RIMSEval

Release 2.0.3.dev16+g7cea290

Reto Trappitsch

Apr 16, 2024

CONTENTS

CHAPTER ONE

WELCOME!

The goal of this project is to give the user an API as well as a GUI interface to evaluate RIMS spectra. All code is available on GitHub.

Please check the sections below for help on installation, usage, etc.

CHAPTER TWO

CONTENTS

2.1 License

MIT License

Copyright (c) 2021-2022 Reto Trappitsch

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

2.2 Installation

The package is available for install via `pip`. To install the latest stable version, you can run:

```
pip install rimseval
```

To get the latest development version, add the `--pre` flag to the installation command. To get the latest developments from GitHub, the package can be installed via:

```
pip install git+https://github.com/RIMS-Code/RIMSEval.git
```

Installations for development are easiest done after cloning the repo. Enter the folder and install the package editable by typing:

```
pip install -e .
```

2.3 General Usage of the API

Note: Here, the general usage of the API is described. Information on filtering data can be found *here*.

2.3.1 Convert LST to CRD

To convert a list file to a CRD file, use the `LST2CRD` class. Details can be found here: `rimseval.processor.data_io.lst_to_crd.LST2CRD class()`. If your list file is at `path/to/file.lst`, you can convert it to a CRD file as following:

```
from pathlib import Path
from rimseval.data_io import LST2CRD
file = Path("path/to/file.lst")
lst = LST2CRD(file_name=file, channel_data=1, tag_data=None)
lst.read_list_file()
lst.write_crd()
```

You will get a *crd* file with the same name and in the same folder.

Note: Depending on the TDC, the channel number that you need to pass for the `channel_data` or `tag_data` variable are not equal to the channel numbers written on front of your TDC. For some TDCs, the manufacturer uses different numbers in the software than on the labels. The following tables shows models that are implemented in the software, for which the channels that you need to set and the stop numbers (labels on the TDC) are different. Please consult your TDC manual!

Table 1: MCS8A

Channel Number	TDC Identifier / Label
9	STOP 1
10	STOP 2
11	STOP 3
12	STOP 4
13	STOP 5
14	STOP 6
15	STOP 7

2.3.2 Work with CRD files

This chapter gradually builds on itself to give you an overview of the package functionality.

Load a CRD file

Let us assume you have a file named `my_file.crd`. Creating a `Path` of this file using `pathlib`, we can then open it as following:

```python
from pathlib import Path
from rimseval import CRDFileProcessor

my_file = Path("my_file.crd")
crd = CRDFileProcessor(my_file)
crd.spectrum_full()
```

The last command in this sequence processes the spectrum such that the time of flight array of data is populated. This command can also be used to reset a dataset at any point.

Mass calibration

To perform a mass calibration, you need to know at least two time of flights (in microseconds) and their corresponding mass values. Group these into a `numpy.ndarray`, in which each line contains first the time of flight and then the associated mass values.

```python
mass_cal_parameters = np.array([[1, 10], [2, 20]])
crd.def_mcal = mass_cal_parameters
```

Note: If more than two values are defined, a linear optimization is run to determine the best mass calibration.

In above example, the time of flights are `1` and `2` with associated masses `10` and `20`, respectively. After setting the parameters, the mass calibration can be executed as:

```python
crd.mass_calibrataion()
```

Note: This calibration can also be performed using a `matplotlib` GUI. For details, see *here*.

Dead time correction

To correct your mass spectrum for dead time effects, you need to know the number of bins d of your TDC that are dead after an ion hits. Assuming $d = 7$, you can then correct your spectrum for dead time effects as following:

```python
crd.dead_time_correction(7)
```

Note: This is only an example of a data evaluation tool. More information on all the tools can be found *here*.

Integrals and Backgrounds

To set integrals, you need to prepare a name for each integral (e.g., the isotope name) and the mass range over which you want to integrate. Assuming you have two peaks, e.g., ^{235}U and ^{238}U, you could define the integrals as following:

```
integral_names = ["235U", "238U"]
integral_ranges = np.array([[234.8, 235.2], [237.8, 238.2]])
crd.def_integrals = integral_names, integral_ranges
```

Warning: Each integral name should only be used once!

In a similar fashion, you can define backgrounds for a peak. For example, if you want to define a 0.2amu background for your ^{235}U peak on the left and right of it, you could set the backgrounds as following:

```
bg_names = ["235U", "235U"]
bg_ranges = np.array([[234.6, 234.8], [235.2, 235.4]])
crd.def_backgrounds = bg_names, bg_ranges
```

Warning: The backgrounds you define must have the same name as the peaks they are defined for. Multiple definitions per background can exist.

Note: Integral and background definitions can also be performed using a `matplotlib` GUI. For details, see *here*.

To apply the integrals, simply run:

```
crd.integrals_calc(bg_corr=True)
```

Background correction can be turned on and off by setting the `bg_corr` variable to either `True` or `False`, respectively. Details on integral background corrections and the math behind it can be found *here*.

Finally, if your integral names follow the format used in the `iniabu` package, you can calculate δ-values for your individual peaks automatically. These values are always calculated with respect to the major isotope. If values are not available, `np.nan` will be written for that specific δ-value. To calculate the δ values, run (after calculating integrals):

```
crd.integrals_calc_delta()
```

This will save the δ-values and associated uncertainties to `crd.integrals_delta`. If packages were defined, an δ-values for each package will also be calculated and stored in `crd.integrals_delta_pkg`. Details on the calculation and error propagation can be found *here*.

Note: While δ-values are generally calculated with respect to the most abundant isotope, this calculation is internally performed using the `iniabu` package. The currently supported version of `iniabu` allows you to select the normalization isotopes. The following gives an example in order to do so.

```
from rimseval.utilities import ini
ini.norm_isos = {"Ti": "Ti-46", "Ba": "Ba-136"}
```

This code would set the normalization isotopes for titanium and barium to ^{46}Ti and ^{136}Ba, respectively. These new normalization isotopes are then respected by subsequent calls to calculate δ-values. Important here is that you import the ini instance that rimseval uses and not any new instance from iniabu. For details on the norm_isos property, see the documentation here.

Results

If everything worked as planned, you can access your processed mass spectrum via multiple variables. An overview of the available variables can be found *here*.

2.4 GUIs

Here, the GUIs that are available in the rimseval packages are described.

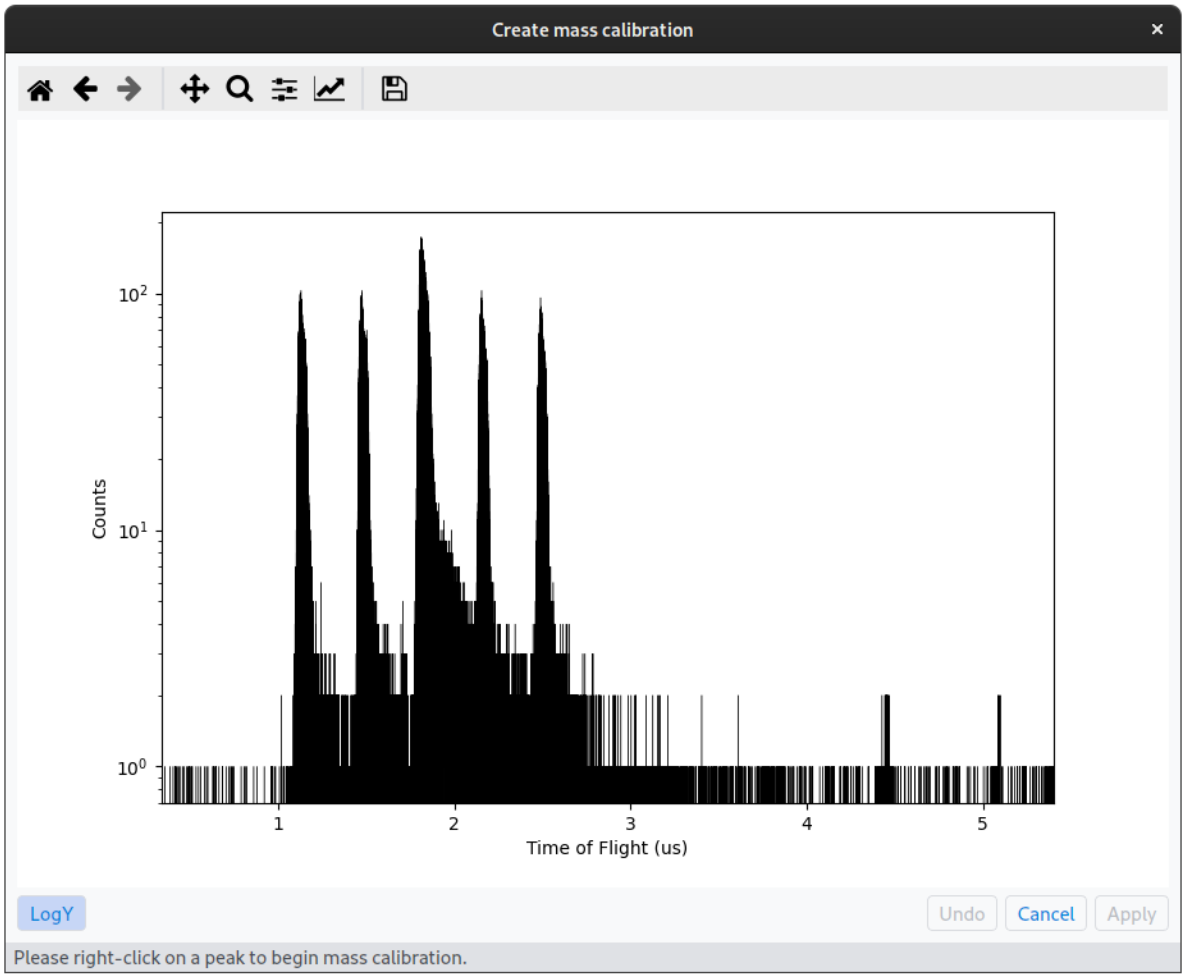
Note: This chapter does not describe the usage of the RIMSMEvalGUI, which is a GUI around the whole package! However, the GUIs described here are also used in the full RIMSEvalGUI. The manual for that GUI can be found *here*

2.4.1 Mass calibration

The mass calibration GUI gives you a graphical user interface to calibrate your masses and apply the mass calibration. Assuming you have CRD file loaded into variable crd, you can bring up the GUI as following:

```python
from rimseval.guis import create_mass_cal_app
create_mass_cal_app(crd)
```

This will bring up a window similar to the following image.



Please right-click on a peak to begin mass calibration.

