converted', ax=ax[0],␣
↪colorbar=False)

magdata.plot_map(
    data_prop='M_per_mass', data_version='processed', ax=ax[1], colorbar=True,
    colorbar_kwargs={'ax': ax, 'fraction': 0.10, 'pad': 0.05}
)

ax[1].set_ylabel('');
```



The second figure demonstrates the natural use of `matplotlib` with the plotting methods. Many additional options are available; see *Plotting Data* for more information.

3.1.7 Writing output

Because data is represented as `DataFrame`s, one can use methods such as `DataFrame.to_csv()` to write output to files. See *Reading Data* for reading in data from delimited files and *Plotting Data* for plotting previously-processed data.

3.2 Examples

See the *Quickstart* for installation, logging, and general usage.

3.2.1 Reading Data

```
from magentropy import MagentroData
```

Constructor

Data must be read when a *MagentroData* object is instantiated via the constructor. Supported input formats are Quantum Design `.dat` data files (default), delimited files, and `DataFrame`s.

QD data files

The default arguments are configured for QD `.dat` files. These files are expected to consist of:

1. A header section with metadata. In particular, the sample mass should be given as `INFO,<sample_mass>, SAMPLE_MASS`, where `<sample_mass>` is replaced by a decimal number.
2. A `\n[Data]\n` tag separating the header section from the data. (Here, `\n` indicates a newline.)
3. The delimited data. The default separator is `','`.
4. Data columns with names `'Comment'`, `'Temperature (K)'`, `'Magnetic Field (Oe)'`, `'Moment (emu)'`, and `'M. Std. Err. (emu)'`.

```
magdata_dat = MagentroData('magdata.dat')
```

```
"[Data]" tag detected, assuming QD .dat file.
The sample mass was determined from the QD .dat file: 0.1
```

Tip: If the column names, delimiter, or sample mass format are different, these can be set manually as described *below*. The `**read_csv_kwargs` keyword arguments will also be applied to the delimited data in `.dat` files.

Delimited files

A delimited input file may be indicated by passing `qd_dat = False` to the constructor. Additionally, different column names can be specified, including the absence of a comment or error column. Here, the comment column is excluded.

```
magdata_csv = MagentroData(
    'magdata.csv', qd_dat=False,
    comment_col=None, T='T', H='H', M='M', M_err='M_err'
)
```

Since the sample mass is not present in delimited files, it is set to the default of 1.0. This can be changed after instantiation (per-mass columns are updated accordingly):

```
print(magdata_csv.sample_mass)
magdata_csv.sample_mass = 0.1
print(magdata_csv.sample_mass)
```

```
1.0
0.1
```

The mass can also be provided in the constructor itself:

```
magdata_csv = MagentroData(
    'magdata.csv', qd_dat=False,
    comment_col=None, T='T', H='H', M='M', M_err='M_err',
    sample_mass=0.1
)
magdata_csv.sample_mass
```

```
0.1
```

Note: It is not strictly necessary to set `qd_dat = False`. The delimited data will be read correctly, though a warning will be printed, and of course the sample mass must still be set manually.

Tip: Delimited data is read using `pandas.read_csv()`. Keyword arguments can be passed to `pandas.read_csv()` as additional keyword arguments to the constructor. For example, if the file is tab-delimited, `magdata_tab = MagentroData(..., sep='\t')`. These will be ignored if the input is a `DataFrame`, described in the *next section*.

DataFrames

If data is in a `DataFrame`, perhaps because preprocessing was required, the procedure is exactly the same as for *delimited files*. Here, we create a new *`MagentroData`* instance using the raw data from `magdata_csv`:

```
magdata_df = MagentroData(
    magdata_csv.raw_df,
    comment_col=None, T='T', H='H', M='M', M_err='M_err',
    sample_mass=0.1
)
```

The column labels and sample mass are specified as before. The `qd_dat` parameter is ignored because a `DataFrame` is detected, so it need not be included.

Missing values

If a comment column label is supplied, any row in which the comment column has a non-`NaN` value is dropped. (i.e., rows with comments are removed, since comments in QD `.dat` output files indicate measurement problems.)

Additionally, any row containing a missing value in the temperature, field, or moment column is dropped. If a moment error column is supplied, any row with a missing value or a value equal to zero in the error column will be dropped.

Viewing data

Raw, converted (SI units), and processed (smoothed) data is available through the attributes *`raw_df`*, *`converted_df`*, and *`processed_df`*. For example:

```
magdata_dat.raw_df
```

```
              T           H         M    M_err  M_per_mass  M_per_mass_err  \
0      1.000000   20.000001  0.002023  0.00005   20.232376             0.5
1      2.000000   20.000000  0.001977  0.00005   19.770351             0.5
2      3.000001   19.999998  0.001969  0.00005   19.691176             0.5
3      4.000000   19.999999  0.001970  0.00005   19.703463             0.5
4      4.999999   20.000001  0.001886  0.00005   18.861315             0.5
..          ...         ...       ...      ...         ...             ...
495   95.999998  100.000002  0.001403  0.00005   14.030179             0.5
496   97.000000   99.999999  0.001446  0.00005   14.458498             0.5
497   98.000001  100.000001  0.001324  0.00005   13.244919             0.5
498   98.999999   99.999999  0.001184  0.00005   11.835327             0.5
499  100.000000  100.000000  0.001172  0.00005   11.722362             0.5

     dM_dT  Delta_SM
0      NaN       NaN
1      NaN       NaN
2      NaN       NaN
3      NaN       NaN
4      NaN       NaN
..     ...       ...
495    NaN       NaN
496    NaN       NaN
497    NaN       NaN
498    NaN       NaN
499    NaN       NaN

[500 rows x 8 columns]
```

Each `DataFrame` attribute contains columns corresponding to temperature, magnetic field strength, moment, moment error, moment per mass, moment per mass error, moment derivative with respect to temperature, and magnetic entropy.

Units can be viewed as a second header level by appending `_with_units` to any of these attributes.

```
magdata_dat.raw_df_with_units
```

```
               T           H         M    M_err M_per_mass M_per_mass_err  \
unit           K          Oe       emu      emu      emu/g          emu/g
0       1.000000   20.000001  0.002023  0.00005  20.232376            0.5
1       2.000000   20.000000  0.001977  0.00005  19.770351            0.5
2       3.000001   19.999998  0.001969  0.00005  19.691176            0.5
3       4.000000   19.999999  0.001970  0.00005  19.703463            0.5
4       4.999999   20.000001  0.001886  0.00005  18.861315            0.5
..           ...         ...       ...      ...        ...            ...
495    95.999998  100.000002  0.001403  0.00005  14.030179            0.5
496    97.000000   99.999999  0.001446  0.00005  14.458498            0.5
497    98.000001  100.000001  0.001324  0.00005  13.244919            0.5
498    98.999999   99.999999  0.001184  0.00005  11.835327            0.5
499   100.000000  100.000000  0.001172  0.00005  11.722362            0.5

          dM_dT Delta_SM
unit cal/K/Oe/g  cal/K/g
0           NaN      NaN
1           NaN      NaN
2           NaN      NaN
3           NaN      NaN
4           NaN      NaN
..          ...      ...
495         NaN      NaN
496         NaN      NaN
497         NaN      NaN
498         NaN      NaN
499         NaN      NaN

[500 rows x 8 columns]
```

Similarly for sample mass:

```
magdata_dat.sample_mass
```

```
0.1
```

```
magdata_dat.sample_mass_with_units
```

```
(0.1, 'mg')
```

Tip: All `DataFrame` attributes are immutable and return copies of the internal instance attributes. If repeated access is required, for example to a `DataFrame`'s columns, it is best to first save the `DataFrame` as a local variable to avoid repeatedly copying large amounts of data.

Don't do this:

```
col_means = [magdata.raw_df['T'].mean(), magdata.raw_df['H'].mean(), magdata.raw_df['M'].
↪mean()]
```

Instead, do this:

```
raw_df = magdata.raw_df
col_means = [raw_df['T'].mean(), raw_df['H'].mean(), raw_df['M'].mean()]
```

Simulating data

The class method *sim_data()* can be used to generate data for testing and examples. A decreasing logistic function with a Gaussian "bump" whose center depends on the field strength is used to "simulate" noisy data. (There are quotation marks because the function has no physical significance.)

The following code returns a DataFrame with columns 'T', 'H', 'M', and 'M_err'. This data is the same data found in the magdata.dat and magdata.csv files used in these examples.

```
import numpy as np

sim_df = MagentroData.sim_data(
    temps=np.linspace(1., 100., 100),
    fields=np.linspace(20., 100., 5),
    sigma_m=5e-5,
    random_seed=0
)
```

Units and presets

It is possible to set presets and units during instantiation. See *Processing Data* and *Units and Conversions* for additional information.

3.2.2 Processing Data

Tip: All of the *presets* metioned below can be set during instantiation:

```
magdata = MagentroData(..., presets={...})
```

```
from magentropy import MagentroData

magdata = MagentroData('magdata.dat')
```

```
"[Data]" tag detected, assuming QD .dat file.
The sample mass was determined from the QD .dat file: 0.1
```

Grouping

Before smoothing, the magnetization data must be grouped by field. Normally, the measured fields are not exact, so groups must be inferred. The method *test_grouping()* can be used to test grouping presets prior to fully processing the data.

```
grouping_presets, grouped_by = magdata.test_grouping()
```

With no arguments passed, the defaults in *presets* are used. The method returns a dictionary of the grouping presets used to perform the grouping and a pandas DataFrameGroupBy object to see the results. In particular, the DataFrameGroupBy.count() method is useful.

```
grouping_presets
```

```
{'fields': array([], dtype=float64), 'decimals': 5, 'max_diff': inf}
```

```
grouped_by['T'].count()
```

```
20.0     100
40.0     100
60.0     100
80.0     100
100.0    100
Name: T, dtype: int64
```

Above, we see that the default grouping presets are an empty array of `fields`, a `decimal` place of 5 for rounding, and infinite `max_diff`. Detailed information on each of these can be found in the *process_data()* documentation.

In this instance, the presets direct the grouping method to group the fields simply by rounding to the 5th decimal place, which accurately determines the field groups, as shown by the `count()` method. There are five fields, each with 100 temperature measurements. In most cases, grouping by rounding should be sufficient.

Smoothing

Reference

J. J. Stickel, Comput. Chem. Eng. 34, 467 (2010)

There are a number of options to control the smoothing. The default presets have been chosen sensibly but can be easily changed. All parameters, including grouping parameters, can be either set as new defaults using *presets* or *set_presets()*, or used for a single *process_data()* run by entering an argument in *process_data()*. See the documentation for a complete description of all parameters. They are summarized below. The use of *set_presets()* is demonstrated in each case with the default presets purely for example; it is not necessary to set presets if the defaults are to be used.

Output temperatures

The smoothed magnetic moment will be evaulated at `npoints` evenly-spaced temperatures in the range `temp_range`. `npoints` expects an integer, and `temp_range` expects an array_like of length 2. The default range `[-numpy.inf, numpy.inf]` adjusts to the maximum range in the data automatically. Additionally, only those fields with at least `min_sweep_len` measured temperatures in their respective temperature sweeps will be processed. The default is 10.

```
from numpy import inf

magdata.set_presets(npoints=1000, temp_range=[-inf, inf], min_sweep_len=10)
```

Regularization

The two most important options for the regularization (smoothing) itself are the derivative order `d_order` and the regularization parameter λ for each field, `lmbds`.

The derivative of the magnetic moment with respect to temperature of order `d_order` is used to quantify the "roughness" of the fitted curves. Generally, 2 or 3 work well. The default is 2.

The regularization parameter determines the empahsis that is given to the roughness regularization penalty. A higher λ results in a smoother curve, and a λ of zero results in interpolation. A λ can be specified for each field (in increasing field order) as an array_like of the same length as the number of fields. Any field with a corresponding λ of `numpy.nan` will have an "optimal" λ determined automatically; see *below*. The default `lmbds` is an array with a single `numpy.nan`, which indicates that an optimal λ should be found for each field. The same behavior occurs if an empty list is given.

```
magdata.set_presets(d_order=2, lmbds=[])
```

Optimal regularization parameters

Numerical optimization is used to determine the optimal regularization parameter for each field without a λ provided. Three metrics are available to quantify the meaning of "optimal":

1. Generalized cross validation (GCV). The GCV variance is minimized. Set `match_err` to `False` (default).
2. Error matching. The standard deviation of the absolute differences between the measured and smoothed magnetic moment points is matched to a value. The squared difference between the standard deviation and this value is minimized. Set `match_err` to a single value to match this value for all fields, an array_like of the same length as the number of fields to match a different value for each field (in order of increasing field), or one of `'min'`, `'mean'`, or `'max'` to use the minimum, mean, or maximum value of the error column for each field as the value.
3. Per-point error matching (experimental). The absolute differences between the measured and smoothed magnetic moment points are computed, and the sum of squared differences between these and the corresponding values in the error column is minimized. Set `match_err` to `True`.

Each of these requires an initial guess, given by `lmbd_guess`. Currently, a single guess to use for all fields is supported. For control over the minimization, keyword arguments to pass to `scipy.optimize.minimize()` can be given as a dictionary to `min_kwargs`. Keep in mind that any values passed to `min_kwargs` should be with respect to $\log_{10}\lambda$, since this is the value that is minimized internally. (However, `lmbd_guess` is the guess for λ itself; no $\log_{10}$.) Lastly, `weight_err` specifies whether to weight measurements by the normalized inverse squares of the errors. The default is `True`.

See *`process_data()`* for full documentation.

```
magdata.set_presets(
    lmbd_guess=1e-4, weight_err=True, match_err=False,
    min_kwargs = {
        'method': 'Nelder-Mead',
        'bounds': ((-inf, inf),),
        'options': {'maxfev': 50, 'xatol': 1e-2, 'fatol': 1e-6}
    }
)
```

Integrating from zero field

The calculation of entropy requires that the derivative of the magnetic moment with respect to temperature be integrated with respect to magnetic field, starting at zero field. Zero field measurements (with zero moment) are prepended before integration during processing, so it is not necessary to include zero field measurements in the input data.

The zeros can be included in *processed_df* if `add_zeros` is set to `True`. It is `False` by default.

```
magdata.set_presets(add_zeros=False)
```

Demonstration

Simple usage of *process_data()* is shown, including the adjustment of the regularization parameters by eye after they have been estimated initially. Plots are used to verify the success of the smoothing. See *Plotting Data* for more information.

```
magdata.process_data()
```

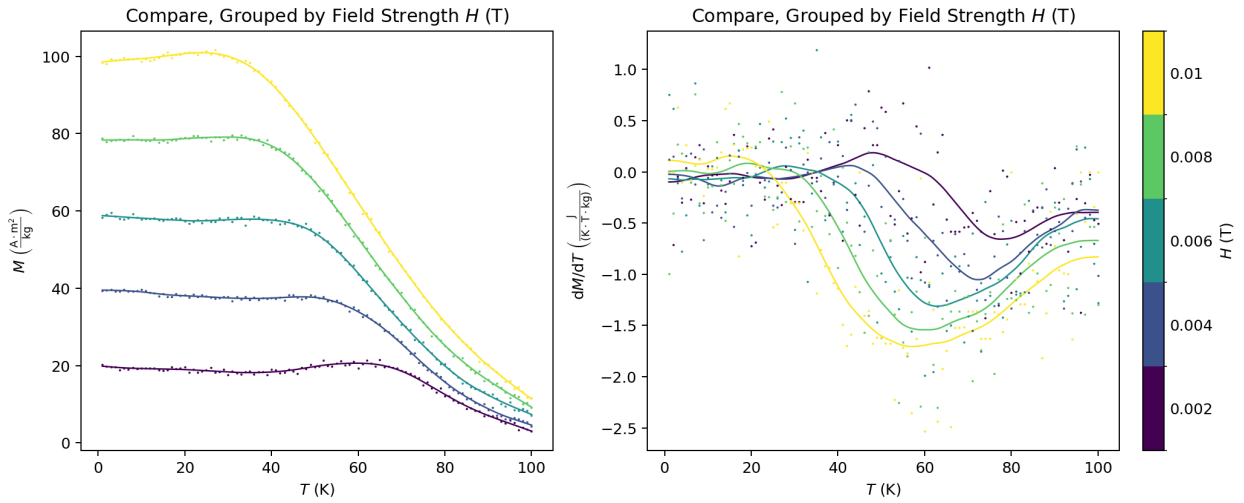
```
The data contains the following 5 magnetic field strengths and observations per field:
20.0     100
40.0     100
60.0     100
80.0     100
100.0    100
Name: T, dtype: int64

Processing data using the following settings:
{
        npoints: 1000,
        temp_range: [-inf  inf],
        fields: [],
        decimals: 5,
        max_diff: inf,
        min_sweep_len: 10,
        d_order: 2,
        lmbds: [nan],
        lmbd_guess: 0.0001,
        weight_err: True,
        match_err: False,
        min_kwargs: {'method': 'Nelder-Mead', 'bounds': ((-inf, inf),), 'options': {
↪'maxfev': 50, 'xatol': 0.01, 'fatol': 1e-06}},
        add_zeros: False
```

(continues on next page)

(continued from previous page)

```
}

scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 20.0
scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 40.0
scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 60.0
scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 80.0
scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 100.0
Calculated raw derivative and entropy.

last_presets set to:
{
        npoints: 1000,
        temp_range: [  0.99999934 100.00000083],
        fields: [ 20.  40.  60.  80. 100.],
        decimals: 5,
        max_diff: inf,
        min_sweep_len: 10,
        d_order: 2,
        lmbds: [0.00091728 0.00054639 0.00072862 0.00091728 0.00095775],
        lmbd_guess: 0.0001,
        weight_err: True,
        match_err: False,
        min_kwargs: {'method': 'Nelder-Mead', 'bounds': ((-inf, inf),), 'options': {
↪'maxfev': 50, 'xatol': 0.01, 'fatol': 1e-06}},
        add_zeros: False
}

Finished.
```

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, 2, figsize=(14, 5))

magdata.plot_lines(data_prop='M_per_mass', data_version='compare', ax=ax[0])

magdata.plot_lines(
    data_prop='dM_dT', data_version='compare', ax=ax[1], colorbar=True,
    colorbar_kwargs={'ax': ax, 'fraction': 0.1, 'pad': 0.02}
);
```



Note: The errors in this fake data are greatly exaggerated! Most instruments will have relative errors much smaller than those shown here.

We can see that the smoothed data (lines) looks much better than the raw data (dots), especially in the derivative plot on the right. Generalized cross validation has done a pretty good job of selecting optimal regularization parameters, which we can view using *last_presets*:

```
magdata.last_presets['lmbds']
```

```
array([0.00091728, 0.00054639, 0.00072862, 0.00091728, 0.00095775])
```

The *presets* are the same as they were before; however, setting them to *last_presets* is simple:

```
magdata.presets = magdata.last_presets
magdata.presets
```

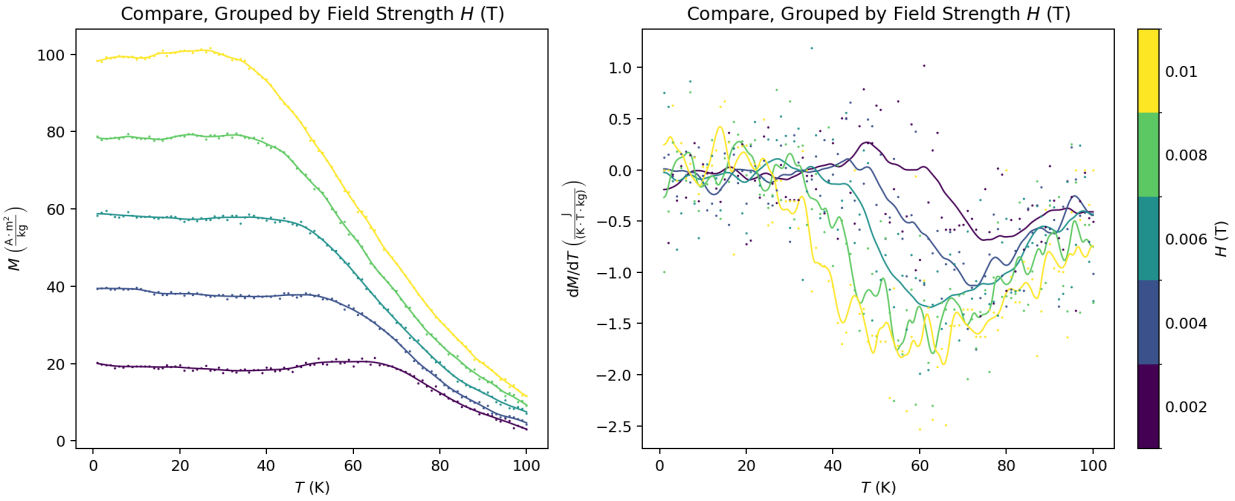
```
{'npoints': 1000,
 'temp_range': array([  0.99999934, 100.00000083]),
 'fields': array([ 20.,  40.,  60.,  80., 100.]),
 'decimals': 5,
 'max_diff': inf,
 'min_sweep_len': 10,
 'd_order': 2,
 'lmbds': array([0.00091728, 0.00054639, 0.00072862, 0.00091728, 0.00095775]),
 'lmbd_guess': 0.0001,
 'weight_err': True,
 'match_err': False,
 'min_kwargs': {'method': 'Nelder-Mead',
  'bounds': ((-inf, inf),),
  'options': {'maxfev': 50, 'xatol': 0.01, 'fatol': 1e-06}},
 'add_zeros': False}
```

We could also adjust `lmbds` for a single run and re-process:

```
magdata.process_data(lmbds=[1e-4, 5e-5, 1e-4, 1e-5, 1e-5])
```

```
The data contains the following 5 magnetic field strengths and observations per field:
20.0     100
40.0     100
60.0     100
80.0     100
100.0    100
Name: T, dtype: int64

Processing data using the following settings:
{
        npoints: 1000,
        temp_range: [  0.99999934 100.00000083],
        fields: [ 20.  40.  60.  80. 100.],
        decimals: 5,
        max_diff: inf,
        min_sweep_len: 10,
        d_order: 2,
        lmbds: [1.e-04 5.e-05 1.e-04 1.e-05 1.e-05],
        lmbd_guess: 0.0001,
        weight_err: True,
        match_err: False,
        min_kwargs: {'method': 'Nelder-Mead', 'bounds': ((-inf, inf),), 'options': {
↪'maxfev': 50, 'xatol': 0.01, 'fatol': 1e-06}},
        add_zeros: False
}

Processed M(T) at field: 20.0
Processed M(T) at field: 40.0
Processed M(T) at field: 60.0
Processed M(T) at field: 80.0
Processed M(T) at field: 100.0
Calculated raw derivative and entropy.

last_presets set to:
{
        npoints: 1000,
        temp_range: [  0.99999934 100.00000083],
        fields: [ 20.  40.  60.  80. 100.],
        decimals: 5,
        max_diff: inf,
        min_sweep_len: 10,
        d_order: 2,
        lmbds: [1.e-04 5.e-05 1.e-04 1.e-05 1.e-05],
        lmbd_guess: 0.0001,
        weight_err: True,
        match_err: False,
        min_kwargs: {'method': 'Nelder-Mead', 'bounds': ((-inf, inf),), 'options': {
↪'maxfev': 50, 'xatol': 0.01, 'fatol': 1e-06}},
        add_zeros: False
}

Finished.
```

```
fig, ax = plt.subplots(1, 2, figsize=(14, 5))

magdata.plot_lines(data_prop='M_per_mass', data_version='compare', ax=ax[0])

magdata.plot_lines(
    data_prop='dM_dT', data_version='compare', ax=ax[1], colorbar=True,
    colorbar_kwargs={'ax': ax, 'fraction': 0.1, 'pad': 0.02}
);
```



The error column in *processed_df* will still be empty after all this. See *Bootstrap Estimates*.

