Python Code Package for 3D VMI Data Acquisition and Analysis —2.0—

Scott Goudreau

November 3, 2025

1 Introduction

In this document, I will outline the main functions of my python 3D VMI code, describing how one might use it to fully collect, process, and analyze data obtained from the NRC/uOttawa 3D VMI spectrometer. Additional description of the techniques used here are contained in my PhD thesis (`http://hdl.handle.net/10393/50915`). I will address each part of the code in the order that it would be typically used when acquiring and processing a new data set. The code discussed here is hosted on the Stolow Group GitHub repository (`https://github.com/scottesg/StolowGroupVMI`).

2 Acquiring Data

The first step is to acquire 3D VMI data, which consists of camera images and digitizer waveform data. While setting up the experiment, one might want to adjust experimental parameters to optimize things like hit count per laser shot, signal-to-noise ratio, etc. During this process, it is helpful to have a preview of the incoming data.

2.1 `VMI3D_LivePreview.py`

This is a script which shows a constant stream of live frames from the camera, as well as some helpful diagnostic information to help with optimizing the experiment (shown in Fig. 1). It was based on a similar script that was part of the Matlab 3D VMI code package.

The preview is not able to display data at the experimental frame rate of 1 kHz. Instead, it pulls frames from the camera at a rate (inversely) specified by the parameter **PAUSETIME**. The maximum framerate of the preview is about 3 fps, limited by the time required to plot the data after each frame. To run the preview at maximum framerate, set **PAUSETIME** to a small value such as 0.01 (seconds). Note that **PAUSETIME** must be non-zero to allow the plots to update properly. If a faster framerate is needed, the variable **NUM_ALLOCATED_FRAMES** can be set to a value greater than 1. This will cause the preview to grab that many frames from the camera between updates (instead of just 1). Each of these frames will be analyzed and their centroids will be added to the total (as described below), but only the last frame in the set will be displayed in the preview.

The preview shows the raw (**DISPLAY_FILTERED_IMAGE** = False) or smoothed and projection-subtracted (**DISPLAY_FILTERED_IMAGE** = True) image in the top left panel. It then finds and extracts regions of interest (ROIs) and shows (top middle panel) the detected ROIs for the frame (the ROI threshold is set by **EXTROI_THRESH**). The coordinates of the ROIs are stored and a cumulative image of all detected ROIs is shown in the top right panel. This helps visualize the full VMI image being detected. The bottom left panel shows a cumulative histogram of hit counts from all prior frames. Finally, the bottom right panel shows the hit count over time for the last 100 frames (the number of frames on screen is set by **HITCOUNTWINDOW**).

The Live Preview has a few keyboard controls, which are printed out when the script starts. It can be (s)topped, (p)aused, (u)npaused, or (r)eset. Most of these are easily understood. The reset command clears the memory of all previous frames and starts accumulating new data. This is useful if one makes a change to the experimental conditions and wants to see a fresh cumulative image or hit count histogram.

For testing purposes, it is possible to run the preview on previously-saved image or ROI data. This is done by setting the **TESTSOURCE_IMG** or **TESTSOURCE_ROI** path and uncommenting the appropriate `pv.set_source` line in the main function.

Camera streaming is enabled by default and can be run either in free-run mode (with a set framerate of **CAM_FRAMERATE**) or by using an external trigger. Typically, we would use a longer exposure for free-run mode and a shorter exposure for triggered mode. The exposure values are set in two variables depending on mode: **CAM_EXPOSURE_FREE** and **CAM_EXPOSURE_TRIG**. The region of the camera being imaged is set by the parameters **CAM_OFFSET_X** and **CAM_OFFSET_Y**. Note that only certain values are valid for the offset parameters; some values will cause the script to fail, while other values may cause bugs in the frame data.

2.2 `VMI3D_DAS2C.py`

Acquiring 3D VMI data is done through this script, based on the Matlab version written by Doug Moffatt. Running this script will create a folder (named after the current date and time) in the specified data directory. It then collects and saves image and waveform data based on a set of user-supplied parameters near the top of the script. The camera acquisition is handled by Doug's C++ script which is called from within a function edited from sample AlazarTech python code. By default, the script collects two-channel data from the AlazarTech digitizer card, with the pickoff on one channel and the photodiode pickoff on the other channel.