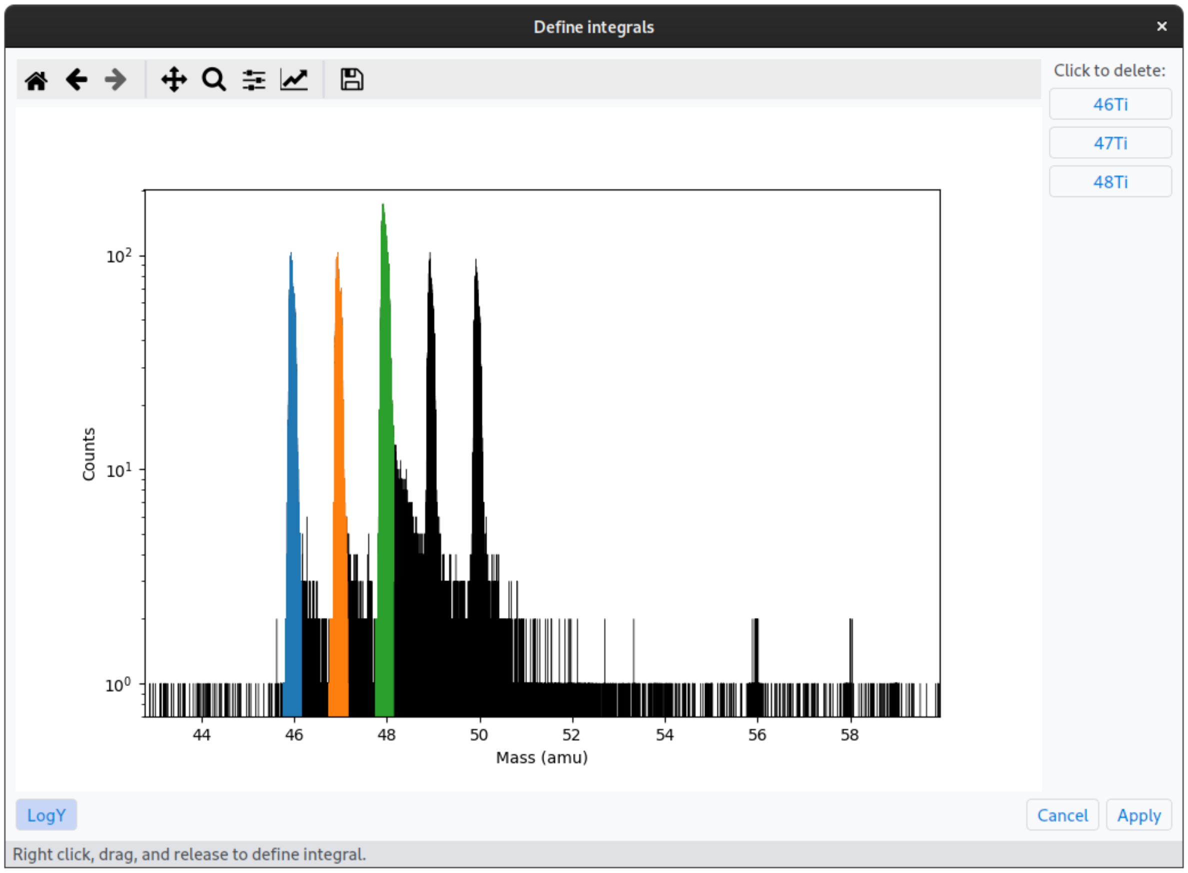
You can use the toolbar on top to zoom, etc. Use your mouse to right-click on a peak. This will bring up a window where you can either enter the mass of the peak or the isotope, e.g., as "46Ti", "Ti46", or "Ti-46". Your entry is not case sensitive. If the peak was found, you can move on to define a second, third, etc. peak. After the second peak, the top axis will turn into a mass calibration axis. Press the apply button will use this mass calibration and apply it to your CRD file.

2.4.2 Integral drawing

Integrals can be defined by drawing in a `matplotlib` GUI. Assuming you have a CRD file loaded into variable `crd`, you can bring up the GUI as following:

```python
from rimseval.guis import define_integrals_app
define_integrals_app(crd)
```

This will bring up a window similar to the following image. Note that several peaks have already been defined here.



Press your right mouse button and hold it down in where you want to start the definition. Then sweep over the peak and let the mouse button go when you want to stop defining the integral. A window will ask you for an integral name. Once you click okay, the defined area will be shaded in a color and the peak name will appear in the list on the right. To delete a peak, simply press its name in the list on the right.

Note: When you click apply, the integral definition will be written to the CRD file. However, you still need to separate calculate the integrals, since this is not done automatically.

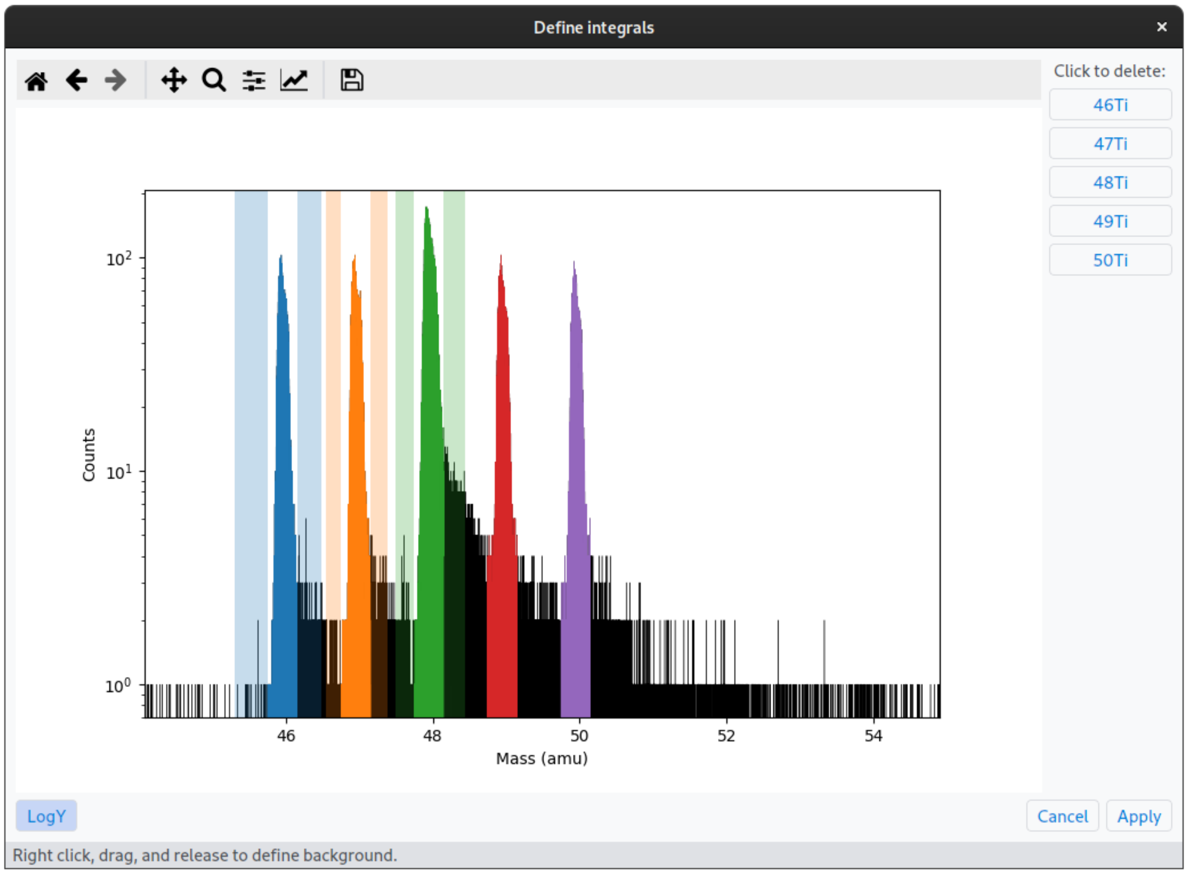
2.4.3 Background drawing

Similar to integrals, you can also define backgrounds using a `matplotlib` GUI. Assuming you have a CRD file loaded into variable `crd`, you can bring up the GUI as following:

```
from rimseval.guis import define_backgrounds_app
define_backgrounds_app(crd)
```

Warning: Integral limits for at least one peak must be defined before you can define backgrounds.

This will bring up a wiwndow similar to the following image. Note that several backgrounds have already been defined.



Right click, drag, and release to define background.

Press your right mouse button and hold it down where you want to start the background. Then sweep over the area you want to define as background and let the mouse button go when done. The GUI will bring up a window asking you for which peak this background applies.

Note: If you want to define a background to the left and right of the peak, you can simply do so by dragging the mouse over the peak. The GUI will automatically use your peak integral limits as the background limits and will define two backgrounds.

The background that is defined will then be shaded in the same color as the peak, but span the whole plot vertically and be lighter. If you want to delete all backgrounds for a peak, simply press the button on the right with the peak name.

2.5 Filters & Data Evaluation Tools

Here, we describe the individual filters that can be applied to the data. Note that these are not just filters in the strict sense, but also further data evaluation tools, e.g., deadtime correction. References to the individual API classes are given as well.

Note: If packages are defined, all regular filters will remove the shots from the existing packages as well. This means that you can end up with packages that contain less shots than what you defined when you created them!

2.5.1 Dead time correction

API Documentation: `rimseval.processor.CRDFileProcessor.dead_time_correction()`

This method applies a dead time correction to the whole spectrum. If packages were set, the dead time correction is performed on each package individually as well. The method takes the number of dead bins after the original bin.

The dead time correction is calculated as described in Stephan et al. (1994). In brief, be $N(i)$ the total ion count in channel i and n_s and total number of shots. We can then calculate a value $N'(i)$ as:

$$N'(i) = n_s - \sum_{i-k}^{i} N(i)$$

Here, k is the number of dead channels after an ion strikes the detector. For every channel and using the quantity $N'(i)$, we can then calculate the corrected number of counts $N_{\text{corr}}(i)$ as:

$$N_{\text{corr}}(i) = -n_s \cdot \log\left(1 - \frac{N(i)}{N'(i)}\right)$$

Note: The dead time correction should be run after all other filters have been applied to the spectrum!

2.5.2 Maximum ions per package

API Documentation: `rimseval.processor.CRDFileProcessor.filter_max_ions_per_pkg()`

This filter acts on packages. It takes one argument, namely the maximum number of ions that are allowed to be in a package. Any package with more ions will be removed from the evaluation process. This filter automatically updates the variable that holds the number of shots that are being evaluated. Furthermore, it also updates the mass spectrum data, which is subsequently used, e.g., to calculate integrals.

Note: If you run this filter multiple times, you must make sure that you are lowering the maximum number of allowed ions. Otherwise, please reset the data set before running a again.

See also:

- `rimseval.processor.CRDFileProcessor.packages()`
- `rimseval.processor.CRDFileProcessor.spectrum_full()`

2.5.3 Maximum ions per shot

API Documentation: `rimseval.processor.CRDFileProcessor.filter_max_ions_per_shot()`

This filter acts on individual. It takes one argument, namely the maximum number of ions that are allowed to be in a shot. Any shot with more ions will be removed from the evaluation process. This filter automatically updates the variable that holds the number of shots that are being evaluated. Furthermore, it also updates the mass spectrum data, which is subsequently used, e.g.,package to calculate integrals.

Note: If you run this filter multiple times, you must make sure that you are lowering the maximum number of allowed ions. Otherwise, please reset the data set before running a again.

Note: If you filter for maximum ions in packages and maximum ions per shot, the preferred way of achieving this is to filter the packages first. Otherwise, packages will be created with shots that are not in order of each other. While in practice, this will most likely not change your setup by a lot, in theory the preferred way should be "more correct".

See also:

- *rimseval.processor.CRDFileProcessor.spectrum_full()*

2.5.4 Filter maximum ions per time window

This filter sorts out shots that contain a more than a given amount of ions per time span. The time span can be anywhere in the ToF window. It takes two arguments: the maximum number of ions and the time window. The time window of interest must be given in microseconds.

See also:

- *rimseval.processor.CRDFileProcessor.filter_max_ions_per_time()*

2.5.5 Filter maximum ions per time of flight window

This filter sorts out shots that contain more than a given amount of ions in a defined time window. The shots will be filtered in the specified time window. Two parameters are required, the maximum number of ions per ToF window and the time window, given as `numpy.ndarray`.

See also:

- *rimseval.processor.CRDFileProcessor.filter_max_ions_per_tof_window()*

2.5.6 Package countrate filter based on Peirce's criterion

Warning: This filter is experimental. Running this filter more than once might lead to weird results!

This method filters packages based on Peirce's criterion, assuming that they should all contain the same number of counts. The user has to decide if such an assumptions makes sense or not.

More details on Peirce's criterion can be found on Wikipedia. The algorithm implemented here is after Ross (2013).

See also:

- *rimseval.processor.CRDFileProcessor.packages()*
- *rimseval.processor.CRDFileProcessor.spectrum_full()*

2.6 Variables

Here we describe the most important variables that are available for the user. These variables can be retrieved by calling them, however, are also of intereste for macro developers.

Throughout this chapter, we will use `crd` as an instance of the `CRDFileProcessor`, see also *`rimseval.processor.CRDFileProcessor()`*.

2.6.1 General

- `crd.crd`: The CRDReader instance of the opened file.
- `crd.fname`: File name of the opened CRD file.

2.6.2 Spectrum Data

- **`crd.data`: The mass spectrum data for an opened file.**
  This data must be a `numpy.ndarray` with the same length as `crd.tof` and `crd.mass`.
- **`crd.mass`: The mass axis of a given spectrum as a `numpy.ndarray`.**
  This array is only available after mass calibration and must be of the same length as `crd.data` and `crd.tof`.
- **`crd.nof_shots`: An integer that contains the number of used shots in the spectrum.**
  This variable is updated when shots are rejected.
- **`crd.tof`: Time of flight data of the spectrum.**
  This data is only available after opening after `crd.full_spectrum()` was run. It is a `numpy.ndarray` and must be of the same length as `crd.data` and `crd.mass`.