3.2.3 Plotting Data

```
import matplotlib.pyplot as plt
from magentropy import MagentroData

magdata = MagentroData('magdata.dat')
magdata.process_data()
```

```
"[Data]" tag detected, assuming QD .dat file.
The sample mass was determined from the QD .dat file: 0.1
The data contains the following 5 magnetic field strengths and observations per field:
20.0     100
40.0     100
60.0     100
80.0     100
100.0    100
Name: T, dtype: int64

Processing data using the following settings:
{
        npoints: 1000,
        temp_range: [-inf  inf],
        fields: [],
        decimals: 5,
```

(continues on next page)

(continued from previous page)

```
        max_diff: inf,
        min_sweep_len: 10,
        d_order: 2,
        lmbds: [nan],
        lmbd_guess: 0.0001,
        weight_err: True,
        match_err: False,
        min_kwargs: {'method': 'Nelder-Mead', 'bounds': ((-inf, inf),), 'options': {
↪'maxfev': 50, 'xatol': 0.01, 'fatol': 1e-06}},
        add_zeros: False
}

scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 20.0
scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 40.0
scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 60.0
scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 80.0
scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 100.0
Calculated raw derivative and entropy.

last_presets set to:
{
        npoints: 1000,
        temp_range: [  0.99999934 100.00000083],
        fields: [ 20.  40.  60.  80. 100.],
        decimals: 5,
        max_diff: inf,
        min_sweep_len: 10,
        d_order: 2,
        lmbds: [0.00091728 0.00054639 0.00072862 0.00091728 0.00095775],
        lmbd_guess: 0.0001,
        weight_err: True,
        match_err: False,
        min_kwargs: {'method': 'Nelder-Mead', 'bounds': ((-inf, inf),), 'options': {
↪'maxfev': 50, 'xatol': 0.01, 'fatol': 1e-06}},
        add_zeros: False
}

Finished.
```

Data may be plotted as lines using *plot_lines()* or as a map using *plot_map()*. In each, the property may be specified by the `data_prop` parameter, which can take values `'M_per_mass'`, `'M_per_mass_err'`, `'dM_dT'`, or `'Delta_SM'`, corresponding to the moment per mass, moment per mass error, derivative with respect to temperature, and entropy, respectively. The data version is specified by the `data_version` parameter, which can take values `'raw'`, `'converted'`, or `'processed'`. Line plots can also include converted and processed data together if `data_version = 'compare'`.

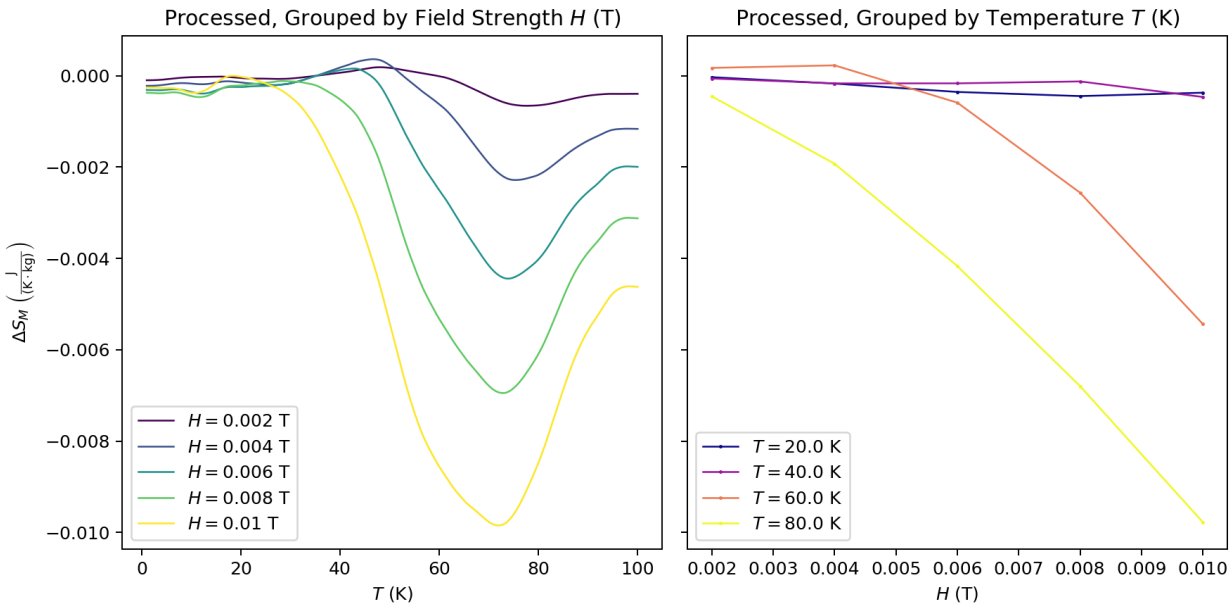
Both methods accept an optional `ax` argument specifying the `Axes` to use, as well as optional `T_range` and `H_range` arguments limiting the temperature and magenetic field range, respectively.

Important: All plotting parameters referring to physical quantities, such as `T_range`, `H_range`, and `offset`, are expected to be in the units indicated by `data_version`.

Lines

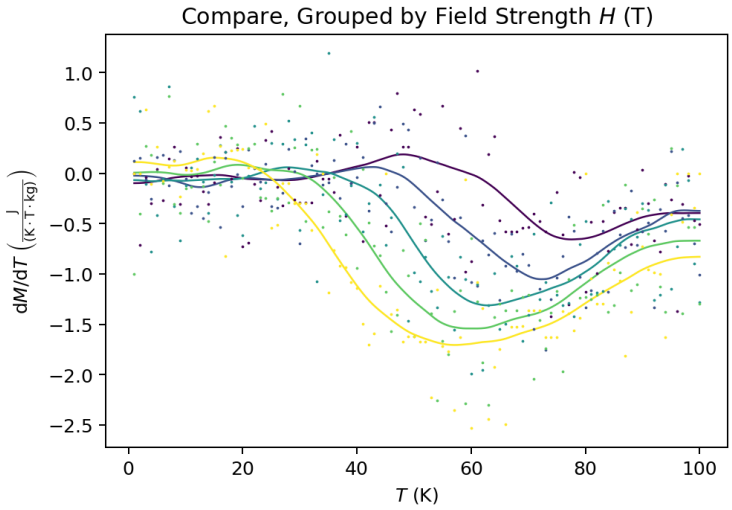
By default, line plots group data by magnetic field using the settings in *last_presets*. The data is grouped by temperature if `at_temps`, a list of temperatures, is supplied. A legend can be added with `legend = True` (default `False`).

```
fig, ax = plt.subplots(1, 2, figsize=(10, 5), sharey=True)

magdata.plot_lines(data_prop='Delta_SM', data_version='processed', ax=ax[0], legend=True)

magdata.plot_lines(
    data_prop='Delta_SM', data_version='processed', ax=ax[1],
    at_temps=[20, 40, 60, 80], legend=True
)

ax[1].set_ylabel('')
fig.tight_layout();
```



Setting `data_version = 'compare'` plots both the converted and processed data as dots and lines, respectively:

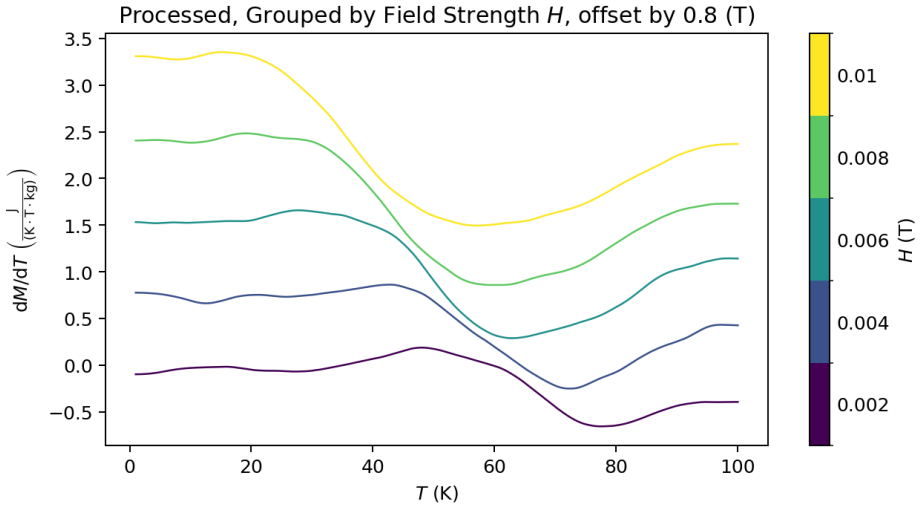
```
fig, ax = plt.subplots(figsize=(6, 4))
magdata.plot_lines(data_prop='dM_dT', data_version='compare', ax=ax);
```



An `offset` can be added to better view the shapes of individual lines, though this must be taken into account when reading off the vertical axis.

A discrete `Colorbar` can be added with `colorbar = True`.

```
fig, ax = plt.subplots(figsize=(8, 4))
magdata.plot_lines(data_prop='dM_dT', data_version='processed', ax=ax, offset=0.8,␣
↪colorbar=True);
```



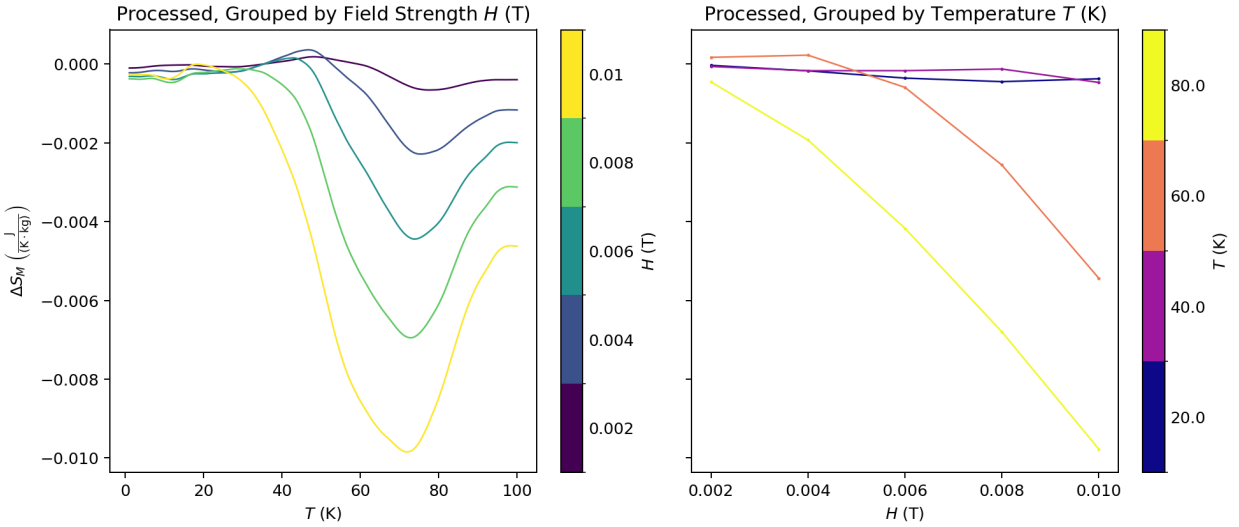
Adding a `Colorbar` is easiest for individual plots. Some extra work is required when adding `Colorbars` to `Figures` with multiple `Axes`.

```
fig, ax = plt.subplots(1, 2, figsize=(12, 5), sharey=True)

magdata.plot_lines(
    data_prop='Delta_SM', data_version='processed', ax=ax[0],
    colorbar=True, colorbar_kwargs={'ax': ax[0], 'fraction': 0.07, 'pad': 0.05}
)

magdata.plot_lines(
```

(continues on next page)

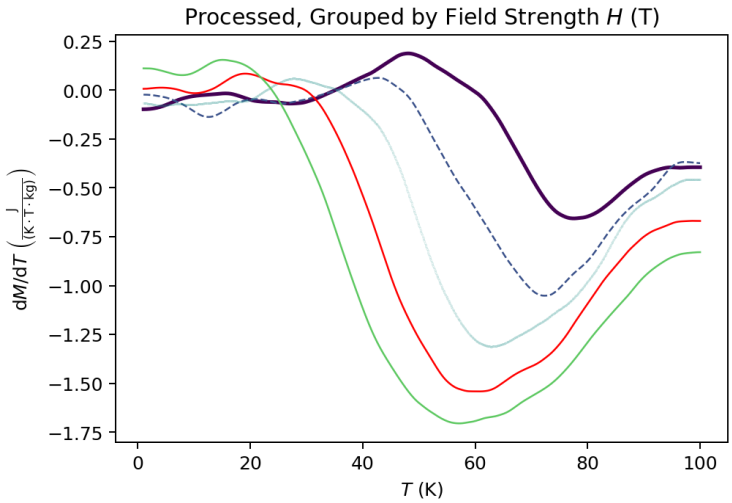
(continued from previous page)

```
    data_prop='Delta_SM', data_version='processed', ax=ax[1], at_temps=[20, 40, 60, 80],
    colorbar=True, colorbar_kwargs={'ax': ax[1], 'fraction': 0.07, 'pad': 0.05}
)

ax[1].set_ylabel('');
```



Note the use of `colorbar_kwargs` to pass keyword arguments directly to `Figure.colorbar()`. Similarly, `plot_kwargs` can pass keyword arguments to `Axes.plot()`. It can be either a single dictionary or a list of dictionaries corresponding to the lines.

```
fig, ax = plt.subplots(figsize=(6, 4))

magdata.plot_lines(
    data_prop='dM_dT', data_version='processed', ax=ax,
    plot_kwargs=[
        {'linewidth': 2},
        {'linestyle': '--'},
        {'linestyle': '', 'markersize': 0.1, 'marker': '.'},
        {'color': 'red'},
        {}
    ]
);
```

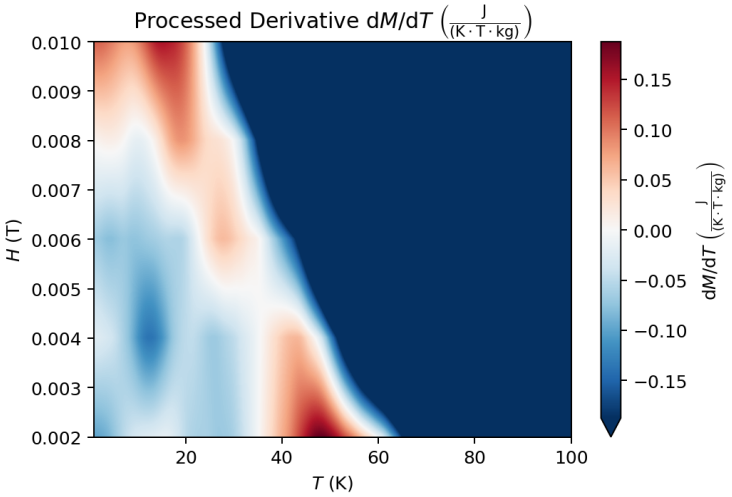


See *plot_lines()* for full documentation.

Maps

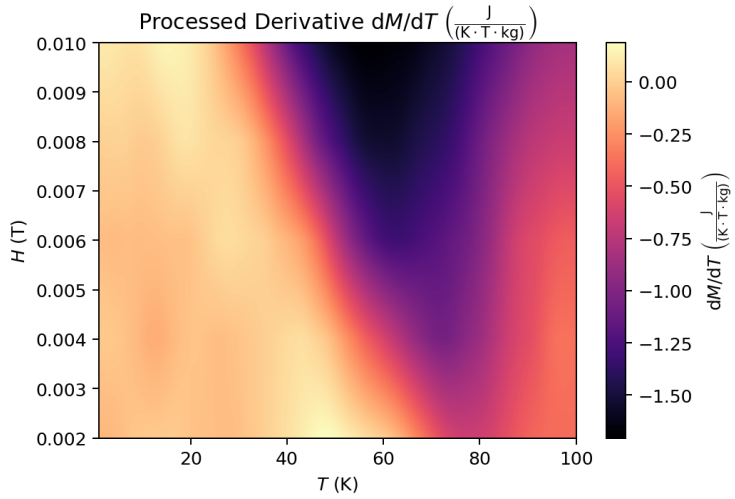
The method *plot_map()* creates a heat map with temperature on the horizontal axis, magnetic field strength on the vertical axis, and `data_prop` as the color. By default, `colorbar` is `True` for maps.

```
fig, ax = plt.subplots(figsize=(6, 4))
magdata.plot_map(data_prop='dM_dT', data_version='processed', ax=ax);
```



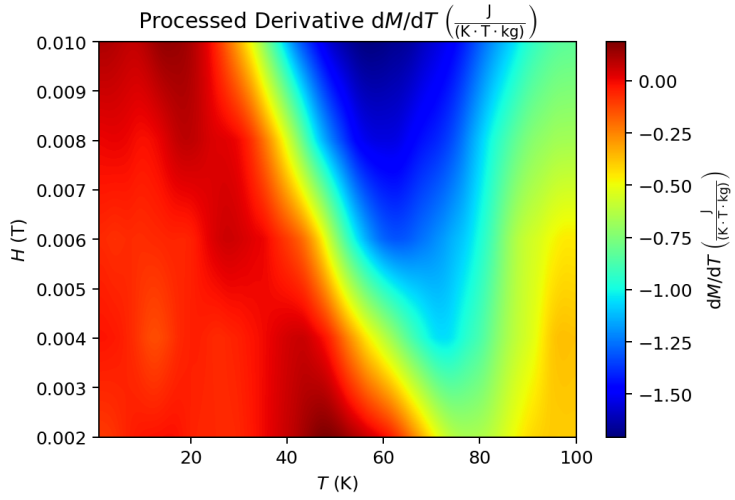
Notice that the color range is automatically centered around zero, clipping extreme values, as indicated by the arrow at the bottom of the `Colorbar`. This is useful in many cases; however, if this behavior is not desired, one can explicitly pass `center = False` to the method.

```
fig, ax = plt.subplots(figsize=(6, 4))
magdata.plot_map(data_prop='dM_dT', data_version='processed', ax=ax, center=False);
```



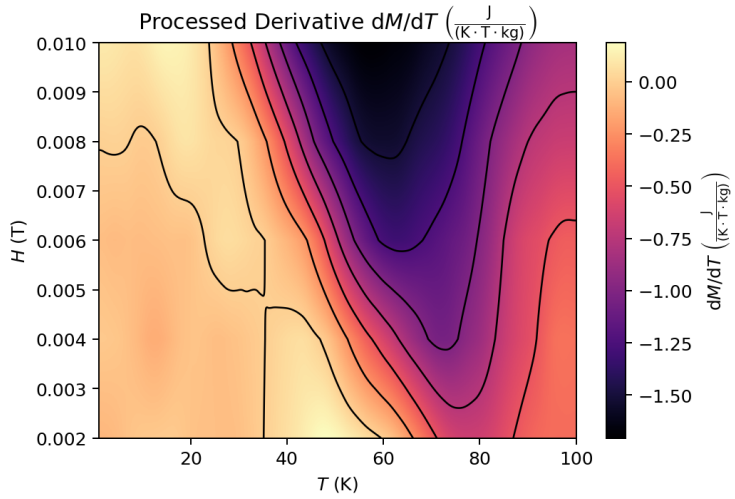
When the colors are not centered around zero, the `cmap` is changed to one that is sequential, rather than diverging. Keyword arguments such as `cmap` can be passed to `Axes.imshow()` via `imshow_kwargs`:

```
fig, ax = plt.subplots(figsize=(6, 4))

magdata.plot_map(
    data_prop='dM_dT', data_version='processed', ax=ax,
    center=False, imshow_kwargs={'cmap': 'jet'}
);
```



Contours can also be added with `contour = True` and customized with `contour_kwargs`, passed to `Axes.contour()`.

```
fig, ax = plt.subplots(figsize=(6, 4))

magdata.plot_map(
    data_prop='dM_dT', data_version='processed', ax=ax, center=False,
    contour=True, contour_kwargs={'linewidths': 1.0}
);
```



The grids used to construct the maps are available directly using *get_map_grid()*. For example,

```
T_grid, H_grid, Delta_SM_grid = magdata.get_map_grid(data_prop='Delta_SM', data_version=
↪'processed')
Delta_SM_grid[:3, :3]
```

```
array([[-9.82653799e-05, -9.82399942e-05, -9.82019155e-05],
       [-9.87532011e-05, -9.87277929e-05, -9.86896804e-05],
       [-9.92410224e-05, -9.92155916e-05, -9.91774454e-05]])
```

```
T_grid[:3, :3]
```

```
array([[0.99999934, 1.09909844, 1.19819754],
       [0.99999934, 1.09909844, 1.19819754],
       [0.99999934, 1.09909844, 1.19819754]])
```

```
H_grid[:3, :3]
```

```
array([[0.002     , 0.002     , 0.002     ],
       [0.00200801, 0.00200801, 0.00200801],
       [0.00201602, 0.00201602, 0.00201602]])
```