Some important parameters are the following: **npts** sets the digitizer time window. Half of the specified number of data points will be pre-trigger and the other half will be post-trigger, with the limitation that the pre-trigger can't be greater than 4088. Once **npts** is greater than 2×4088, the remaining points will be allocated entirely to post-trigger. **nframes** sets the number of laser shots per run and **nruns** sets the number of runs. **nstacks** is mostly a legacy parameter can be left at 1. **iexposure** is the camera exposure and the camera window is defined by **roix0** and **roiy0** which are equivalent to the offset parameters above. Many of the other parameters of the camera are set as well (defined in **params**) to ensure that everything is consistent for each acquisition.

While running, the script displays feedback from each run in the form of a series of plots. These include an integrated 2D VMI image, a cross-correlation between the camera and digitizer data, a 2D image of stacked 1D waveform data, and an average of all of the waveforms. These are updated after each run is collected so that the quality of the data can be monitored during an experiment.

TODO: Currently, the real-time plotting does not currently update in between runs, only displaying once the full acquisition has finished. Also, the plotting is not coded efficiently and the data volume causes long delays for large values of **nframes**.

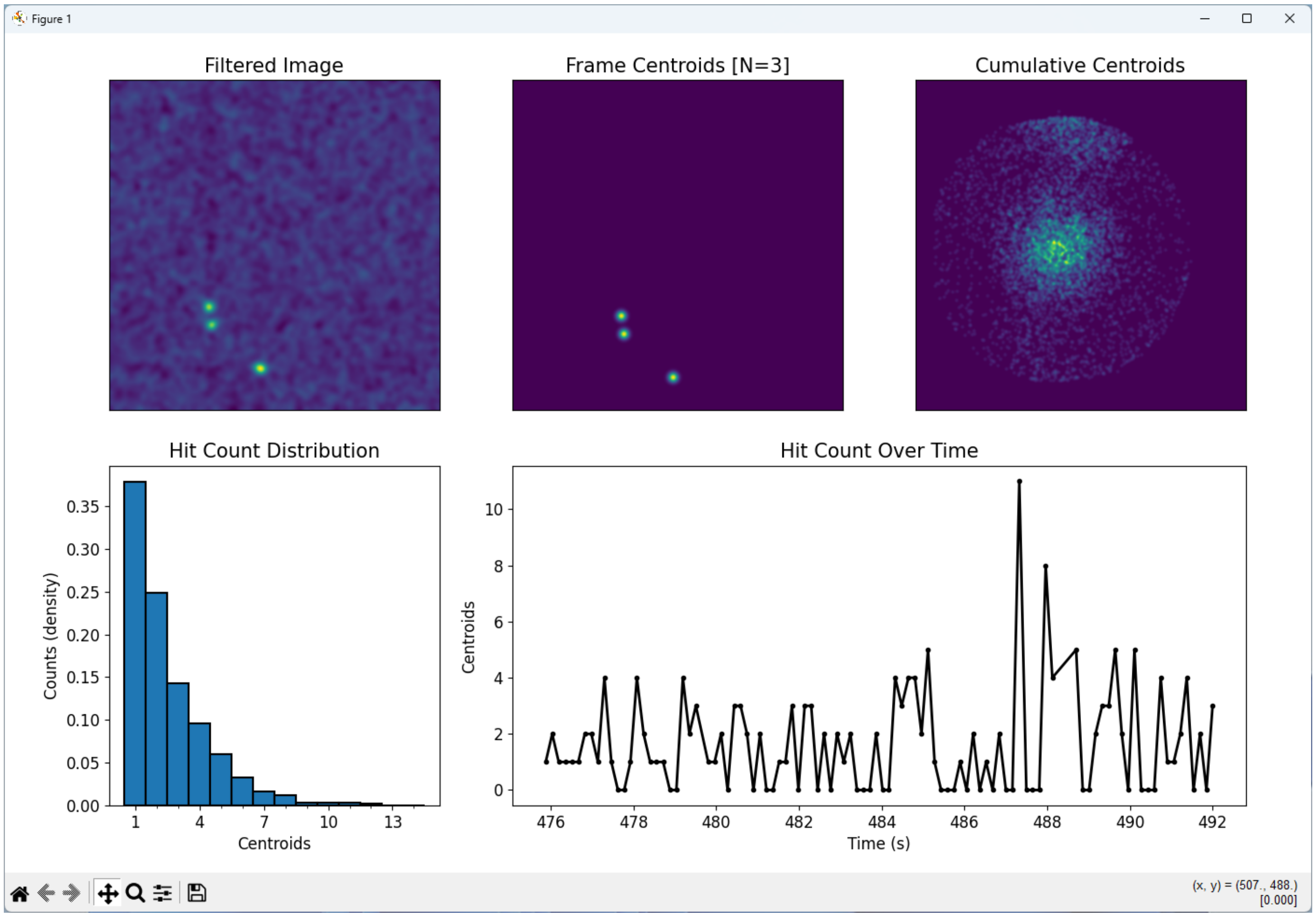

Figure 1: Appearance of the Live Preview script while running.

3 Processing the Data

The main steps of data processing include: centroiding the raw ROIs to obtain coordinates and amplitude for each event, peak-finding the waveform data to obtain time stamps and amplitude for each event, and performing assignment of the spatial and temporal coordinates to obtain 3D VMI data in the form of (x, y, t) coordinates for each event. These steps should be performed using the following scripts.

3.1 Step 1: Centroiding the Image Data

The script `VMI3D_Centroiding.py` will centroid all of the data contained in the current folder. The folder should contain one or more sub-folders named "run1", "run2", etc., as is created by the acquisition script `VMI3D_DAS2C.py`. Each of these run folders should contain a file (roisposb1.single) containing the real-time ROI data, and the average projection of the images along the x-axis (projsb1.int16). Both of these files should be created when running the acquisition script.

The centroids will be saved in each run sub-folder as "ctrs.npy". One must specify the number of frames per run (**nframes**) which may change between experiments. Other parameters will likely remain unchanged. Optionally, **fastctrs** can be set to True to get faster (but far less accurate) centroids.

