Multi-Orientation MAXWELL; Example Protocol

Sierra Dean
E-mail: ccnd@live.com

05 December 2024

This document serves as a general walkthrough of how to use the Multi-Orientation MAXWELL analysis method. In addition to custom software hosted as various python classes, simulated data is provided to ensure a thorough understanding of the software capabilities.

Software:
https://github.com/SierraD/Multi-Orientation-Maxwell/
Simulated Data:
https://figshare.com/authors/Sierra_Dean/20319102

For this example, all classes composing the Multi-Orientation MAXWELL software are copied directly from the online repository into Jupyter Notebook cells. The classes can also be ran through the command line interface. Both the code, as well as the output, are recorded here as a general recommendation on how to execute the analysis method.

1 Imports and Supplied Data

The data used for this documentation is hosted on Figshare, as tiff datatype files simulating lightsheet images acquired with known variables. In addition, ThunderSTORM analysis results tables have been provided as csv files using the MAXWELL camera values and default ThunderSTORM analysis parameters.

The simulated data employs a 230 nm pixel size, which imitates a 20x magnification lens with MAXWELL apparatus. This data simulates a 60 nm step between images, depicting the in-depth "pixel" size when the data is rotated. The scintillation events have a FWHM between 2-5 pixels in size for the in-plane imaging, with intensity between 10,000-20,000 photons per event.

```
# Used in Preparation;
import pandas
# Used in Overlap;
import numpy
# Used in Plotting;
import plotly
from plotly import subplots
# Used in Precision;
import fitter
import matplotlib
import seaborn
import scipy
```

Listing 1: *The python imports required for the Multi-Orientation MAXWELL software classes.*

2 Preparation Class

The **Preparation.py** class is used to format the two different orientation ThunderSTORM result files into readable data that can be called by the other classes in the software.

The input for this class is two different orientation ThunderSTORM results files, namely XY and XZ. When performing ThunderSTORM analysis, the pixel size parameter is set to 1 to accommodate the asymmetric voxel size between the two different orientations. This class is used to correct the isotropic voxel size to the correct nanometer units, which requires input of the in-plane pixel size, obtained from the camera and magnification, as well as the in-depth pixel size, which is the step distance between successive images. These are titled as *pixelsize_xy* and *pixelsize_xz*, respectively.

This class also enables re-centering the data around zero, and limiting the data to a dedicated ROI to reduce the computation time.

The class is initiated with the following code:

```
Prepare = preparation()
Prepare = Prepare.setting("ThunderSTORM_Results_XY.csv",
                          "ThunderSTORM_Results_XZ.csv",
                          magnification=20,
                          pixelsize_xy=230,
                          pixelsize_xz=60)
```

Listing 2: *The code to initiate the Preparation.py class, using two ThunderSTORM results files obtained by analyzing two different orientations of the same data. This class requires the camera pixel size, which is the XY orientation, in addition to the step size between images, which is designated as the XZ pixel size.*

The data inside of the class is not visible until an internal parameter is called. The table below shows all of the internal parameters for the **Preparation.py** class which can be used to display the data.

Parameter	**Description**
name_xy	The name of the file used for the XY data.
name_xz	The name of the file used for the XZ data.
magnification	The magnification input by the user.
xzpix	The in-plane pixel size.
zpix	The in-depth pixel size, which was also the step size between successive images during experimentation.
dfxy	A dataframe of all data obtained from the XY results table.
dfxz	A dataframe of all data obtained from the XZ results table.
dfxy["X_XY"], dfxz["X_XZ"]	The X position data obtained from the XY, and XZ data files.
dfxy["Y_XY"], dfxz["Y_XZ"]	The Y position data obtained from the XY, and XZ data files.
dfxy["Z_XY"], dfxz["Z_XZ"]	The Z position data obtained from the XY, and XZ data files.
dfxy["I_XY"], dfxz["I_XZ"]	The intensity obtained from the XY, and XZ data files.
dfxy["O_XY"], dfxz["O_XZ"]	The offset obtained from the XY, and XZ data files.
dfxy["B_XY"], dfxz["B_XZ"]	The standard deviation of the background, obtained from the XY, and XZ data files.
dfxy["U_XY"], dfxz["U_X"] , dfxz["U_Z"]	The standard deviation of the localization uncertainty obtained from the XZ, or XZ data files, with two options for the XZ data due to the asymmetric pixel size.
dfxy["S_XY"], dfxz["S_X"] , dfxz["S_Z"]	The standard deviation of the Gaussian fitted on the peak during localization, obtained from the XZ, or XZ data files, with two options for the XZ data due to the asymmetric pixel size.