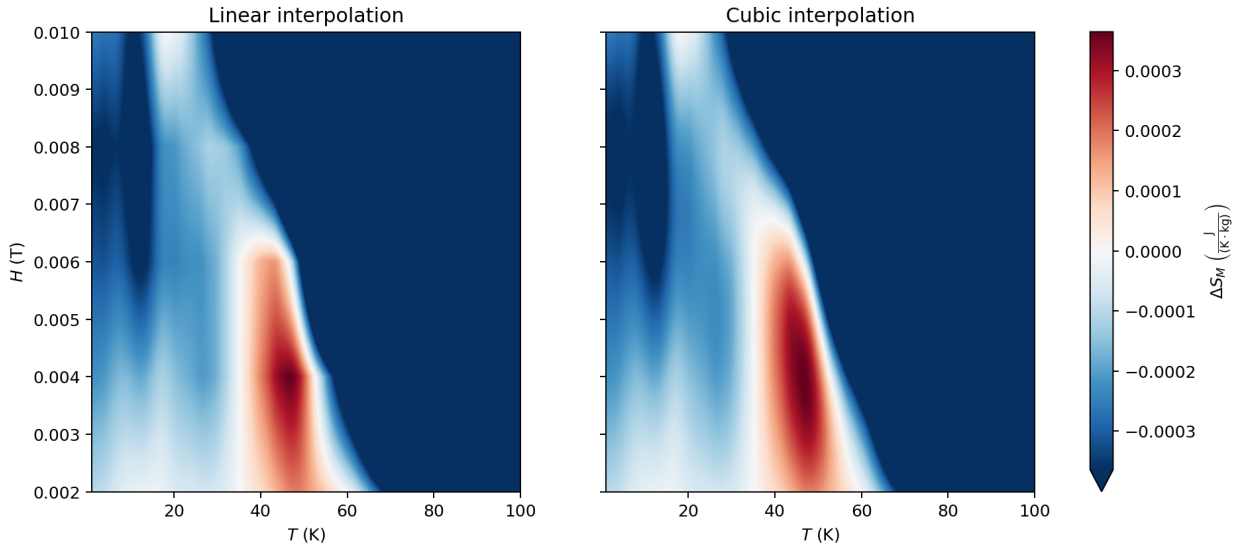
By default, linear interpolation is used to create the grids. It is recommended to at least start with linear interpolation, as cubic interpolation can occasionally result in artifacts. The interpolation method, passed to `scipy.interpolate.griddata()`'s `method` parameter, can be specified in *get_map_grid()* or in *plot_map()* with `interp_method`:

```
fig, ax = plt.subplots(1, 2, figsize=(12, 5), sharey=True)

magdata.plot_map(
    data_prop='Delta_SM', data_version='processed', ax=ax[0],
    colorbar=False,
    interp_method='linear'
)

magdata.plot_map(
```

(continues on next page)

(continued from previous page)

```
    data_prop='Delta_SM', data_version='processed', ax=ax[1],
    colorbar_kwargs={'ax': ax, 'fraction': 0.1, 'pad': 0.05},
    interp_method='cubic'
)

ax[0].set_title('Linear interpolation')
ax[1].set_title('Cubic interpolation')
ax[1].set_ylabel('');
```

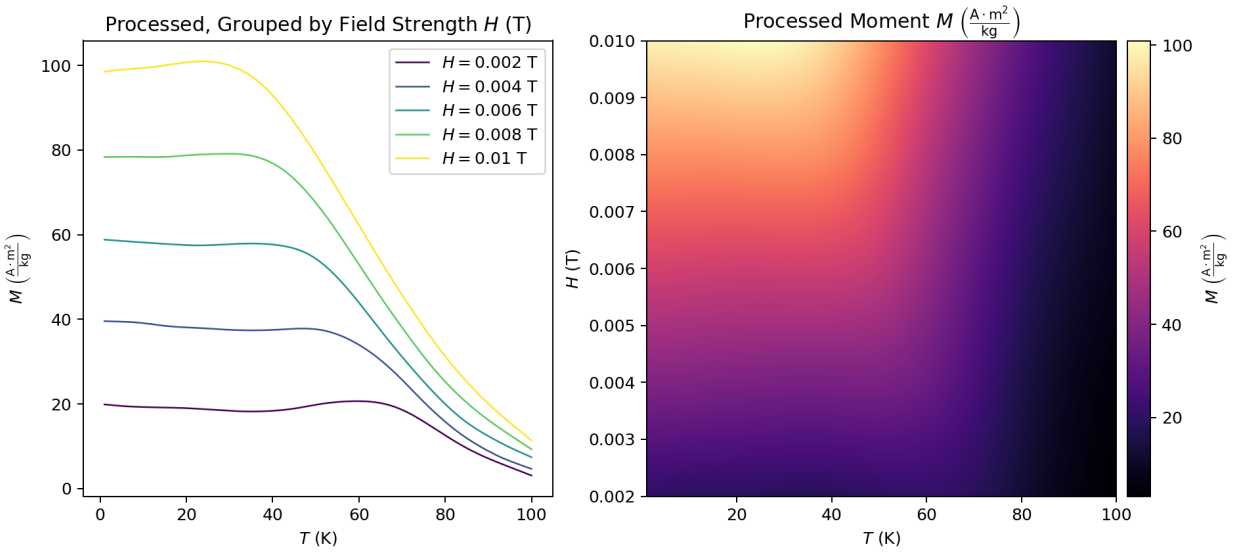


We can see that cubic interpolation produces a smoother result than linear interpolation, with no noticeable artifacts in this case.

Previously-processed data

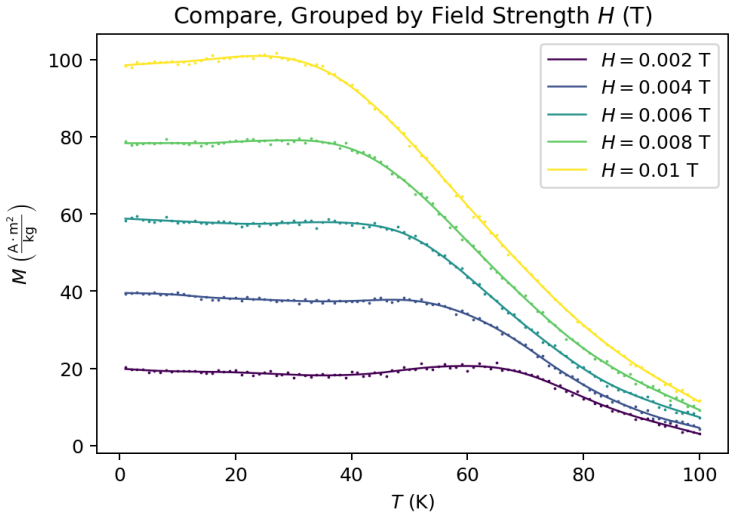
It is possible to plot previously-processed data contained in `DataFrame`s using the class methods *plot_processed_lines()* and *plot_processed_map()*. This allows one to avoid having to re-process data. As an example, we'll use the processed data from above.

```
processed_df = magdata.processed_df

fig, ax = plt.subplots(1, 2, figsize=(12, 5))

MagentroData.plot_processed_lines(processed_df, data_prop='M_per_mass', ax=ax[0],␣
↪legend=True)

MagentroData.plot_processed_map(
    processed_df, data_prop='M_per_mass', ax=ax[1],
    colorbar_kwargs={'ax': ax, 'fraction': 0.05, 'pad': 0.01}
);
```



In *plot_processed_lines()*, a `compare_df` can also be supplied for comparison plots.

```
converted_df = magdata.converted_df

fig, ax = plt.subplots(figsize=(6, 4))

MagentroData.plot_processed_lines(
    processed_df, compare_df=converted_df,
    data_prop='M_per_mass', ax=ax, legend=True
);
```



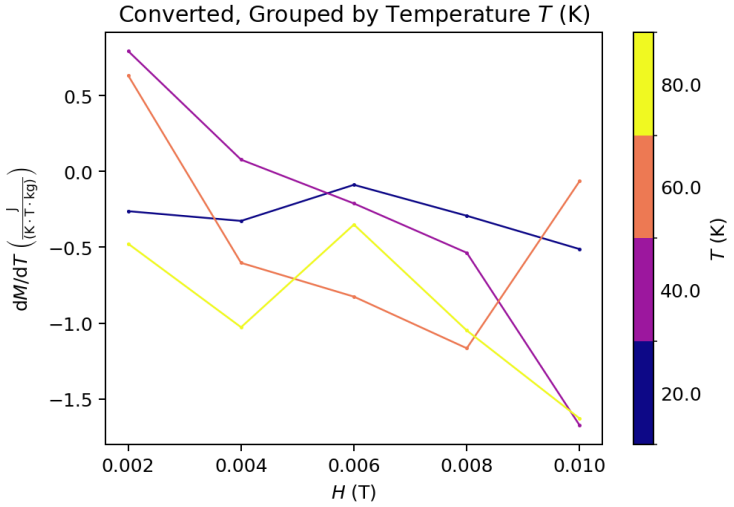
With the exception of `data_version`, the rest of the parameters for *plot_lines()* and *plot_map()* are also available for these two methods.

Convenience methods

Three convenience plotting methods are available.

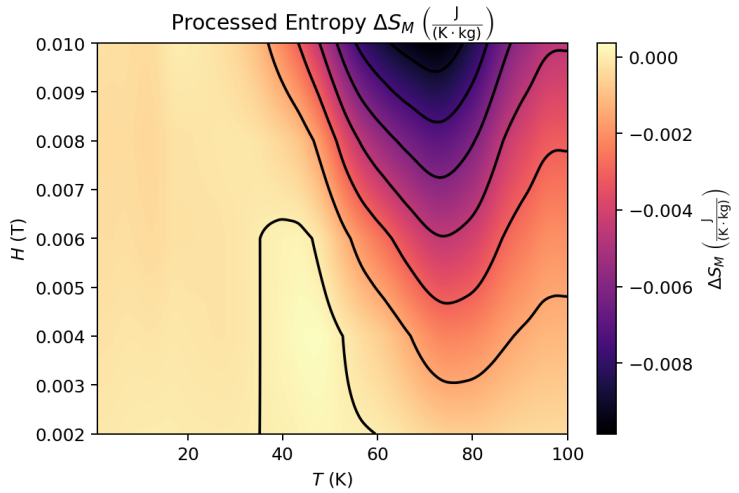
plot() passes keyword arguments to *plot_lines()* or *plot_map()* depending on whether `plot_type` is `'lines'` or `'map'`.

```
fig, ax = plt.subplots(figsize=(6, 4))

magdata.plot(
    plot_type='lines', data_prop='dM_dT', data_version='converted',
    ax=ax, at_temps=[20, 40, 60, 80], colorbar=True
);
```



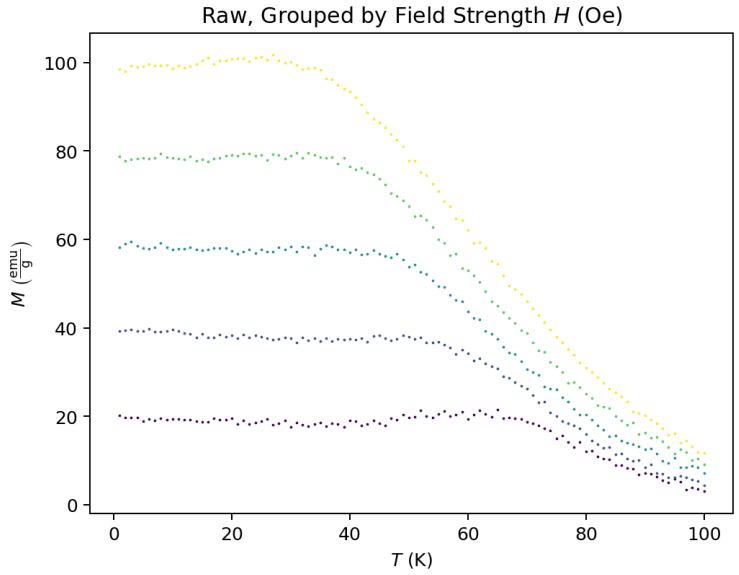
Similarly, *plot_processed()* passes keyword arguments to *plot_processed_lines()* or *plot_processed_map()*.

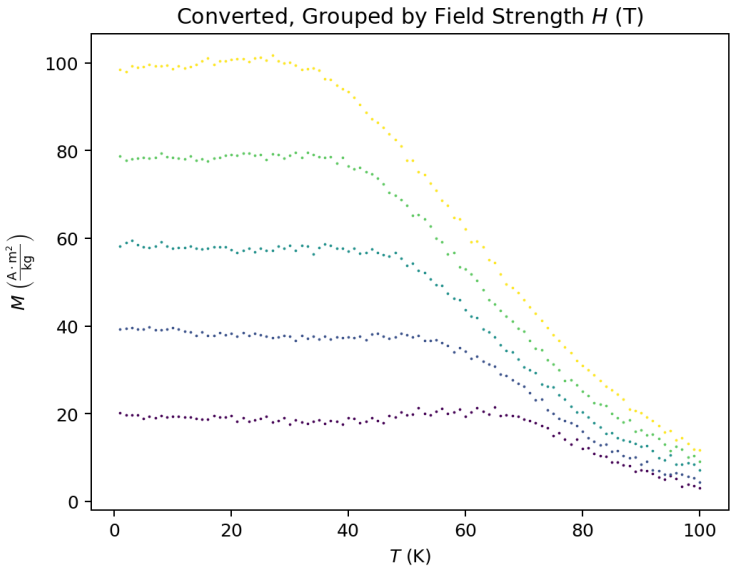
```
fig, ax = plt.subplots(figsize=(6, 4))

MagentroData.plot_processed(
    plot_type='map', processed_df=processed_df, data_prop='Delta_SM',
    ax=ax, center=False, contour=True
);
```

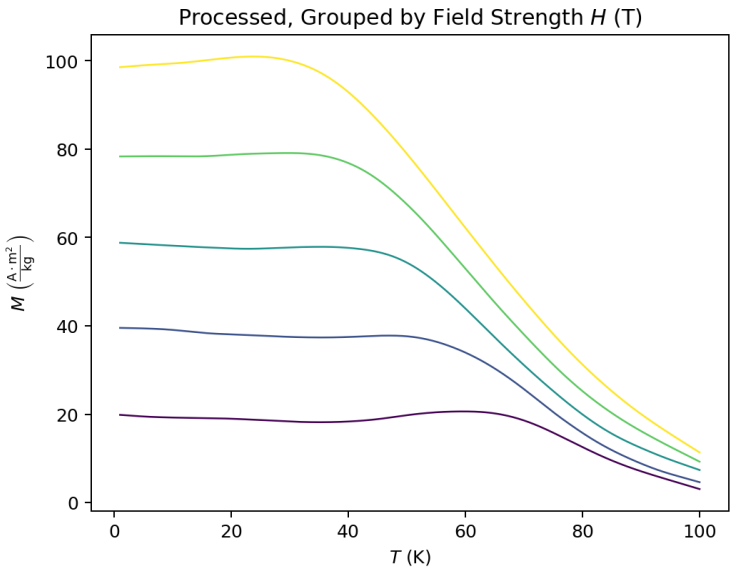


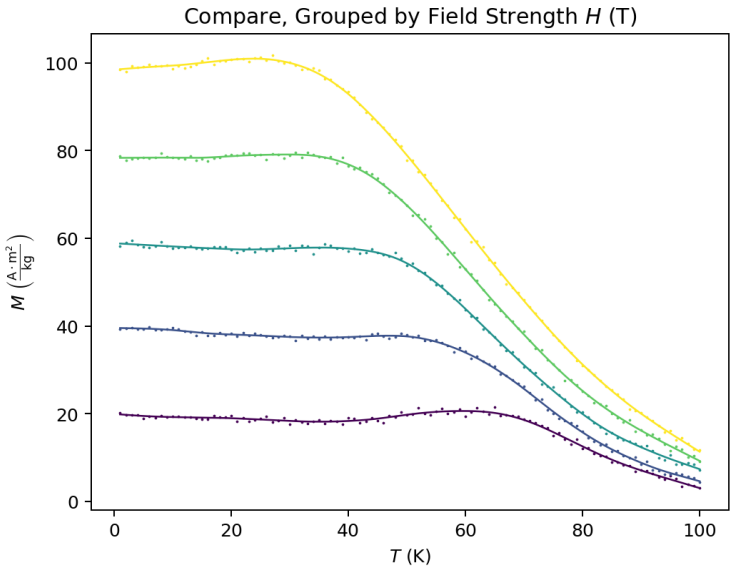
Lastly, *plot_all()* accepts no arguments and plots every plot combination with the default arguments, as well as some line plots grouped by temperature. This can be used as an initial inspection of all the data after *process_data()* is run. However, the output is quite long. Be sure to include a semicolon afterwards if running in a notebook to suppress unwanted text output!

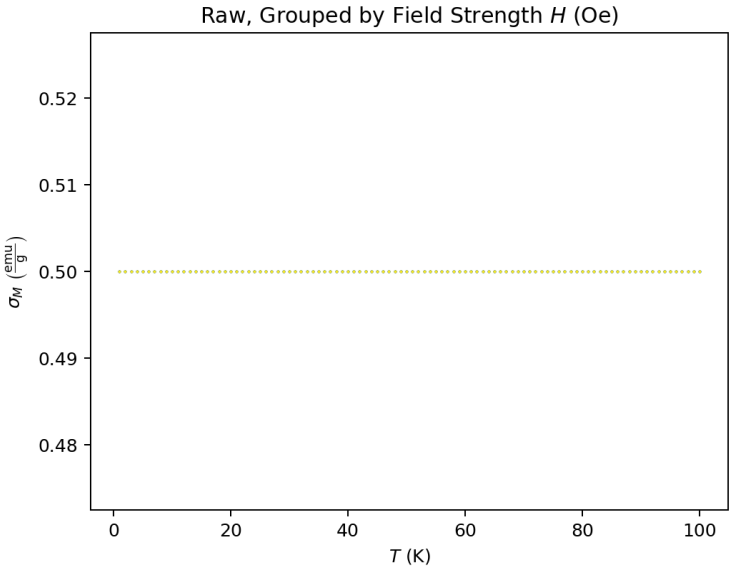
```
magdata.plot_all();
```

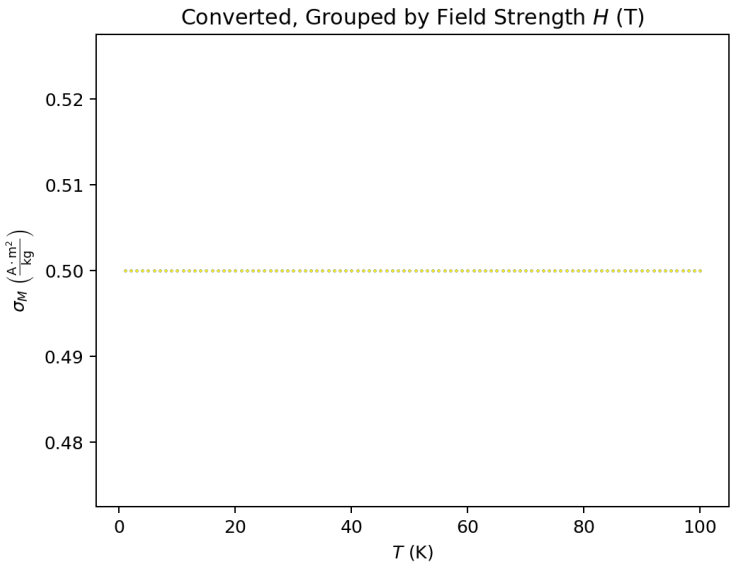


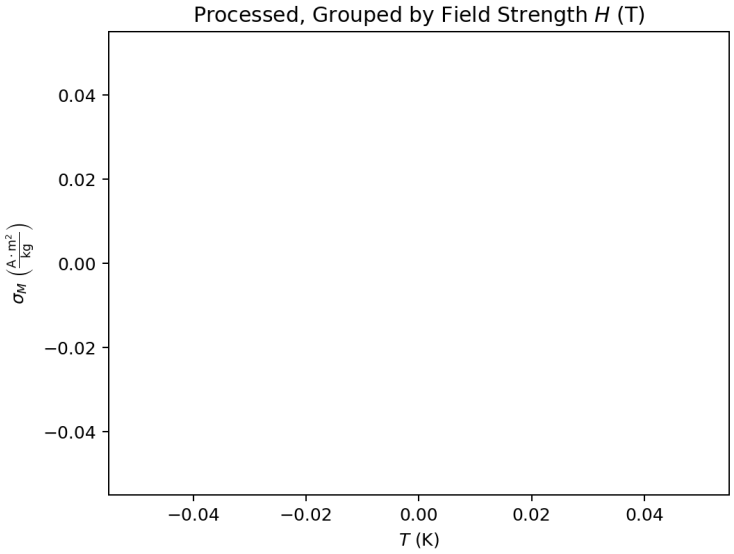

Converted, Grouped by Field Strength H (T)
M $\left(\frac{A \cdot m^2}{kg}\right)$
T (K)
0
20
40
60
80
100