3.2 Step 2: Peak-Finding the Waveform Data

There are two scripts for extracting time-stamps from the waveform data, depending on whether the data uses two channels (separate data and reference pulse) or a single channel (combined data and reference pulse). These scripts are `VMI3D_PeakFind_2Ch.py` and `VMI3D_PeakFind_1Ch.py`. The use of both scripts are similar, so I will only discuss the two-channel version in detail.

At this point, the data directory should be restructured. All of the centroids files for each run should be moved into a single directly "ctr/" and numbered as "ctrs1.npy", "ctrs2.npy", etc. The same should be done with the raw waveform data, which should be moved to "wfm/" and numbered as "daqdata1.uint16" "daqdata2.uint16", etc. There are functions located in `VMI3D_IO.py` called *copywfms* and *copyctrs* which can help with this.

One might need to change some of the parameters at the top of the peak-finding script, such as the number of shots per run (**stksize**), the length of each trace (**tracelen**), and the approximate position (**t00**) or width (**refwidth**) of the reference peak. The other parameters will probably not change.

The scripts are divided into blocks which should be run one-at-a-time to evaluate the results before proceeding to the next. At the top of the first block (step 0), one must specify the data directory to be processed. This step extracts an average zero-hit background and an average reference peak, then creates a filter function incorporating both of these as well as Gaussian smoothing. The average reference peak and background are displayed after running, so ensure these are acceptable before proceeding.

The second block (step 1) extracts and displays an average single hit function. As before, ensure this is acceptable before proceeding.

The final block (step 2) performs the full peak-finding process on all of the data in the specified folder. This is a time-consuming task, so first set the **test** variable to True and run the block. This will show the peak-finding results for the first 20 traces. Examine these to make sure everything is working correctly. In particular, one may want to change the **SNR** variable to change the peak detection threshold. If you wish to look at traces other than the first 20, you can change the starting run (**n0**) or trace number (**idx0**). Once the performance is as desired, change **test** to False and run the block. The peaks for each run are saved in text form in a folder called "hits/" in the data directory and are named "THits_1.txt", "THits_2.txt", etc.