2.6.3 Package Data

- **`crd.data_pkg`: A data array for each individual package.**
  For a data length n and p packages, this `numpy.ndarray` is of size $n \times p$.
- **`crd.nof_shots_pkg`: The number of shots in the package.**
  The data are given as a `numpy.ndarray` of integers. It describes the number of shots in each individual package.

2.6.4 Integrals

- **`crd.integrals`: Sum of counts for each defined integral.**
  For each defined integral, this array will contain the sum of counts in the defined area and its uncertainty. m defined peaks, it will therefore be of size $m \times 2$. The second entry of each integral is the uncertainty of the counts.
- **`crd.integrals_delta`: δ-values for all integrals and uncertainties.**
  The format is the same as for `crd.integrals`.
- **`crd.integrals_pkg`: Integral data for packages.**
  For a total of p packages and m defined integrals, this `numpy.ndarray` will be of size $p \times m \times 2$. As for the integrals, the sum of counts for each peak and its uncertainty are given.

- **crd.integrals_delta_pkg: δ-values for all packages.**
  The format is the same as for crd.integrals_pkg.

2.6.5 Single Shot Data

In order to avoid storing the whole spectrum for every shot, the data is ordered in two arrays with one helper array.

- **crd.all_tofs: Recorded time of arrival bins in the TDC.**
  This numpy.ndarray contains all recorded time of flights in order of arrival. It is therefore of the same length as the total number of recorded ions. The time of flights are not stored in microseconds, but the TDC bin of arrival is stored as an integer.
- **crd.ions_per_shot: Number of ions recorded per shot.**
  This numpy.ndarray is of length self.nof_shots and records for the number of ions that arrived in each shot as an integer. The sum of this array is the total number of ions that are in the spectrum.
- **crd.ions_to_tof_map: Mapper array to find arrival bins for each shot.**
  This 2D numpy.ndarray contains one entry per shot and is therefore of the same length as crd.ions_per_shot. Each entry describes in which range of crd.all_tofs the arrival bins are stored. For example, if you want to retrieve the arrival bins of the ions of the 10th shot of your data set, you could acquire them by calling:

```
tof_range = crd.ions_per_shot[9]  # the first shot is index 0
arrival_bins = crd.all_tofs[tof_range[0]:tof_range[1]]
```

2.7 Special

The rimseval package has some special functions incorporated, which allow you to perform various analysis on your data and your RIMS setup.

2.7.1 Excel Workup file writer

Sometimes, it is useful to further process multiple evaluated spectra in an Excel file. You can use the *rimseval.data_io.excel_writer.workup_file_writer()* to create a workbook to continue working on your data. This workbook will contain integrals, δ-values.

Here is an example usage for a file that would contain zirconium data. Note that δ-values would usually be normalized to the most abundant ^{90}Zr, however, we would like to normalize here to ^{94}Zr. This is accomplished by setting the according normalization isotope of iniabu.

The following code assumes that crd is an instance of CRDFileProcessor and that a mass calibration has been performed and integrals have been set for zirconium isotopes.

```
from pathlib import Path

from rimseval.data_io import excel_writer
from rimseval.utilities import ini

# set Zr normalization isotope to Zr-94
ini.norm_isos = {"Zr": "Zr-94"}

# Write the excel file
```

(continues on next page)

(continued from previous page)

```
my_output_file = Path("workup_file.xlsx")
excel_writer.workup_file_writer(crd, my_output_file)
```

2.7.2 Hist. ions per shot

For appropriate dead time correction we assume that all counts are Poisson distributed. This means that the histogram of number of ions per shot should follow the following distribution:

$$f(k) = \exp(-\mu)\frac{\mu^k}{k!}$$

Here, k is the number of ions per shot, $f(k)$ is the frequency of ions in bin k, and μ is the number of ions divided by the number of shots.

There is a GUI function implemented that allows you to directly plot a histogram of ions per shot and compare it to the theoretical assumption. The following code shows you how to do this:

```
from pathlib import Path

from rimseval import CRDFileProcessor
from rimseval.guis import hist_nof_shots

my_file = Path("path/to/my_file.crd")
crd = CRDFileProcessor(crd)
crd.spectrum_full()

nof_ions_per_shot(crd)
```

This will open a `matplotlib` window and display the histogram.

2.7.3 Hist. time differences

To debug your system, i.e., to determine if the detector is ringing, it can be useful to determine the time difference between all ions in individual shots that have more than one ion arriving.

For every shot with more than one ion, we determine the time difference between these shots and create a histogram of all of these time differences. For a shot with n ions arriving, there will be $\frac{(n-1)n}{2}$ time differences determined.

Warning: This is different from the previous `LIONEval` software, where time differences were only determined between subsequent ions. Here, all ion time differences are taken into account now.

To calculate and display this plot, some example code is given below. Note that `max_ns=100` will set the upper limit of the x-axis to 100ns. This number is of course user-defined and can be omitted.

```
from pathlib import Path

from rimseval import CRDFileProcessor
from rimseval.guis import dt_ions

my_file = Path("path/to/my_file.crd")
```

(continues on next page)

(continued from previous page)

```
crd = CRDFileProcessor(crd)
crd.spectrum_full()

dt_ions(crd, max_ns=100)
```

2.7.4 Integrals per package

If you have split your spectrum into packages and have defined integrals, this routine allows you to show a figure of all integrals per package versus the number of the package. This is especially interesting to find bursts in your measurements, i.e., when measuring with the desorption laser.

The following example shows how the plot is generated:

```
from pathlib import Path

from rimseval import CRDFileProcessor
from rimseval.guis import integrals_packages

my_file = Path("path/to/my_file.crd")
crd = CRDFileProcessor(crd)
crd.spectrum_full()

integrals_packages(crd)
```

2.8 Verbosity

The package has the possibility to go into verbose mode. To do so, change the `rimseval.DEBUG`. This variable is by default set to `0`, which means no excessive warnings.

Setting the debug level to 1 will show the following warnings in addition:

- Automatic mass calibration optimization did not find enough peaks and was skipped.

Setting the debug level to 2 will show the following warnings in addition:

- Division by zero warnings in δ-value calculation
- `scipy` optimize routine (peak fitting in mass calibration) reached maximum iterations.

2.9 Installation

2.9.1 Executable

To install an executable file of the software on your computer, download the latest release version for your operating system from the GitHub repository. You do not need to have an existing python environment, since all dependencies will be installed along.

Warning: Installing a new version

If you already have a previous version of the software installed, you should first uninstall it before installing a new version. This is to ensure that all dependencies are correctly updated.

Note: Issues on macOS

Once you moved the *RIMSEvaluation* program from the *dmg* archive into your *Applications* folder, your macOS might still tell you that when opening the program that it is damaged and needs to be moved to the trash. This happens since the program is not officially signed with a valid Apple ID account. More information can be found in this issue. To still use the *RIMSEvaluation* executable, open a terminal and run the following command:

```
xattr -cr /Applications/RIMSEvaluation.app
```

This assumes that the program is in fact installed into the main installation folder. If not, replace `/Applications/` with the path to the correct installation folder. The *RIMSEvaluation* software should now run.

2.9.2 Anaconda

Preapring your Environment

If you work with Anaconda3, you should have the Anaconda Prompt installed on your system. Open the Anaconda Prompt. First, we want to set up a virtual environment to use for the `rimseval` GUI. In the Anaconda Prompt, type:

```
conda create -n rimseval python=3.9
conda activate rimseval
```

Next let's check if `git` is available. Type:

```
git version
```

You should see the version of `git` that you have installed. If you see an error saying that `git` was not found, install it from the Anaconda Prompt via:

```
conda install -c anaconda git
```

Note: If you do not have `git` installed system wide and install it according to the order above, it will only be installed in your virtual environment. This is fine, simply ensure that your virtual environment is activated in any of the steps below (which it should be anyway).

Installing RIMSEvalGUI

With the rimseval virtual environment activated, move to a folder where you want to put the RIMSEvalGUI source code. Then clone the GitHub repository by typing:

```
git clone https://github.com/RIMS-Code/RIMSEvalGUI.git
cd RIMSEvalGUI
```

The last command will enter the newly created folder. Then install the necessary requirements by typing:

```
pip install -r requirements.txt
```

If everything worked, you should be able to start the GUI by typing:

```
python RIMSEvalGUI.py
```

Running RIMSEvalGUI

If you start the Anaconda Prompt anew, you can run the program the next time by first moving to your installation folder. Then activate the virtual environment and run the python script. The following gives a summary of the steps to run the RIMSEvalGUI. Note that the path_to_folder should be replaced with the folder where the RIMSEvalGUI folder lies.

```
cd path_to_folder/RIMSEvalGUI
conda activate rimseval
python RIMSEvalGUI.py
```

The GUI should start. The Anaconda Prompt in the background will show you any warnings and errors that the program throws.

Updating your installation

Updating your installation, e.g., when a new version comes out, can be easily done with git. The steps to do so are as following form the Anaconda Prompt. We assume that you have already activated the rimseval virtual environment and changed directory into the RIMSEvalGUI folder on your computer (see above).

```
git pull
pip install -r requirements.txt --upgrade
```

Now you can start the new GUI as described above. Double check that the latest version is indeed displayed in the window title.

The above procedure gives you the latest development version. If you rather prefer the latest version that was officially released, check the releases here. Each release has a so-called tag associated with it, which is equal to the version number of the release. For example, to check out version v2.0.0 and not go to the latest development version, proceed as following:

```
git pull
git checkout tags/v2.0.0
pip install -r requirements.txt --upgrade
```

To switch back to the main branch / latest development version, you can simply type:

```
git checkout main
git pull
git install -r requirements.txt --upgrade
```

Note: Newer versions of the GUI can depend on development versions of `rimseval`. This means that you might see unexpected and wrong behavior. The packaging tool of the GUI does not allow for specifically labeling of such versions. Therefore, it is up to he user to ensure that you have the version that you like. New versions that depend on development versions of `rimseval` will always be labeled on GitHub as pre-releases. They will therefore not show up in the update reminder of the software.

2.9.3 Python

Note: If you are used to *git* and *python*, these instructions should work great for you. Otherwise, it might be recommendable that you install Anaconda and follow the instructions above.

To setup the RIMSEval GUI on regular python, make sure that you have Python 3.9 installed installed. Then create a virtual environment. Instructions can, e.g., found here.

After activating your new virtual environment, install the requirements by typing:

```
pip install -r requirements
```

The RIMSEval GUI can then be started by typing:

```
python RIMSEvalGUI.py
```

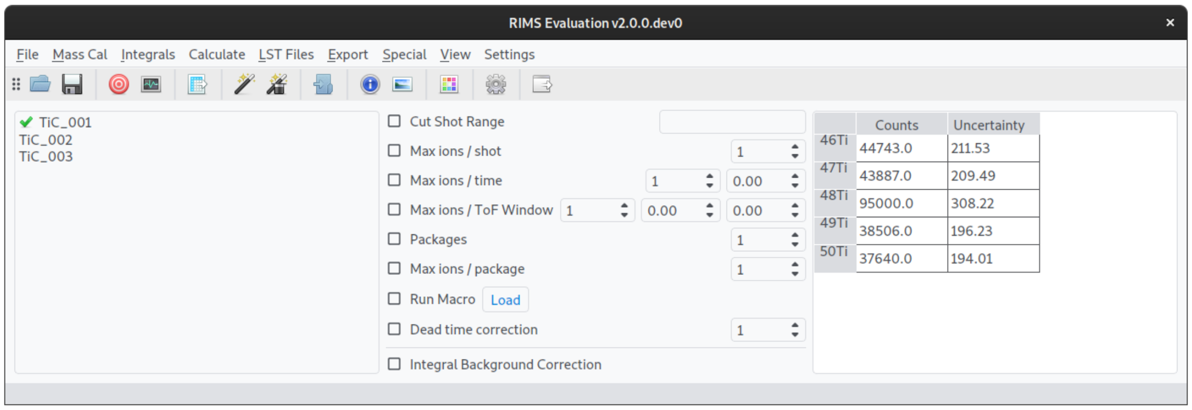
To update the RIMSEval GUI, refresh the folder from github and then upgrade the dependencies. From the shell you can accomplish this from within the RIMSEvalGUI folder, assuming you have initially cloned the folder from GitHub:

```
git pull
pip install -r requirements.txt --upgrade
```

2.10 Graphical User Interface

In this chapter, we briefly discuss the main functions of the graphical user interface (GUI). A more detailed description of the functionality that lies behind the GUI can be found in the *filters documentation*.