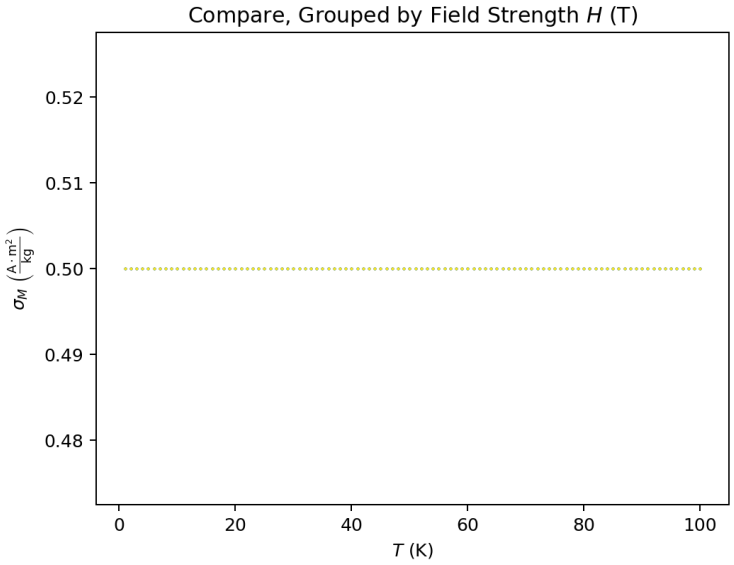

Processed, Grouped by Field Strength H (T)
M $\left(\frac{A \cdot m^2}{kg}\right)$
T (K)
0
20
40
60
80
100

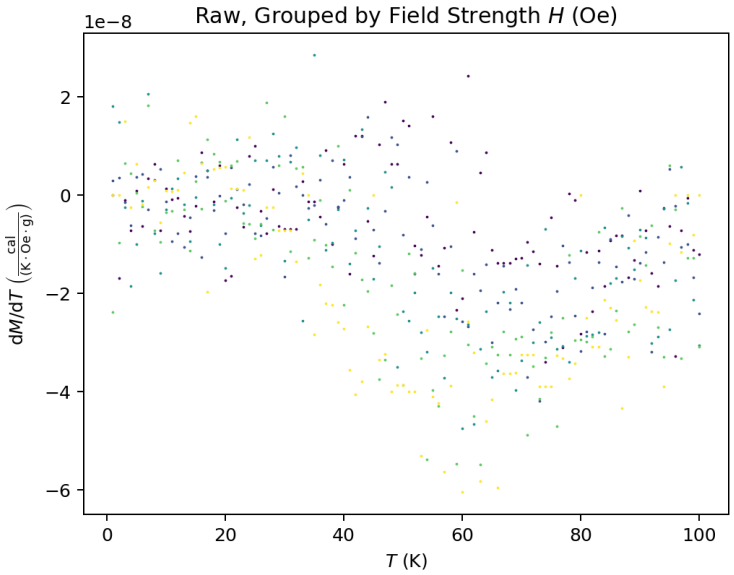

Compare, Grouped by Field Strength H (T)
M $\left(\frac{A \cdot m^2}{kg}\right)$
0
20
40
60
80
100
T (K)

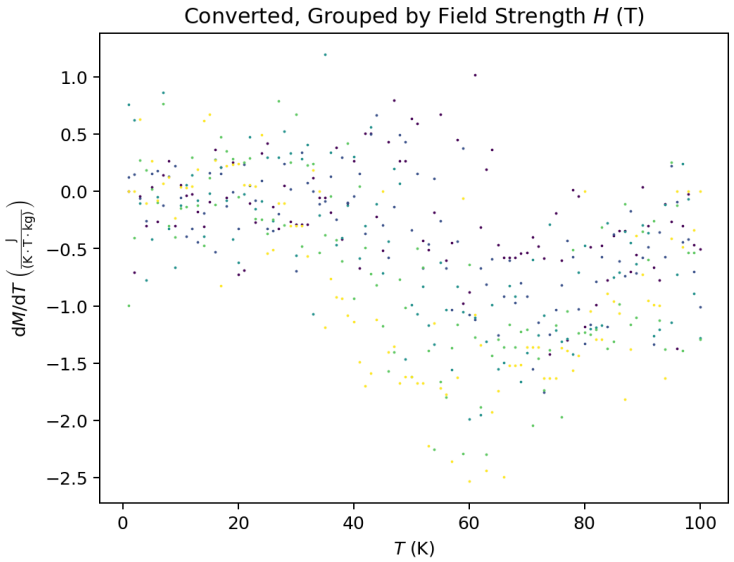

Raw, Grouped by Field Strength H (Oe)
σ_M $\left(\frac{emu}{g}\right)$
0.48
0.49
0.50
0.51
0.52
0
20
40
60
80
100
T (K)

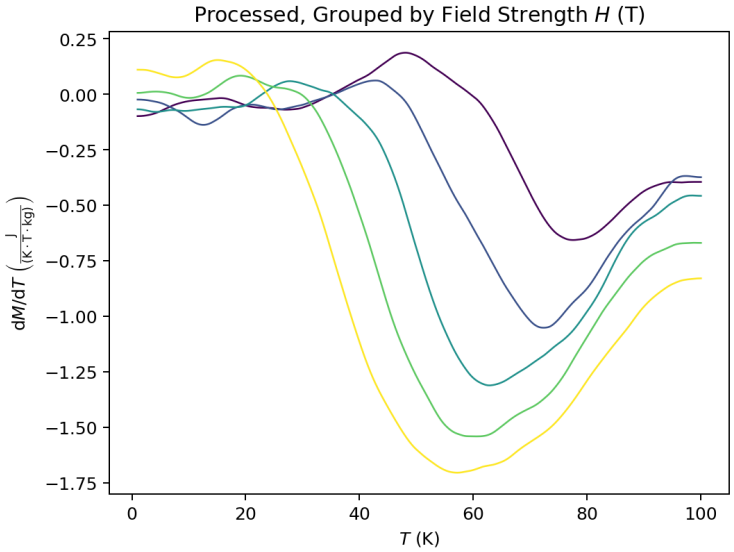

Converted, Grouped by Field Strength H (T)
0.52
0.51
0.50
0.49
0.48
$\sigma_M \left(\frac{A \cdot m^2}{kg}\right)$
0
20
40
60
80
100
T (K)

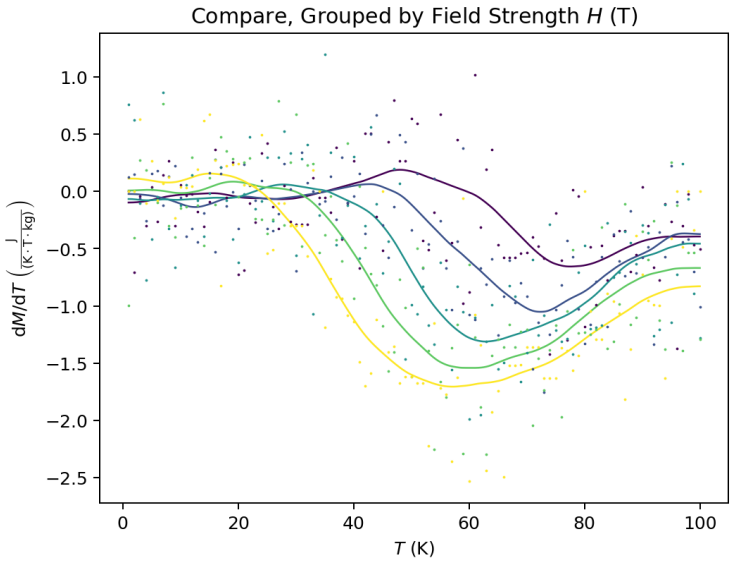

Processed, Grouped by Field Strength H (T)
0.04
0.02
0.00
−0.02
−0.04
$\sigma_M \left(\frac{A \cdot m^2}{kg}\right)$
−0.04
−0.02
0.00
0.02
0.04
T (K)

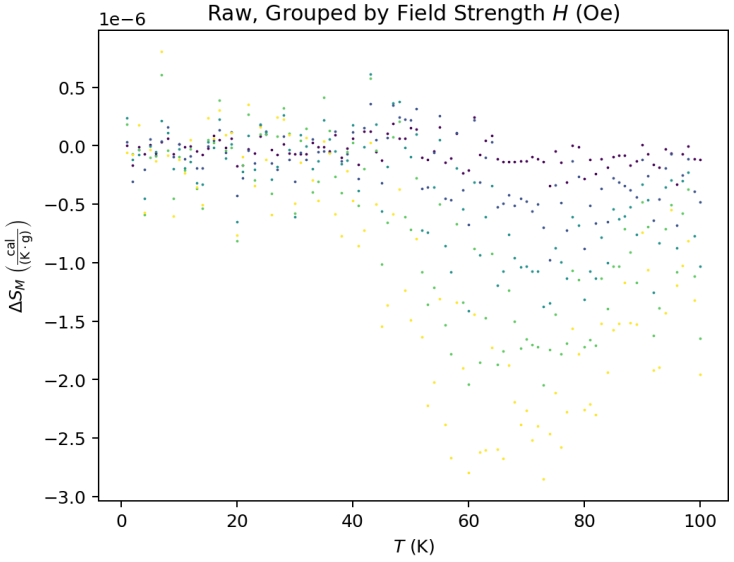

Compare, Grouped by Field Strength H (T)
σ_M $\left(\frac{A \cdot m^2}{kg}\right)$
0.52
0.51
0.50
0.49
0.48
0
20
40
60
80
100
T (K)

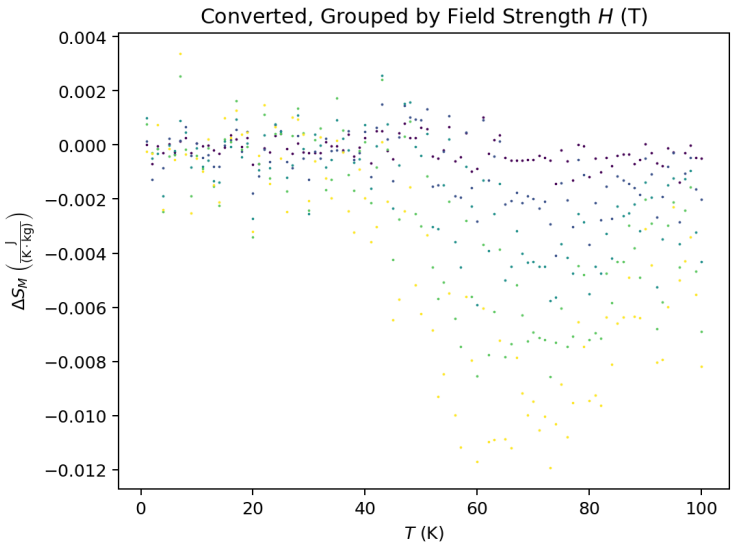

Raw, Grouped by Field Strength H (Oe)
1e−8
dM/dT $\left(\frac{cal}{(K \cdot Oe \cdot g)}\right)$
2
0
−2
−4
−6
0
20
40
60
80
100
T (K)

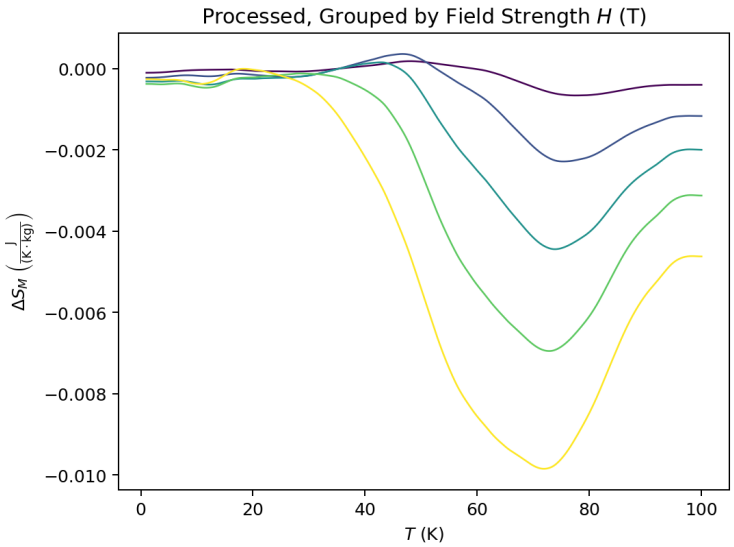

Converted, Grouped by Field Strength H (T)
dM/dT $\left(\frac{J}{(K \cdot T \cdot kg)}\right)$
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
0
20
40
60
80
100
T (K)

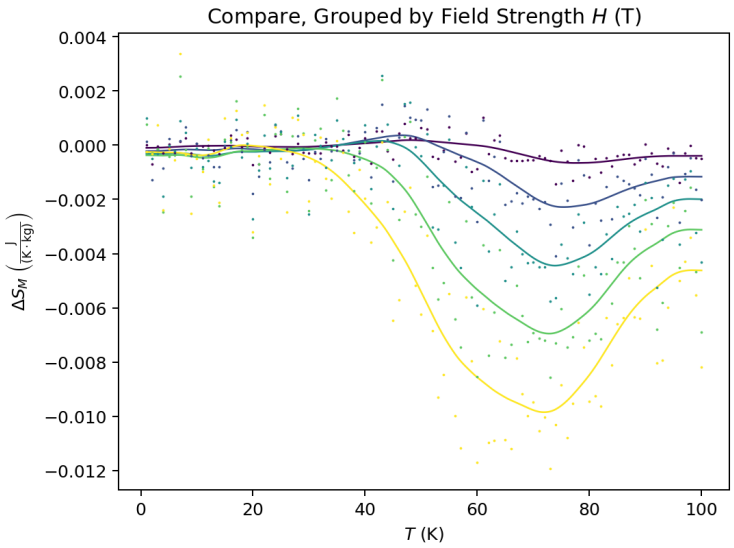

Processed, Grouped by Field Strength H (T)
dM/dT $\left(\frac{J}{(K \cdot T \cdot kg)}\right)$
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
−1.50
−1.75
0
20
40
60
80
100
T (K)

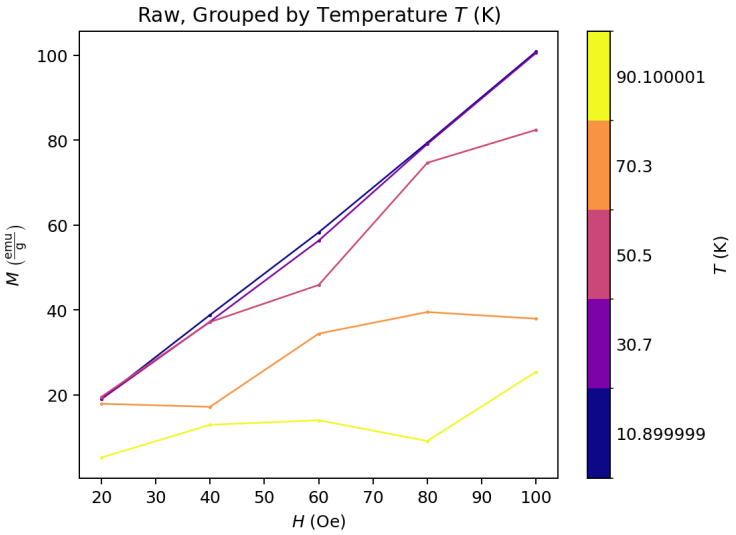

Compare, Grouped by Field Strength H (T)
dM/dT (J/(K·T·kg))
T (K)

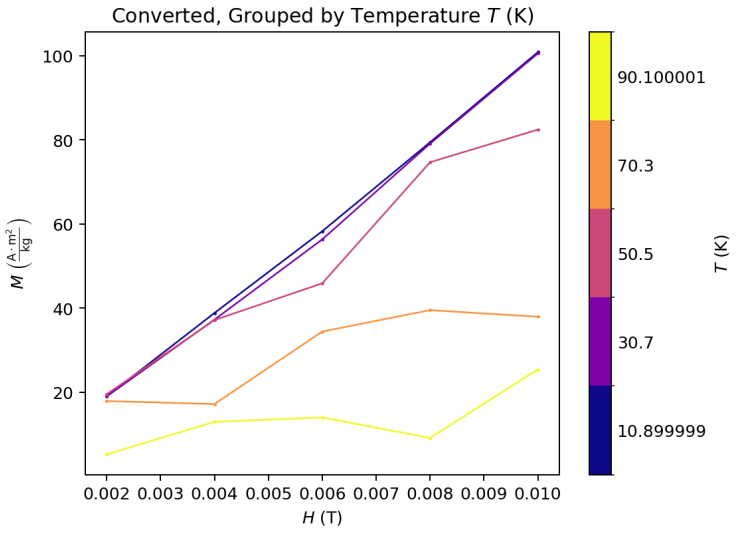

Raw, Grouped by Field Strength H (Oe)
1e−6
ΔS_M (cal/(K·g))
T (K)

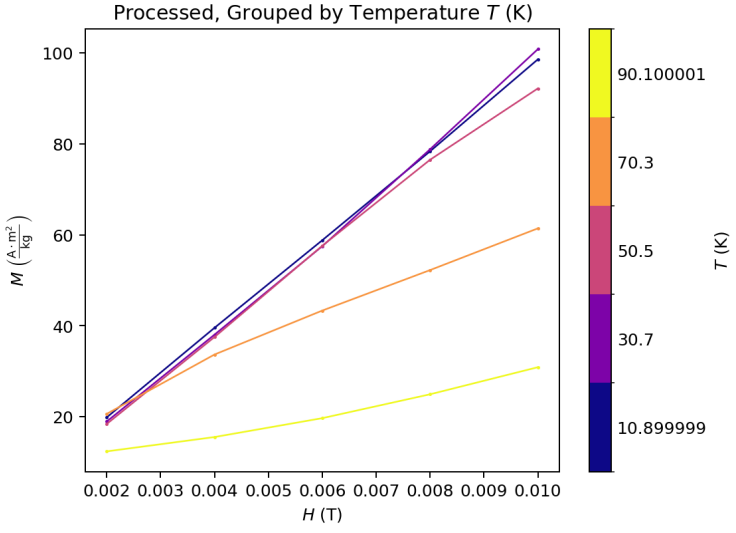

Converted, Grouped by Field Strength H (T)
$\Delta S_M \left(\frac{J}{(K \cdot kg)}\right)$
T (K)

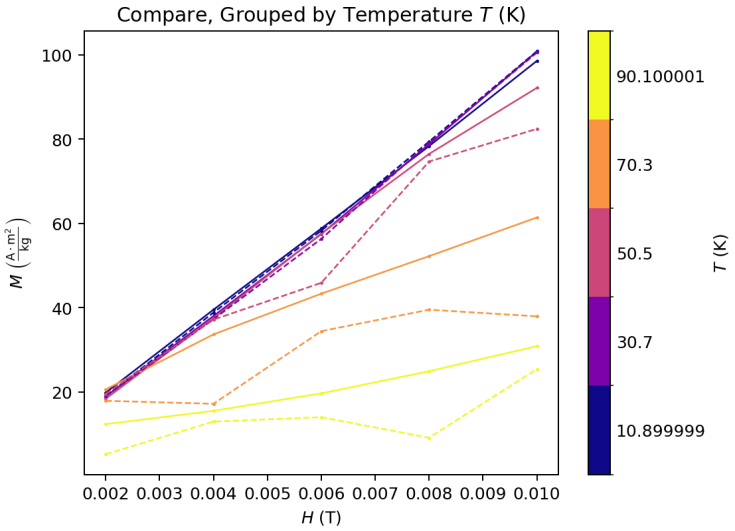

Processed, Grouped by Field Strength H (T)
$\Delta S_M \left(\frac{J}{(K \cdot kg)}\right)$
T (K)

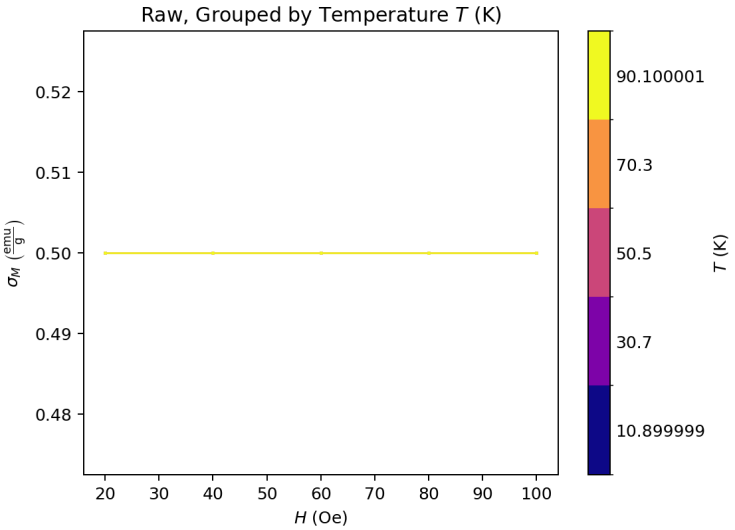

Compare, Grouped by Field Strength H (T)
$\Delta S_M \left(\frac{J}{(K \cdot kg)}\right)$
T (K)

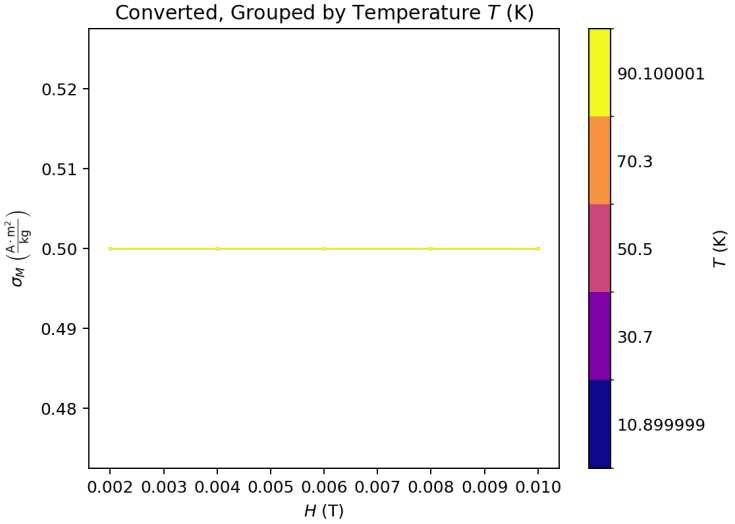

Raw, Grouped by Temperature T (K)
M $\left(\frac{\text{emu}}{\text{g}}\right)$
H (Oe)
T (K)
90.100001
70.3
50.5
30.7
10.899999