3.3 Step 3: Assigning Time-Stamps

The assignment step is performed by running the script `VMI3D_AssignHits.py`. This script is also divided into blocks. In general, one should proceed by running each block in order. Under the "Paths and Parameters" block, the data directory should be specified. If one wishes to attempt the assignment multiple times with different parameters, the **name** variable can be changed which will save the results in a different sub-directory. As usual, **stksize** should be set to the number of shots per run. It is also important to set the **dt** variable to the correct time step (0.5 for two-channel data, 0.25 for one-channel data).

The next block will combine the text files created by the time-stamping script into a single numpy file containing the hits from all runs. Note that the number of runs must be set using the **n** variable.

In some runs, the shot numbering of the camera and digitizer data is offset. This offset must be detected for each run so that the time-stamps can be assigned to the correct centroids. The next block checks for these offsets and saves them for use in later steps. It shows a plot with the detected offsets and the associated correlation amplitudes. The offsets should either be 1 or 0, and the amplitudes should be relatively constant.

The next block performs the initial time-stamping of the single-hit data. While running, it prints out the number of detected hits for each run. After completion, some diagnostic images are produced, including a VMI image from all of the single-hit centroids, and pulse-height distributions for the camera and digitizer data. It also produces a histogram of event arrival times. Importantly, it shows the correlation plot of camera and digitizer amplitudes for the single-hit data (the SHAD). The SHAD will not be particularly narrow at this stage, but the correlation should be clear. The single-hit assigned data is saved as a ".pt.npy" file along with a text log file ".params.txt" containing the parameters used in the analysis.

The next two blocks perform (and save) the correction for a spatially-varying slope over the MCP, which will result in a much more narrow SHAD. One must specify the grid dimension for the correction (**gsize**) and the minimum number of hits that must be in one of the spatial bins for a slope to be calculated (**tilelim**). You can run the first of these two blocks until satisfied, then run the second block to save the result.

The next two blocks show (and save) a movie of the single-hit data. One must correctly set the time range of the movie. The time histogram produced above can help with this. The number of frames, speed, and smoothing can also be set.

The next block performs a series of fits to characterize the SHAD. However, this step is not necessary since the following step now uses the SHAD directly with no need for fitting. If one wishes to use the fit-based treatment in the next step, this block must be used first.

The final major block ("Load3D with correlation") performs the full assignment of the multi-hit data. Currently, the only parameter one might wish to change here is the maximum number of hits per shot (**nctrmax**) to attempt to assign. The maximum recommended value is 20, however, one can speed up the processes by lowering this value. The multi-hit assigned data is saved as a "...C.pt.npy" file along with a text log file "...C.params.txt" containing the parameters used in the analysis and some additional statistics. This analysis also saves a ".bad.npy" file containing all of the centroid which were not able to be time-stamped, and a ".csc.npy" file containing additional information about the assignment results for each shot.

The final blocks after this produce another movie, this time using the full multi-hit data set. The usage is the same as the movie above, except that one can now specify a rotation angle to apply to the data.

4 Analyzing the Data

The analysis of the data is less structured, since it depends on what the user's objectives are. I will instead provide instructions on how to perform a 'standard' analysis. The scripts for these steps are not as generalized as the ones above. Instead of having a single script that can be pointed at the data and executed, it is better to make copies of existing scripts and adjust them to fit the desired analysis.

The scripts I will use as examples in the following are:
`3DVMICalibration_Scripts_NO205_2CH_20240207_141739.py`,
`3DVMICalibration_Scripts_NO205_2CH_20240207_141739_Part2.py`,
`3DVMICalibration_Scripts_NO205_2CH_20240207_141739_Rot.py`, and
`3DVMICalibration_Scripts_NO205_2CH_20240207_141739_betafit.py`

These scripts are for analysis of two-channel data and begin from where the processing left off, with the ".pt.npy" assigned data file (renamed in this case to "dataS.npy"). Since these scripts were edited to suit a particular analysis and weren't written with the intention of being reused as they are, the following instructions should be taken as a general approach to the analysis rather than a rigid set of instructions.

4.1 Step 1: Rotational Alignment

In general, the data will need to be rotated to align with the experimental axes. Note that this process requires an existing momentum calibration function. If no such function currently exists, use approximate values for the angles for now and skip to the next section. After preparing a calibration function, return here and properly fit the rotational angles.