To view any of the data withheld inside the **Preparation.py** class, the initiated variable can be called with the following code, which will print all of the X positions obtained from the XY orientation. This code can be altered to print any of the internal parameters shown in the table. The output from the code is shown below the code, with a green background.

```
print(Prepare.dfxy["X_XY"])
```

```
0            653.130625
1           1090.070063
2            897.483837
3           1283.333925
4           1399.128039
...
25089     109889.442185
25090     109958.660024
25091     110285.011456
25092     110284.970581
25093     112719.640519
Name: X_XY, Length: 25094, dtype: float64
```

The **Preparation.py** class also allows for the data to be re-centered around zero in all directions by using the *set_to_center* command.

```
# Before the data is centered
print(Prepare.dfxy["X_XY"].min())
print(Prepare.dfxy["X_XY"].max())
# After the data is centered
Centered = Prepare.set_to_center()
print(Centered.dfxy["X_XY"].min())
print(Centered.dfxy["X_XY"].max())
```

```
# Before the data is centered
444.873388773216 # Minimum X
114495.27946649647 # Maximum X
# After the data is centered
-57025.20303886163 # Minimum X
57025.203038861626 # Maximum X
```

In addition, if the ThunderSTORM results files contain a large number of localizations, the data can be reduced to a region of interest to reduce the computation time.

```
Data = Centered.limiting("X", 15000, "less")
Data = Data.limiting("X", -15000, "more")
Data = Centered.limiting("Y", 15000, "less")
Data = Data.limiting("Y", -15000, "more")
print(Data.dfxy["X_XY"].min())
print(Data.dfxy["X_XY"].max())
print(Data.dfxy["Y_XY"].min())
print(Data.dfxy["Y_XY"].max())
```

```
-14809.500107940548 # Minimum X
14999.995595139713 # Maximum X
-14946.366838933172 # Minimum Y
14051.911396848314 # Maximum Y
```

The data, formatted by the **Preparation.py** class, can be downloaded as a csv file using the *download_dataframe* command.

```
Data.download_dataframe(filename="Preparation_Dataframe")
```

3 Overlap Class

The ***Overlap.py*** class is used to determine the position matching events between the two different orientations. This requires the data to have been formatted in the correct nanometer units with the ***Preparation.py*** class. To find the position matched events, the indexes of the overlapped events are first determined, then the values associated with those indexes are saved for future filtering.

To determine overlapped, or position matched events between the two orientations, the position in all three dimensions is evaluated. As the localization analysis provided by ThunderSTORM is inherently two-dimensional, both orientations will have a direction with a very large positional uncertainty, shown in the table below. For the XY analysis, the Z position is very uncertain, with the step size between images providing the positional uncertainty. Similarly, when the data is rotated to provide the XZ orientation, the in-plane pixel size becomes the step size between images, and also the positional uncertainty for the Y direction.

XY Orientation		
Position	**Uncertainty**	**Uncertainty Meaning**
X	ΔX	X positional uncertainty calculated and returned by ThunderSTORM
Y	ΔY	Y positional uncertainty calculated and returned by ThunderSTORM
Z	ΔZ	Step size between successive images, experimental step parameter
XZ Orientation		
Position	**Uncertainty**	**Meaning**
X	ΔX	X positional uncertainty calculated and returned by ThunderSTORM
Y	ΔY	Step size between successive images, experimental camera pixel size
Z	ΔZ	Z positional uncertainty calculated and returned by ThunderSTORM

To find the position matched events, the three-dimensional position of each localization in the first orientation is compared with every localization from the second orientation. A pair of localizations between XY and XZ is only considered a position matched pair if their X, Y, and Z positions all overlap, within positional uncertainty values.