The main GUI should look something similar to this:


RIMS Evaluation v2.0.0.dev0
File Mass Cal Integrals Calculate LST Files Export Special View Settings
Load

✓ TiC_001
TiC_002
TiC_003

☐ Cut Shot Range	
☐ Max ions / shot	1
☐ Max ions / time	1, 0.00
☐ Max ions / ToF Window	1, 0.00, 0.00
☐ Packages	1
☐ Max ions / package	1
☐ Run Macro	
☐ Dead time correction	1
☐ Integral Background Correction	

	Counts	Uncertainty
46Ti	44743.0	211.53
47Ti	43887.0	209.49
48Ti	95000.0	308.22
49Ti	38506.0	196.23
50Ti	37640.0	194.01

Here, a file is already loaded and all buttons in the toolbar are active. The window title shows the current version of the `rimseval` package that is used in the GUI. The menu bar and tool bar below present all functions. If you browse through the menu bar you will see that many more functions are available than can be seen in the tool bar. In the toolbar, only the most frequently used functions are made available.

Below the tool bar, the GUI has three sections. On the left is the file dialog, which indicates what files are open. The file indicated with a green tick is the one that is currently active. Double-click one of the other names to activate that file. The center pane shows the possible filters that are available. Hovering over individual filters will show tool tips on what they do. Finally, the right side shows the integrals of defined peaks and the associated uncertainties.

At the bottom of the GUI is the status bar. Here, helpful information will be displayed when you hover over tool bar buttons.

All functions of the GUI contain tool tips, so hover with your mouse over buttons and dials, and some help should be displayed.

2.11 Macro Development

Here we describe how you can develop macros to run. In the GUI, the macros are run with the given interface. In the `CRDFileProcessor`, you can run your own macros using the `run_macro` routine: *`rimseval.processor.CRDFileProcessor.run_macro()`*

Templates for macros and example python files can be found on GitHub.

An overview of useful variables can be found in *here*.

2.11.1 Introduction

A macro requires a specific structure. If you need to import packages, please follow best norms.

Warning: Do not import full packages into the namespace. This could yield to odd behavior down the line.

For example, if you want to use `numpy` functions, you should import is as:

```
import numpy as np
```

If you want to import other routines from `rimseval`, use absolute imports.

Your macro can have as many subroutines as you like. These will all be imported by the macro reader. You must have a routine called `calc` that runs your calculations. If you have more subroutines, these need to be called from within your main calculation function. The following template lines out the start of a macro. If you use an editor that supports type hints, autocompletion will work. Information on type hints can be found here. Note the usage of absolute imports to enable type hints.

```python
from rimseval.processor import CRDFileProcessor

def calc(crd: CRDFileProcessor) -> None:
    """Macro template.

    :param crd: CRD file processor that will be passed to the macro.
    """
    # your code goes here
```

2.11.2 Basic examples

Below are two examples that show you how to implement, e.g., how to filter on full data sets and how to filter packages.

Example: Filter maximum ions per shot

Here is a re-implementation of the *rimseval.processor.CRDFileProcessor.filter_max_ions_per_shot()* method in the form of a macro.

```python
import numpy as np

from rimseval.processor import CRDFileProcessor


def calc(crd: CRDFileProcessor) -> None:
    """Macro to filter out all shots with more than 3 ions.

    :param crd: CRD file processor that will be passed to the macro.
    """
    # maximum ions per shot to be filtered out
    max_ions = 3

    # create a numpy mask that selects the filters we want to filter
    shots_rejected = np.where(crd.ions_per_shot > max_ions)[0]

    # pass the rejected shots array to the routine that will filter everything
    crd.apply_individual_shots_filter(shots_rejected)
```

As discussed above, the macro must be in the given template form. We define here a `shots_rejected` array (a `numpy.ndarray`), which lists the shots that we want to reject. Finally, we pass the array of rejected shots to the in-built function that processes the shots further and handles everything down the line. This is the `crd.apply_individual_shots_filter()` routine. More detailed documentation on this routine can be found in *rimseval.processor.CRDFileProcessor.apply_individual_shots_filter()*.

Example: Filter maximum ions per package

Here is a macro that re-implements the maximum number of ions per package filter. Details on the filter can be found in *rimseval.processor.CRDFileProcessor.filter_max_ions_per_shot()*.

```python
import numpy as np

from rimseval.processor import CRDFileProcessor


def calc(crd: CRDFileProcessor) -> None:
    """Macro to filter out all packages with more than 4 ions.

    Here we assume, that you have already created packages using the built-in function.
    Package data is therefore available.

    :param crd: CRD file processor that will be passed to the macro.
    """
    # maximum ions per pkg to be filtered out
    max_ions = 4

    # calculate total ions in all the packages
    total_ions_per_pkg = np.sum(crd.data_pkg, axis=1)

    # find packages that need deletion
    pkg_to_delete = np.where(total_ions_per_pkg > max_ions)[0]

    # now delete the packages that are not needed
    crd.data_pkg = np.delete(crd.data_pkg, pkg_to_delete, axis=0)

    # update the number of shots per package array: delete entries of deleted packages
    crd.nof_shots_pkg = np.delete(crd.nof_shots_pkg, pkg_to_delete, axis=0)

    # finally, we need to update the ``crd.data`` and ``crd.nof_shots`` entries
    crd.data = np.sum(crd.data_pkg, axis=0)
    crd.nof_shots = np.sum(crd.nof_shots_pkg)
```

First we calculate the sum of ions per package by summing over the `crd.data_pkg` array. Note that we need to specify the axis in this case since we want to sum over packages, not over the spectra. Using the `numpy.where` function, we then locate packages that meet the rejection criterion. The `pkg_to_delete` variable stores the index of the packages. Using `numpy.delete`, we then delete the rejectable packages from `crd.data_pkg` and `crd.nof_shots_pkg`. Finally, we need to update `crd.data` and `crd.nof_shots` such that the non-packaged data is updated as well.

Note: Before the macro runs, data must have been packaged using the built-in method.

2.12 Delta Values

For all integrals, δ-values can be calculated automatically as described in the *usage documentation*. Delta values are always calculated (1) with respect to the major isotope and (2) using the NIST database of the iniabu package. If no value can be calculated, i.e., because the isotope is unknown to `iniabu` or because the natural abundance of the specified isotope is unknown, `np.nan` will be returned as the δ-value.

Note: All δ-values are reported in per mil (‰).

2.12.1 Calculation δ-value

Be i and j the number of counts in the nominator and denominator isotopes, respectively, and r the NIST isotopic ratio of the same isotopes. The δ-value can then be calculated as:

$$\delta = \left(\frac{i/j}{r} - 1 \right) \times 1000 \qquad ()$$

2.12.2 Uncertainties

Be σ_i and σ_j the uncertainties of values i and j, respectively. The uncertainty of the δ-value σ_δ can be calculated as:

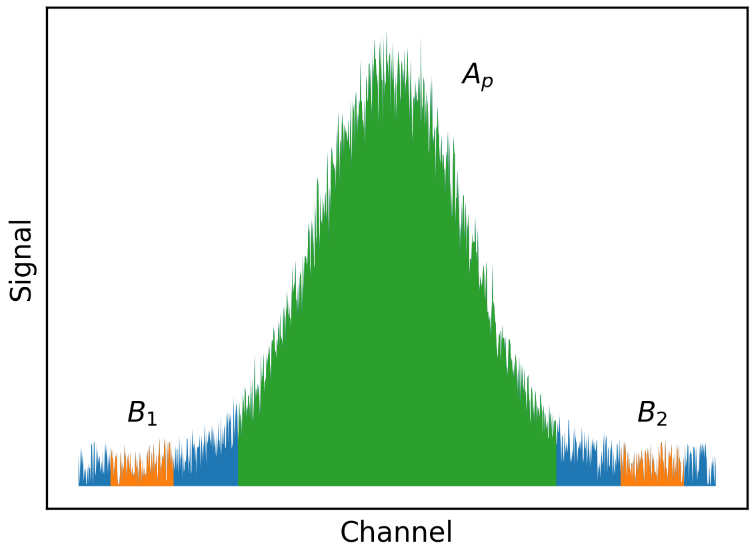
$$\sigma_\delta = \frac{1000}{r} \left[\left(\frac{\sigma_i}{j} \right)^2 + \left(\frac{i\sigma_j}{j^2} \right)^2 \right]^{1/2} \qquad ()$$

2.13 Integrals

This page explains the strategy and mathematical derivations for spectra processing. It discusses the thinking behind routines, and not the routines and algorithms used themselves.

2.13.1 Background correction

Below figure shows a peak in green and two backgrounds left and right surrounding the peak.



Let us define A_p as the peak area (green) and B_1 and B_2 as two backgrounds, left and right, respectively. Furthermore, let us generalize the backgrounds as B_i, where i represents any background. In above example, $i \in \{1, 2\}$. Furthermore, let's define the number of channels in the peak as n_p and the channels in background i as n_i.

Without background corrections and using counting statistics, we can write the total area under the peak and its uncertainty as $A_p \pm \sqrt{A_p}$. With background correction, let us define the peak area as A_corr and its uncertainty as σ_{A_corr}.

To perform a background subtraction, we first have to define the channel normalized background, i.e., the amount of background that is present per channel. This can be written as:

$$B_{\text{ch},i} = \frac{B_i}{n_i}$$

From this, we can derive the average background by averaging over all i, such that:

$$B_\text{ch} = \frac{1}{N} \sum_{i=1}^{N} \frac{B_i}{n_i}$$

Its uncertainty, based on counting statistics, is defined as:

$$\sigma_{B_\text{ch}} = \frac{1}{N} \sum_{i=1}^{N} \frac{\sqrt{B_i}}{n_i}$$

Here, we assume that the number of channels is known without any uncertainty. This is generally true when dealing with mass spectra.

From this formalism, we can now calculate the background corrected peak as:

$$\begin{aligned} A_\text{corr} &= A_p - n_p \cdot B_\text{ch} \\ &= A_p - \frac{n_p}{N} \sum_{i=1}^{N} \frac{B_i}{n_i} \end{aligned}$$

Propagating the uncertainties, we can calculate the uncertainty of the background corrected peak as:

$$\sigma_{A_\text{corr}} = \left[\left(\sqrt{A_p} \right)^2 + \left(\frac{n_p}{N} \sum_{i=1}^{N} \frac{\sqrt{B_i}}{n_i} \right)^2 \right]^{1/2}$$

2.14 Development Guide

This developers guide briefly goes through setting up the project. It is not intended as a full guide for development of this project. Most of the guidelines are copied from the developer's guide of the `iniabu` project, which can be found here.

2.14.1 Testing

Full testing, linting, etc. is built-in using `nox`. To make it work, ensure that `nox` is installed by running:

```
pip install nox
```

Then you can invoke nox by simply calling it within the project folder via:

```
nox
```

You can also set up your IDE to run any of the tests. Required dependencies if you prefer not testing with `nox` can be found in the `requirements-dev.txt` file.

2.14.2 Pre-commit

Pre-commit will format code according to specs prior to committing it to GitHub. To install the pre-commit hooks, go to the code folder and run the following command (after installing pre-commit using pip or pipx):

$ pre-commit install

This will install the hooks that are defined in *.pre-commit-config.yaml*.

2.14.3 Test Coverage

Coveralls and pytest-cov is used to automatically determine the test coverage.

Note: If you are using PyCharm, you should set up your testing environment such that it contains in the *Additional Arguments* section the flag `--no-cov`. Otherwise, coverage reporting in PyCharm will not work.

Warning: Code that contains the `@njit` or any numba JIT decorator is currently excluded from coverage using the `# pragma: nocover`. Make sure you include useful tests for these routines nevertheless! However, due to existing issues, these jited methods would not show up as covered.

2.14.4 Documentation

This documentation is written in reText and automatically generated using `sphinx`. If you would like to run it locally, install the `requirements-dev.txt` packages. From within the `docs` folder, you can then generate the documentation via:

```
sphinx-build -b html docs docs/_build/html/
```

This creates the the *html* documentation in the `docs/_build` folder.

2.15 API Reference

Some words on API reference.

Contents:

2.15.1 RIMSEval Processor and Utilities

`CRDFileProcessor`: Processor class

class `rimseval.processor.`**`CRDFileProcessor`**(*fname: Path*)

Process a CRD file in this class, dead time corrections, etc.