In the first script, `...20240207_141739.py`, we first set the XY-plane rotation angle (**rotangle**) to 0 so it can be determined later. Also specify the approximate time centre (**t0**), image centre (**cen**), and approximate time duration of the Newton sphere (**dt**). The shuffle-based covariance method (**covshuffle** = True) is faster and has a similar performance to the slow method. The script starts by trimming the data to a **dt**-sized window, moves the distribution so the centre is at a value of **T0**, centers the distribution in space, and rotates it (although not at the moment, since we have set the angle to 0). The first plot in the script can help to optimize the time-centre (**t0**) and window (**dt**). The following covariance treatment removes statistically insignificant data and saves the remaining data. There are plots here which show the results of the covariance. There is also a plot which shows a center time-slice through the distribution, and can be used to optimize the spatial centre (**cen**). At this stage we stop progressing through this script and open the second script: `...20240207_141739_Part2.py`.

The covariance results, saved as "...P.npz" are loaded in to the "part 2" script. This script produces Cartesian and polar momentum and energy maps of the data. The code block which produces the polar maps contains the rotation angles which are to be applied to the polar data. The polar rotation angle **a** represents rotation about the y axis and should be set to 0 for this step. After running the polar map block, the results are saved as "...000.npz". If you wish to apply mirroring to the data, the following block can be used. This will mirror the data according to the specified angular indices and save the result as "...000M.npz".

We then open the third script `...20240207_141739_Rot.py`. This script loads the zero-rotation polar momentum map and fits Legendre polynomials to determine the correct rotation angles to align with the experimental axes. One can adjust the widths of the momentum and angular slices here if desired. After running this script with the path pointing to the "...000.npz" file produced in the previous step, the results should be displayed in a figure like Fig. 2. Take note of these angles. The XY-slice angle should be entered as the variable **rotangle** in the first script. The XZ-slice angle should be entered as

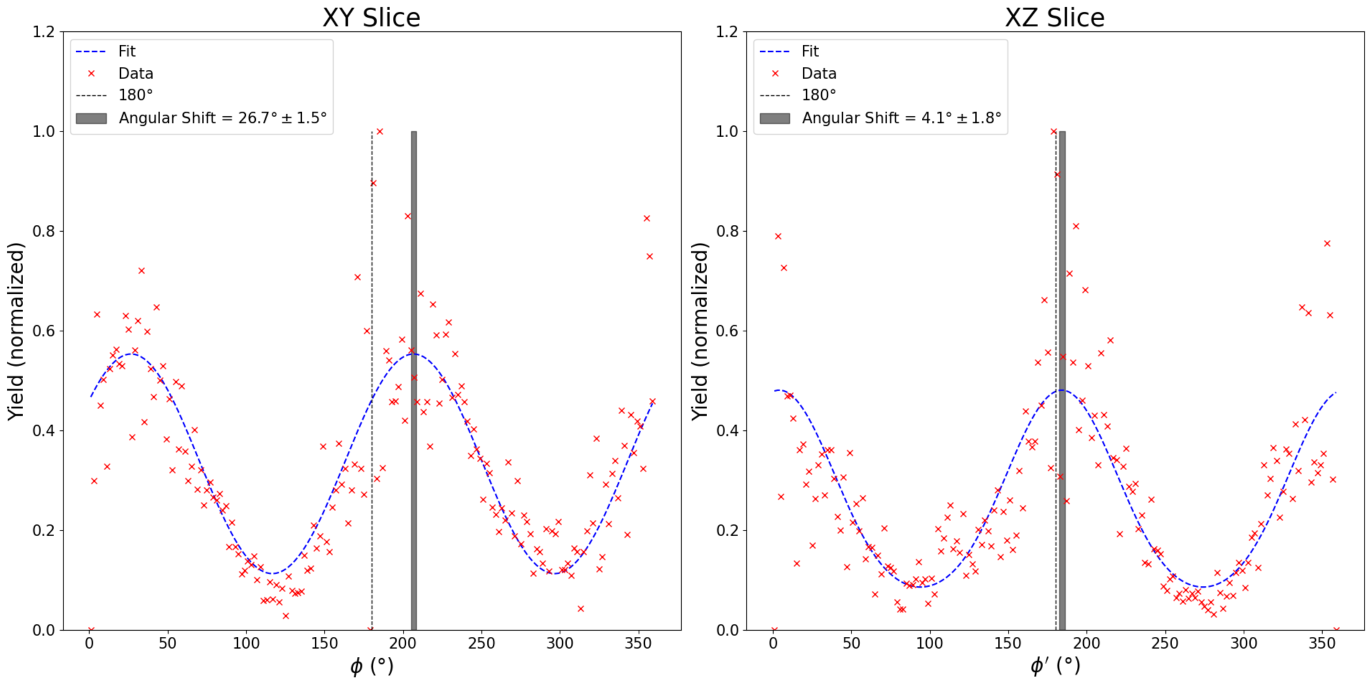

Figure 2: Result of running the script "...20240207_141739_Rot.py".