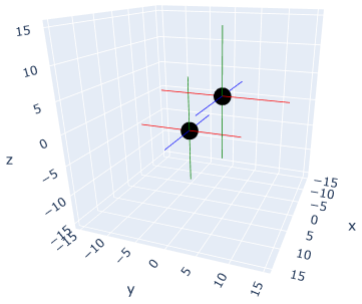

Figure 1: *A demonstration showing a position-matched pair of localizations. The three-dimensional position of the localization is shown with a black dot, with bars depicting the X (blue), Y (red), and Z (green) positional uncertainty values. The two localizations are selected as a position matched pair due to all of their XYZ positional uncertainty values overlapping.*

In the XY orientation, the X and Y positional uncertainty values are calculated by ThunderSTORM, and are deemed accurate. On the other hand, the Z positional uncertainty is determined by the step size, and is much larger. Similarly with XZ, the Y positional uncertainty is the pixel size, and is very large. To increase or decrease the large positional uncertainty values used to determine the overlap between XY and XZ localizations, the *z_range* and *xy_range* can be provided when initiating the **Overlap.py** class. Generally, the experimental parameters are provided, as shown below. The indexes of the overlapped events are first determined with the *indexes* command, with the localization information then assigned with the *values* command.

```
Matched = overlap(Data)
Indexes = Matched.indexes(z_range=Matched.zpix,
                          xy_range=Matched.xypix)
Values = Indexes.values()
```

Listing 3: *The code to initiate the Overlap.py class. This class requires two parameters, z_range and xy_range, which designate the allowable overlap between positional uncertainty values. The indexes command will determine the indexes of the overlapped events, with the values command then assigning the ThunderSTORM values to the indexes.*

Similarly to the **Preparation.py** class, the internal parameters can be viewed using any of the following commands:

Parameter	Description
magnification	The magnification input by the user.
xzpix	The in-plane pixel size.
zpix	The in-depth pixel size, which was also the step size between successive images during experimentation.
Indexes	
XY_indexes	A list of indexes from the XY data which are position matched pairs with the list of indexes from the XZ data.
XZ_indexes	A list of indexes from the XZ data which are position matched pairs with the list of indexes from the XY data.
Values	
df	A dataframe including the position matched data from both orientations.
dfxy	A dataframe including the position matched data only from the XY orientation results file.
dfxz	A dataframe including the position matched data only from the XZ orientation results from.

The *indexes* command creates a dataframe of all of the indexes for the position matched pairs. In this example, the first position matched pair of localizations is the 61st localization from the XY results file, and the 10th localization from the XZ results file.

```
print(Indexes.XY_indexes[0])
print(Indexes.XZ_indexes[0])
```

```
61
10
```

The *values* command reads the indexes, and acquires the data from the original ThunderSTORM results files for those designated indexes, returning them as a dataframe. In the following code, it can be seen that the dataframe from the *values* command gives the same information as the **Preparation.py** dataframe when using the *indexes* from the indexes command.

```
print(Values.df["X_XY"][0])
print(Values.df["U_XY"][0])
print(Values.df["X_XZ"][0])
print(Values.df["U_X"][0])
```

```
11541.408424869674
4.5305378438347095
11543.13812577413
5.4460202423270445
```

```
print(Data.dfxy["X_XY"][60])
print(Data.dfxy["U_XY"][60])
print(Data.dfxz["X_XZ"][10])
print(Data.dfxz["U_X"][10])
```

```
11541.408424869674
4.5305378438347095
11543.13812577413
5.4460202423270445
```