Computationally expensive routines are sourced out into processor_utils.py for jitting.

Example:

```
>>> my_file = Path("crd_files/my_file.crd")
>>> crd = CRDFileProcessor(my_file)  # open the file
>>> crd.spectrum_full()  # create a spectrum
```

`adjust_overlap_background_peaks`(*other_peaks: bool = False*) → None

Routine to adjust overlaps of backgrounds and peaks.

By default, this routine checks if the backgrounds overlap with the peaks they are defined for and removes any background values that interfer with the peak that is now defined. It also checks for overlap with other peaks and if it finds any, warns the user. If *other_peaks* is set to *True*, the routine will not warn the user, but automatically correct these bad overlaps.

Parameters
`other_peaks` – Automatically correct for overlap with other peaks?

Returns
None

`apply_individual_shots_filter`(*shots_rejected: ndarray*)

Routine to finish filtering for individual shots.

This will end up setting all the data. All routines that filter shots only have to provide a list of rejected shots. This routine does the rest, including. the handling of the data if packages exist.

ToDo: rejected shots should be stored somewhere.

Parameters
`shots_rejected` – Indices of rejected shots.

calculate_applied_filters()

Check for which filters are available and then recalculate all from start.

dead_time_correction(*dbins: int*) → None

Perform a dead time correction on the whole spectrum.

If packages were set, the dead time correction is performed on each package individually as well. :param dbins: Number of dead bins after original bin (total - 1).

Warning.warn
There are no shots left in the package. No deadtime correction can be applied.

property def_backgrounds: Tuple[List[str], ndarray]

Background definitions for integrals.

The definitions consist of a tuple of a list and a np.ndarray. The list contains first the names of the integrals. The np.ndarray then contains in each row the lower and upper limit in amu of the peak that needs to be integrated.

Note: The format for defining backgrounds is the same as the format for defining integrals, except that peaks can occur multiple times for multiple backgrounds.

Returns
Background definitions.

Raises
ValueError – Data Shape is wrong

Example:

```
>>> data = CRDFileProcessor("my_data.crd")
>>> peak_names = ["54Fe", "54Fe"]
>>> peak_limits = np.array([[53.4, 53.6], [54.4, 54.6]])
>>> data.def_integrals = (peak_names, peak_limits)
```

property def_integrals: Tuple[List[str], ndarray]

Integral definitions.

The definitions consist of a tuple of a list and a np.ndarray. The list contains first the names of the integrals. The np.ndarray then contains in each row the lower and upper limit in amu of the peak that needs to be integrated. If backgrounds overlap with the peaks themselves, they will be automatically adjusted.

Returns
Integral definitions.

Raises

- **ValueError** – Data Shape is wrong
- **ValueError** – More than one definition exist for a given peak.

Example:

```
>>> data = CRDFileProcessor("my_data.crd")
>>> peak_names = ["54Fe", "64Ni"]
>>> peak_limits = np.array([[53.8, 54.2], [63.5, 64.5]])
>>> data.def_integrals = (peak_names, peak_limits)
```

property def_mcal: ndarray

Mass calibration definitions.

Returns

Mass calibration definitions. The columns are as following: 1st: ToF (us) 2nd: Mass (amu)

Raises

- `TypeError` – Value is not a numpy ndarray
- `ValueError` – At least two parameters must be given for a mass calibration.
- `ValueError` – The array is of the wrong shape.

filter_max_ions_per_pkg(*max_ions: int*) → None

Filter out packages with too many ions.

Note: Only run more than once if filtering out more. Otherwise, you need to reset the dataset first.

Parameters

`max_ions` – Maximum number of ions per package.

Raises

- `ValueError` – Invalid range for number of ions.
- `OSError` – No package data available.

filter_max_ions_per_shot(*max_ions: int*) → None

Filter out shots that have more than the max_ions defined.

Note: Only run more than once if filtering out more. Otherwise, you need to reset the dataset first.

Parameters

`max_ions` – Maximum number of ions allowed in a shot.

Raises

`ValueError` – Invalid range for number of ions.

filter_max_ions_per_time(*max_ions: int*, *time_us: float*) → None

Filter shots with >= max ions per time, i.e., due to ringing.

Parameters

- `max_ions` – Maximum number of ions that is allowed within a time window.
- `time_us` – Width of the time window in microseconds (us)

filter_max_ions_per_tof_window(*max_ions: int*, *tof_window: ndarray*) → None

Filer out maximum number of ions in a given ToF time window.

Parameters

- `max_ions` – Maximum number of ions in the time window.
- `tof_window` – The time of flight window that the ions would have to be in. Array of start and stop time of flight (2 entries).

Raises

`ValueError` – Length of *tof_window* is wrong.

filter_pkg_peirce_countrate() → None

Filter out packages based on Peirce criterion for total count rate.

Fixme: This needs more thinking and testing Now we are going to directly use all the integrals to get the sum of the counts, which we will then feed to the rejection routine. Maybe this can detect blasts.

Warning: Running this more than once might lead to weird results. You have been warned!

integrals_calc(*bg_corr=True*) → None

Calculate integrals for data and packages (if present).

The integrals to be set per peak are going to be set as an ndarray. Each row will contain one entry in the first column and its associated uncertainty in the second.

Parameters

bg_corr – If false, will never do background correction. Otherwise (default), background correction will be applied if available. This is a toggle to switch usage while leaving backgrounds defined.

Raises

- `ValueError` – No integrals were set.
- `ValueError` – No mass calibration has been applied.

integrals_calc_delta() → None

Calculate delta values for integrals and save them in class.

This routine uses the `iniabu` package to calculate delta values for defined integrals. It reads the peak names and calculates delta values for isotopes that can be understood `iniabu`, and calculates the delta values with respect to the major isotope. These values are then saved to the class as `integrals_delta` and `integrals_delta_pkg`, if packages were defined. Uncertainties are propagated according to Gaussian error propagation. The format of the resulting arrays are identical to the `integrals` and `integrals_pkg` arrays.

Raises

`ValueError` – No integrals were calculated.

property integrals_overlap: bool

Check if any of the integrals overlap.

Returns

Do any integrals overlap?

Example:

```
>>> data = CRDFileProcessor("my_data.crd")
>>> peak_names = ["54Fe", "64Ni"]
>>> peak_limits = np.array([[53.8, 54.2], [63.5, 64.5]])
>>> data.def_integrals = (peak_names, peak_limits)
>>> data.integrals_overlap
False
```

mass_calibration() → None

Perform a mass calibration on the data.

Let m be the mass and t the respective time of flight. We can then write:

$$t \propto \sqrt[a]{m}$$

Usually it is assumed that $a=2$, i.e., that the square root is taken. We don't have to assume this though. In the generalized form we can now linearize the mass calibration such that:

$$\log(m) = a \log(t) + b$$

Here, a is, as above, the exponent, and b is a second constant. With two values or more for m and t, we can then make a linear approximation for the mass calibration $m(t)$.

Raises

`ValueError` – No mass calibration set.

property name

Get the name of the CRD file.

optimize_mcal(*offset: float = None*) → None

Take an existing mass calibration and finds maxima within a FWHM.

This will act on small corrections for drifts in peaks.

Parameters

`offset` – How far do you think the peak has wandered? If None, it will be set to the FWHM value.

packages(*shots: int*) → None

Break data into packages.

Parameters

`shots` – Number of shots per package. The last package will have the rest.

Raises

`ValueError` – Number of shots out of range

property peak_fwhm: float

Get / Set the FWHM of the peak.

Returns

FWHM of the peak in us.

run_macro(*fname: Path*) → None

Run your own macro.

The macro will be imported here and then run. Details on how to write a macro can be found in the documentation.

Parameters

`fname` – Filename to the macro.

sort_backgrounds() → None

Sort all the backgrounds that are defined.

Takes the backgrounds and the names and sorts them by proton number (first order), then by mass (second order), and finally by start of the background (third order). All backgrounds that cannot be identified with a clear proton number are sorted in at the end of the second order sorting, and then sorted by starting mass. If no backgrounds are defined, this routine does nothing.

Example:

```
>>> crd.def_backgrounds
["56Fe", "54Fe"], array([[55.4, 55.6], [53.4, 53.6]])
>>> crd.sort_backgrounds()
>>> crd.def_backgrounds
["54Fe", "56Fe"], array([[53.4, 53.6], [55.4, 55.6]])
```

sort_integrals(*sort_vals: bool = True*) → None

Sort all the integrals that are defined by mass.

Takes the integrals and the names and sorts them by proton number (first order), then by mass (second order). All integrals that cannot be identified with a clear proton number (e.g., molecules) are sorted in at the end of the primary sorting, then sorted by mass. The starting mass of each integral is used for sorting. If no integrals are defined, this routine does nothing.

Parameters
sort_vals – Sort the integrals and integral packages? Default: True

Example:

```
>>> crd.def_integrals
["Fe-56", "Ti-46"], array([[55.8, 56.2], [45.8, 46.2]])
>>> crd.sort_integrals()
>>> crd.def_integrals
["Ti-46", "Fe-56"], array([[45.8, 46.2], [55.8, 56.2]])
```

spectrum_full() → None

Create ToF and summed ion count array for the full spectrum.

The full spectrum is transfered to ToF and ion counts. The spectrum is then saved to: - ToF array is written to *self.tof* - Data array is written to *self.data*

Warnings
Time of Flight and data have different shape

spectrum_part(*rng: Tuple[Any] | List[Any]*) → None

Create ToF for a part of the spectra.

Select part of the shot range. These ranges will be 1 indexed! Always start with the full data range.

Parameters
rng – Shot range, either as a tuple (from, to) or as a tuple of multiple ((from1, to1), (from2, to2), ...).

Raises

- **ValueError** – Ranges are not defined from, to where from < to
- **ValueError** – Tuples are not mutually exclusive.

- **IndexError** – One or more indexes are out of range.

property timestamp: datetime

Get the time stamp when the recording was started.

Returns
Timestamp of the CRD file.

Example:

```
>>> crd = CRDFileProcessor(Path("my_file.crd"))
>>> crd.timestamp
datetime.datetime(2021, 7, 10, 11, 41, 13)
```

property us_to_chan: float

Conversion factor for microseconds to channel / bin number.

Returns
Conversion factor

2.15.2 Processor Utilities

Utility functions for the CRD file processor `CRDFileProcessor`. Many of the mathematically heavy routines are outsourced here for JITing with numba.

create_packages()

rimseval.processor_utils.**create_packages**(*shots: int*, *tofs_mapper: ndarray*, *all_tofs: ndarray*) → Tuple[ndarray, ndarray]

Create packages from data.

Parameters

- **shots** – Number of shots per package
- **tofs_mapper** – mapper for ions_per_shot to tofs
- **all_tofs** – all arrival times / bins of ions

Returns
Data array where each row is a full spectrum, each line a package and a shot array on how many shots are there per pkg

dead_time_correction()

rimseval.processor_utils.**dead_time_correction**(*data: ndarray*, *nof_shots: ndarray*, *dbins: int*) → ndarray

Calculate dead time for a given spectrum.

Parameters

- **data** – Data array, histogram in bins. 2D array (even for 1D data!)
- **nof_shots** – Number of shots, 1D array of data
- **dbins** – Number of dead bins after original bin (total - 1).

Returns
Dead time corrected data array.

gaussian_fit_get_max()

`rimseval.processor_utils.gaussian_fit_get_max`(*xdata: ndarray*, *ydata: ndarray*) → float

Fit a Gaussian to xdata and ydata and return the xvalue of the peak.

Parameters

- **xdata** – X-axis data
- **ydata** – Y-axis data

Returns
Maximum mof the peak on the x-axis

integrals_bg_corr()

`rimseval.processor_utils.integrals_bg_corr`(*integrals: ndarray*, *int_names: ndarray*, *int_ch: ndarray*, *bgs: ndarray*, *bgs_names: ndarray*, *bgs_ch: ndarray*, *int_pkg: ndarray = None*, *bgs_pkg: ndarray = None*) → Tuple[ndarray, ndarray]

Calculate background correction for integrals with given backgrounds.

This takes the integrals that already exist and updates them by subtracting the backgrounds. Multiple backgrounds per integral can be defined. Important is that the names of the backgrounds are equal to the names of the integrals that they need to be subtracted from and that the names of the integrals are unique. The latter point is tested when defining the integrals.

Note: This routine currently cannot be jitted since we are using an `np.where` statement. If required for speed, we can go an replace that statement. Most likely, this is plenty fast enough though.

