Preprocessor Manual

Tyler C. Sterling[1] *and Dmitry Reznik*[1,2]

[1] *Department of Physics, University of Colorado Boulder*
[2] *Center for Experiments on Quantum Materials, University of Colorado Boulder*

May 16, 2024

1 Introduction

We use PHONON-EXPLORER to look for phonon peaks in constant-$\boldsymbol{Q}$ cuts in large datasets. We typically need to look at many $\boldsymbol{Q}$-points for a given project. The old method we used was to loop over the list of user-requested $\boldsymbol{Q}$ and call MANTID once per $\boldsymbol{Q}$-point. MANTID integrates constant-$\boldsymbol{Q}$ cuts from raw event-based data stored in objects called `MDevent` workspaces. The workspaces are stored in `.mde` files.

Calling MANTID once for every $\boldsymbol{Q}$-point is time consuming and keeping all events (there is one per neutron!) and meta-data in the `.mde` files is disk-consuming ($\sim$100 GB). The time-consuming problem is particularly bad when we are looking at *many* $\boldsymbol{Q}$-points. We could, in principle, preintegrate data onto a fine grid and then manually reintegrate later: however, there are non-trivial issues with this that we want to avoid so we dismiss this approach.

As a work around, we created a PREPROCESSOR that preintegrates the entire dataset with user-specified binning: when we want to cut data at a particular $\boldsymbol{Q}$-point later, we can simply look it up and read it from the preintegrated file. The preintegrated files are custom `.hdf5` files. Reading data from the `.hdf5` files doesn't require specialized software and reading thousands of $\boldsymbol{Q}$-points takes less than a second. The preprocessing can be done on e.g. the SNS server rather quickly and the resulting files `.hdf5` are small ($\sim$1 GB) and can be readily copied to and used on local machines. Moreover, this scheme avoids needing to install MANTID on a local machine (which can be pretty annoying).

The reason the PREPROCESSOR method is faster is as follows. MANTID has to read the whole `.mde` file to integrate even a single $\boldsymbol{Q}$-point. This is slow and takes a huge amount of memory, so doing it once per $\boldsymbol{Q}$-point is a drain on performance. Moreover, the MANTID algorithms are parallelized such that integrating many $\boldsymbol{Q}$-points on a uniform grid isn't much slower than integrating a single $\boldsymbol{Q}$-point. This means preintegrating an entire (or large part of) `.mde` file isn't much more costly than integrating one or a few $\boldsymbol{Q}$-pobts. I.e preintegrating means the `.mde` file only has to be loaded once and integrating the whole dataset is relatively fast.

We suggest to create several prehistogrammed files with different binnings and then explore these files to understand what is going on in the data. You can even use different `u`,

v, and w vectors in different files to make cuts through reciprocal space in arbitrary directions. Once you have a clear understanding of the physics, you can settle on a good binning scheme, create a final file, and cut and fit all the interesting $\boldsymbol{Q}$-points *very* quickly.

2 Instructions

Here, we explain how to use the PREPROCESSOR. There are two different types of PREPROCESSORS: EVENTS (`PreprocessEvents.py`) and NXSPE (`PreprocessNXSPE.py`). The NXSPE type is designed to preprocess a group of `.nxspe` files that are preintegrated in energy only. This is for a custom data reduction scheme for e.g. data from JPARC. There are several instrument specific options that must be set in `PreprocessNXSPE.py`. We won't cover these here. Otherwise, the only difference in the PREPROCESSORS is that `.nxspe` files are prehistogrammed in energy, so we don't need to specify energy binning. For the EVENTS version, we have to specify energy integration too.

2.1 Example file

Here, we give an example of a PREPROCESSOR file followed by a description of the variables. This version is specific to the EVENTS type, but the NXSPE version is very similar.

```
from file_tools.m_MDE_tools import bin_MDE

# raw MDE file from SNS experiment
MDE_file_name = f'../../merged_mde/LSNO25_Ei_120meV_300K.nxs'

# binned sparse hdf5 output file
merged_file_name = f'LSNO25_300K_test.hdf5'

# binning projections. same meaning as in NormMD
u = [ 1, 0, 0]
v = [ 0, 1, 0]
w = [ 0, 0, 1]

# Q binning args
H_lo = 5
H_hi = 7
H_step = 0.10
H_bin = 2

K_lo = -2
K_hi = 2
K_step = 0.10
K_bin = 2

L_lo = -4
L_hi = 4
L_step = 2
L_bin = 4

# E binning args
```

```
E_lo = -50
E_hi = 100
E_step = 1

# split binning over these chunks
num_chunks = [1,1,1]

# load the raw event dataset. dont change this.
bin_MDE(MDE_file_name,H_lo,H_hi,H_bin,K_lo,K_hi,K_bin,L_lo,L_hi,L_bin,
        E_lo,E_hi,E_bin,H_shifts,K_shifts,L_shifts,
        merged_file_name,u,v,w,num_chunks)
```

`MDE_file_name` : the path to the `.mde` file containing the raw event data.