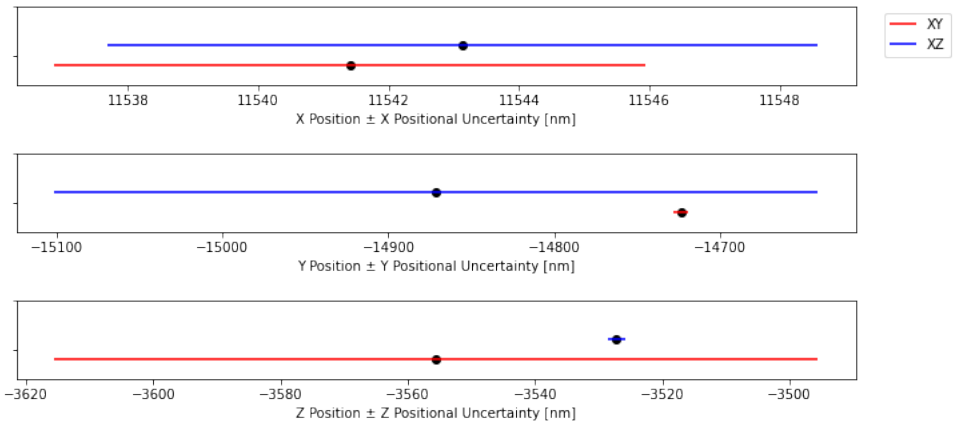

Figure 2: *A demonstration showing the three dimensional overlap of the first position matched pair. The X, Y, and Z positions are displayed with a black circle, with the positional uncertainty for the XY orientation shown as red bars, and XZ orientation shown as blue bars. The figure includes the overlap from the X positions ± X positional uncertainty (Top), Y positions ± Y positional uncertainty (Middle), and Z positions ± Z positional uncertainty (Bottom).*

4 Filtering Class

The **Filtering.py** class is used to filter the position matched events to ensure no duplicate events exist. Duplicate events are seen when one localization from the XY dataframe is position matched with various localizations from the XZ dataframe, or vise versa. Duplicate events can occur most frequently when scanned with the X-ray light sheet tail.

It has been observed that when duplicate events occur, the multiple localizations from a single set have drastically increasing or decreasing positional uncertainty, allowing for the selection of the lowest positional uncertainty as the particle's true location out of all of the duplicates.

The **Filtering.py** class requires a *merge* command first, to group events which overlap with multiple other points. Once grouped together, the *selection* command allows the duplicates to be filtered out either by *uncertainty*, as described above, which is the recommended selection, or by *intensity*, which will choose the closest intensity values.

Lastly, the *points* command is used to determine the three-dimensional localizations, by combining the most accurate information from the position matched events.

```
Filtered = filtering(Values)
Filtered = Filtered.merge()
Filtered = Filtered.selection(selection_type="uncertainty")
Filtered = Filtered.points()
```

Listing 4: *The code to initiate the Filtering.py class. This class requires one parameter, the selection to be used for the filtering. The options for the parameter include "uncertainty" or "intensity", which will select the true location among duplicated events based on minimum positional uncertainty or matching intensity, respectively.*

Similar to the previous classes, the internal parameters can be viewed with any of the following commands, which are oriented similar to the results file from ThunderSTORM analysis.

Parameter	Data Description
points	A dataframe of all the three-dimensional localizations, after filtering
X [nm]	The X position of the localizations.
Y [nm]	The Y position of the localizations.
Z [nm]	The Z position of the localizations.
Uncertainty XY [nm]	The positional uncertainty on the X and Y positions.
Uncertainty Z [nm]	The positional uncertainty on the Z position.
Sigma XY [nm]	The standard deviation of the Gaussian fitted on the peak in the X and Y directions.
Sigma Z [nm]	The standard deviation of the Gaussian fitted on the peak in the Z direction.
Intensity XY [Photons]	The intensity observed from the XY point spread function.
Intensity XZ [Photons]	The intensity observed from the XZ point spread function.
Offset XY [Photons]	The offset obtained from the XY point spread function.
Offset XZ [Photons]	The offset obtained from the XZ point spread function.
Bkgstd XY [Photons]	The standard deviation of the background, obtained from the XY point spread function.
Bkgstd XZ [Photons]	The standard deviation of the background, obtained from the XZ point spread function.