the variable **a** in the polar map block in the "part 2" script. The first script should then be run again to the same point as before with the changed angle.

4.2 Step 2: Preparing the Calibration Function

The remaining part of the first script fits the momentum distribution to obtain a calibration for reconstructing the initial momentum vectors. If you wish to do this for the current data set, resume running the script from the "Find 2D radius" block. This block fits the radial distribution of the central slice data to obtain the maximum radius of the slice. In the next block, the 2D calibration constant **K** is calculated (and saved) from the 2D radius and the known kinetic energy of the feature **ke**. The code blocks after this fit slices through the transverse momentum ($p_\perp$) VS time plot to obtain coordinates ($p_\perp$, t) describing the feature. The widths, positions, and number of slices can be adjusted here. Once these coordinates are found, the transverse momentum is converted into longitudinal momentum (p_z) using the known energy. Finally, the calibration function is found by fitting p_z VS t using a second order polynomial, as shown in Fig. 3.

4.3 Step 3: Gain Correction and Mirroring

Using a calibration source with known energy and angular distribution, we can create a radial correction function to account for the common gain degradation near the centre of the MCP. The first step is to create a simulated data set that mimics the calibration feature. For this, we use the script `gaincorrection_simulation.py`. We first specify the energy (**eref**), bandwidth (**dp**) and asymmetry parameter β_2 (**beta**). Currently, this simulation only allows β_2, but it could be easily generalized to higher order asymmetry parameters. This simulated data will be compared directly to the experimental data, which will in general have a rotational offset as determined above. We specify the XZ-plane rotation (**a**) so it can be applied to the simulated data. Note that the angle here must be the negative of

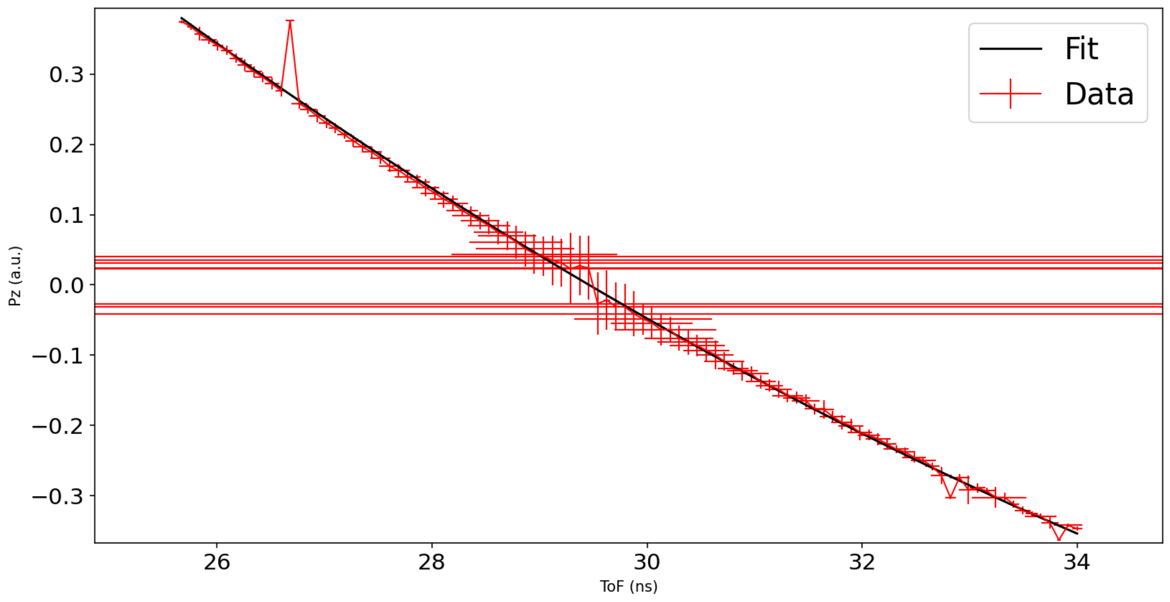

Figure 3: Example of fitting a calibration function. Note that this not an optimized result.

the angle used in the script above. Running this script will now produce a simulated distribution with the specified parameters. It will be saved as "mod_X_Y.npz" where X is the rotation angle and Y is the β_2 multiplied by 100.