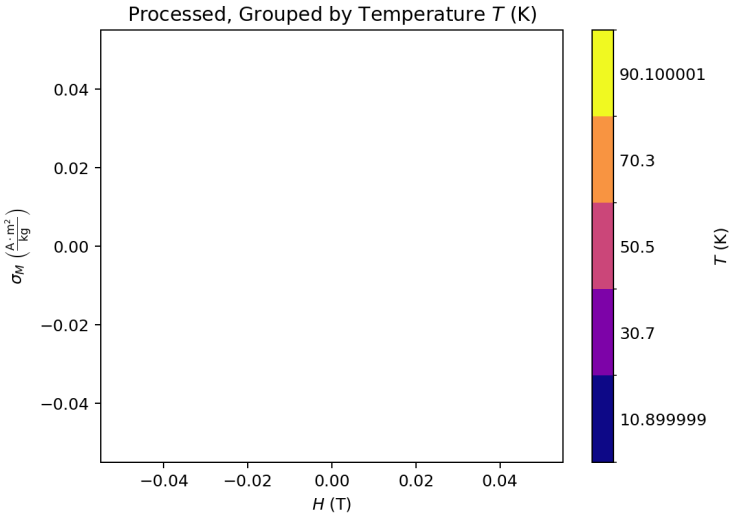

Converted, Grouped by Temperature T (K)
M $\left(\frac{\text{A}\cdot\text{m}^2}{\text{kg}}\right)$
H (T)
T (K)
90.100001
70.3
50.5
30.7
10.899999

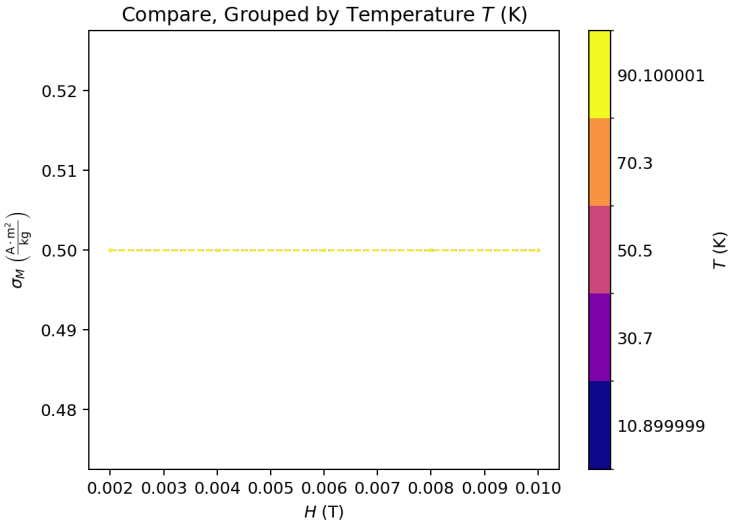

Processed, Grouped by Temperature T (K)
M $\left(\frac{\text{A}\cdot\text{m}^2}{\text{kg}}\right)$
H (T)
T (K)
90.100001
70.3
50.5
30.7
10.899999

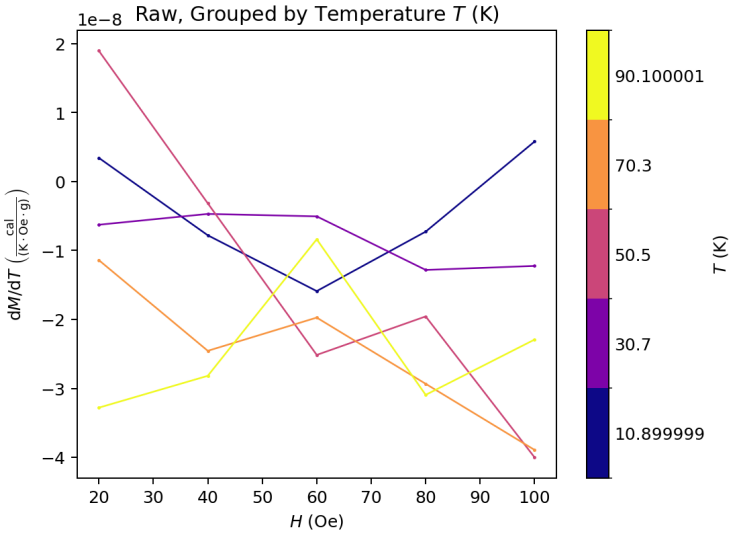

Compare, Grouped by Temperature T (K)
M $\left(\frac{\text{A}\cdot\text{m}^2}{\text{kg}}\right)$
100
80
60
40
20
0.002 0.003 0.004 0.005 0.006 0.007 0.008 0.009 0.010
H (T)
90.100001
70.3
50.5
30.7
10.899999
T (K)

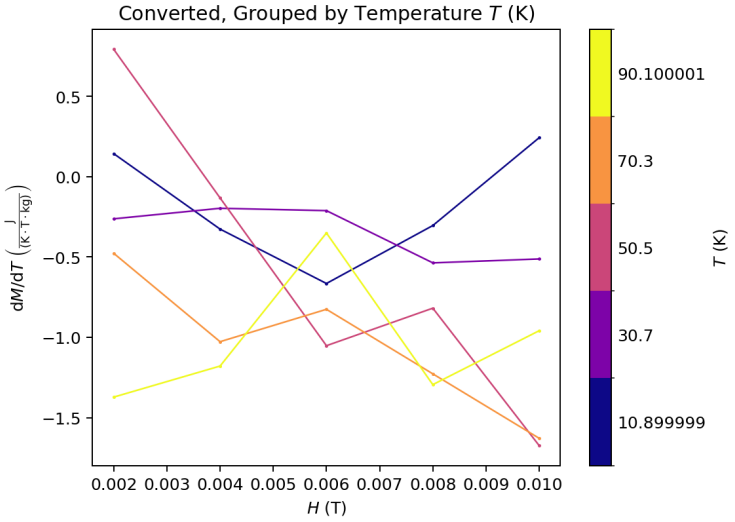

Raw, Grouped by Temperature T (K)
σ_M $\left(\frac{\text{emu}}{\text{g}}\right)$
0.52
0.51
0.50
0.49
0.48
20 30 40 50 60 70 80 90 100
H (Oe)
90.100001
70.3
50.5
30.7
10.899999
T (K)

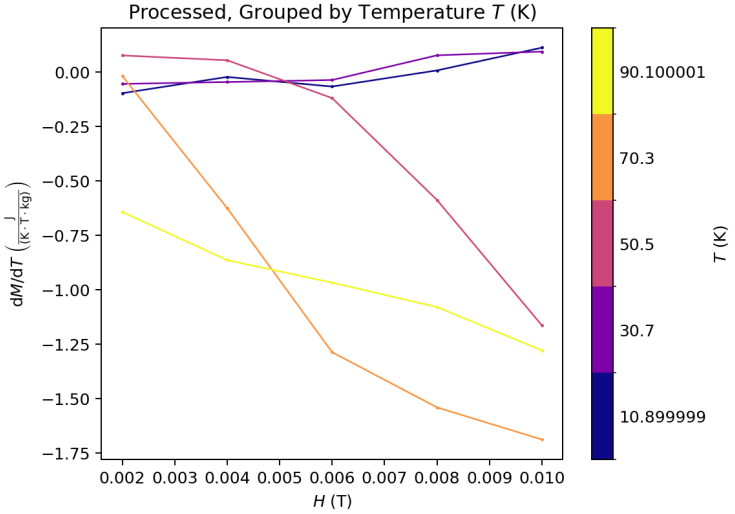

Converted, Grouped by Temperature T (K)
σ_M $\left(\frac{\text{A}\cdot\text{m}^2}{\text{kg}}\right)$
0.52
0.51
0.50
0.49
0.48
0.002 0.003 0.004 0.005 0.006 0.007 0.008 0.009 0.010
H (T)
90.100001
70.3
50.5
30.7
10.899999
T (K)

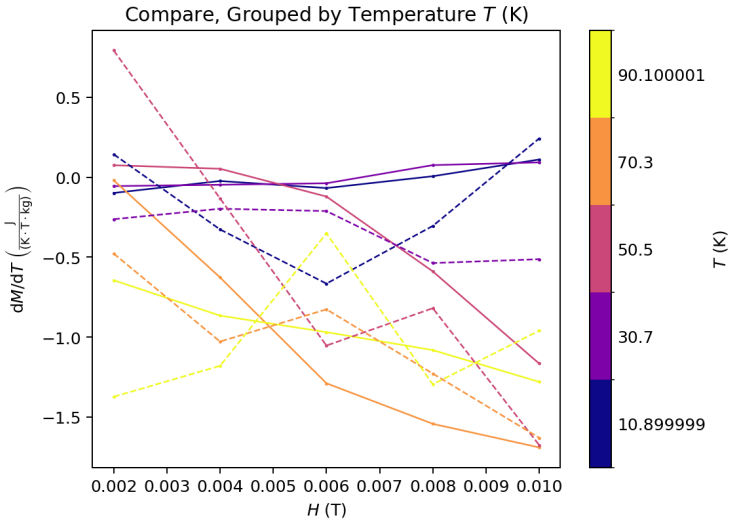

Processed, Grouped by Temperature T (K)
0.04
0.02
0.00
−0.02
−0.04
$\sigma_M \left(\frac{A \cdot m^2}{kg}\right)$
−0.04
−0.02
0.00
0.02
0.04
H (T)
90.100001
70.3
50.5
30.7
10.899999
T (K)

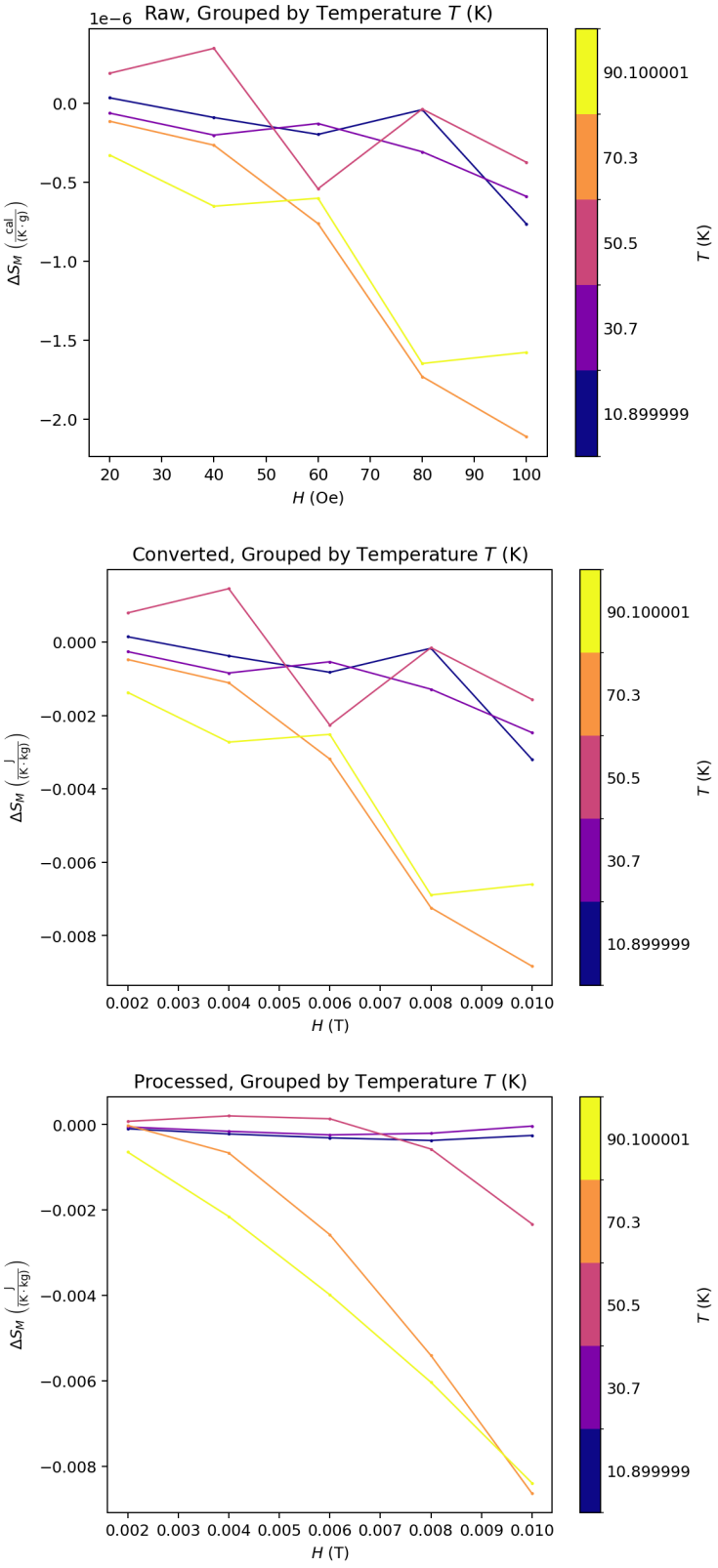

Compare, Grouped by Temperature T (K)
0.52
0.51
0.50
0.49
0.48
$\sigma_M \left(\frac{A \cdot m^2}{kg}\right)$
0.002 0.003 0.004 0.005 0.006 0.007 0.008 0.009 0.010
H (T)
90.100001
70.3
50.5
30.7
10.899999
T (K)

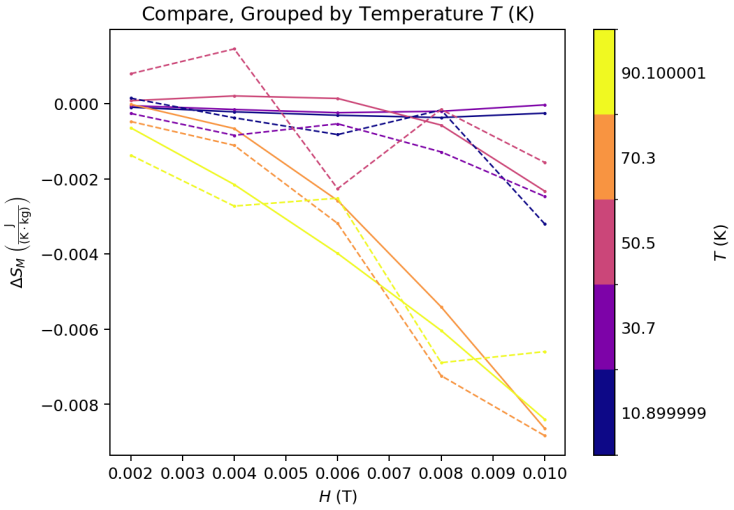

1e−8
Raw, Grouped by Temperature T (K)
2
1
0
−1
−2
−3
−4
$dM/dT \left(\frac{cal}{(K \cdot Oe \cdot g)}\right)$
20 30 40 50 60 70 80 90 100
H (Oe)
90.100001
70.3
50.5
30.7
10.899999
T (K)

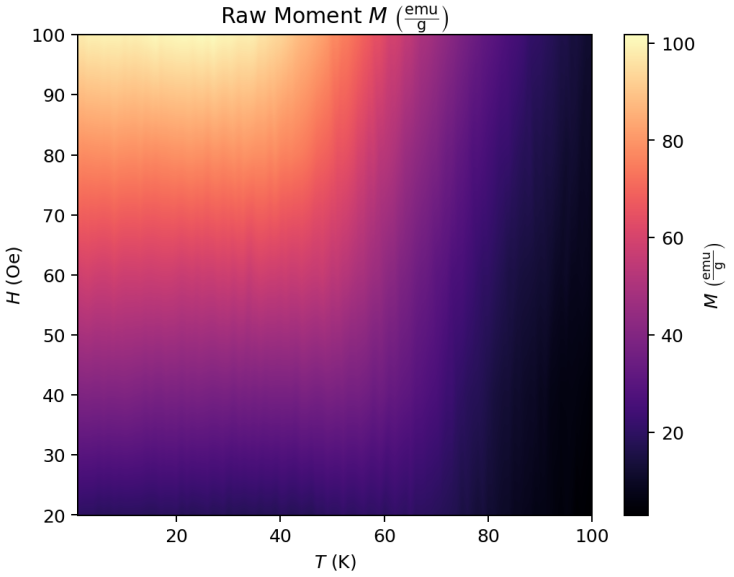

Converted, Grouped by Temperature T (K)
$dM/dT\ \left(\frac{J}{(K \cdot T \cdot kg)}\right)$
0.5
0.0
−0.5
−1.0
−1.5
0.002 0.003 0.004 0.005 0.006 0.007 0.008 0.009 0.010
H (T)
90.100001
70.3
50.5
30.7
10.899999
T (K)

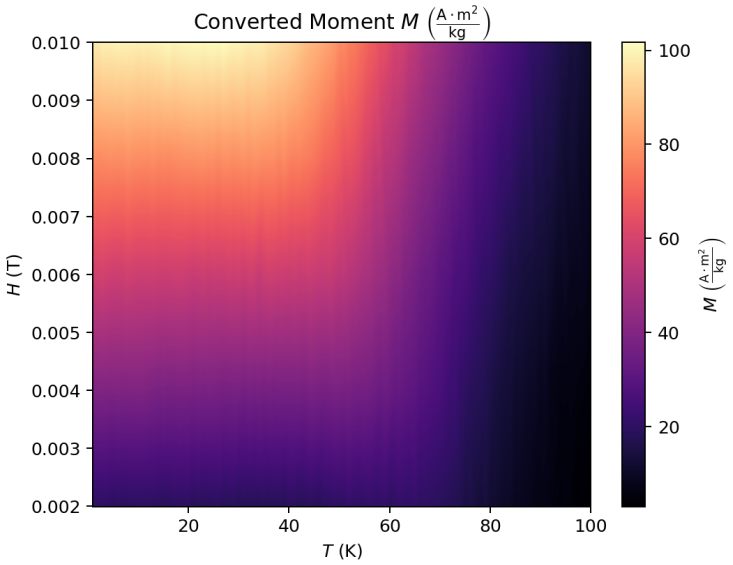

Processed, Grouped by Temperature T (K)
$dM/dT\ \left(\frac{J}{(K \cdot T \cdot kg)}\right)$
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
−1.50
−1.75
0.002 0.003 0.004 0.005 0.006 0.007 0.008 0.009 0.010
H (T)
90.100001
70.3
50.5
30.7
10.899999
T (K)

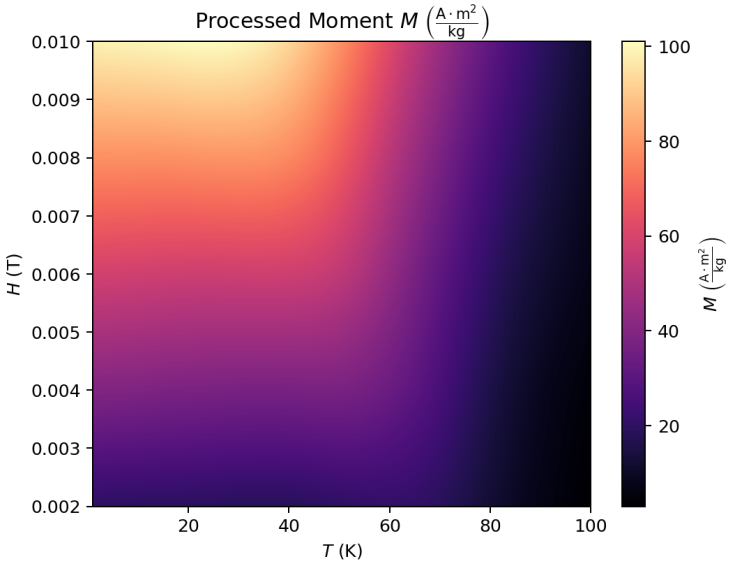

Compare, Grouped by Temperature T (K)
$dM/dT\ \left(\frac{J}{(K \cdot T \cdot kg)}\right)$
0.5
0.0
−0.5
−1.0
−1.5
0.002 0.003 0.004 0.005 0.006 0.007 0.008 0.009 0.010
H (T)
90.100001
70.3
50.5
30.7
10.899999
T (K)

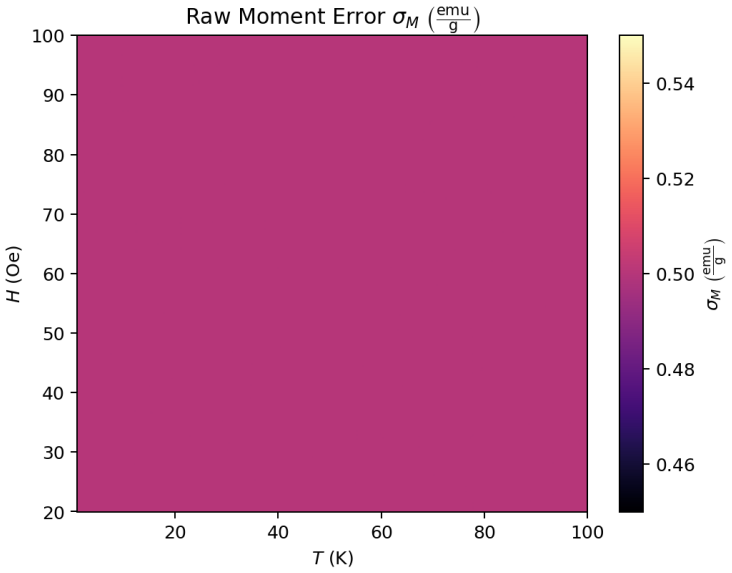

1e−6
Raw, Grouped by Temperature T (K)
ΔS_M (cal/(K·g))
H (Oe)
Converted, Grouped by Temperature T (K)
ΔS_M (J/(K·kg))
H (T)
Processed, Grouped by Temperature T (K)
ΔS_M (J/(K·kg))
H (T)
T (K)
90.100001
70.3
50.5
30.7
10.899999

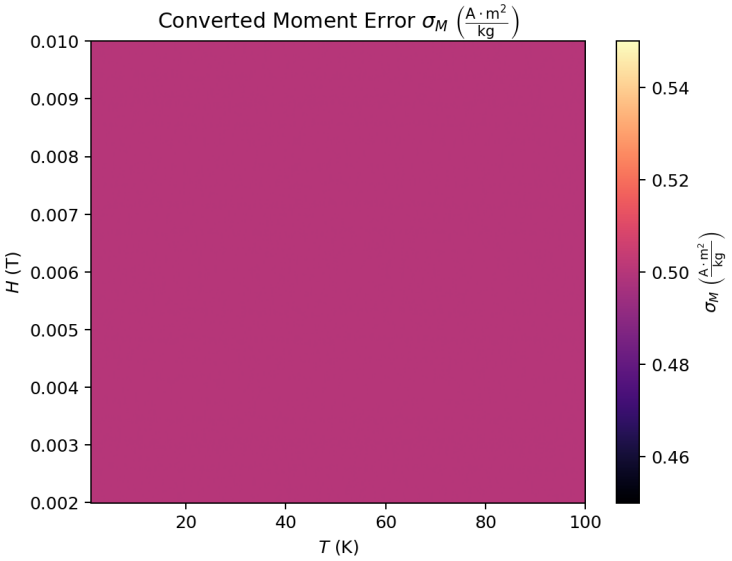

Compare, Grouped by Temperature T (K)
$\Delta S_M \left(\frac{J}{(K \cdot kg)}\right)$
0.000
−0.002
−0.004
−0.006
−0.008
0.002 0.003 0.004 0.005 0.006 0.007 0.008 0.009 0.010
H (T)
90.100001
70.3
50.5
30.7
10.899999
T (K)