The following code demonstrates how many points are contained within the finalized dataframe, as well as the minimum value for the positional uncertainty in the X and Y directions, as well as the minimum positional uncertainty in the Z direction.

```
print(len(Filtered.points["X [nm]"]))
print(min(Filtered.points["Uncertainty Z [nm]"]))
print(min(Filtered.points["Uncertainty XY [nm]"]))
```

```
98
1.100363163786924
3.344546645000296
```

5 Plotting Class

The **Plotting.py** class is used as a convenient plotting tool which expedites plotting the results from any of the described classes in both two and three-dimensional options.

```
# For data from Filtering.py:
Visualize = plotting(Filtered)
# For data from Overlap.py:
Overlap_Visualize = plotting(Values)
# For data from Preparation.py:
Prepare_Visualize = plotting(Data)
```

Listing 5: *The code to initiate the Plotting.py class. This class requires one parameter, which is a dataframe of localizations created by any of the described classes.*

The *Tricolumn* command can be used to visualize the data using three two-dimensional plots, showing XY, XZ, and YZ orientations. For this command, if the data being plotted is the results from the **Overlap.py** or **Preparation.py** classes, the *dimensions* option should be set to 2, as the data is two sets of two dimensional data. Conversely, if the data being plotted is from the **Filtering.py** class, the *dimensions* option should be set to 3, as the data is three dimensional.

```
Visualize = Visualize.tricolumn(dimensions = 3)
Overlap_Visualize = Overlap_Visualize.tricolumn(dimensions = 2)
Prepare_Visualize = Prepare_Visualize.tricolumn(dimensions = 2)
```