Parameters

- **integrals** – Integrals and uncertianties for all defined peaks.
- **int_names** – Name of the individual peaks. Must be unique values!
- **int_ch** – Number of channels for the whole peak width.
- **bgs** – Backgrounds and their uncertianties for all defined backgrounds.
- **bgs_names** – Peaks each backgrounds go with, can be multiple.
- **bgs_ch** – Number of channels for background width.
- **int_pkg** – Packaged integrals, if exist: otherwise provide `None`
- **bgs_pkg** – Packaged backgrounds, if exist: otherwise provide `None`

Returns
Corrected data and data_packages.

integrals_summing()

`rimseval.processor_utils.`**`integrals_summing`**(*data: ndarray*, *windows: Tuple[ndarray]*, *data_pkg: ndarray = None*) → Tuple[ndarray, ndarray]

Sum up the integrals within the defined windows and return them.

Parameters

- **data** – Data to be summed over.
- **windows** – The windows to be investigated (using numpy views)
- **data_pkg** – Package data (optional), if present.

Returns

integrals for data, integrals for data_pkg

mask_filter_max_ions_per_time()

`rimseval.processor_utils.`**`mask_filter_max_ions_per_time`**(*ions_per_shot: array*, *tofs: array*, *max_ions: int*, *time_chan: int*) → array

Return indices where more than wanted shots are in a time window.

Parameters

- **ions_per_shot** – How many ions are there per shot? Also defines the shape of the return array.
- **tofs** – All ToFs. Must be of length ions_per_shot.sum().
- **max_ions** – Maximum number of ions that are allowed in channel window.
- **time_chan** – Width of the window in channels (bins).

Returns

Boolean array of shape like ions_per_shot if more are in or not.

mask_filter_max_ions_per_tof_window()

`rimseval.processor_utils.`**`mask_filter_max_ions_per_tof_window`**(*ions_per_shot: array*, *tofs: array*, *max_ions: int*, *tof_window: array*) → array

Return indices where more than wanted shots are in a given ToF window.

Parameters

- **ions_per_shot** – How many ions are there per shot? Also defines the shape of the return array.
- **tofs** – All ToFs. Must be of length ions_per_shot.sum().
- **max_ions** – Maximum number of ions that are allowed in channel window.
- **tof_window** – Start and stop time of the ToF window in channel numbers.

Returns

Boolean array of shape like ions_per_shot if more are in or not.

`mass_calibration()`

`rimseval.processor_utils.mass_calibration`(*params: array*, *tof: array*, *return_params: bool = False*) → array | Tuple[array]

Perform the mass calibration.

Parameters

- **params** – Parameters for mass calibration.
- **tof** – Array with all the ToFs that need a mass equivalent.
- **return_params** – Return parameters as well? Defaults to False

Returns
Mass for given ToF.

`mass_to_tof()`

`rimseval.processor_utils.mass_to_tof`(*m: ndarray | float*, *tm0: float*, *const: float*) → ndarray | float

Functional prescription to turn mass into ToF.

Returns the ToF with the defined functional description for a mass calibration. Two parameters are required. The equation, with parameters defined as below, is as following:

$$t = \sqrt{m} \cdot \mathrm{const} + t_0$$

Parameters

- **m** – mass
- **tm0** – parameter 1
- **const** – parameter 2

Returns
time

`multi_range_indexes()`

`rimseval.processor_utils.multi_range_indexes`(*rng: array*) → array

Create multi range indexes.

If a range is given as (from, to), the from will be included, while the to will be excluded.

Parameters
rng – Range, given as a numpy array of two entries each.

Returns
A 1D array with all the indexes spelled out. This allows for viewing numpy arrays for multiple windows.

`remove_shots_from_filtered_packages_ind()`

`rimseval.processor_utils.`**`remove_shots_from_filtered_packages_ind`**`(`*shots_rejected: array*, *len_indexes: int*, *filtered_pkg_ind: array*, *pkg_size: int*`)` → Tuple[array, array]

Remove packages that were already filtered pkg from ion filter indexes.

This routine is used to filter indexes in case a package filter has been applied, and now an ion / shot based filter needs to be applied.

Parameters

- **`shots_rejected`** – Array of indexes with rejected shots.
- **`len_indexes`** – length of the indexes that the rejected shots are from.
- **`pkg_size`** – Size of the packages that were created.
- **`filtered_pkg_ind`** – Array with indexes of packages that have been filtered.

Returns

List of two Arrays with shots_indexes and shots_rejected, but filtered.

`remove_shots_from_packages()`

`rimseval.processor_utils.`**`remove_shots_from_packages`**`(`*pkg_size: int*, *shots_rejected: array*, *ions_to_tof_map: array*, *all_tofs: array*, *data_pkg: array*, *nof_shots_pkg: array*, *pkg_filtered_ind: array = None*`)` → Tuple[array, array]

Remove shots from packages.

This routine can take a list of individual ions and remove them from fully packaged data. In addition, it can also take a list of packages that, with respect to the raw data, have previously been removed. This is useful in order to filter individual shots from packages after packages themselves have been filtered.

Parameters

- **`pkg_size`** – How many shots were grouped into a package originally?
- **`shots_rejected`** – Index array of the rejected shots.
- **`ions_to_tof_map`** – Mapping array where ions are in all_tof array.
- **`all_tofs`** – Array containing all the ToFs.
- **`data_pkg`** – Original data_pkg before filtering.
- **`nof_shots_pkg`** – Original nof_shots_pkg before filtering.
- **`pkg_filtered_ind`** – Indexes where the filtered packages are.

Returns

Filtered data_pkg and nof_shots_pkg arrays.

sort_data_into_spectrum()

`rimseval.processor_utils.`**`sort_data_into_spectrum`**(*ions: ndarray*, *bin_start: int*, *bin_end: int*) → ndarray

Sort ion data in 1D array into an overall array and sum them up.

Parameters

- **`ions`** – Arrival time of the ions - number of time bin
- **`bin_start`** – First bin of spectrum
- **`bin_end`** – Last bin of spectrum

Returns

arrival bins summed up

tof_to_mass()

`rimseval.processor_utils.`**`tof_to_mass`**(*tm: ndarray | float*, *tm0: float*, *const: float*) → ndarray | float

Functional prescription to turn ToF into mass.

Returns the mass with the defined functional description for a mass calibration. Two parameters are required. The equation, with parameters defined as below, is as following:

$$m = \left(\frac{tm - tm_0}{\mathrm{const}}\right)^2$$

Parameters

- **`tm`** – time or channel
- **`tm0`** – parameter 1
- **`const`** – parameter 2

Returns

mass m

2.15.3 Interfacer

Functions to talk to settings, calibratinos, GUIs, etc.

read_lion_eval_calfile()

`rimseval.interfacer.`**`read_lion_eval_calfile`**(*crd:* CRDFileProcessor, *fname: Path = None*) → None

Read a LIONEval calibration file and set it to instance of crd.

LIONEval is the first, Python2.7 version of the data evaluation software. This routine takes an old calibration file if requested by the user and sets the mass calibration, integrals, and background correction information if present.

Parameters

- **`crd`** – Instance of the CRDFileProcessor, since we need to set properties
- **`fname`** – Filename to mass calibration file. If *None*, try the same file name as for the CRD file, but with *.cal* as an extension.

Raises
`OSError` – Calibration file does not exist.

`load_cal_file()`

`rimseval.interfacer.`**`load_cal_file`**`(`*crd: CRDFileProcessor*, *fname: Path = None*`)` → None

Load a calibration file from a specific path / name.

Parameters

- **crd** – CRD Processor class to load into
- **fname** – Filename and path. If *None*, try file with same name as CRD file but *.json* suffix.

Raises

- `OSError` – Calibration file does not exist.
- `OSError` – JSON file cannot be decoded. JSON error message is returned too.

`save_cal_file()`

`rimseval.interfacer.`**`save_cal_file`**`(`*crd: CRDFileProcessor*, *fname: Path = None*`)` → None

Save a calibration file to a specific path / name.

Note: The new calibration files are *.json* files and not *.cal* files.

Parameters

- **crd** – CRD class instance to read all the data from.
- **fname** – Filename to save to to. If None, will save in folder / name of original crd file name, but with '.cal' ending.

2.15.4 Data Input and Output (I/O)

The routines in this folder deal with data input and output. Transformers for converting list files to *CRD* files can also be found here. Also export functions are found here.

`CRDReader`

class `rimseval.data_io.crd_reader.`**`CRDReader`**`(`*fname: Path*`)`

Read CRD Files and make the data available.

Example:

```
>>> fname = Path("folder/my_crd_file.crd")
>>> crd_file = CRDReader(fname)
>>> crd_file.nof_ions
13281
>>> crd_file.nof_shots
5000
```

`property all_data: Tuple[ndarray, ndarray]`

Get the data.

Returns
1D array with ions per shot, 1D array with bin in which ions arrived

`property all_tofs: ndarray`

Get all time of flight bins.

Returns
1D array with bin in which ions arrived

`property ions_per_shot: ndarray`

Get ions per shot array.

Returns
1D array with ions per shot

`property ions_to_tof_map: ndarray`

Get the index mapper for ions_per_shot -> tof.

Returns
Mapper with indexes where tofs are in all_tofs

`property nof_ions: int`

Get the number of shots.

Returns
Number of shots.

`property nof_shots: int`

Get the number of ions.

Returns
Number of ions.

`parse_data`(*data: bytes*) → None

Parse the actual data out and put into the appropriate array.

For this parsing to work, everything has to be just right, i.e., the number of shots have to be exactly defined and the data should have the right length. If not, this needs to throw a warning and move on to parse in a slower way.

Parameters
`data` – Binary string of all the data according to CRD specification.

Warning
Number of Shots do not agree with the number of shots in the list or certain ions are outside the binRange. Fallback to slower reading routine.

Warning
There is more data in this file than indicated by the number of Shots.

`parse_data_fallback`(*data: bytes*) → None

Slow reading routine in case the CRD file is corrupt.

Here we don't assume anything and just try to read the data into lists and append them. Sure, this is going to be slow, but better than no data at all.

Parameters
`data` – Array of all the data.

`read_data()` → None

Read in the data and parse out the header.

The header information will be stored in the header dictionary. All entry names are as specified in the CRD format file for version 1.0.

Raises

- `KeyError` – Header is not available.
- `OSError` – Corrupt data length.

CRD File Utilities

Header definitions, enums, and JIT compiled functions for processing *CRD* files.

Definitions

`property` `rimseval.data_io.crd_utils.`**`CURRENT_DEFAULTS Current default CRD header information.`**

`property` `rimseval.data_io.crd_utils.`**`HEADER_START Defaults sizes for the header values to read in.`**

`CRDHeader`

`class` `rimseval.data_io.crd_utils.`**`CRDHeader`**(*value*, *names=None*, *, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Enum class for CRD header.

The start must always be the same as HEADER_START above, however, the other fields might vary depending on the header that is being used. The header is called by its version number. Note that the letter *v* precedes the number and that the period is replaced with the letter *p*.

Format is always: Name, length, struct unpack qualifier

`v1p0 = (('shotPattern', 4, '<I'), ('tofFormat', 4, '<I'), ('polarity', 4, '<I'), ('binLength', 4, '<I'), ('binStart', 4, '<I'), ('binEnd', 4, '<I'), ('xDim', 4, '<I'), ('yDim', 4, '<I'), ('shotsPerPixel', 4, '<I'), ('pixelPerScan', 4, '<I'), ('nofScans', 4, '<I'), ('nofShots', 4, '<I'), ('deltaT', 8, '<d'))`

`shot_to_tof_mapper()`

`rimseval.data_io.crd_utils.`**`shot_to_tof_mapper`**(*ions_per_shot: array*) → array

Mapper for ions_to_shot to all_tofs.

Takes ions_per_shot array and creates a mapper that describes which ranges in the all_tofs array a given shot refers to.

Parameters

`ions_per_shot` – Ion per shots array.

Returns

Mappeing array shots to ToF.

LST to CRD File Transformer

class rimseval.data_io.lst_to_crd.LST2CRD(*file_name: Path = None*, *channel_data: int = None*, *channel_tag: int = None*)

Convert list files to CRD files.