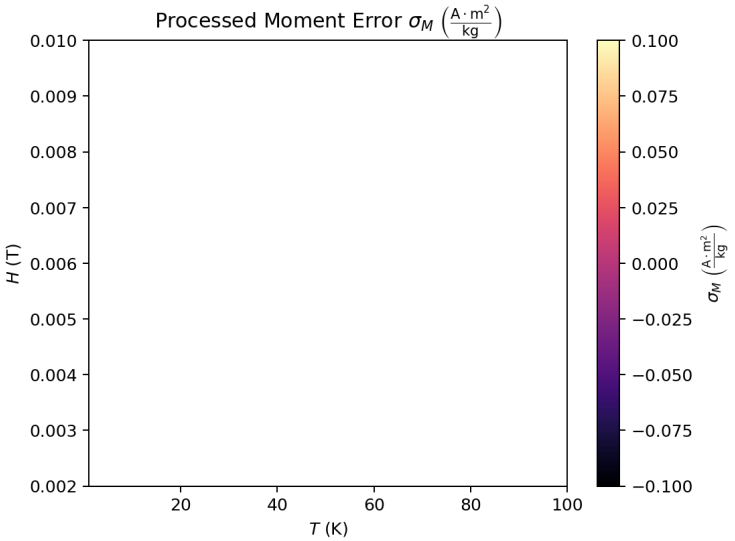

Raw Moment M $\left(\frac{emu}{g}\right)$
H (Oe)
100
90
80
70
60
50
40
30
20
20 40 60 80 100
T (K)
100
80
60
40
20
M $\left(\frac{emu}{g}\right)$

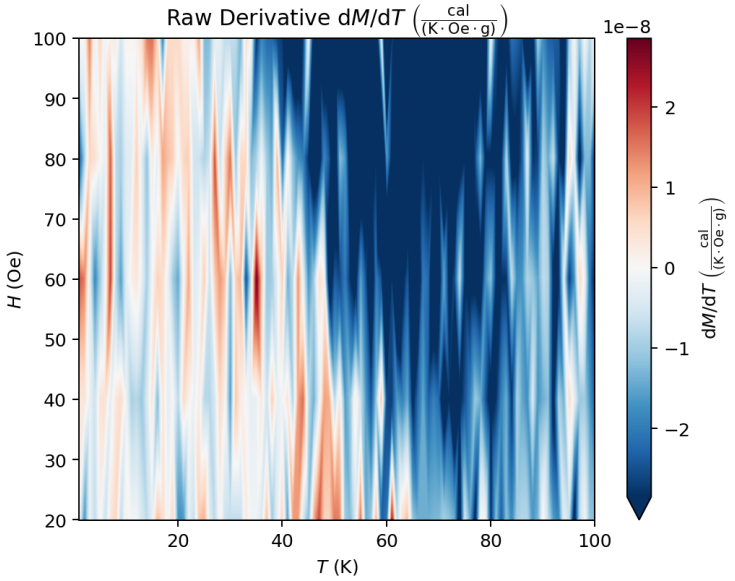

Converted Moment M $\left(\frac{\mathrm{A}\cdot\mathrm{m}^2}{\mathrm{kg}}\right)$
0.010
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
H (T)
20
40
60
80
100
T (K)
100
80
60
40
20
M $\left(\frac{\mathrm{A}\cdot\mathrm{m}^2}{\mathrm{kg}}\right)$

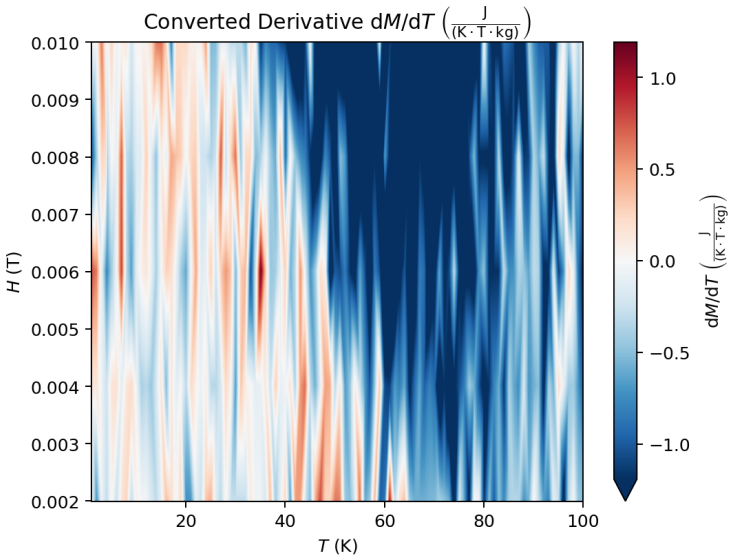

Processed Moment M $\left(\frac{\mathrm{A}\cdot\mathrm{m}^2}{\mathrm{kg}}\right)$
0.010
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
H (T)
20
40
60
80
100
T (K)
100
80
60
40
20
M $\left(\frac{\mathrm{A}\cdot\mathrm{m}^2}{\mathrm{kg}}\right)$

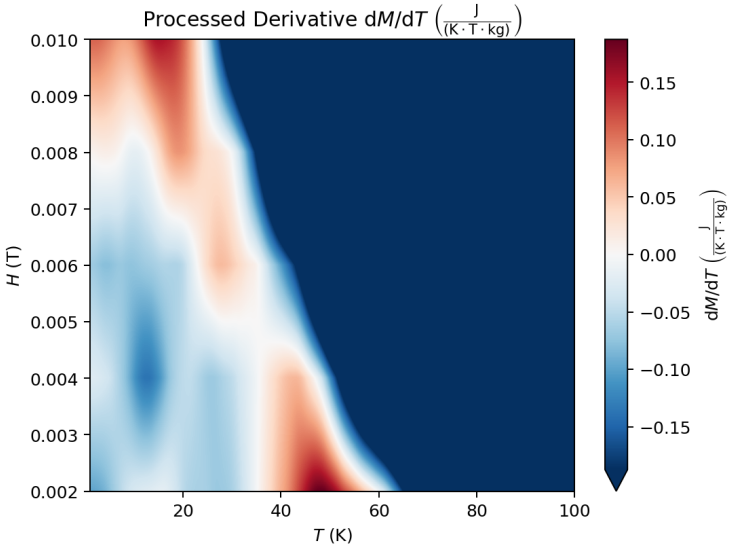

Raw Moment Error σ_M $\left(\frac{\text{emu}}{\text{g}}\right)$
H (Oe)
T (K)
σ_M $\left(\frac{\text{emu}}{\text{g}}\right)$

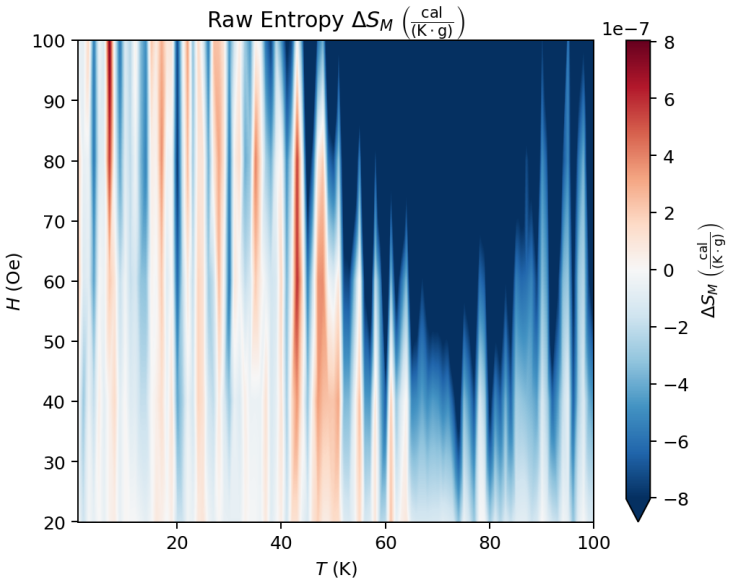

Converted Moment Error σ_M $\left(\frac{\text{A}\cdot\text{m}^2}{\text{kg}}\right)$
H (T)
T (K)
σ_M $\left(\frac{\text{A}\cdot\text{m}^2}{\text{kg}}\right)$

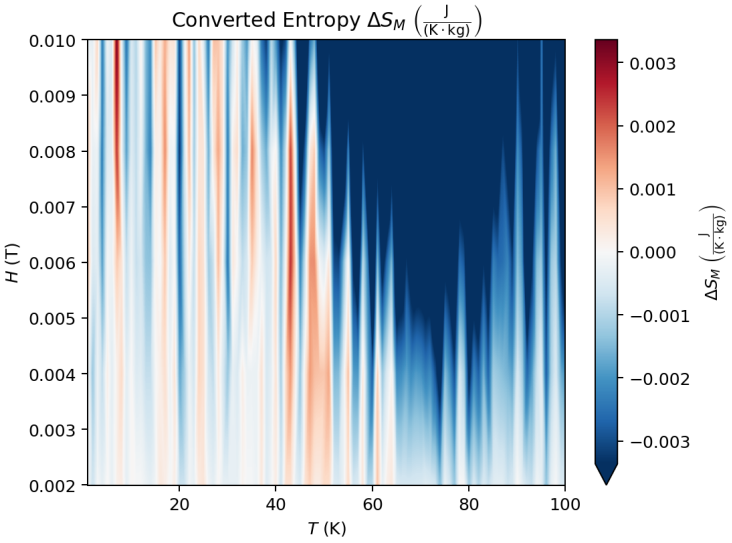

Processed Moment Error σ_M $\left(\frac{\mathrm{A \cdot m^2}}{\mathrm{kg}}\right)$
0.010
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
H (T)
20
40
60
80
100
T (K)
0.100
0.075
0.050
0.025
0.000
−0.025
−0.050
−0.075
−0.100
σ_M $\left(\frac{\mathrm{A \cdot m^2}}{\mathrm{kg}}\right)$

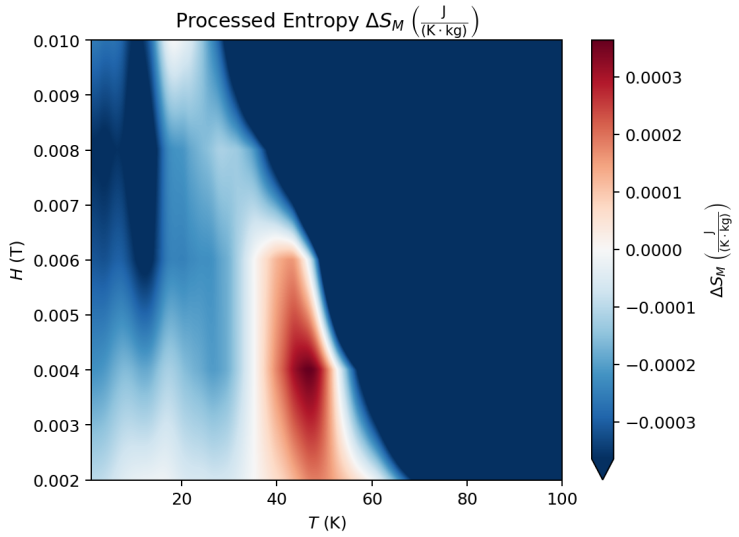

Raw Derivative dM/dT $\left(\frac{\mathrm{cal}}{\mathrm{(K \cdot Oe \cdot g)}}\right)$
100
90
80
70
60
50
40
30
20
H (Oe)
20
40
60
80
100
T (K)
1e−8
2
1
0
−1
−2
dM/dT $\left(\frac{\mathrm{cal}}{\mathrm{(K \cdot Oe \cdot g)}}\right)$


Converted Derivative dM/dT $\left(\frac{\mathrm{J}}{(\mathrm{K}\cdot\mathrm{T}\cdot\mathrm{kg})}\right)$
H (T)
T (K)
dM/dT $\left(\frac{\mathrm{J}}{(\mathrm{K}\cdot\mathrm{T}\cdot\mathrm{kg})}\right)$


Processed Derivative dM/dT $\left(\frac{\mathrm{J}}{(\mathrm{K}\cdot\mathrm{T}\cdot\mathrm{kg})}\right)$
H (T)
T (K)
dM/dT $\left(\frac{\mathrm{J}}{(\mathrm{K}\cdot\mathrm{T}\cdot\mathrm{kg})}\right)$


Raw Entropy ΔS_M $\left(\frac{\text{cal}}{(\text{K}\cdot\text{g})}\right)$
1e−7
8
6
4
2
0
−2
−4
−6
−8
ΔS_M $\left(\frac{\text{cal}}{(\text{K}\cdot\text{g})}\right)$
100
90
80
70
60
50
40
30
20
H (Oe)
20
40
60
80
100
T (K)


Converted Entropy ΔS_M $\left(\frac{\text{J}}{(\text{K}\cdot\text{kg})}\right)$
0.003
0.002
0.001
0.000
−0.001
−0.002
−0.003
ΔS_M $\left(\frac{\text{J}}{(\text{K}\cdot\text{kg})}\right)$
0.010
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
H (T)
20
40
60
80
100
T (K)



3.2.4 Units and Conversions

Note: MagentroPy uses the `pint` package internally for unit conversions, as well as the `pint_pandas` package to extend functionality to `DataFrame`s. If you happen to be using `pint_pandas` as well, be aware that *`MagentroData`*'s `pint.UnitRegistry` is set to `pint_pandas.PintType.ureg`, and it is reset with the necessary definitions and conversion contexts each time conversions are performed.

Tip: Raw data units can be set during instantiation:

```
magdata = MagentroData(..., raw_data_units={...})
```

```
from IPython.display import display, HTML
from magentropy import MagentroData

magdata = MagentroData('magdata.csv', qd_dat=False, comment_col=None, T='T', H='H', M='M
↪', M_err='M_err')
magdata.process_data()

def data_heads():
    display(HTML('<p>Raw data:</p>'))
    display(magdata.raw_df_with_units.head(3))
    display(HTML('<p>Converted data:</p>'))
    display(magdata.converted_df_with_units.head(3))
    display(HTML('<p>Processed data:</p>'))
    display(magdata.processed_df_with_units.head(3))
```

```
The data contains the following 5 magnetic field strengths and observations per field:
20.0     100
40.0     100
60.0     100
```

(continues on next page)

(continued from previous page)

```
80.0     100
100.0    100
Name: T, dtype: int64

Processing data using the following settings:
{
        npoints: 1000,
        temp_range: [-inf  inf],
        fields: [],
        decimals: 5,
        max_diff: inf,
        min_sweep_len: 10,
        d_order: 2,
        lmbds: [nan],
        lmbd_guess: 0.0001,
        weight_err: True,
        match_err: False,
        min_kwargs: {'method': 'Nelder-Mead', 'bounds': ((-inf, inf),), 'options': {
↪'maxfev': 50, 'xatol': 0.01, 'fatol': 1e-06}},
        add_zeros: False
}

scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 20.0
scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 40.0
scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 60.0
scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 80.0
scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 100.0
Calculated raw derivative and entropy.

last_presets set to:
{
        npoints: 1000,
        temp_range: [  0.99999934 100.00000083],
        fields: [ 20.  40.  60.  80. 100.],
        decimals: 5,
        max_diff: inf,
        min_sweep_len: 10,
        d_order: 2,
        lmbds: [0.00091728 0.00054639 0.00072862 0.00091728 0.00095775],
        lmbd_guess: 0.0001,
        weight_err: True,
        match_err: False,
        min_kwargs: {'method': 'Nelder-Mead', 'bounds': ((-inf, inf),), 'options': {
↪'maxfev': 50, 'xatol': 0.01, 'fatol': 1e-06}},
        add_zeros: False
}
```

(continues on next page)

(continued from previous page)

```
Finished.
```

Above, we've read a `.csv` file and processed the data. The default sample mass of 1.0 mg was used. Luckily, the sample mass and all of the raw data units can be set at any time, before or after processing, with the appropriate conversions applied automatically.

Right now, the data looks like this:

```
data_heads()
```

```
<IPython.core.display.HTML object>
```

```
             T          H         M    M_err M_per_mass M_per_mass_err  \
unit         K         Oe       emu      emu      emu/g          emu/g
0     1.000000  20.000001  0.002023  0.00005   2.023238           0.05
1     2.000000  20.000000  0.001977  0.00005   1.977035           0.05
2     3.000001  19.999998  0.001969  0.00005   1.969118           0.05

             dM_dT      Delta_SM
unit    cal/K/Oe/g       cal/K/g
0     4.245595e-23  8.491189e-22
1    -1.692885e-09 -1.692885e-08
2    -9.980543e-11 -9.980543e-10
```

```
<IPython.core.display.HTML object>
```

```
             T      H         M         M_err M_per_mass M_per_mass_err  \
unit         K      T      A·m²          A·m²    A·m²/kg        A·m²/kg
0     1.000000  0.002  0.000002  5.000000e-08   2.023238           0.05
1     2.000000  0.002  0.000002  5.000000e-08   1.977035           0.05
2     3.000001  0.002  0.000002  5.000000e-08   1.969118           0.05

             dM_dT      Delta_SM
unit      J/K/T/kg        J/K/kg
0     1.776357e-15  3.552714e-18
1    -7.083031e-02 -7.083031e-05
2    -4.175859e-03 -4.175859e-06
```

```
<IPython.core.display.HTML object>
```

```
             T      H         M M_err M_per_mass M_per_mass_err     dM_dT  \
unit         K      T      A·m²  A·m²    A·m²/kg        A·m²/kg  J/K/T/kg
0     0.999999  0.002  0.000002   NaN   1.984700            NaN -0.009827
1     1.099098  0.002  0.000002   NaN   1.983727            NaN -0.009824
2     1.198198  0.002  0.000002   NaN   1.982753            NaN -0.009820

     Delta_SM
unit   J/K/kg
0    -0.00001
1    -0.00001
2    -0.00001
```

Sample mass

To change the sample mass, use the *sample_mass* attribute:

```
magdata.sample_mass = 0.1
data_heads()
```

```
<IPython.core.display.HTML object>
```

```
             T          H         M    M_err M_per_mass M_per_mass_err  \
unit         K         Oe       emu      emu      emu/g          emu/g
0     1.000000  20.000001  0.002023  0.00005  20.232376            0.5
1     2.000000  20.000000  0.001977  0.00005  19.770351            0.5
2     3.000001  19.999998  0.001969  0.00005  19.691176            0.5

             dM_dT      Delta_SM
unit    cal/K/Oe/g       cal/K/g
0     4.245595e-22  8.491189e-21
1    -1.692885e-08 -1.692885e-07
2    -9.980543e-10 -9.980543e-09
```

```
<IPython.core.display.HTML object>
```

```
             T      H         M         M_err M_per_mass M_per_mass_err  \
unit         K      T      A·m²          A·m²    A·m²/kg        A·m²/kg
0     1.000000  0.002  0.000002  5.000000e-08  20.232376            0.5
1     2.000000  0.002  0.000002  5.000000e-08  19.770351            0.5
2     3.000001  0.002  0.000002  5.000000e-08  19.691176            0.5

             dM_dT      Delta_SM
unit      J/K/T/kg        J/K/kg
0     1.776357e-14  3.552714e-17
1    -7.083031e-01 -7.083031e-04
2    -4.175859e-02 -4.175859e-05
```

```
<IPython.core.display.HTML object>
```

```
             T      H         M M_err M_per_mass M_per_mass_err     dM_dT  \
unit         K      T      A·m²  A·m²    A·m²/kg        A·m²/kg  J/K/T/kg
0     0.999999  0.002  0.000002   NaN  19.847003             NaN -0.098265
1     1.099098  0.002  0.000002   NaN  19.837267             NaN -0.098240
2     1.198198  0.002  0.000002   NaN  19.827533             NaN -0.098202

      Delta_SM
unit    J/K/kg
0    -0.000098
1    -0.000098
2    -0.000098
```

All of the per-mass columns (`'M_per_mass'`, `'M_per_mass_err'`, `'dM_dT'`, and `'Delta_SM'`) have increased by a factor of 10.

Similarly, one can change the units:

```
magdata.set_raw_data_units(sample_mass='g')
data_heads()
```

```
sample_mass will have units of g
```

```
<IPython.core.display.HTML object>
```

```
             T          H         M    M_err M_per_mass M_per_mass_err  \
unit         K         Oe       emu      emu      emu/g          emu/g
0     1.000000  20.000001  0.002023  0.00005   0.020232         0.0005
1     2.000000  20.000000  0.001977  0.00005   0.019770         0.0005
2     3.000001  19.999998  0.001969  0.00005   0.019691         0.0005

             dM_dT      Delta_SM
unit    cal/K/Oe/g       cal/K/g
0     4.245595e-25  8.491189e-24
1    -1.692885e-11 -1.692885e-10
2    -9.980543e-13 -9.980543e-12
```

```
<IPython.core.display.HTML object>
```

```
             T      H         M         M_err M_per_mass M_per_mass_err  \
unit         K      T      A·m²          A·m²    A·m²/kg        A·m²/kg
0     1.000000  0.002  0.000002  5.000000e-08   0.020232         0.0005
1     2.000000  0.002  0.000002  5.000000e-08   0.019770         0.0005
2     3.000001  0.002  0.000002  5.000000e-08   0.019691         0.0005

             dM_dT      Delta_SM
unit      J/K/T/kg        J/K/kg
0     1.776357e-17  3.552714e-20
1    -7.083031e-04 -7.083031e-07
2    -4.175859e-05 -4.175859e-08
```

```
<IPython.core.display.HTML object>
```

```
             T      H         M M_err M_per_mass M_per_mass_err     dM_dT  \
unit         K      T      A·m²  A·m²    A·m²/kg        A·m²/kg  J/K/T/kg
0     0.999999  0.002  0.000002   NaN   0.019847             NaN -0.000098
1     1.099098  0.002  0.000002   NaN   0.019837             NaN -0.000098
2     1.198198  0.002  0.000002   NaN   0.019828             NaN -0.000098

          Delta_SM
unit        J/K/kg
0    -9.826538e-08
1    -9.823999e-08
2    -9.820192e-08
```

The per-mass columns have now decreased by a factor of 1000.