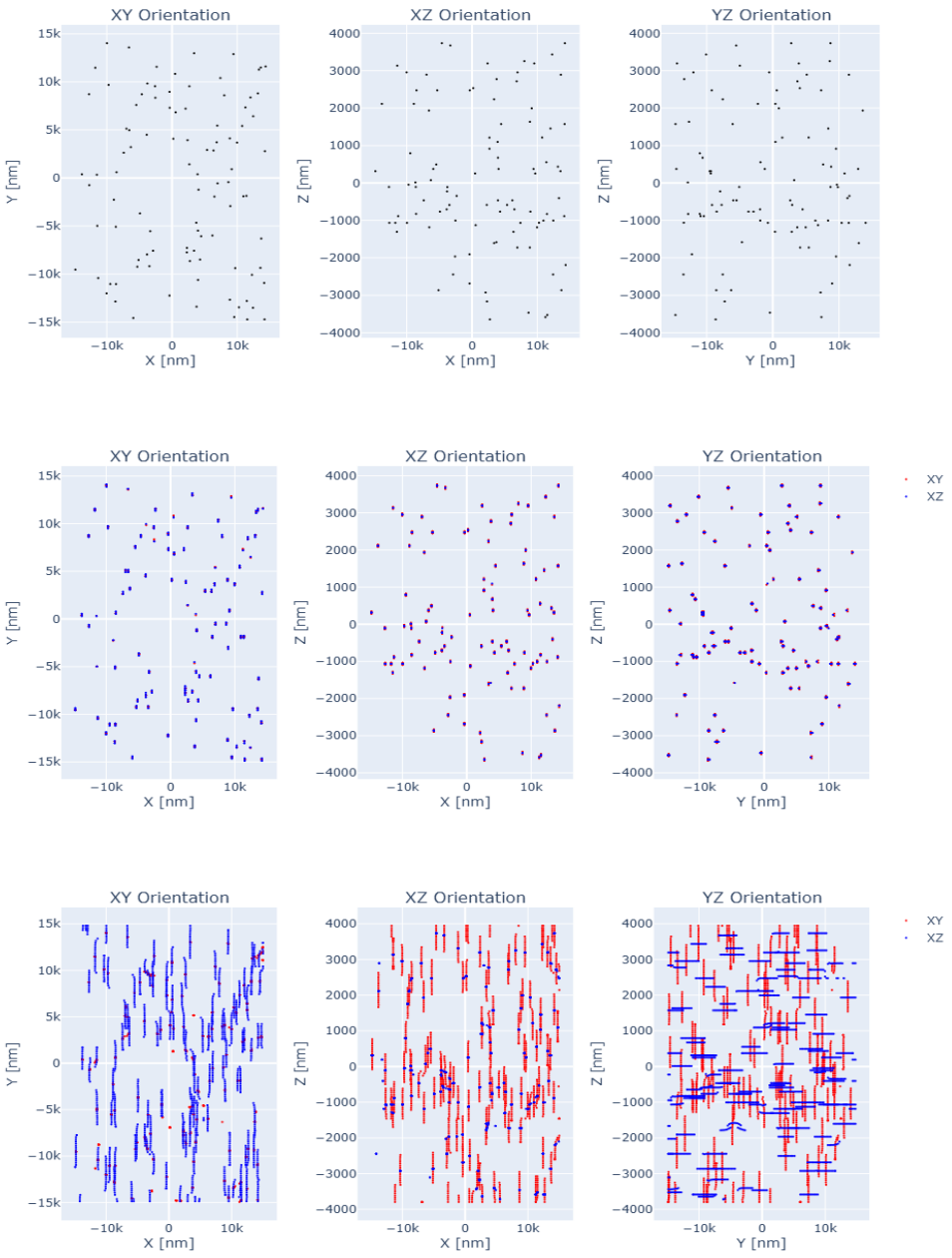

Figure 3: *Tricolumn plots obtained from the dataframe obtained from* ***Filtering.py*** *(Top),* ***Overlap.py*** *(Middle), and* ***Preparation.py*** *(Bottom). For the plots containing localizations from various orientations, the XY is depicted in red, and XZ in blue.*

For the *Tricolumn* command, the data is visualized with an assigned point size, to save computational time, but using the *Tricolumn_Sigma* command instead, the points will be visualized by their individual standard deviation values of the Gaussian fitted on the peak during localization, understood as the unique point spread function of the localization, which is more computationally demanding.

```
Visualize = Visualize.tricolumn_sigma(dimensions = 3)
```

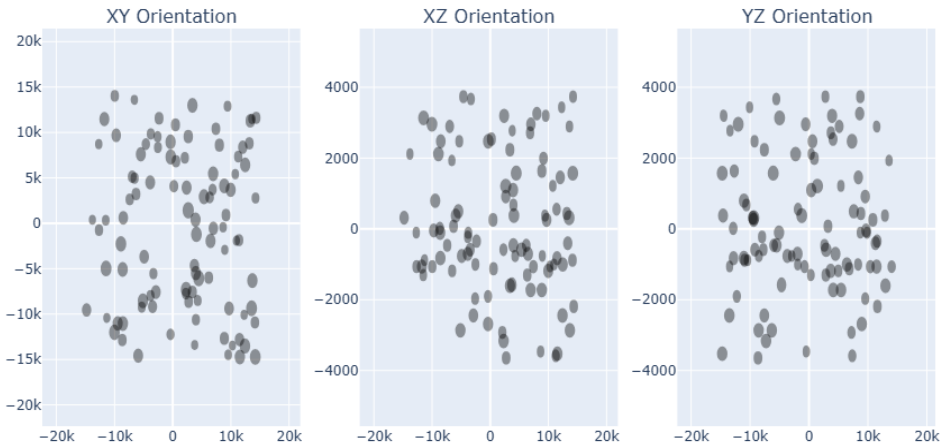

Figure 4: *Tricolumn_sigma plots obtained from* ***Filtering.py*** *dataframe, which depicts the size of each localization as the standard deviation of the Gaussian fitted on the peak during localization.*

Similarly, the *Three_dimensional* command can be used to visualize all localizations from the **Filtering.py** dataframe, with an arbitrary point size. The *Three_dimensional_sigma* command allows the points to be visualized from their point spread functions.

```
Visualize = Visualize.three_dimensional(dimensions = 3)
```