Next, we compare this simulated data set with the corresponding experimental one to derive the correction function. This is done using the script `gaincorrection_fig.py`. Here, we specify the paths of the simulated data set (**spath**), the experimental zero-rotation ("...000.npz" or, if mirrored, "...000M.npz") data (**dpath**), and the 2D VMI calibration constant K (**kpath**). This script will normalize both the simulated and experiential data sets, show the ratio of them, then extract the radial dependence of the gain. A damping function is then fit to the radial gain dependence and saved for later use. This process is shown in Fig. 4.

Once the correction function is saved, we move back to the "part 2" script to apply the correction. This script can now be run all the way through, making sure that the rotation angle **a** has been correctly set and that the path for the radial gain correction is specified (after the first mirroring block). This will save a polar momentum map of the data with and without mirroring (is desired) and with and without the radial gain correction.

4.4 Step 4: Fitting the Asymmetry Parameters

Fitting asymmetry parameters to the data set is done using the script `...20240207_141739_betafit.py`. Begin by specifying the path to the polar momentum map that you wish to fit as well as the constant **K**. This script will extract a slice in total momentum which includes the feature of interest, then fit the entire integrated distribution as well as multiple angular slices through it. You can specify here the width of the momentum slice (**n0**) and angular slices (**nth**). The kinetic energy of the feature (**ke**) will also need to be set. From here, simply running the script will produce a plot similar to Fig. 5. The β_2 and β_4 values are shown for the integrated distribution, XY- and XZ- slices. In addition, the variation in asymmetry parameters are shown with 10-degree slices through the entire distribution.

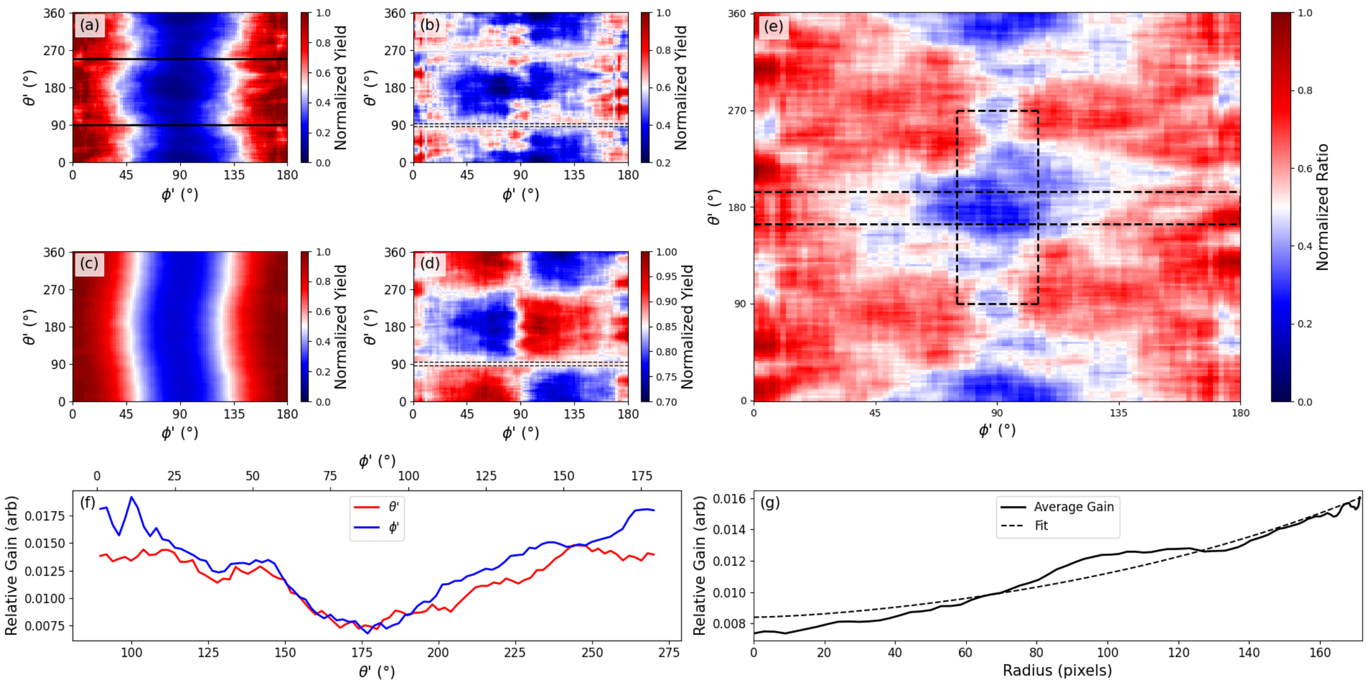
Figure 4: Example of fitting a gain correction function.