`pint`'s unit string parsing is used to change the units. The logged output confirms that the unit change is as expected.

```
magdata.set_raw_data_units(sample_mass='gram')
magdata.set_raw_data_units(sample_mass='grams')
```

```
sample_mass will have units of g
sample_mass will have units of g
```

Both the magnitude and the units can be set at once using *sample_mass_with_units*:

```
magdata.sample_mass_with_units
```

```
(0.1, 'g')
```

```
magdata.sample_mass_with_units = (0.5, 'milligrams')
data_heads()
```

```
sample_mass will have units of mg
```

```
<IPython.core.display.HTML object>
```

```
             T          H         M    M_err M_per_mass M_per_mass_err  \
unit         K         Oe       emu      emu      emu/g          emu/g
0     1.000000  20.000001  0.002023  0.00005   4.046475            0.1
1     2.000000  20.000000  0.001977  0.00005   3.954070            0.1
2     3.000001  19.999998  0.001969  0.00005   3.938235            0.1

             dM_dT      Delta_SM
unit    cal/K/Oe/g       cal/K/g
0     8.491189e-23  1.698238e-21
1    -3.385770e-09 -3.385770e-08
2    -1.996109e-10 -1.996109e-09
```

```
<IPython.core.display.HTML object>
```

```
             T      H         M         M_err M_per_mass M_per_mass_err  \
unit         K      T      A·m²          A·m²    A·m²/kg        A·m²/kg
0     1.000000  0.002  0.000002  5.000000e-08   4.046475            0.1
1     2.000000  0.002  0.000002  5.000000e-08   3.954070            0.1
2     3.000001  0.002  0.000002  5.000000e-08   3.938235            0.1

             dM_dT      Delta_SM
unit      J/K/T/kg        J/K/kg
0     3.552714e-15  7.105427e-18
1    -1.416606e-01 -1.416606e-04
2    -8.351719e-03 -8.351719e-06
```

```
<IPython.core.display.HTML object>
```

```
             T      H         M M_err M_per_mass M_per_mass_err     dM_dT  \
unit         K      T      A·m²  A·m²    A·m²/kg        A·m²/kg  J/K/T/kg
0     0.999999  0.002  0.000002   NaN   3.969401            NaN -0.019653
```

(continues on next page)

(continued from previous page)

```
1     1.099098  0.002  0.000002   NaN   3.967453            NaN -0.019648
2     1.198198  0.002  0.000002   NaN   3.965507            NaN -0.019640

     Delta_SM
unit   J/K/kg
0    -0.00002
1    -0.00002
2    -0.00002
```

```
magdata.sample_mass_with_units
```

```
(0.5, 'mg')
```

Column units

The input temperature, magnetic field, and magnetic moment units can be changed using *set_raw_data_units()*. The moment error column will get the same units as the moment column. The last four columns are fixed in cgs units.

```
magdata.set_raw_data_units(T='degC', H='G', M='erg/G')
data_heads()
```

```
T will have units of ˚C
H will have units of G
M will have units of erg/G
```

```
<IPython.core.display.HTML object>
```

```
             T          H         M    M_err M_per_mass M_per_mass_err  \
unit        ˚C          G     erg/G    erg/G      emu/g          emu/g
0     1.000000  20.000001  0.002023  0.00005   4.046475            0.1
1     2.000000  20.000000  0.001977  0.00005   3.954070            0.1
2     3.000001  19.999998  0.001969  0.00005   3.938235            0.1

             dM_dT      Delta_SM
unit    cal/K/Oe/g       cal/K/g
0     8.491189e-23  1.698238e-21
1    -3.385770e-09 -3.385770e-08
2    -1.996109e-10 -1.996109e-09
```

```
<IPython.core.display.HTML object>
```

```
               T      H         M         M_err M_per_mass M_per_mass_err  \
unit           K      T      A·m²          A·m²    A·m²/kg        A·m²/kg
0     274.150000  0.002  0.000002  5.000000e-08   4.046475            0.1
1     275.150000  0.002  0.000002  5.000000e-08   3.954070            0.1
2     276.150001  0.002  0.000002  5.000000e-08   3.938235            0.1

             dM_dT      Delta_SM
unit      J/K/T/kg        J/K/kg
```

(continues on next page)

(continued from previous page)

```
0     3.552714e-15  7.105427e-18
1    -1.416606e-01 -1.416606e-04
2    -8.351719e-03 -8.351719e-06
```

```
<IPython.core.display.HTML object>
```

```
                T      H         M M_err M_per_mass M_per_mass_err     dM_dT  \
unit            K      T      A·m²  A·m²    A·m²/kg        A·m²/kg  J/K/T/kg
0     274.149999  0.002  0.000002   NaN   3.969401             NaN -0.019653
1     274.249098  0.002  0.000002   NaN   3.967453             NaN -0.019648
2     274.348198  0.002  0.000002   NaN   3.965507             NaN -0.019640

     Delta_SM
unit   J/K/kg
0    -0.00002
1    -0.00002
2    -0.00002
```

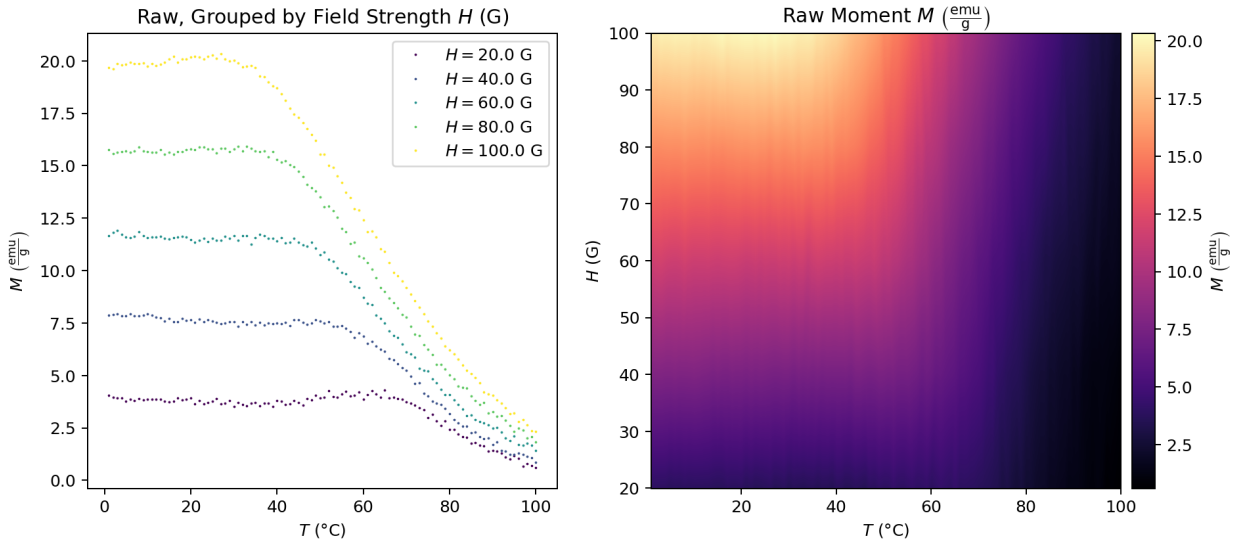
A few things to note:

1. The short string for "degrees Celcius" is `'degC'`; `'C'` stands for "Coulombs".
2. Changing the input units does not convert the input values themselves. Rather, it determines how they are converted to cgs and SI units. For example, the converted and processed data now begin at about 274.15 K. (Compare to the previous data output.)
3. In a vacuum, the conversion from Gauss to Tesla is the same as that from Oersted to Tesla (divide by 10000), so the magnetic field column is unchanged.
4. Similarly, 1 emu is defined to be 1 erg/Gauss in the context of measuring magnetic moments, so the magnetic moment column is unchanged.
5. Concluding the perhaps underwhelming lack of changes in the data, the unit scale is the same for Celcius as it is for Kelvin, so the `'dM_dT'` and `'Delta_SM'` columns, which depend only on the derivative with respect to temperature, are unchanged.

Plotting

Temperature and field units will be reflected in the default plot labels and titles when plotting the raw data.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, 2, figsize=(12, 5))

magdata.plot_lines(data_prop='M_per_mass', data_version='raw', ax=ax[0], legend=True)

magdata.plot_map(
    data_prop='M_per_mass', data_version='raw', ax=ax[1],
    colorbar_kwargs={'ax': ax, 'fraction': 0.05, 'pad': 0.01}
);
```



A note about regularization parameters

In general, regularization parameters are dependent on the scale of the data. One might think it necessary, then, to re-process the data after changing the sample mass or raw data units in order to appropriately choose regularization parameters. Internally, however, smoothing is done using the raw values themselves, and conversions happen afterwards. This removes the need to re-process after conversions.

3.2.5 Bootstrap Estimates

```
from magentropy import MagentroData

magdata = MagentroData('magdata.dat')
magdata.process_data()
```

```
"[Data]" tag detected, assuming QD .dat file.
The sample mass was determined from the QD .dat file: 0.1
The data contains the following 5 magnetic field strengths and observations per field:
20.0     100
40.0     100
60.0     100
80.0     100
100.0    100
Name: T, dtype: int64

Processing data using the following settings:
{
        npoints: 1000,
        temp_range: [-inf  inf],
        fields: [],
        decimals: 5,
        max_diff: inf,
        min_sweep_len: 10,
```

(continues on next page)

(continued from previous page)

```
        d_order: 2,
        lmbds: [nan],
        lmbd_guess: 0.0001,
        weight_err: True,
        match_err: False,
        min_kwargs: {'method': 'Nelder-Mead', 'bounds': ((-inf, inf),), 'options': {
↪'maxfev': 50, 'xatol': 0.01, 'fatol': 1e-06}},
        add_zeros: False
}

scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 20.0
scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 40.0
scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 60.0
scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 80.0
scipy.optimize.minimize: Optimization terminated successfully.
Processed M(T) at field: 100.0
Calculated raw derivative and entropy.

last_presets set to:
{
        npoints: 1000,
        temp_range: [  0.99999934 100.00000083],
        fields: [ 20.  40.  60.  80. 100.],
        decimals: 5,
        max_diff: inf,
        min_sweep_len: 10,
        d_order: 2,
        lmbds: [0.00091728 0.00054639 0.00072862 0.00091728 0.00095775],
        lmbd_guess: 0.0001,
        weight_err: True,
        match_err: False,
        min_kwargs: {'method': 'Nelder-Mead', 'bounds': ((-inf, inf),), 'options': {
↪'maxfev': 50, 'xatol': 0.01, 'fatol': 1e-06}},
        add_zeros: False
}

Finished.
```

```
magdata.processed_df
```

```
                T      H         M  M_err  M_per_mass  M_per_mass_err  \
0        0.999999  0.002  0.000002    NaN   19.847003             NaN
1        1.099098  0.002  0.000002    NaN   19.837267             NaN
2        1.198198  0.002  0.000002    NaN   19.827533             NaN
3        1.297297  0.002  0.000002    NaN   19.817803             NaN
4        1.396396  0.002  0.000002    NaN   19.808082             NaN
...           ...    ...       ...    ...         ...             ...
```

(continues on next page)

(continued from previous page)

```
4995   99.603604  0.010  0.000001    NaN   11.673323             NaN
4996   99.702704  0.010  0.000001    NaN   11.591120             NaN
4997   99.801803  0.010  0.000001    NaN   11.508924             NaN
4998   99.900902  0.010  0.000001    NaN   11.426733             NaN
4999  100.000001  0.010  0.000001    NaN   11.344545             NaN

         dM_dT  Delta_SM
0    -0.098265 -0.000098
1    -0.098240 -0.000098
2    -0.098202 -0.000098
3    -0.098138 -0.000098
4    -0.098050 -0.000098
...        ...       ...
4995 -0.829550 -0.004619
4996 -0.829466 -0.004620
4997 -0.829407 -0.004620
4998 -0.829371 -0.004620
4999 -0.829347 -0.004620

[5000 rows x 8 columns]
```

The problem of estimating true statistical model parameters using a single data set is commonly approached using bootstrap procedures. Given data of length N, bootstrap resampling involves repeatedly sampling N points from the data *with replacement*, fitting a model to each of the N_B data samples, and computing the parameter of interest from the N_B fitted models.

In our case, we want to estimate the error at each output point of the smoothed magnetic moment. To do this, the standard deviation of each smoothed magnetic moment point is computed from the values of N_B fitted models at each point. Every model is computed using a subset (again, sampled with replacement) of the original data, though the smoothed moment is evaluated at the same linearly-spaced points every time. (The output points are specified in `presets` as part of *data processing*.)

There are a few significant caveats associated with this approach. Each caveat get its own little admonition below. Please read!

Attention: The bootstrap method presented here is purely experimental and is not detailed in either of the sources listed on the *homepage*.

Caution: N_B regularization problems must be solved for every temperature sweep taken at a particular field strength. As such, this method is computationally expensive and can take upwards of ten minutes to run on typical magnetization data, depending on the size of the data and how many models are fitted at each field.

Important: Bootstrap estimates in the context of regularization are dependent on the chosen regularization parameter λ. These error estimates should not be viewed as "true" estimates but rather as the estimates for a *given* λ. This method should only be used once the user is confident their λ's are appropriate.

Caveats aside, the method is simple, if time-consuming. Two arguments are supported: `n_bootstrap` (the number of models to fit at each field) and `random_seed` (for reproducibility).

```
magdata.bootstrap(n_bootstrap=100, random_seed=0)
```

```
Performing bootstrap calculations...
Calculated bootstrap estimates at field: 20.0
Calculated bootstrap estimates at field: 40.0
Calculated bootstrap estimates at field: 60.0
Calculated bootstrap estimates at field: 80.0
Calculated bootstrap estimates at field: 100.0
Finished.
```

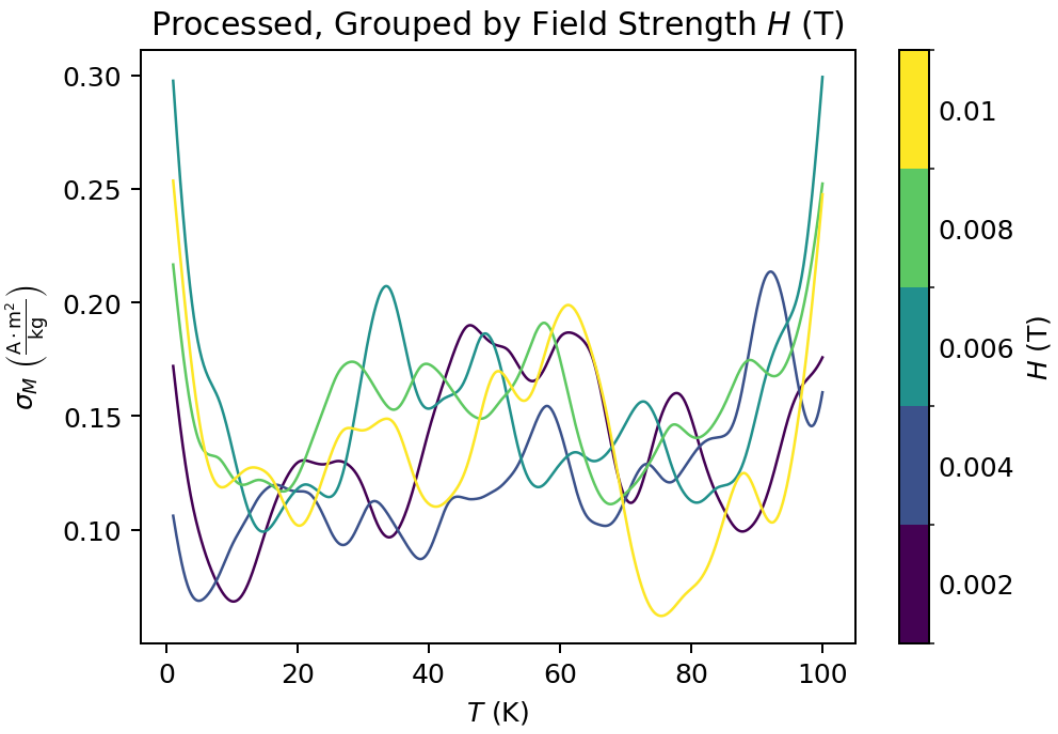
The error columns in *processed_df* are now filled:

```
magdata.processed_df
```

```
              T      H         M         M_err  M_per_mass  M_per_mass_err  \
0      0.999999  0.002  0.000002  1.720299e-08   19.847003        0.172030
1      1.099098  0.002  0.000002  1.694624e-08   19.837267        0.169462
2      1.198198  0.002  0.000002  1.669156e-08   19.827533        0.166916
3      1.297297  0.002  0.000002  1.643916e-08   19.817803        0.164392
4      1.396396  0.002  0.000002  1.618925e-08   19.808082        0.161893
...         ...    ...       ...           ...         ...             ...
4995  99.603604  0.010  0.000001  2.348720e-08   11.673323        0.234872
4996  99.702704  0.010  0.000001  2.380203e-08   11.591120        0.238020
4997  99.801803  0.010  0.000001  2.411919e-08   11.508924        0.241192
4998  99.900902  0.010  0.000001  2.443854e-08   11.426733        0.244385
4999 100.000001  0.010  0.000001  2.475996e-08   11.344545        0.247600

         dM_dT  Delta_SM
0    -0.098265 -0.000098
1    -0.098240 -0.000098
2    -0.098202 -0.000098
3    -0.098138 -0.000098
4    -0.098050 -0.000098
...        ...       ...
4995 -0.829550 -0.004619
4996 -0.829466 -0.004620
4997 -0.829407 -0.004620
4998 -0.829371 -0.004620
4999 -0.829347 -0.004620

[5000 rows x 8 columns]
```

We can easily plot the errors:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(6, 4))

magdata.plot_lines(data_prop='M_per_mass_err', data_version='processed', ax=ax,␣
↪colorbar=True);
```



3.3 API Reference

3.3.1 MagentroData

class magentropy.**MagentroData**

Representation of DC magnetization data.

Magnetization data is collected for a sample by varying the temperature monotonically for each of many magnetic field strengths. This class provides methods for reading, processing, and plotting the data.

Uses the `pint` package internally for unit convertsions.

Notes

All DataFrame attributes (*raw_df*, *converted_df*, etc.) are immutable and return copies of the internal instance attributes. If repeated access is required, for example to a DataFrame's columns, it is best to first save the DataFrame as a local variable to avoid repeatedly copying large amounts of data.

Attributes

converted_df	A copy of the converted data (SI units).
converted_df_with_units	A copy of the converted data (SI units) with a second header level indicating units.
last_presets	The most recently used *process_data()* presets, or None if *process_data()* has not been run.
presets	The current *process_data()* presets.
processed_df	A copy of the processed data.
processed_df_with_units	A copy of the processed data with a second header level indicating units.
raw_df	A copy of the raw data.
raw_df_with_units	A copy of the raw data with a second header level indicating units.
sample_mass	The magnitude of the sample mass.
sample_mass_with_units	The magnitude and units of the sample mass.

Methods

bootstrap([n_bootstrap, random_seed])	Calculate bootstrap estimates of the errors in the smoothed magnetic moment output and fill *processed_df*'s `'M_err'` and `'M_per_mass_err'` columns.
get_map_grid([data_prop, data_version, ...])	Return the temperature, field, and property grids used to construct maps.
get_presets()	Alias for attribute *presets*.
get_raw_data_units()	The raw data units for T, H, M, and sample mass.
plot(plot_type, **kwargs)	Plot property as lines or as a map.
plot_all()	Plot all combinations of *data_prop* and *data_version* for both line plots and maps with default settings.
plot_lines([data_prop, data_version, ax, ...])	Plot the moment per mass, derivative with respect to temperature, or entropy as lines.
plot_map([data_prop, data_version, ax, ...])	Plot the moment per mass, derivative with respect to temperature, or entropy as a map.
plot_processed(plot_type, **kwargs)	Plot processed property as lines or as a map.
plot_processed_lines(processed_df[, ...])	Plot processed data from a `DataFrame` as lines.
plot_processed_map(processed_df[, ...])	Plot processed data from a `DataFrame` as a map.
process_data([npoints, temp_range, fields, ...])	Smooth magnetic moment and calculate raw, converted, and processed derivative and entropy.
set_presets(**kwargs)	Set *presets* for *process_data()*.
set_raw_data_units([T, H, M, sample_mass])	Set the units of the raw data.
sim_data(temps, fields[, sigma_t, sigma_h, ...])	Simulate data for testing and example purposes.
test_grouping([fields, decimals, max_diff])	Test grouping parameters before processing data, if desired.
*test_grouping_*(raw_df[, fields, decimals, ...])	Class method corresponding to *test_grouping()*.
to_html(**kwargs)	Render as an HTML table.
to_string(**kwargs)	Render as console-friendly output.

__init__(*file_or_df*, *qd_dat=True*, *comment_col='Comment'*, *T='Temperature (K)'*, *H='Magnetic Field (Oe)'*, *M='Moment (emu)'*, *M_err='M. Std. Err. (emu)'*, *sample_mass=None*, *units_level=None*, *raw_data_units=None*, *presets=None*, ***read_csv_kwargs*)

Initialize data with a source file or `DataFrame`.