Example:

```
>>> from pathlib import Path
>>> from rimseval.data_io import LST2CRD
>>> file = Path("path/to/file.lst")
>>> lst = LST2CRD(file_name=file, channel_data=1, tag_data=None)
>>> lst.read_list_file()
>>> lst.write_crd()
```

class ASCIIFormat(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Available formats for this routine that are already implemented.

Various formats that are implemented when dealing with ASCII data. The value is composed of a tuple of 2 entries.

- 0: entry: width of the binary number (binary_width)
- 1: Tuple of tuples with start, stop on where to read 0: sweep - 1: time - 2: channel

ASC_1A = (48, ((0, 16), (16, 44), (45, 48)))

ASC_9 = (64, ((1, 21), (21, 59), (60, 64)))

class BinWidthTDC(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bin width defined by instrument.

These are only FASTComTec instruments. The name of each entry is equal to the identifier that can be found in a datafile (lst). The entries of the enum are as following:

- binwidth in ps

MCS8A = 80

MPA4A = 100

class DATFormat(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Available formats (time_patch) for binary data.

Various binary data formats are incorporated. Value is compmosed of 2 entries:

- 0: Data length in bytes
- 1: Encoding of the binary value to read with struct.unpack()
- 2: Tuple, Where in the decoded list are: 0: sweep - 1: time - 2: channel

DAT_9 = (8, '<')

property channel_data: int

Get / set the channel number of the data.

Returns
Channel number of data

Raises
`TypeError` – Channel number is not an integer.

`property channel_tag: int`

Get / set the channel number of the tag.

Returns
Channel number of tag

Raises
`TypeError` – Channel number is not an integer.

`property data_format:` *`ASCIIFormat`*

Select the data format to use to convert the LST file to CRD.

Returns
The currently chosen data format.

Raises
`TypeError` – Data format is not a DataFormat enum.

`property file_name: Path`

Get / set the file name for the file to be read / written.

Returns
The path and file name to the selected object.

Raises
`TypeError` – Path is not a *pathlib.Path* object.

`read_list_file()` → None

Read a list file specified in *self.file_name*.

This routine sets the following parameters of the class:

- self._file_data
- self._tag_data (if a tag was selected)

This routine sets the following information parameters in self._file_info:

- "bin_width": Sets the binwidth in ps, depending on the instrument
- "calfact": Calibration factor, to scale range to bins
- "data_type": Sets the data type, 'ascii' for ASCII or 'dat' for binary, str
- "shot_range": shot range
- "timestamp": Time and date of file recording
- "time_patch": Data format, as reported by Fastcomtec as time_patch. as str

Raises

- `ValueError` – File name not provided.
- `ValueError` – Channel for data not provided.
- `OSError` – The Data Format is not available / could not be found in file.
- `NotImplementedError` – The current data format is not (yet) implemented.

`set_data_format()`

Set the data format according to what is saved in file_info dictionary.

The "data_type" and "time_patch" values must be present in the dictionary. Writes the information to itself, to the *_data_format* variable.

Raises

- `NotImplementedError` – Binary data are currently not supported.
- `ValueError` – Needs to be binary or ASCII data.

`write_crd()` → None

Write CRD file(s) from the data that are in the class.

Note: A file must have been read first. Also, this routine doesn't actually write the crd file itself, but it handles the tags, etc., and then sources the actual writing task out.

Raises

- `ValueError` – No data has been read in.
- `OSError` – Data is empty.

LST File Utilities

JIT compiled functions for processing *LST* files.

`ascii_to_ndarray()`

`rimseval.data_io.lst_utils.`**`ascii_to_ndarray`**(*data_list: List[str]*, *fmt:* ASCIIFormat, *channel: int*, *tag: int = None*) → Tuple[ndarray, ndarray, List]

Turn ASCII LST data to a numpy array.

Takes the whole data block and returns the data in a properly formatted numpy array. If channels other than the selected ones are available, these are written to a List and also returned as `other_channels`.

Parameters

- `data_list` – Data, directly supplied from the TDC block.
- `fmt` – Format of the data
- `channel` – Channel the data is in
- `tag` – Channel the tag is in, or None if no tag

Returns

Data, Tag Data, Other Channels available

`get_sweep_time_ascii()`

`rimseval.data_io.lst_utils.`**`get_sweep_time_ascii`**`(`*data: str*, *sweep_b: Tuple[int, int]*, *time_b: Tuple[int, int]*`)` → Tuple[int, int]

Get sweep and time from a given ASCII string.

Parameters

- **`data`** – ASCII string
- **`sweep_b`** – Boundaries of sweep
- **`time_b`** – Boundaries of time

Returns
sweep, time

`transfer_lst_to_crd_data()`

`rimseval.data_io.lst_utils.`**`transfer_lst_to_crd_data`**`(`*data_in: ndarray*, *max_sweep: int*, *ion_range: int*`)` → Tuple[ndarray, ndarray, bool]

Transfer lst file specific data to the crd format.

Parameters

- **`data_in`** – Array: One ion per line, two entries: sweep first (shot), then time
- **`max_sweep`** – the maximum sweep that can be represented by data resolution
- **`ion_range`** – Valid range of the data in multiples of 100ps bins

Returns
Array of how many ions are in each shot, Array of all arrival times of these ions, and a bool if there are any ions out of range

Excel Writer

`workup_file_writer()`

`rimseval.data_io.excel_writer.`**`workup_file_writer`**`(`*crd:* CRDFileProcessor, *fname: Path*, *timestamp: bool = False*`)` → None

Write out an Excel workup file.

This is for the user to write out an excel workup file, which will already be filled with the integrals of the given CRD file.

Parameters

- **`crd`** – CRD file processor file to write out.
- **`fname`** – File name for the file to write out to.
- **`timestamp`** – Create a column for the time stamp? Defaults to `False`

Integral export / import

`export()`

`rimseval.data_io.integrals.`**`export`**(*crd: CRDFileProcessor*, *fname: Path = None*) → None

Export integrals to csv file.

If no file name is given, the file name of the CRD file is used, '_int' is added, and '.csv' is used as the extension.

Header lines start with a #, all data lines are comma separated and labels for rows and columns are given within the data.

Parameters

- **`crd`** – CRD file.
- **`fname`** – File name to export to (optional): csv file.

Raises

`ValueError` – CRD file has no integrals.

`load()`

Export functions

`tof_spectrum()`

`rimseval.data_io.export.`**`tof_spectrum`**(*crd: CRDFileProcessor*, *fname: Path*, *bins: int = 1*) → None

Export time of flight spectra to csv file.

Parameters

- **`crd`** – CRD file.
- **`fname`** – File name to export to.
- **`bins`** – How many data to bin.

`mass_spectrum()`

`rimseval.data_io.export.`**`mass_spectrum`**(*crd: CRDFileProcessor*, *fname: Path*, *bins: int = 1*) → None

Export time of flight and mass spectra to csv file.

Parameters

- **`crd`** – CRD file.
- **`fname`** – File name to export to.
- **`bins`** – How many data to bin.

2.15.5 GUIs

These GUIs are part of the API package in order to bring some functionality to the user when working on the command line. Here, we solely use `matloplib` and Qt frontends. This is **not** the description of the main GUI, which can be found in *Graphical User Interface*.

Mass Calibration GUIs

Define the mass calibration by clicking on a plot.

`CreateMassCalibration`

`class rimseval.guis.mcal.`**`CreateMassCalibration`**(*crd:* CRDFileProcessor, *logy=True*, *mcal: array = None*, *theme=None*)

QMainWindow to create a mass calibration.

`append_to_mcal`(*tof: float*, *mass: float*) → None

Append a given value to the mass calibration.

Parameters

- **`tof`** – Time of flight.
- **`mass`** – Mass.

`apply`()

Apply the mass calibration and return it.

`check_mcal_length`()

Check length of mcal to set button statuses, start guessing.

`guess_mass`(*tof: float*) → str

Guess the mass depending on what is currently set.

If not drawn from iniabu, I assume the user wants even masses!

Parameters

`tof` – Time of Flight (us)

Returns

Guessed mass.

`query_mass`(*tof: float*) → float | None

Query mass from user.

Query a mass from the user using a QInputDialog.

Parameters

`tof` – Time of flight of the clicked area.

Returns

Mass of the peak as given by user.

`right_click_event`(*xpos: float*, **args*, ***kwargs*) → None

Act on an emitted right click event.

secondary_axis(*params: List = None*, *visible: bool = False*) → None

Toggle secondary axis.

Parameters

- **params** – Parameters for the transfer functions t0 and const. Only required when visible=True.
- **visible** – Turn it on? If so, params are required.

signal_calibration_applied

pyqtSignal(*types, name: str = …, revision: int = …, arguments: Sequence = …) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

undo_last_mcal()

Undo the last mass calibration by popping the last entry of list.

create_mass_cal_app()

rimseval.guis.mcal.**create_mass_cal_app**(*crd: CRDFileProcessor*, *logy: bool = True*, *theme=None*) → None

Create a PyQt5 app for the mass cal window.

Parameters

- **crd** – CRD file to calibrate for.
- **logy** – Should the y axis be logarithmic? Defaults to True.
- **theme** – Theme of GUI, requires `pyqtdarktheme` to be installed

find_closest_iso()

rimseval.guis.mcal.**find_closest_iso**(*mass: float*, *key: List = None*) → Tuple[str, float]

Find closest iniabu isotope to given mass and return its name.

If a key is given, will only consider that element, otherwise all.

Parameters

- **mass** – Mass of the isotope to look for.
- **key** – An element or isotope key that is valid for iniabu.

Returns

Closest isotpoe, name and mass as tuple.

Integrals & Background

Classes to define integrals and backgrounds. These are both very similar in nature, therefore, one superclass is created and individual routines subclass this one further.

`DefineAnyTemplate`

class `rimseval.guis.integrals.`**`DefineAnyTemplate`**(*crd: CRDFileProcessor*, *logy=True*, *theme: str = None*)

Template to define integrals and backgrounds.

`apply`()

Apply the mass calibration and return it.

`button_pressed`(*name: str*)

Define action for left click of a peak button.

Parameters
`name` – Name of the peak.

`check_peak_overlapping`(*peak_pos: array*) → List | None

Check if a given peak position overlaps with any other peaks.

Parameters
`peak_pos` – Position of peak, 2 entry array with from, to.

Returns
List of all peak names that it overlaps or `None`.

`clear_layout`(*layout*) → None

Clear a given layout of all widgets, etc.

`create_buttons`()

Create the buttons in the right menubar.

`mouse_right_press`(*xpos: float*) → None

Act on right mouse button pressed.

Parameters
`xpos` – Position on x axis.

`mouse_right_released`(*xpos: float*) → None

Right mouse button was released.

Parameters
`xpos` – Position on x axis.

`peaks_changed`()

Go through the list of peaks, make buttons and shade areas.

`shade_peaks`()

Shade the peaks with given integrals.

`sort_integrals`()

Sort the names and integrals using routine from processor_utilities.

user_input(*peak_pos: array*, *name: str = ''*) → None

Query user for position.

Parameters

- **peak_pos** – Sorted array, left and right position of peak.
- **name** – Name to preset the line-edit with.

DefineBackgrounds

class rimseval.guis.integrals.**DefineBackgrounds**(*crd:* CRDFileProcessor, *logy=True*, *theme: str = None*)

QMainWindow to define backgrounds.

apply()

Apply the mass calibration and return it.

button_pressed(*name: str*)

Delete a peak from consideration names and values list.

Parameters
name – Name of the peak.

peaks_changed()

Go through the list of peaks, make buttons and shade areas.

shade_backgrounds()

Go through background list and shade them.

Note: Canvas is not cleared prior to this!

signal_backgrounds_defined

pyqtSignal(*types, name: str = …, revision: int = …, arguments: Sequence = …) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

user_input(*bg_pos: array*, *name: str = ''*) → None

Query user for position of background.

Parameters

- **bg_pos** – Sorted array, left and right position of background.
- **name** – Name of the peak, for reloading the UI.

`DefineIntegrals`

class `rimseval.guis.integrals.`**`DefineIntegrals`**(*crd: CRDFileProcessor*, *logy=True*, *theme: str = None*)

QMainWindow to define integrals.

`apply`()

Apply the mass calibration and return it.

`button_pressed`(*name: str*)

Delete a peak from consideration names and values list.

Parameters
`name` – Name of the peak.

`peaks_changed`()

Go through the list of peaks, make buttons and shade areas.