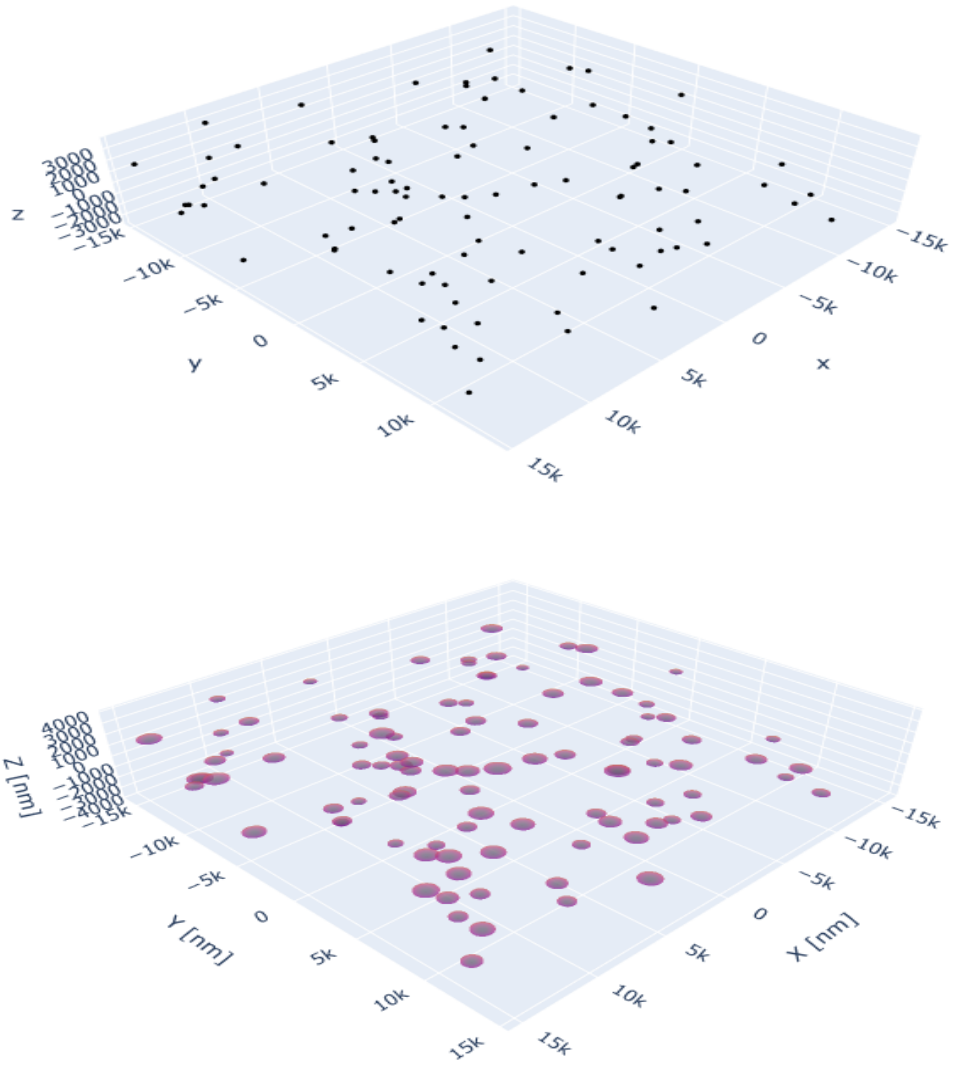

Figure 5: *Three_dimensional and three_dimensional_sigma plots obtained from the* ***Filtering.py*** *dataframe, which depicts the size of each localization as an arbitrary size, or as the standard deviation of the Gaussian fitted on the peak during localization, respectively.*

6 Precision Class

The **Precision.py** class is used to determine the method precision for the analyzed data. The precision can be generated for the Bkgstd, Uncertainty, or Sigma columns from the **Filtering.py** class, as these values are all standard deviation values.

To perform the analysis, a histogram is created of the chosen data, which is then fit with PDF fittings from common distributions to find the most accurate fit for the data. The PDF provides insights into the histogram properties. In the case of standard deviation results, the precision can be obtained by determining the most likely value of the standard deviation, which will be the peak of the PDF fitting applied to the histogram.

```
Z_Precision = precision(Filtered,
                        data_type="Uncertainty Z [nm]")
```

Listing 6: *The code to initiate the **Precision.py** class. This class requires a dataframe of three-dimensional localizations, formatted in the **Filtering.py** class, as well as a column specified by the data_type parameter.*

The *get_precision* command is used to obtain a histogram of the specified data column, then fit the histogram with PDF distributions. If the most accurate distribution is not known, the *distribution* parameter can be specified as *distribution = fitter.get_distributions()*, which will fit the histogram with various distributions, which are visible in the Python Fitter Package documentation [https://pypi.org/project/fitter/]. If no distribution is specified, only the Fitter common distributions will be fit. If the best fit option is known, the distribution can be set to save computation time.

```
Z_Precision = Z_Precision.get_precision(distribution=
                                        "norminvgauss")
```

```
                  sumsquare_error     aic             bic
norminvgauss      13.88973            336.197606      346.537476
                  kl_div              ks_statistic    ks_pvalue
norminvgauss      inf                 0.074514        0.620843
```