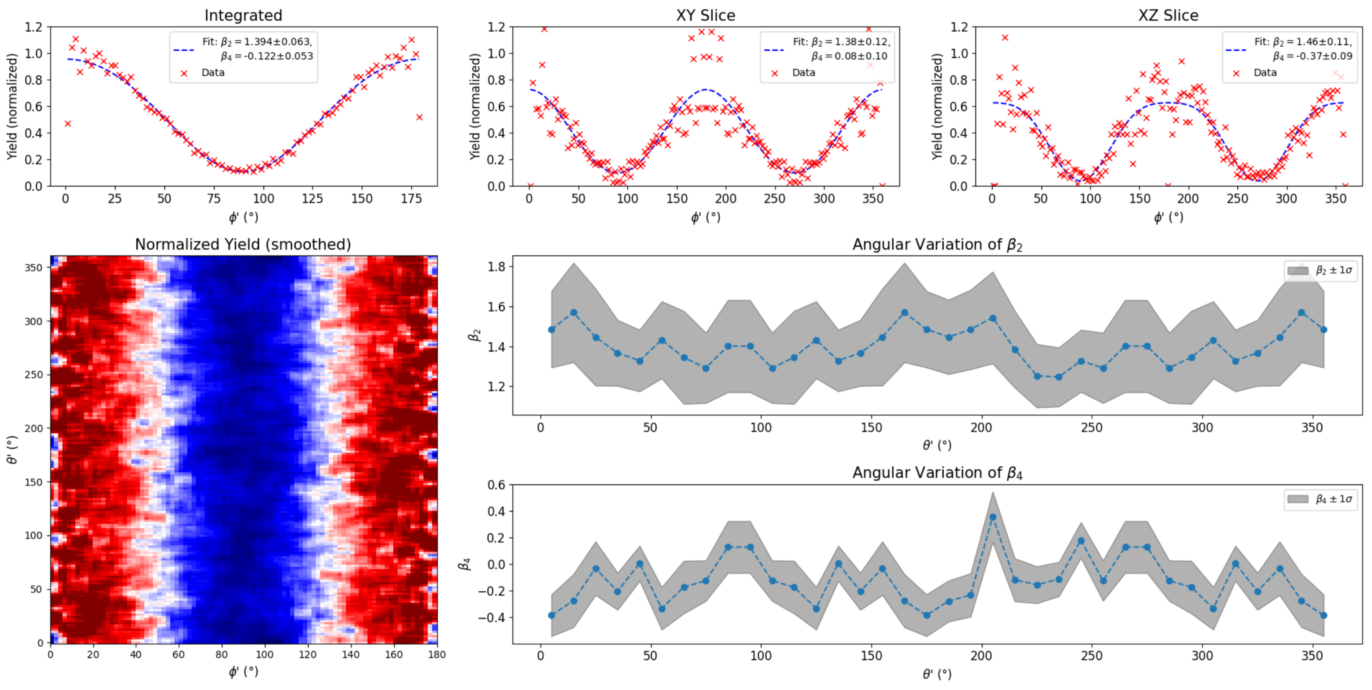

Figure 5: Example of fitting asymmetry parameters to an experimental data set.