`signal_integrals_defined`

pyqtSignal(*types, name: str = …, revision: int = …, arguments: Sequence = …) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

`user_input`(*peak_pos: array*, *name: str = ''*) → None

Query user for position.

Parameters

- **`peak_pos`** – Sorted array, left and right position of peak.
- **`name`** – Name to pre-populate line-edit with.

`define_backgrounds_app()`

`rimseval.guis.integrals.`**`define_backgrounds_app`**(*crd: CRDFileProcessor*, *logy: bool = True*) → None

Create a PyQt5 app for defining backgruonds.

Parameters

- **`crd`** – CRD file to calibrate for.
- **`logy`** – Should the y axis be logarithmic? Defaults to True.

`define_integrals_app()`

`rimseval.guis.integrals.`**`define_integrals_app`**(*crd: CRDFileProcessor*, *logy: bool = True*) → None

Create a PyQt5 app for defining integrals.

Parameters

- **`crd`** – CRD file to calibrate for.
- **`logy`** – Should the y axis be logarithmic? Defaults to True.

`tableau_color()`

`rimseval.guis.integrals.`**`tableau_color`**`(`*it: int = 0*`)` → str

Return nth color from matplotlib TABLEAU_COLORS.

If out of range, start at beginning.

Parameters
`it` – Which tableau color to get.

Returns
Matplotlib color string.

Matplotlib Canvas Classes

These classes create spectra plotters and handling for theses specific tasks. Uses the matplotlib `Qt5Agg` backend.

`PlotSpectrum`

Plots the spectrum and serves it as a matplotlib figure. It adds toolbar and canvas (see below) plus makes two layouts available, a bottom layout and a right layout. This allows the addition to QWidgets into this layouts later on.

The plot widget adds one button in the bottom layout to toggle logarithmic axes for the vertical / signal axis.

`class rimseval.guis.mpl_canvas.`**`PlotSpectrum`**`(`*crd: CRDFileProcessor*, *logy: bool = True*, *theme: str = None*`)`

QMainWindow to plot a ToF or mass spectrum.

`logy_toggle``()`
Toggle logy.

`plot_ms``()`
Plot mass spectrum.

`plot_tof``()`
Plot ToF spectrum.

`MplCanvasRightClick`

Handle right-click on `matplotlib` canvas. Releases to signals: one on right mouse button press and one on right mouse button release. These signals send the x and y position where the mouse event took place.

`class rimseval.guis.mpl_canvas.`**`MplCanvasRightClick`**`(`*figure: Figure*`)`

MPL Canvas reimplementation to catch right click.

On right click, emits the coordinates of the position in axes coordinates as a signal of two floats (x_position, y_position).

`emit_mouse_position``(`*event*, *case*`)`
Emit a signal on a right mouse click event.

Here, bring up a box to ask for the mass, then send it, along with the time the mass is at, to the parent class receiver.

Parameters

- **event** – PyQt event.
- **case** – Which case are we handling? Currently implemented are “pressed”, “released”.

mouse_right_press_position

pyqtSignal(*types, name: str = ..., revision: int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal’s arguments.

mouse_right_release_position

pyqtSignal(*types, name: str = ..., revision: int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal’s arguments.

MyMplNavigationToolbar

Re-implementation of the matplotlib navigation toolbar. After zooming in, the zoom function is automatically deactivated.

class rimseval.guis.mpl_canvas.**MyMplNavigationToolbar**(**args*, ***kwargs*)

My own reimplementation of the navigation toolbar.

Features: - untoggle zoom button after zoom is finished.

release_pan(*event*)

Run a normal pan release event and then untoggle button.

release_zoom(*event*)

Run a normal zoom release event and then untoggle button.

Plots

These routines allow for specialty plots, i.e., to plot figures that are enot regularly used or needed during data evaluation but can give useful further information on the instrument, etc.

dt_ions()

Create a PyQt app and run it.

rimseval.guis.plots.**dt_ions**(*crd: CRDFileProcessor*, *logy: bool = False*, *theme: str = None*, *max_ns: float = None*) → None

Plot ToF difference between ions for shots with 2+ ions.

Parameters

- **crd** – CRD file to process.

- **`logy`** – Plot with logarithmic y axis? Defaults to `True`
- **`theme`** – Theme to plot in, defaults to `None`.
- **`max_ns`** – Maximum time to plot in ns. If None, plots all.

`integrals_packages()`

Create a PyQt app and run it.

`rimseval.guis.plots.`**`integrals_packages`**`(`*crd: CRDFileProcessor*, *logy: bool = False*, *theme: str = None*`)` → None

Plot all the integrals versus package number for data split into packages.

Parameters

- **`crd`** – CRD file to process.
- **`logy`** – Plot with logarithmic y axis? Defaults to `True`
- **`theme`** – Theme to plot in, defaults to `None`.

`nof_ions_per_shot()`

Create a PyQt app and run it.

`rimseval.guis.plots.`**`nof_ions_per_shot`**`(`*crd: CRDFileProcessor*, *logy: bool = False*, *theme: str = None*`)` → None

Plot a histogram of the number of shots in a given crd file.

The histogram is compared with the theoretical curve based on poisson statistics.

Parameters

- **`crd`** – CRD file to process.
- **`logy`** – Plot with logarithmic y axis? Defaults to `True`
- **`theme`** – Theme to plot in, defaults to `None`.

`PlotFigure`

Class to plot a figure.

`class rimseval.guis.plots.`**`PlotFigure`**`(`*logy: bool = False*, *theme: str = None*`)`

QMainWindow to plot a Figure.

`logy_toggle``()`

Toggle logy.

DtIons

Matplotlib PyQt figure to plot histogram for arrival time differences between ions.

class rimseval.guis.plots.**DtIons**(*crd: CRDFileProcessor*, *logy: bool = False*, *theme: str = None*, *max_ns: float = None*)

Plot time differences between ions.

calc_and_draw() → None

Calculate the required data and draw it.

IntegralsPerPackage

Matplotlib PyQt figure to integrals per package.

class rimseval.guis.plots.**IntegralsPerPackage**(*crd: CRDFileProcessor*, *logy: bool = False*, *theme: str = None*)

Plot integrals of all packages versus package number.

calc_and_draw() → None

Create the plot for all the defined integrals.

IonsPerShot

Matplotlib PyQt figure to plot histogram of ions per shot.

class rimseval.guis.plots.**IonsPerShot**(*crd: CRDFileProcessor*, *logy: bool = False*, *theme: str = None*)

Plot histogram for number of ions per shot.

calc_and_draw() → None

Calculate the histogram and plot it.

2.15.6 Multiple File Processor

In this class, the user can process multiple CRD files simultaneously. It is mainly intended as an easy interface class for the RIMSEvalGUI.

MultiFileProcessor: Multi File Processor class

class rimseval.multi_proc.**MultiFileProcessor**(*crd_files: List[Path]*)

Class to process multiple CRD files at ones.

Example:

```
>>> file_names = [Path("file1.crd"), Path("file2.crd"), Path("file3.crd")]
>>> mfp = MultiFileProcessor(file_names)
>>> mfp.num_of_files
3
>>> mfp.peak_fwhm = 0.02  # set full with half max for all files
```

apply_to_all(*id: int*, *opt_mcal: bool = False*, *bg_corr: bool = False*)

Take the configuration for the ID file and apply it to all files.

Parameters

- **id** – Index where the main CRD file is in the list
- **opt_mcal** – Optimize mass calibration if True (default: False)
- **bg_corr** – Perform background correction?

close_files() → None

Destroys the files and frees the memory.

close_selected_files(*ids: List[int]*, *main_id: int = None*) → int

Close selected files.

Close the ided files and free the memory. If the main_id is given, the program will return the new ID of the main file in case it got changed. If the main file is gone or no main file was provided, zero is returned.

Parameters

- **ids** – List of file IDs, i.e., where they are in `self.files`.
- **main_id** – ID of the main file, an updated ID will be returned if main file is present, otherwise zero will be returned.

Returns

New ID of main file if present or given. Otherwise, return zero.

property files: List[*CRDFileProcessor*]

Return a list of files.

If the files are not opened, it will open and read them.

Returns

List of CRDFileProcessor instances with files opened

static load_calibration_single(*crd:* CRDFileProcessor, *secondary_cal: Path = None*) → None

Load a single calibration for a CRD file.

Loads the primary calibration (i.e., the one with the same name but .json qualifier) if it exists. Otherwise, if a secondary calibration is given, it will try to load that one. If that file does not exist either, nothing will be done.

Parameters

- **crd** – CRD file to load the calibration for.
- **secondary_cal** – Optional, calibration to fall back on if no primary calibration is available.

load_calibrations(*secondary_cal: Path = None*) → None

Load calibration files for all CRDs.

This routine checks first if a calibration file with the same name exists. If so, it will be loaded. If no primary calibration is present and a secondary calibration filename is given, the program will try to load that one if it exists. Otherwise, no calibration will be loaded.

Parameters

secondary_cal – File for secondary calibration. Will only be used if it exists.

property num_of_files: int

Get the number of files that are in the multiprocessor.

open_additional_files(*fnames: List[Path]*, *read_files=True*, *secondary_cal=None*) → None

Open additional files to the ones already opened.

Files will be appended to the list, no sorting.

Parameters

- **fnames** – List of filenames for the additional files to be opened.
- **read_files** – Read the files after opening?
- **secondary_cal** – Secondary calibration file for loading calibrations.

open_files() → None

Open the files and store them in the list.

property peak_fwhm: float

Get / Set FWHM of each peak.

The getter returns the average, the setter sets the same for all.

Returns

Average peak FWHM in us.

read_files() → None

Run spectrum_full on all CRD files.

signal_processed

pyqtSignal(*types, name: str = …, revision: int = …, arguments: Sequence = …) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

2.15.7 Utility functions

This folder contains further functions that help with various tasks. These functions are split up into individual files, depending on their specific tasks.

Fitting functions

gaussian()

rimseval.utilities.fitting.**gaussian**(*xdat: float | ndarray*, *coeffs: ndarray*) → float | ndarray

Return the value of a Gaussian for given parameters.

Parameters

- **xdat** – X value
- **coeffs** – Coefficients, [mu, sigma, height]

Returns
Value of the Gaussian.

`residuals_gaussian()`

`rimseval.utilities.fitting.`**`residuals_gaussian`**`(`*coeffs: ndarray*, *ydat: ndarray*, *xdat: ndarray*`)` → ndarray

Calculate residuals and return them.

Parameters

- **`coeffs`** – Coefficients for model
- **`ydat`** – Y data to compare with
- **`xdat`** – X data for model fit

Returns
Residual of the Gaussian.

Peirce's rejection criterion

Functions to deal with Peirce's rejection criterion. See Wikipedia and this PDF file for details.

`peirce_criterion()`

`rimseval.utilities.peirce.`**`peirce_criterion`**`(`*n_tot: int*, *n: int*, *m: int = 1*`)` → float

Peirce's criterion.

Returns the threshold error deviation for outlier identification using Peirce's criterion based on Gould's methodology. This routine is heavily copied from Wikipedia

Parameters

- **`n_tot`** – Total number of observations.
- **`n`** – Number of outliers to be removed.
- **`m`** – Number of model unknowns, defaults to 1.

Returns
Error threshold *R* (Ross, 2003) / Square root of *x**2* (Gould, 1955)

`reject_outliers()`

`rimseval.utilities.peirce.`**`reject_outliers`**`(`*data: ndarray*, *m: int = 1*`)` → Tuple[float, float, ndarray, ndarray]

Apply Peirce's criterion to reject outliers.

Algorithm implmeneted as given by Ross (2003).

Parameters

- **`data`** – All data points.
- **`m`** – Number of model unknowns, defaults to 1.

Returns
New average and standard deviation, Array with the outliers.

Utilities

Further utility routines that do not fit anywhere else.

`rimseval.utilities.utils.`**`not_index`**(*ind: array*, *length: int*) → array

Reverse an index.

After filtering ions, e.g., using *np.where*, and keeping the indices of an array, this routine creates the opposite, i.e., an array of the indices that have not been filtered.

Parameters

- **`ind`** – Array of all the indexes.
- **`length`** – Length of the original index to take out of.

Returns
The reversed index.

Raises
`ValueError` – Max index is larger than the total length, which should not be.

Example:

```
>>> a = np.arange(7)
>>> b = np.where(a < 4)[0]
>>> b
array([0, 1, 2, 3])
>>> not_index(b)
array([4, 5, 6])
```

INDEX