The *fitted_precision* command is used to fit the histogram with the best fit determined, and to obtain the histogram properties. This command includes the *kde* parameter, which will also fit the histogram with a Kernel Density Estimate for reference. In addition, the number of histogram bins can be specified with *bins*, and the best fit distribution determined with the previous command should be specified.

```
Z_Precision = Z_Precision.fitted_precision(kde=False, bins=100,
                                           distribution="norminvgauss")
```

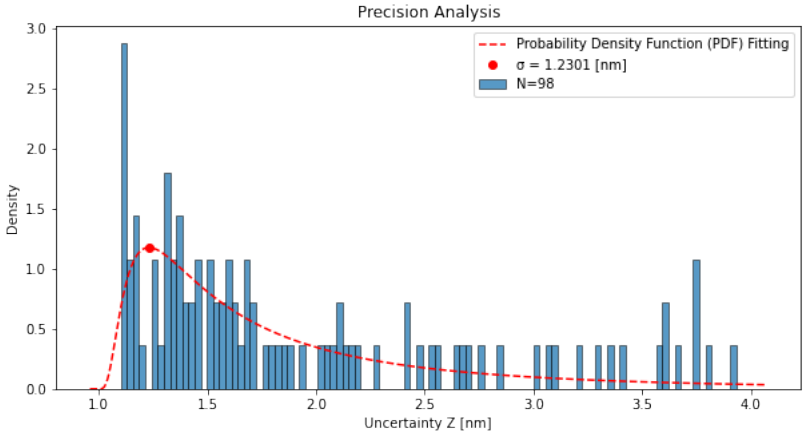

Figure 6: *The histogram and PDF fitting obtained from the* ***Precision.py*** *dataframe, depicting the results from the Uncertainty Z [nm] column. The localizations are depicted as a histogram, with the specified column displayed on the X axis. The number of events is recorded in the legend. The chosen PDF, a Normal Inverse Gaussian for this example, is plotted in red, with the maximum value of the PDF fitting shown as a red circle. The precision is described as the maximum value of the PDF fit on a histogram of standard deviation values.*

This command can be repeated for other columns, to determine the precision for other localization values as well, such as the positional uncertainty in the X and Y directions.

```
XY_Precision = precision(Filtered,
                         data_type="Uncertainty Z [nm]")
XY_Precision = XY_Precision.get_precision(distribution="burr")
XY_Precision = XY_Precision.fitted_precision(kde=False,
                                             bins=100,
                                             distribution="burr")
```

```
          sumsquare_error     aic                 bic
burr      0.333999            1500.496955         1510.836825
          kl_div              ks_statistic        ks_pvalue
burr      inf                 0.080205            0.527615
```

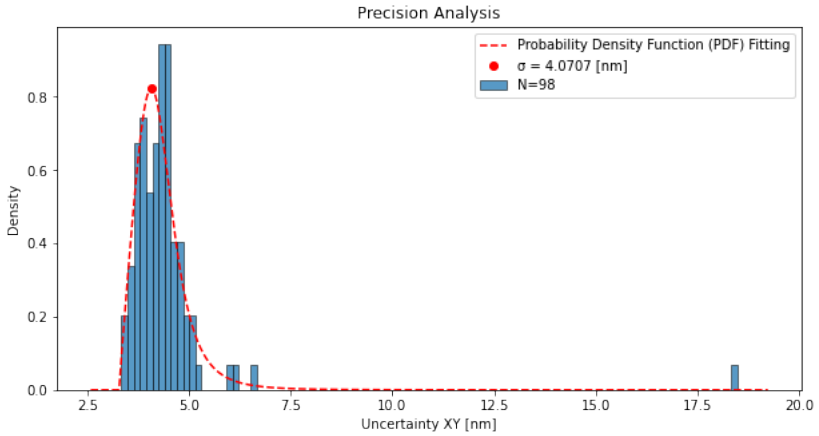

Figure 7: *The histogram and PDF fitting obtained from the* ***Precision.py*** *dataframe, depicting the results and precision from the Uncertainty XY [nm] column.*