Parameters

file_or_df
[`str`, path `object`, file-like `object`, or `DataFrame`] An input file or `DataFrame`. A `DataFrame` should have the specified columns (parameters *comment_col* through *M_err*). Files will be read by `pandas.read_csv()` with additional arguments given in ***read_csv_kwargs*, and the resultant `DataFrame` should have the specified columns.

qd_dat
[bool, default `True`] If `True` and *file_or_df* is not a `DataFrame`, the input file is assumed to be a Quantum Design `.dat` file with the sample mass given in the header as "`INFO,<sample_mass>,SAMPLE_MASS`" and the delimited data separated from the header by "`\n[Data]\n`". The delimited data will then be read by `pandas.read_csv()` with additional arguments given in ***read_csv_kwargs*.

comment_col
[label, optional, default 'Comment'] The name of the input `DataFrame`'s comment column.

If a row has a non-`NaN` value in the comment column, it will be omitted. Set to `None` to ignore (do not omit any rows based on comments).

T
[label, default 'Temperature (K)'] The name of the input temperature column.

H
[label, default 'Magnetic Field (Oe)'] The name of the input magnetic field strength column.

M
[label, default 'Moment (emu)'] The name of the input magnetic moment column. (Moment only, not per mass unit.)

M_err
[label, optional, default 'M. Std. Err. (emu)'] The name of the input moment standard error column.

sample_mass
[`float`, optional] The mass of the sample that was measured. If supplied, this will override any value determined from an input file when *qd_dat* is `True`. Defaults to 1.0. Keep default of 1.0 if magnetic moment was already measured per mass unit, and set the units of moment and sample mass so that the dimensionality is correct.

units_level
[`int` or `str`, optional] If supplied, data is expected to have units specified in this level of the column index. The column name parameters should still account for this level so they each refer to a single `Series` (i.e., include all levels in the column names).

raw_data_units
[`dict`, optional] Keyword arguments specifying the units of the raw data that will be passed to *set_raw_data_units*. If supplied, this will override any units determined from column levels when *units_level* is supplied.

presets
[`dict`, optional] Keyword arguments to pass to *set_presets()*. See *set_presets()* and *process_data()* for more info.

****read_csv_kwargs**
Passed to `pandas.read_csv()` for reading delimited data.

property raw_df

A copy of the raw data.

property raw_df_with_units

A copy of the raw data with a second header level indicating units.

property converted_df

A copy of the converted data (SI units).

property converted_df_with_units

A copy of the converted data (SI units) with a second header level indicating units.

property processed_df

A copy of the processed data.

property processed_df_with_units

A copy of the processed data with a second header level indicating units.

property sample_mass

The magnitude of the sample mass.

Can be set with a `float`.

property sample_mass_with_units

The magnitude and units of the sample mass.

Can be set with a `tuple` containing a `float` (magnitude) and a `str` (units).

get_raw_data_units()

The raw data units for T, H, M, and sample mass.

Returns

`dict`
Raw data units.

set_raw_data_units(*T=None*, *H=None*, *M=None*, *sample_mass=None*)

Set the units of the raw data.

After the units are set, all other data is converted accordingly, so there is no need to re-process data if units are changed retroactively.

Parameters

T, H, M, sample_mass
[`str`, optional] New units for temperature, magnetic field strength, measured moment, and sample mass, respectively. Moment is not per mass. Parameters left as `None` will not change the corresponding units.

property presets

The current *process_data()* presets.

Can be set with a `dict`.

get_presets()

Alias for attribute *presets*.

set_presets(***kwargs*)

Set *presets* for *process_data()*.

Parameters left as `None` will not change the corresponding preset.

Parameters

****kwargs**
See *process_data()* for valid parameters and parameter info.

property last_presets

The most recently used *process_data()* presets, or `None` if *process_data()* has not been run.

to_string(***kwargs*)

Render as console-friendly output.

Parameters

****kwargs**
[`Any`] Passed to `DataFrame.to_string()` for rendering *raw_df_with_units*, *converted_df_with_units*, and *processed_df_with_units*. Excludes `buf` parameter.

Returns

str
Console-friendly output.

to_html(***kwargs*)

Render as an HTML table.

Parameters

****kwargs**
[Any] Passed to DataFrame.to_html() for rendering *raw_df_with_units*, *converted_df_with_units*, and *processed_df_with_units*. Excludes buf parameter.

Returns

str
HTML table.

classmethod sim_data(*temps*, *fields*, *sigma_t=1e-06*, *sigma_h=1e-06*, *sigma_m=1e-06*, *random_seed=None*, *m_max=0.01*, *slope=1.5*, *bump_height=0.1*)

Simulate data for testing and example purposes.

The simulated data model function is a decreasing logistic function with maximum *m_max* plus a tiny Gaussian bump, the center of which varies linearly with field strength.

The moment error column will the filled with *sigma_m*.

Parameters

temps, fields
[array_like] Temperatures and fields at which to generate data.

sigma_t, sigma_h, sigma_m
[float, default 1e-6] Standard deviation of random normally-distributed errors added to the temperatures, fields, and moments, respectively.

random_seed
[None, int, array_like[ints], SeedSequence, BitGenerator, or Generator] Passed to numpy.random.default_rng().

m_max
[float, default 0.01] The limit as temperature goes to -inf of the moment for the highest field strength.

slope
[float, default 1.5] Controls the steepness of the moment curves. Higher slope results in a faster decrease with temperature.

bump_height
[float, default 0.1] Amplitude of the Gaussian bump as a proportion of *m_max*.

Returns

df
[DataFrame] Simulated data for temperature, field, moment, and moment error.

test_grouping(*fields=None*, *decimals=None*, *max_diff=None*)

Test grouping parameters before processing data, if desired.

See *process_data()* for parameter info. Default parameters are those in *presets*.

Returns

grouping_presets
[dict] The field grouping parameters *fields*, *decimals*, and *max_diff* after being checked and any defaults are used.

grouped_by
[DataFrameGroupBy] Object on which one may test the results of the grouping.

Notes

The pandas.core.groupby.DataFrameGroupBy.count() method is useful for viewing the number of observations in each field group. For example, `test_grouping(...)['T'].count()` returns a DataFrame of the group counts. Groups with less than *min_sweep_len* observations are ignored in *process_data()*.

classmethod test_grouping_(*raw_df*, *fields=None*, *decimals=None*, *max_diff=None*)

Class method corresponding to *test_grouping()*.

See *test_grouping()* and *process_data()* for parameters following *raw_df* and return values. Default parameters are the class defaults.

A copy of *raw_df* is grouped, so the returned DataFrameGroupBy cannot modify *raw_df*.

Parameters

raw_df
[DataFrame] DataFrame on which to test grouping.

process_data(*npoints=None*, *temp_range=None*, *fields=None*, *decimals=None*, *max_diff=None*, *min_sweep_len=None*, *d_order=None*, *lmbds=None*, *lmbd_guess=None*, *weight_err=None*, *match_err=None*, *min_kwargs=None*, *add_zeros=None*)

Smooth magnetic moment and calculate raw, converted, and processed derivative and entropy.

Groups raw data, smooths magnetic moment using Tikhonov regularization, and fills *processed_df*. Calculates derivative `'dM_dT'` and entropy `'Delta_SM'` for raw and converted data without smoothing, and for processed data using smoothed moment.

Requires that all sweeps are taken on cooling, or all sweeps are taken on warming (monotonic). Warming and cooling sweeps should not both be included in the data.

Note: Rows of zero field and zero moment are prepended to the data before integration, so it is not necessary to include measurements at zero field in the input data. Whether or not the zeros are added to *processed_df* after processing can be controlled with *add_zeros*.

Parameters left as the default None will use the corresponding values in *presets*. All parameters should be given in raw data units if applicable (*temp_range*, *max_diff*, etc.).

Parameters

npoints
[int, optional] Number of temperature points in *temp_range* to use to output smoothed `'M_per_mass'`, `'dM_dT'`, and `'Delta_SM'` for each field strength.

temp_range
[(2,) array_like, optional] Temperature range (inclusive) in raw data units over which to analyze the data. Bounds less than or greater than the lowest or highest given temperatures, respectively, will be adjusted to the data range when creating output temperatures.

fields
[array_like, optional] Expected field strengths for grouping data. If *fields* has length zero, the groups are determined automatically based on *decimals* and/or *max_diff*.

decimals
[int, optional] The decimal place to which to round the automatically determined field groups. (A negative integer specifies the number of positions to the left of the decimal point. See numpy.around().) Ignored if *fields* has length greater than zero. If *fields* has length zero and *max_diff* is numpy.inf, groups will be determined solely by rounding to this decimal place.

max_diff
[float, optional] If *fields* has length greater than zero, *max_diff* is the maximum difference allowed between each raw field strength and the closest field in *fields*. Raw fields too far away from any field group will be omitted, unless *max_diff* is numpy.inf. If *fields* has length zero, *max_diff* is the maximum difference allowed between any two items in each field group, which is used to determine groups automatically. Generally, *decimals* is enough to determine groups, but *max_diff* can be used for finer control, e.g. to get exact means.

min_sweep_len
[int, optional] Minimum number of observations required for a field to be included in the smoothed output. Field groups with less than this number will be skipped.

d_order
[int, optional] Order of derivative used to calculate roughness during regularization. For example, if *d_order* is 2, the second temperature derivative of magnetic moment is used to calculate the roughness. Generally 2 or 3 work well. Choice of *d_order* will change optimal regularization parameter λ.

lmbds
[array_like, optional] Specifies regularization parameter λ. If *lmbds* is a single number (or array_like of length 1), it will be applied to all magnetic field strengths. If *lmbds* is numpy.nan or length 0, each λ will be determined automatically. If *lmbds* is the same length as the number of field strengths, each element (numerical or numpy.nan) will be applied to the corresponding field, in order of increasing field strength.

lmbd_guess
[float, optional] Initial guess for regularization parameter λ when determining automatically.

weight_err
[bool, optional] If True, weight measurements by the normalized inverse squares of the errors.

match_err
[bool, array_like, or one of {'min', 'mean', 'max'}, optional] Ignored if λ is given in *lmbds*. If *match_err* is False, use generalized cross validation (GCV) to find optimal λ. If *match_err* is True, find optimal λ by matching absolute differences between the measured and smoothed values with the errors. If *match_err* is a single number (or array_like of length 1), match the standard deviation of the absolute differences with this number. If *match_err* is the same length as the number of field strengths, each element (numeric) will be applied to the corresponding field, in order of increasing field strength. If *match_err* is one of 'min', 'mean', or 'max', match the standard deviation of the absolute differences with the minimum, mean, or maximum error for each field.

min_kwargs
[dict, optional] Keyword arguments to pass to scipy.optimize.minimize() when de-

termining optimal λ. The parameters `fun`, `x0`, and `args` will be ignored if included. Note that $\log_{10} \lambda$ is passed to `scipy.optimize.minimize()`, so arguments such as `bounds` should be adjusted accordingly. (The same is not true, however, for *lmbd_guess*.)

add_zeros
[bool, optional] If `True`, rows of zeros corresponding to zero field and zero moment will be prepended to *processed_df* after processing.

bootstrap(*n_bootstrap=100*, *random_seed=None*)

Calculate bootstrap estimates of the errors in the smoothed magnetic moment output and fill *processed_df*'s `'M_err'` and `'M_per_mass_err'` columns.

Parameters

n_bootstrap
[`int`, default 100] The number of times to sample from the data and fit a model.

random_seed
[`None`, `int`, array_like[`ints`], `SeedSequence`, `BitGenerator`, or `Generator`] Passed to `numpy.random.default_rng()`.

Notes

Bootstrap procedures involve repeatedly sampling N points from data of length N *with replacement*, fitting a model to each data sample, and computing the parameter of interest from the *n_bootstrap* fitted models. In this case, the standard deviation of each smoothed magnetic moment point is computed from the values of the *n_bootstrap* models at each point.

Attention: The bootstrap method presented here is purely experimental and is not detailed in either of the sources listed on the *homepage*.

Caution: This method is computationally expensive and can take upwards of ten minutes to run on typical magnetization data.

Important: Bootstrap estimates in the context of regularization are dependent on the chosen regularization parameter λ. These error estimates should not be viewed as "true" estimates but rather as the estimates for a given λ. This should only be used once the user is confident their λ's are appropriate.

classmethod plot_processed_lines(*processed_df*, *compare_df=None*, *data_prop='M_per_mass'*, *ax=None*, *T_range=array([-inf, inf])*, *H_range=array([-inf, inf])*, *offset=0*, *at_temps=None*, *fields=None*, *decimals=None*, *max_diff=None*, *colormap=None*, *legend=False*, *colorbar=False*, *plot_kwargs=None*, *compare_kwargs=None*, *colorbar_kwargs=None*)

Plot processed data from a `DataFrame` as lines.

This class method allows already-processed data to be easily plotted so that raw data needn't be re-processed.

See *plot_lines()* for parameters following *compare_df* and return values.

Parameters

processed_df
[DataFrame] Processed data. Expected to have column names matching those of the *processed_df* attribute of an instance of *MagentroData*.

compare_df
[DataFrame, optional] Converted data. Expected to have column names matching those of the *converted_df* attrubute of an instance of *MagentroData*. Expected to use the same units as *processed_df*.

`plot_lines`(*data_prop='M_per_mass'*, *data_version='raw'*, *ax=None*, *T_range=array([-inf, inf])*, *H_range=array([-inf, inf])*, *offset=0*, *at_temps=None*, *colormap=None*, *legend=False*, *colorbar=False*, *plot_kwargs=None*, *compare_kwargs=None*, *colorbar_kwargs=None*)

Plot the moment per mass, derivative with respect to temperature, or entropy as lines.

All parameter units should correspond to those of the data specified by *data_version*.

Parameters

data_prop
[{'M_per_mass', 'M_per_mass_err', 'dM_dT', 'Delta_SM'}, default 'M_per_mass'] The property to plot.

data_version
[{'raw', 'converted', 'processed', 'compare'}, default 'raw'] The version of the data to plot. If `'compare'`, converted and processed data will be plotted together.

ax
[Axes, optional] Axes on which to plot. If None, new Axes will be created from the current Figure.

T_range, H_range
[(2,) array_like] Temperature and magnetic field strength ranges to display.

offset
[float, default 0] If a nonzero *offset* is supplied, successive lines (fields or temperatures) will have an offset added to them. Good for seeing curve shapes at different fields or temperatures.

at_temps
[array_like, optional] Temperatures to group data. If supplied, data will be plotted versus magnetic field strength instead of temperature, at the temperatures in the data that are closest to the supplied *at_temps*.

colormap
[Colormap or str, optional] Color map to cycle through when plotting lines.

legend
[bool, default False] If True, add a legend to the Axes with `Axes.legend()`.

colorbar
[bool, default False] If True, add a discrete color bar to the Figure containing *ax* with `Figure.colorbar()`.

plot_kwargs
[dict or array_like of dicts, optional] Keyword arguments for `Axes.plot()`. A single dict will be applied to each line. Multiple dicts will be applied to successive lines. Not checked for length; a dict is applied to each line until either the end is reached or there are no more lines to plot.

compare_kwargs
[dict or array_like of dicts, optional] *plot_kwargs* for converted data, used if *data_version* is `'compare'`.

colorbar_kwargs
[dict, optional] Keyword arguments for `Figure.colorbar()`, excluding `mappable`.

Returns

ax
[Axes] If *colorbar* is `False`, the Axes on which the data is plotted.

ax, cbar
[Axes, Colorbar] If *colorbar* is `True`, the Axes on which the data is plotted and the Colorbar.

classmethod plot_processed_map(*processed_df*, *data_prop='M_per_mass'*, *ax=None*, *T_range=array([-inf, inf])*, *H_range=array([-inf, inf])*, *T_npoints=1000*, *H_npoints=1000*, *interp_method='linear'*, *center=None*, *contour=False*, *colorbar=True*, *imshow_kwargs=None*, *contour_kwargs=None*, *colorbar_kwargs=None*)

Plot processed data from a DataFrame as a map.

This class method allows already-processed data to be easily plotted so that raw data needn't be re-processed.

See *plot_map()* for parameters following *processed_df* and return values.

Parameters

processed_df
[DataFrame] Processed data. Expected to have column names matching those of the *processed_df* attribute of an instance of *MagentroData*.

get_map_grid(*data_prop='M_per_mass'*, *data_version='raw'*, *T_range=array([-inf, inf])*, *H_range=array([-inf, inf])*, *T_npoints=1000*, *H_npoints=1000*, *interp_method='linear'*)

Return the temperature, field, and property grids used to construct maps.

See *plot_map()* for parameters.

Returns

T_grid, H_grid, grid
[ndarray] Grids corresponding to temperature, field, and property, respectively.

plot_map(*data_prop='M_per_mass'*, *data_version='raw'*, *ax=None*, *T_range=array([-inf, inf])*, *H_range=array([-inf, inf])*, *T_npoints=1000*, *H_npoints=1000*, *interp_method='linear'*, *center=None*, *contour=False*, *colorbar=True*, *imshow_kwargs=None*, *contour_kwargs=None*, *colorbar_kwargs=None*)

Plot the moment per mass, derivative with respect to temperature, or entropy as a map.

All parameter units should correspond to those of the data specified by *data_version*.

Note: Different default colormaps are used depending on *center*. The colormap can be specified manually in *imshow_kwargs*. For example, `imshow_kwargs = {'cmap': 'RdBu_r'}`.

Parameters

data_prop
[{'M_per_mass', 'M_per_mass_err', 'dM_dT', 'Delta_SM'}, default 'M_per_mass'] The property to plot.

data_version
[{'raw', 'converted', 'processed'}, default 'raw'] The version of the data to plot. (`'compare'` is not available for maps.)

ax
[`Axes`, optional] `Axes` on which to plot. If `None`, new `Axes` will be created from the current `Figure`.

T_range, H_range
[(2,) array_like] Temperature and magnetic field strength ranges to display.

T_npoints, H_npoints
[`int`, default 1000] Number of points to use for grid interpolation in the horizontal (T) and vertical (H) directions.

interp_method
[{'linear', 'nearest', 'cubic'}, default 'linear'] Map grid interpolation method. See `scipy.interpolate.griddata()`'s `method` parameter. The `'cubic'` method may give a smoother result, but it is recommended to start with `'linear'` interpolation, as artifacts can occasionally occur in the output when using higher-order interpolation.

center
[bool, optional] If `True`, center the pixel values around zero, setting values beyond the central range to the values at the boundaries of the range. This is helpful for ignoring extreme values. `None` defaults to `False` when *data_prop* is `'M_per_mass'` or `'M_per_mass_err'` and `True` otherwise.

contour
[bool, default `False`] If `True`, add contours to the plot with `Axes.contour()`.

colorbar
[bool, default `True`] If `True`, add a continuous color bar to the `Figure` containing *ax* with `Figure.colorbar()`.

imshow_kwargs
[`dict`, optional] Keyword arguments for `Axes.imshow()`.

contour_kwargs
[`dict`, optional] Keyword arguments for `Axes.contour()`.

colorbar_kwargs
[`dict`, optional] Keyword arguments for `Figure.colorbar()`, excluding `mappable`.

Returns

ax
[`Axes`] If *colorbar* is `False`, the `Axes` on which the data is plotted.

ax, cbar
[`Axes`, `Colorbar`] If *colorbar* is `True`, the `Axes` on which the data is plotted and the `Colorbar`.

`classmethod plot_processed`(*plot_type*, ***kwargs*)

Plot processed property as lines or as a map.

See *plot_processed_lines()* or *plot_processed_map()* for parameters and return values.

Parameters

plot_type
[{'lines', 'map'}] Plot lines or map.

****kwargs**
[dict, optional] Passed to *plot_processed_lines()* or *plot_processed_map()*, depending on *plot_type*.

plot(*plot_type*, ***kwargs*)

Plot property as lines or as a map.

See *plot_lines()* or *plot_map()* for parameters and return values.

Parameters

plot_type
[{'lines', 'map'}] Plot lines or map.

****kwargs**
[dict, optional] Passed to *plot_lines()* or *plot_map()*, depending on *plot_type*.

plot_all()

Plot all combinations of *data_prop* and *data_version* for both line plots and maps with default settings.

Line plots grouped by temperature are also plotted, with five evenly-spaced temperature groups.

Each returned Axes gets its own Figure. All Figures are plotted immediately if run in a notebook, so this is a "quick-and-dirty" way to view every plot after initial processing.

Tip: If using a notebook, be sure to put a semicolon (;) after this method to suppress nasty-looking text output!

Returns

tuple[Axes, ...]
The Axes, one for each plot.

3.3.2 Errors

Exception classes.

exception magentropy.errors.**MagentroError**

Base exception class for *MagentroData*.

exception magentropy.errors.**UnitError**

Exception class for invalid units or conversions.

exception magentropy.errors.**MissingDataError**

Exception class for attempting to plot or operate on empty data.

3.3.3 Typing

Type definitions.

class magentropy.typedefs.**ColumnDataDict**

Type for `dict` describing column data, such as temperature and magnetic moment.

Methods

clear()	
copy()	
fromkeys(iterable[, value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(key[, default])	If the key is not found, return the default if given; otherwise, raise a KeyError.
popitem(/)	Remove and return a (key, value) pair as a 2-tuple.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

T

H

M

M_err

M_per_mass

M_per_mass_err

dM_dT

Delta_SM

class magentropy.typedefs.**Presets**

Type for *process_data()* presets.

Implementation note

These should all have defaults in _DEFAULT_PRESETS, be parameters in *process_data()*, and be verified and returned in _validation.check_presets().

Methods

clear()	
copy()	
fromkeys(iterable[, value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(key[, default])	If the key is not found, return the default if given; otherwise, raise a KeyError.
popitem(/)	Remove and return a (key, value) pair as a 2-tuple.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

npoints

temp_range

fields

decimals

max_diff

min_sweep_len

d_order

lmbds

lmbd_guess

weight_err

match_err

min_kwargs

add_zeros

class magentropy.typedefs.**SetterPresets**

Same as *Presets*, except all are optional.

For *presets* setter typing.

Methods

clear()	
copy()	
fromkeys(iterable[, value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(key[, default])	If the key is not found, return the default if given; otherwise, raise a KeyError.
popitem(/)	Remove and return a (key, value) pair as a 2-tuple.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

npoints

temp_range

fields

decimals

max_diff

min_sweep_len

d_order

lmbds

lmbd_guess

weight_err

match_err

min_kwargs

add_zeros

3.4 License

3.5 Index

CHAPTER

FOUR

LICENSE

This project is *licensed* under the MIT License.

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

D

F

G

H

L

M

N

P

R

S

T

U

W