`merged_file_name` : the name for the preintegrated `.hdf5` file that will be written out.

`u`, `v`, `w` : projections to integrate along (in units of reciprocal lattice vectors).

E.g. `u = [1, 0, 0]`, `v = [0, 1, 0]`, and `w = [0, 0, 1]` has $\boldsymbol{H} = H\boldsymbol{a}^*$, $\boldsymbol{K} = K\boldsymbol{b}^*$, and $\boldsymbol{L} = L\boldsymbol{c}^*$ with $\boldsymbol{a}^*$ etc. the reciprocal lattice vectors. As another example, `u = [1, 1, 0]` and `v = [-1, 1, 0]` means $\boldsymbol{H} = H\boldsymbol{a}^* + H\boldsymbol{b}^*$ and $\boldsymbol{K} = -K\boldsymbol{a}^* + K\boldsymbol{b}^*$, i.e. a 45° rotation around the $\boldsymbol{c}^*$ axis. Note, if `w` isn't given, it is `w`=`u`×`v`.

`H_lo`, `H_hi`, `H_step` : determines the bin centers and widths for the binning along H that you want in the file.

E.g. suppose you want to integrate from $H = 0$ to $H = 1$ with bin sizes 0.1 rlu. Then `H_lo=0.0`, `H_hi=1.0`, and `H_step=0.1` will produce the following bin centers: `H_bin_centers = [0.00, 0.10, 0.20, ···, 0.90, 1.00]`.

Similarly for K, L, and E. The bins along H, K, and L are meshed so that if e.g. `H_bin_centers = [0, 1]`, `K_bin_centers = [0, 1]`, `L_bin_centers = [0, 1]`, the file will contain bin centers $\boldsymbol{Q}$ =`([0, 0, 0], [0, 0, 1], [0, 1, 0], ···, [1, 1, 0], [1, 1, 1])`.

`K_lo`, `K_hi`, `K_step` : see explanation for `H_lo`, `H_hi`, `H_step`.

`L_lo`, `L_hi`, `L_step` : see explanation for `H_lo`, `H_hi`, `H_step`.

`E_lo`, `E_hi`, `E_step` : determines the binning along E for each $\boldsymbol{Q}$-point in the `.hdf5` file. Also see explanation for `H_lo`, `H_hi`, `H_step`.

`H_bin`, `K_bin`, `L_bin` : after integrating data on the bin centers specified by `H_lo`, `H_hi`, `H_step` etc., shift bin centers and integrate again to allow overlapping voxels. How much to shift is specified by `H_bin` etc. These give the *number* of sub-divisions in single bin to shift bin centers by. Shifts are calculated as `H_shift[n] = n*H_step/H_bin`.

Example: If `H_bin=1` (default) and `H_lo=0.0`, `H_hi=1.0`, and `H_step=0.1`, only unshifted bin centers are used and `H_bin_centers = [0.00, 0.10, 0.20, ···, 0.90, 1.00]`.

Example: If `H_bin=2` and `H_lo=0.0`, `H_hi=1.0`, and `H_step=0.1`, first unshifted bin centers (`H_shift[0] = 0*H_step/H_bin = 0.0`) are integrated with `H_bin_centers[0] = [0.00, 0.10, 0.20, ···, 0.90, 1.00]` and then a shift (`H_shift[1] = 1*H_step/H_bin = 0.05`) is used with `H_bin_centers[0] = [0.05, 0.15, 0.25, ···, 0.95, 1.05]`. The results from all shifts are appended to the same file so that all bin centers `H_bin_centers = [0.00, 0.05, 0.10, 0.15, ···, 0.95, 0.90, 1.00, 1.05]` are included. Bin centers are integrated from -0.05 to 0.05, 0.00 to 0.10, ..., -0.95 to 1.05, and 1.00 to 1.10.

`num_chunks` : because of the way MANTID works, it loads the whole dataset to integrate even a single bin; integrating multiple bins is done in a single `MDNorm` call and is split-up and parallelized on the backend. If we have want very many bins, we are likely to run out memory calling `MDNorm` since a lot of data needs to be copied bin.

To use less memory, we can automatically divide the integration requested region into smaller chunks along each H, K, and L direction. `num_chunks` specifies the number of chunks to split the integration region into. The histogrammed data from each chunk are appended to the same file.

E.g. if `H_bin_centers = [0.0, 1.0, 2.0, 3.0]` and `num_chunks[0] = 2`, bin centers along H will be subdivided into `H_bin_centers[0] = [0.0, 1.0]` and then `H_bin_centers[1] = [2.0, 3.0]` with each chunk being integrated one after the other.